Processing Multimedia Workloads on Heterogeneous Multicore Architectures

Håkon Kvale Stensland

February 2015

Abstract

Processor architectures have been evolving quickly since the introduction of the central processing unit. For a very long time, one of the important means of increasing performance was to increase the clock frequency. However, in the last decade, processor manufacturers have hit the so-called power wall, with high heat dissipation. To overcome this problem, processors were designed with reduced clock frequencies but with multiple cores and, later, heterogeneous processing elements. This shift introduced a new challenge for programmers: Legacy applications, written without parallelization in mind, gain no benefits from moving to multicore and heterogeneous architectures. Another challenge for the programmers is that heterogeneous architecture designs are very different with respect to caches, memory types, execution unit organization, and so forth and, in the worst case, a programmer must completely rewrite the application to obtain the best performance on the new architecture.

Multimedia workloads, such as video encoding, are often time sensitive and interactive. These workloads differ from traditional batch processing workloads with no real-time requirements. This work investigates how to use modern heterogeneous architectures efficiently to process multimedia workloads. To do so, we investigate both simple and complex workloads on multiple architectures to learn about the properties of these architectures. When programing multimedia workloads, it is very important to know how the algorithms perform on the target architecture. In addition, achieving high performance on heterogeneous architectures is not a trivial task, often requiring detailed knowledge about the architecture. We therefore evaluate several optimizations so we can learn how best to write programs for these architectures and avoid potential pitfalls. We later use the knowledge gained to propose a framework design and language called Parallel Processing Graph (P2G). The P2G framework is designed for multimedia workloads and supports heterogeneous architectures. To demonstrate the feasibility of the framework, we construct a proof-of-concept implementation. Two simple workloads show that we can express multimedia workloads in the system. We also demonstrate the scalability of the designed solution.

Acknowledgements

Working with the PhD has been a long journey, at times, it has been both frustrating and stressful, but it has mostly been lots of fun. I would like to thank my supervisors, Professor Carsten Griwodz and Professor Pål Halvorsen for interesting discussions, their deep insights and valuable feedback over the years.

I would also like to thank my colleagues Håvard Espeland and Paul Beskow who I have shared office with for good collaboration and inspiring discussions on the topic of processing multimedia workloads. During the work with this thesis I have supervised several master students. I would like to thank all of them, the discussions have been very inspiring, and have helped me with this thesis.

The work environment at Simula Research Laboratory and the Media-department has also been excellent. Here, I would like to thank Andreas Petlund, Kristian Evensen, Ragnhild Eg, Preben Olsen and Vamsidhar Gaddam for making Simula a great place to be.

Finally I would like to thank my family, friends and especially my wife Marianne, for being patient and always supporting me no matter what I decide to do.

Contents

x

List of Figures

List of Tables

Part I

Overview

Chapter 1

Introduction

1.1 Background and Motivation

Multimedia applications are one class of applications that typically follow the trend of increasing processing demands to continuously increase quality and perceived experience. For example, live interactive multimedia services are steadily growing in volume. In this respect, Internet users uploaded over 100 hours of video to YouTube every minute in 2014 [139]. In the future, consumers will demand features such as interactively refined video search, dynamic participation in video conferencing systems, and user-controlled views in live media content. To support these features, we must be able to process compute-intensive workloads such as those required in extracting video features to identify objects, in the calculation of three-dimensional depth information from camera arrays, or in generating Freeview video from multiple camera sources in real time. This adds further magnitudes of processing requirements to already computationally intensive tasks such as the traditional video encoding of high-definition videos.

Over the last decade, we have witnessed two paradigm shifts within modern processor architectures. The first shift was when single-core processors reached their power and frequency limits, forcing chip designers to start focusing on on-chip parallelism. This started with the introduction of IBM's POWER4 dual-core processor [118] and was followed by the introduction of Hyper-Threading Technology [60] on Intel's Pentium 4 processors. Today, dual- and quad-core processors from Intel and AMD are a commodity in desktop and laptop computers. Mobile devices have followed the same trend and several processor designs, such as Nvidia's Tegra 4 mobile system on a chip (SoC) [91], have a quad-core general-purpose processor.

The other paradigm shift was the introduction of heterogeneous processing architectures, such as the Cell Broadband Engine [54] from Sony, Toshiba, and IBM (STI) and the graphics processing units (GPUs) of Nvidia, AMD, and other vendors.

Heterogeneous processing architectures provide more computing power than traditional general-purpose single- and multicore systems. The processing cores have different instruction sets, use several different types of memory, and provide different programming abstractions compared to traditional desktop processors. Heterogeneous processing cores are often designed very differently from general-purpose cores, being more specialized toward solving specific tasks. Because of this, the cores in a heterogeneous architecture can utilize more of the die space on the chip for arithmetic and logic units (ALUs) and use

less space for caches and control logics. We are also witnessing the trend of heterogeneous processing in mobile devices, where all SoCs have dedicated processors for audio, video, and imaging. Even the latest generation of Intel x86 processors has a dedicated coprocessor called Quick Sync [61] for video encoding, decoding, and transcoding.

Today, programmers who want to utilize these heterogeneous processing architectures face several challenges. They must write their applications with a very detailed knowledge about the target architecture and, when the target architecture is changed or even upgraded to a new generation by the hardware vendors, applications optimized for one architecture will in some cases require a complete rewrite for utilization or at least efficient execution on the new architecture. We therefore need abstractions and programming concepts that will ease the development of applications for heterogeneous processing architectures. This thesis is meant as a step toward this goal.

1.1.1   Heterogeneous Architectures

Processor architectures have been evolving quickly since the introduction of the central processing unit (CPU) and vendors often release either a new architecture or an updated architecture every year. For a very long time, one of the important means of increasing performance was to increase the clock frequency of the processors. However, in the beginning of the 2000s, this approach started to become problematic [79]. As processors evolved, the chips also shrank, with the manufacturing process packing more and more transistors onto smaller areas. With the introduction of the 90nm process in 2004, several vendors hit the so-called power wall: The transistors, working at very high frequencies, leaked power and, to make them run stably, the voltage had to be increased, resulting in higher heat dissipation.

The CPU vendor's first solution to this challenge was to place multiple cores onto a single chip and AMD was the first to introduce such a processor for consumers, with the Athlon 64 X2 in 2004 [3]. Multi-core processors were nothing new and had previously been provided by multiple vendors, but only in high-end systems, with multiple processor sockets connected to the buses. The introduction of multicore systems was a paradigm shift for the average developer, who now had to parallelize applications to scale performance.

A different approach to continuously scale performance is to add simple cores or more specialized cores that can carry out certain tasks more quickly and more efficiently—hence the term *heterogeneous architectures*. Heterogeneous architectures in a simple form have been around for a long time. One example is from 1985, when Intel's 80386 processor had the option of adding an x87 floating-point coprocessor. This coprocessor was later integrated into the main processor core as an independent pipeline when the 80486 processor was launched in 1989. In Chapter 2, we examine the x86 core in more detail and see that what seems to be a single core is in fact built up of several heterogeneous elements.

Another example of heterogeneous architecture is the use of processor cores with different capabilities and instruction sets. One example of such an architecture is Intel's specialized network processor IXP1200, which was launched in 1999 [56]. This architecture had one general-purpose core to run the operating system and execute the control plane in the network and several specialized cores for packet processing. Another example of such a system in the consumer market is the Sony PlayStation 3, launched in 2006 [55].

The PlayStation 3 featured the Cell Broadband Engine, a heterogeneous processor optimized for floating point operations, which are important in both computer games and multimedia processing. Perhaps the most common heterogeneous architecture in computers today, however, consists of GPUs working together with the CPU. Over the last decade, GPUs have evolved from simple fixed-function pipelines to fully programmable processors. With the launch of CUDA in 2007 [92], Nvidia introduced low-cost high-performance computing to the masses.

However, programming heterogeneous architectures has proven to be a significant challenge. Legacy applications written without parallelization gain no benefit from the move to multicore and heterogeneous architectures. In some cases, legacy applications may even perform worse on modern architectures, because the architectures are focused toward adding more cores at a lower frequency instead of having fewer cores at a high frequency. Another challenge with heterogeneous architectures (e.g., GPUs) is that they can be very different in design in regard to caches, types of memory, execution unit organization, and so forth, and, in the worst case, the programmer must completely rewrite the application to obtain the best performance from a new architecture. Chapters 3 and 4 present our experience with developing multimedia workloads for some of these heterogeneous architectures.

1.1.2 Multimedia Workloads

A multimedia workload is often characterized as being time sensitive and iterative. Video processing is an example that involves a multimedia workload and it is explored later in this thesis. Encoding and decoding video are very computationally intensive operations, where calculations have to be done for all the pixels in a frame. Video encoding is a common operation in video processing; raw data from the imaging sensor are often compressed according to bandwidth or storage requirements defined by the scenario. The most common standard for video coding today is H.264 [131], which was defined by the International Telecommunication Union (ITU) and the International Organization for Standardization (ISO) as a video codec for everything from streaming video to mobile devices to TV broadcasting and high-definition videos stored on Blu-ray disks.

Since H.264 is currently the de facto standard for video coding in the industry, a great deal of research has been performed on how to optimize its encoding and decoding. Most of the devices available have dedicated hardware to either assist the processor or carry out the decoding. The same trend appears for encoding. Dedicated hardware implementations are fast and have low energy requirements. However, they lack the flexibility of software implementation, they are complex and expensive to manufacture, and, when parameters (e.g., the resolution or the frame rate) are changed, the hardware must also be changed in most cases. Multimedia workloads developed in software are more flexible and are easier to update and replace compared to hardware implementations, where the entire chip has to be changed for a different implementation. This thesis therefore focuses on software implementations of multimedia workloads.

The software multimedia workloads investigated in this thesis are independent filters and operations on video data organized in pipelines, often with real-time processing requirements. One example of such a multimedia workload is the real-time panorama stitching pipeline in the Bagadus soccer analysis system [114], which is investigated fur-

ther in Chapter 4.
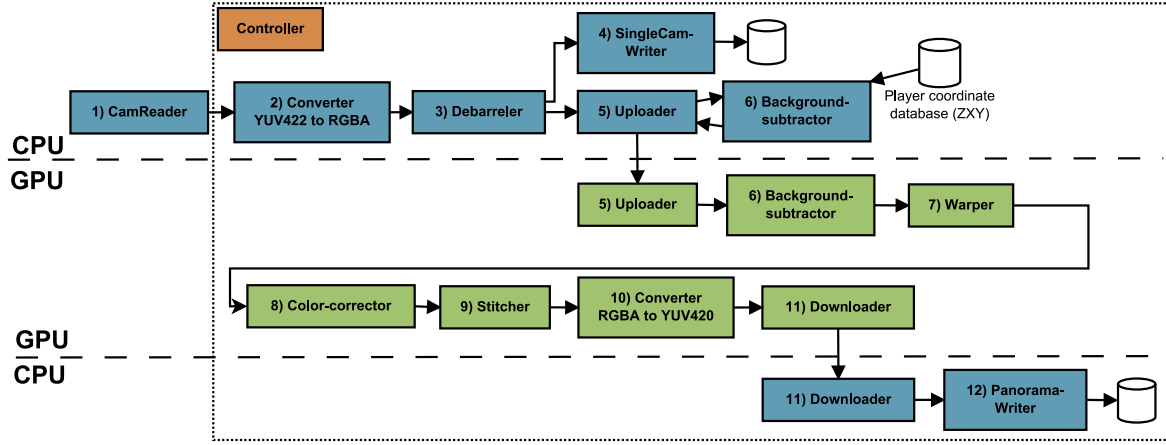
Figure 1.1: The real-time panorama video stitching pipeline in the Bagadus soccer analysis system [114].

Figure 1.1 provides an overview of the Bagadus pipeline. In the Bagadus pipeline, raw video data from multiple video cameras is read over the network in real time. The pipeline comprises several independent steps for converting the data between different formats, optimizing the videos, and finally stitching them together into a panoramic video. Most of the steps in the pipeline have dependencies that have to be satisfied before they can start processing and each step also has different degrees of parallelization potential. To fulfill the system's real-time requirements of 30 frames per second, the pipeline has to deliver an encoded panoramic video frame every 33 milliseconds. Another challenge of the Bagadus pipeline is the scalability of the video data, where adding more cameras, increasing the resolution of existing cameras, or increasing the system's frame rate will increase its computational demands. The addition of more and other more complex steps such as high dynamic range (HDR) rendering in the pipeline will also increase the complexity of the system.

1.2 Problem Statement

When used efficiently, modern heterogeneous architectures provide the processing power required by resource-hungry multimedia workloads. However, the diversity of resources to which developers are exposed makes it very hard to develop programs that are portable and scalable on multiple architectures. Even with new languages such as OpenCL, which are supposed to be a "recompile-only" solution, the applications must be tuned and in many cases hand optimized for the different heterogeneous architectures. In examining this area of computing, we proceeded according to the following problem statement:

> **How can programmers efficiently develop multimedia workloads for modern heterogeneous multicore architectures?**

The problem stated in this thesis focuses on how to use modern heterogeneous architectures to efficiently process multimedia workloads. We want to learn how programmers

need to think when writing their applications for these architectures. Many details about the architectures are also undocumented, so we need to learn how the architectures behave when processing multimedia workloads. We approached the problem statement in three steps, presented in Chapters 3, 4, and 5 and briefly formulated as follows:

- Learn about the behavior of heterogeneous architectures by implementing and evaluating prototypes where simple multimedia workloads run on a single heterogeneous architecture.

- Learn how to use multiple heterogeneous architectures to process a complex pipeline with several multimedia workloads running in real time.

- Propose new ways of designing and developing multimedia workloads with a framework for processing multimedia workloads on heterogeneous architectures.

1.3   Limitations

To reduce the scope of this thesis, we limited the number of heterogeneous architectures investigated. We also examine GPUs from only one vendor, Nvidia. The main reason for focusing on only one vendor is the availability of programming tools and documentation when the project started. The multimedia workloads we consider are data intensive and have good parallelization potential.

We focus only on processing multimedia workloads on the constrained resources of a single machine, not those of large-scale distributed systems. Many small devices today—such as smartphones, tablets, laptops, and desktop PCs—can process multimedia workloads, which is also expected by the users of such devices. Our main focus is therefore the efficient utilization of resources on these platforms.

1.4   Research Method

As defined by the Association for Computing Machinery (ACM) Education Board [25] in 1989, the discipline of computer science is divided into three major paradigms. Each of these paradigms has its roots in different areas of science and all can be applied to computing and computer science. The ACM Education Board states that the paradigms are intertwined and that it is irrational to say that any one of the paradigms is fundamental. These three computer science paradigms are as follows.

- *Theory*: This paradigm has its roots in mathematics. It defines objects of study and hypothesizes their interrelations; it then determines whether these relations are true and interprets the results. This paradigm is concerned with the ability to describe and prove relationships among the objects of study.

- *Abstraction*: This paradigm is from the experimental research field and consists of four stages. A scientist forms a hypothesis, constructs a model, makes a prediction before designing an experiment, and collects data. This paradigm is concerned with the ability to use predictions that can be compared with real-world situations.

- *Design*: This paradigm is from the field of engineering and involves building a device or system to solve the given problem. The scientist states the requirements and specifications of a design and the system's implementation. The system is then tested and the previous steps can be repeated if the system does not meet the requirements. This paradigm is concerned with the ability to implement specific instances and use them to perform useful actions.

All three of these paradigms are applied in our field of multimedia systems research, the one to use depending on the problem to be solved. In our case, in which we want to investigate how to use heterogeneous multicore architectures for processing multimedia workloads, we first determined if the *theory* paradigm could be applied. The *theory* paradigm requires a precise definition and modeling of the multimedia software and the heterogeneous multicore hardware we want to use. This is particularly challenging with hardware, since many of the details about the low-level behavior of schedulers and the execution pipeline are known only to the hardware vendors. We therefore rejected this paradigm for this thesis. The next paradigm is the *abstraction* paradigm. This paradigm is often used, typically in combination with simulators. One can model the behavior of both the software and the hardware architecture in a simulator to try to capture the system's behavior. We used this paradigm in one of the initial investigations on using heterogeneous architectures for multimedia workloads. However, the challenge is that simulators are no better than the models used and, as mentioned previously, because of the black boxes in the hardware, many of the details about the architectures are unknown. It is therefore impossible to know how good the simulated behavior is without conducting experiments on real systems and we rejected this theory as well. The last paradigm is the *design* paradigm. With this paradigm, we specified the requirements and built prototype systems. These prototypes were evaluated on real-world systems and, based on the data gathered, the prototype improved over multiple iterations. One of the challenges with this paradigm is that it can take significant effort to implement these prototypes, which in itself is not necessarily scientific work. The fact that many of the details of the hardware are not published by their vendors is a challenge in our investigation of how to program multimedia workloads for modern heterogeneous multicore architectures. We therefore have to make many assumptions and simplifications if we want to simulate exactly how the hardware works. We therefore decided to use the *design* paradigm in this thesis. It requires more engineering work, but we were not limited by inaccurate models as a basis for the simulations.

1.5 Main Contributions

The main research question in Section 1.2 states the challenge on how to efficiently use modern heterogeneous multicore architectures to process multimedia workloads. This problem statement is addressed from both a low-level standpoint, where we carry out experiments with simple multimedia workloads on different heterogeneous multicore architectures, and a more high-level one, where we present, implement, and evaluate a programming model to support the real-time processing of multimedia data. The papers that are part of the contribution of this thesis have been published in a number of peer-reviewed conference proceedings and international journals and are included in full

in Chapter 6. Several other papers related to processing multimedia workloads were not included in the thesis to limit its scope. A summary of the main contributions included is as follows.

- We learned about the behavior of heterogeneous architectures with simple multimedia workloads. The first architecture we experimented with was Intel's IXP network processor. This architecture was used for experiments involving network protocol translation [32]. The next architecture was the x86 processor architecture. Here, the first case study involved the efficient implementation of Motion JPEG (MJPEG) encoding [112]. We also conducted case studies on using multiple x86 cores for multi-rate video encoding [34] and running a multi-threaded game server prototype [102]. For the GPU architecture, we conducted case studies on its memory [94], optimization of host-to-device communication [13], and cheat detection [117]. We also revisited the MJPEG workload with both the GPU [112] and Cell architectures. The IXP architecture provided experience in working with a state-of-the-art (at the time) heterogeneous architecture with several special features. With the MJPEG workload on the x86 architecture, we learned the importance of selecting algorithms optimized for the target architecture and the benefits of using the vector unit available on all modern processors. The shared memory model on the x86 revealed the importance of reusing parts of computationally heavy multimedia workloads and of using the optimal number of threads for the number of cores available. On the GPU architecture, we learned about the importance of using the memory architecture correctly and optimizing transfers between the host CPU and the GPU with asymmetrical transfers. We also observed that when offloading multimedia workloads to a GPU, they have to be large enough to compensate for the added latency of data transfers and launching kernels on the GPU. With the MJPEG workload, as on the x86, we learned the importance of optimizing algorithms for the target architecture. On the Cell architecture, we also learned the same lessons as on the x86 and GPUs: It is important to select an algorithm suited for the target architecture and always vectorize your workload when possible. Our MJPEG experiments also suggested that programmers prefer programming models exposed by the GPU compared to models exposed by the vector unit on x86 and the Cell.

- The knowledge obtained from investigating simple multimedia workloads was used to investigate a more complex multimedia workload, that is, part of the Bagadus soccer analysis system, which has three components: a tracker subsystem, an analytics subsystem, and a video subsystem [114]. The complex workload used in these experiments was that of a video subsystem that involved the real-time capture, pre-processing, stitching, and encoding of a panoramic video stream from a soccer stadium [113]. Here, we had to optimize the workload for multiple heterogeneous architectures to run the workload in real time. By implementing the subsystem as a pipeline and optimizing for both the x86 architecture and GPUs, we were able to capture the five 720p streams and stitch them into a panoramic video on a single commodity gaming PC.

- We used our knowledge from processing both simple and complex multimedia workloads on heterogeneous systems and from our evaluation of multicore scheduling

mechanisms [115] as follows. We proposed a programming language and framework that exposes the parallelization opportunities of a multimedia workload for a runtime that allows for the efficient execution of a workload on the available heterogeneous hardware [31]. We also developed a prototype of this system running on a single machine with support for multicore x86 processors, together with several simple multimedia workloads running on the system. In addition, we conducted experiments demonstrating the system's usability for multimedia workloads.

There are many opportunities for further work within this field; however, we aim to show the essential challenges of using heterogeneous multicore architectures for processing multimedia workloads and the essential considerations that programmers must make when choosing both the architectures and the algorithms.

1.6   Outline

This thesis is written as a collection of paper and it is therefore organized in two parts. Part I provides an introduction and places the research papers in context and Part II includes the selected full papers.

The first part is organized as follows: In Chapter 2, we introduce the heterogeneous architectures used in our research. We also look at the low-level programming abstractions that are used when programming the different heterogeneous architectures. In Chapter 3, we research several simple multimedia workloads on the heterogeneous architectures and we discuss how to use the architectures for the different multimedia workloads and how to structure the workloads to obtain the best performance from the architectures. This is followed up in Chapter 4 with an investigation of a more complex pipeline, with several components running on different heterogeneous architectures. Based on our results in Chapters 3 and 4, we introduce in Chapter 5 the P2G framework for running multimedia workloads on heterogeneous architectures. We benchmark and evaluate a set of multimedia workloads within the P2G framework. Chapter 6 gives an overview of the research papers and, finally, Chapter 7 provides a conclusion.

Chapter 2

Heterogeneous Computing

In this chapter, we introduce the heterogeneous hardware architectures used for experiments in this thesis. This introduction provides insight into the history of the different architectures; it gives a basic introduction to the architecture by looking at the state-of-the-market products available in each of the architectures.

The architectures we have chosen are very different with respect to the amount of available computational resources—floating point units, arithmetic and logic units (ALUs), and so forth—and how these are connected in the architecture's execution pipeline. The memory types and layouts are also very different, some of the architectures having a shared memory model that hides the memory management from programmers and other architectures having an explicit memory model, which gives programmers full control over the memory. We also examine caches, how they are organized, and whether programmers have any control over how they are used. Next, we look at the buses that connect the resources within the processors and as well as the processors with each other and resources in the machine. Finally, we look at what these heterogeneous architectures expose to programmers. We investigate three examples of what the architectures expose to programmers and how the programming model is used to hide some of the architecture's complexity.

2.1 Hardware Architectures

In this section, we take a more detailed look at some heterogeneous architectures. First, we look at the family of x86 processors, specifically those produced by Intel. Next, we introduce the Intel IXP2400 network processing unit (NPU) with a heterogeneous architecture before looking into the Cell Broadband Engine (CBE). Finally, we take a look at Nvidia graphics processing units (GPUs) used for general-purpose programming.

2.1.1 Intel x86 Processor Architecture

The x86 processor architecture has a long history, dating back to Intel's 8086 central processing unit (CPU) released in 1978 as a fully 16-bit processor. One of the reasons this instruction set succeeded in becoming the dominant instruction set in the mainstream computer market was the fact that IBM selected the 8086 for the original IBM PC. Over the years, the x86 instruction set has undergone many extensions (32 bit in 1985 and

64 bit in 2003) and additions. However, the instruction set has always been backward compatible with previous versions. A modern x86 processor from 2014 is still able to execute 16-bit code compiled for the original 8086. Over the years, several vendors have also been designing and manufacturing CPUs compatible with the x86 instruction set (e.g., Intel, AMD, VIA, and Cyrix). However, today only two remain and, of those, Intel is the dominant vendor.

Originally, x86 was a little-endian, variable instruction length complex instruction set computer (CISC) design. However, over the years, the processors executing the instruction set changed greatly. The introduction of superscalar pipelines, where a CPU with a single instruction stream can dynamically check data dependencies and process multiple instructions per clock cycle, made it possible for the x86 processors to execute more operations in parallel in a single clock cycle (i.e., fetch, decode, execute, memory, and write back). Modern x86 architectures are able to decode x86 instructions into smaller operations called micro-operations (μops). The processors then use out-of-order execution to reorder those μops. This approach combined with a superscalar pipeline enables modern processors to extract parallelism out of the code stream for improved performance. A great deal of responsibility is left to the instruction decoders. It is the decoder's job to make the execution as efficient as possible, as well as all the instructions and μops analyzed by branch predictors. If branches are detected, the processors will use speculative execution to try to prevent miss predicted branches stalling the pipeline. In the same way, a great deal of effort is also put into prefetching data from memory into the different caches, so that as much data as possible is ready in the caches for execution.

A technique called symmetric multiprocessing (SMP) is used to implement multiple processor cores in a x86 system. Here, two or more identical processors, all of which have full access to the I/O devices, connect to a single shared main memory and are all controlled by a single operating system. Even though the architectures of x86 processors have evolved over the years, the principles behind SMP are still the same, leading to several challenges when trying to scale the numbers of processing cores in an SMP system. One of the challenges is memory. In early SMP systems, all cores shared the same memory controller, but with integration of the memory controller onto the die of processors, the access times to the different parts of memory are not the same. Another challenge is cache coherency. Since all processor cores have the same access to the main memory, all the caches on the processors must be kept up-to-date. If one core changes data, this change must be broadcasted to all the other cores that work with the same data. The first x86 implementations with multiple cores had one processor core per socket and multiple sockets on a motherboard. The first attempts at processing more than one thread simultaneously on a single die used simultaneous multithreading (SMT) on Intel's Pentium 4 processors. Because of the very long pipeline on Pentium 4 processors, several parts of the pipeline were often idle. To use more resources, Intel implemented Hyper-Threading Technology [60], it's version of hardware multithreading. Intel's first true multicore processor, with two separate independent cores on a single die, was the Pentium D, introduced in 2005. Today, most commodity desktop machines and laptops have two or four processor cores on a single die. In the server and workstation space, up to 18 cores are fitted onto a single die.

During work on this thesis, several generations of processor architectures were released by Intel, a list of which can be found in Table 2.1. The processor roadmap used by Intel

Architecture	Codename	Fabrication process	Released
Core	Merom	65 nm	2006
	Perryn	45 nm	2007
Nehalem	Nehalem	45 nm	2008
	Westmere	32 nm	2010
Sandy Bridge	Sandy Bridge	32 nm	2011
	Ivy Bridge	22 nm	2012
Haswell	Haswell	22 nm	2013
	Broadwell	14 nm	2014

Table 2.1: Roadmap of Intel processors used in our experiments.

today was introduced in 2007 and is called the Tick Tock CPU roadmap [58]. The idea of this roadmap is to follow every new architecture, referred to as a "tock," with a shrink in fabrication processes referred to as a "tick." One of the advantages with this strategy is the reduction of risk when moving to a completely new fabrication process, since the architecture is already known, and vice versa when developing a new processor architecture.

(a) Traditional system bus   (b) Point-to-point links
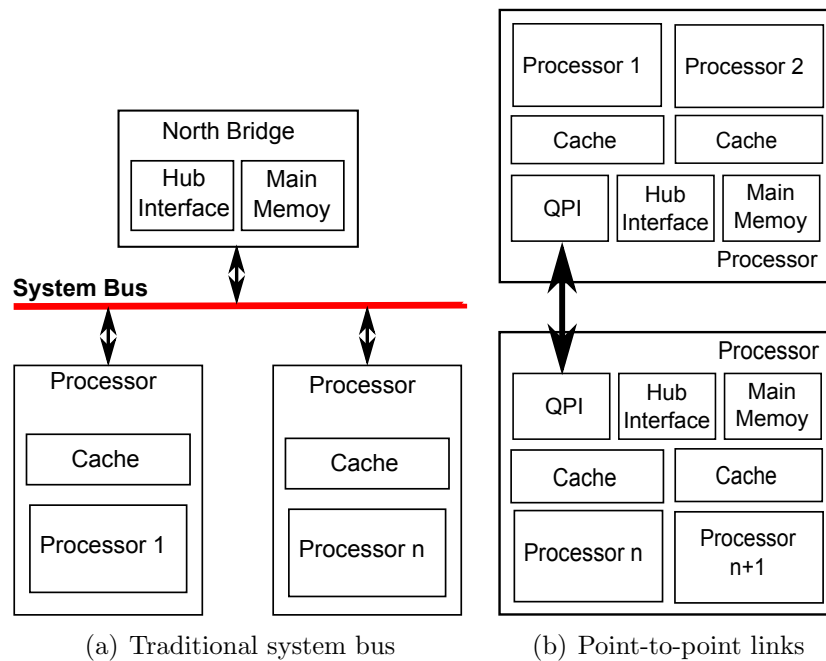
Figure 2.1: Comparison of SMP architectures.

Over the period of this thesis, the architecture connecting multiple processors and external devices has also changed and an overview of the two different architectures can be seen in Figure 2.1. During the 1990s and the 2000s, until the Nehalem architecture, Intel used a simple bus called the front-side bus (FSB) to connect multiple CPUs and to connect these to the so-called north bridge (Figure 2.1(a)). The north bridge is also referred to as the memory controller hub. This is where the memory controller is located and also where you would connect external devices that require fast access to main memory. These

devices are often connected by a PCI, AGP, and, later, PCI Express bus. Such I/O devices as network and hardware are connected on the "south bridge" via the hub interface, also referred to as the DMI. One of the challenges with the FSB was that it is a shared bus and was quickly becoming a bandwidth bottleneck when multiple CPUs were connected. With the Nehalem architecture, the FSB was replaced by several point-to-point interconnects, called QuickPath Interconnect (QPI) [59] (Figure 2.1(b)), and the memory controller was integrated onto the same die as the processor cores were. To better share the memory controller between multiple cores on the die, a level 3 cache was added. Furthermore, the PCI Express bus used to connect external devices was also added to the same die, thus eliminating the need for a north bridge. With the Sandy Bridge architecture, a GPU was also integrated into the same die as the processor cores were. This GPU also shares the memory controller with the processor cores and uses the same level 3 cache to connect to the memory.
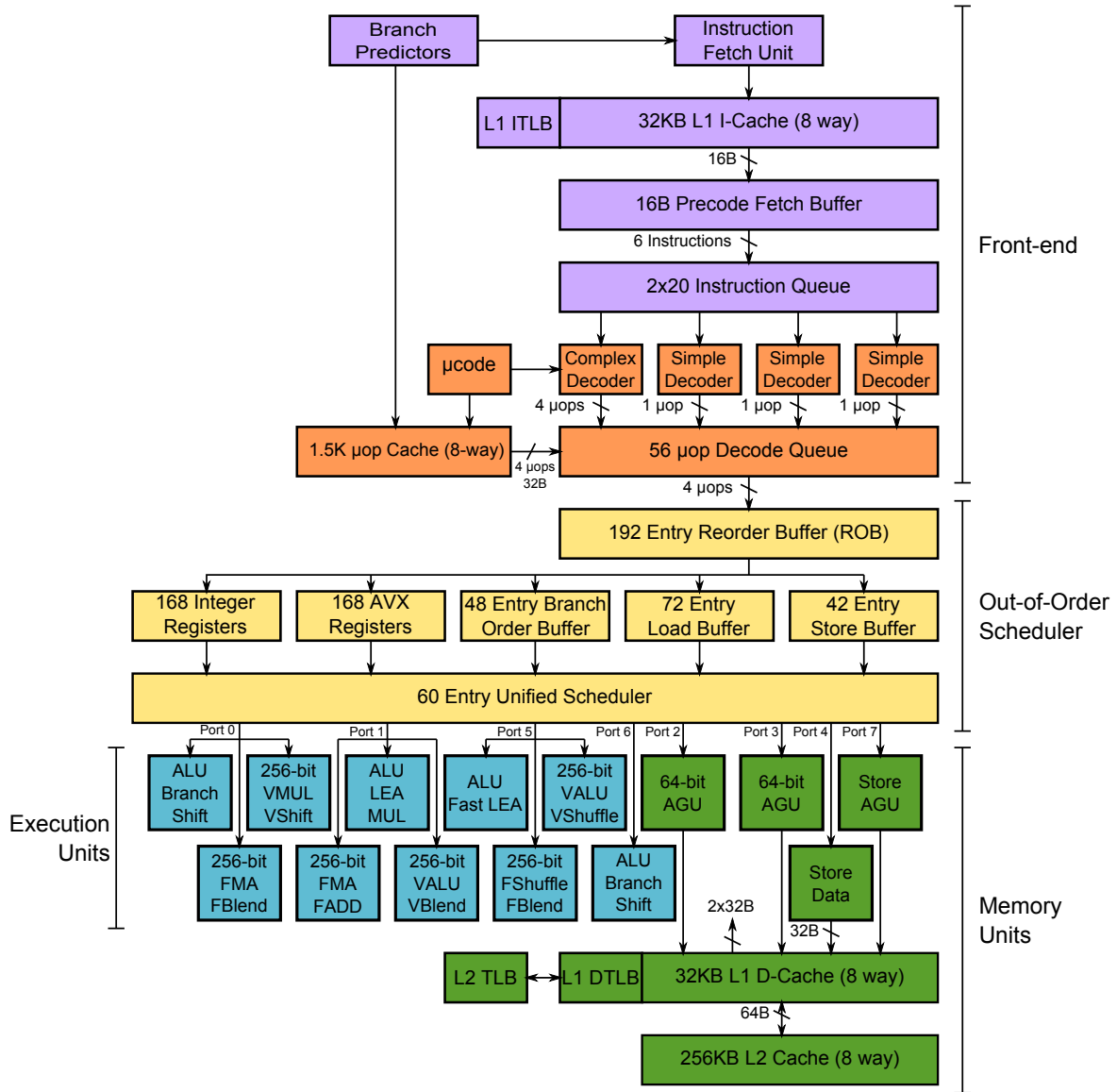
Figure 2.2: Intel Haswell architecture diagram.

The Haswell Architecture

The latest x86 architecture from Intel is called Haswell [104]. It was released in 2013 and, according to the Intel CPU roadmap [58], it was a tock, that is, a new architecture. A detailed overview of a single Haswell core is shown in Figure 2.2.

The first part of the Haswell processor architecture, orange and purple in Figure 2.2, is called the *front end*. This part of the processor is perhaps one of the most complex parts of a modern x86 CPU. The instruction cache is kept coherent with the data cache. Haswell has four decoders to decode the x86 instructions, including three simple decoders that are able to decode one fused μop and one complex decoder that is capable of decoding four fused μops. With μops fusion, the decoders are able to combine x86 instructions that can be executed in parallel (i.e., jump and compare). The front end also includes other important features, such as the branch predictor; however, details of the branch prediction unit are not published by Intel. The next part of the pipeline, in yellow in Figure 2.2, is the *out-of-order scheduler*. Haswell has a 192-instruction out-of-order window. The first part of out-of-order execution is renaming. Here, the renaming allocates resources and maps the source and destination x86 registers onto the underlying physical register files. The idea of out-of-order execution is to optimally reorder the instructions for execution, making sure that as many dependencies as possible are met before the instruction is executed, and, after reordering, to place the μops in a unified scheduler queue ready for execution. Execution is split into two parts: *Execution units*, in blue, and *memory units*, in green in Figure 2.2. In total, the Haswell architecture has eight execution ports, meaning that each cycle the oldest eight non-conflicting μops that are ready for execution are taken out of the unified scheduler and dispatched to the execution ports. Computational μops are sent to port 0, 1, 5, and 6. Three of the four computational execution ports are also capable of executing 256-bit single instruction multiple data (SIMD) instructions. Memory operation μops are sent to ports 2, 3, 4, and 7. The memory hierarchy on the Haswell core is similar to that of earlier Intel processors. For each core, there is a 32-kB level 1 data cache and a 256-kB unified level 2 cache, both private for the core. The level 3 cache, however, is shared with the other cores on the die and with the on-die GPU.

A single Haswell x86 core can therefore have up to two hardware threads that execute up to eight instructions in parallel. This is carried out with techniques such as superscalar pipelines, multithreading, and out-of-order-execution.

Intel Many Integrated Core (Intel MIC) Architecture

Intel MIC is a multicore architecture developed by Intel incorporating work from the now defunct Larrabee GPU architecture [109] and the Single-chip Cloud Computer research project [78]. The first commercial product released in the MIC family had the codename Knights Corner and was later branded Intel Xeon Phi [21].

The Xeon Phi architecture consists of up to 61 simple x86 cores and a basic overview of the architecture is shown in Figure 2.3. The processor architecture used is the P54C [21] architecture, originally used in the Pentium processor from 1993. The cores have been modified with support for the 64-bit x86 instruction set and support for four-way SMT. The process still keeps its original in-order execution pipeline design and the coherent level 2 cache has been extended to 512 kB per core. The main change in the architecture from the original P54C cores lies in the floating-point pipeline. The P54C has a simple
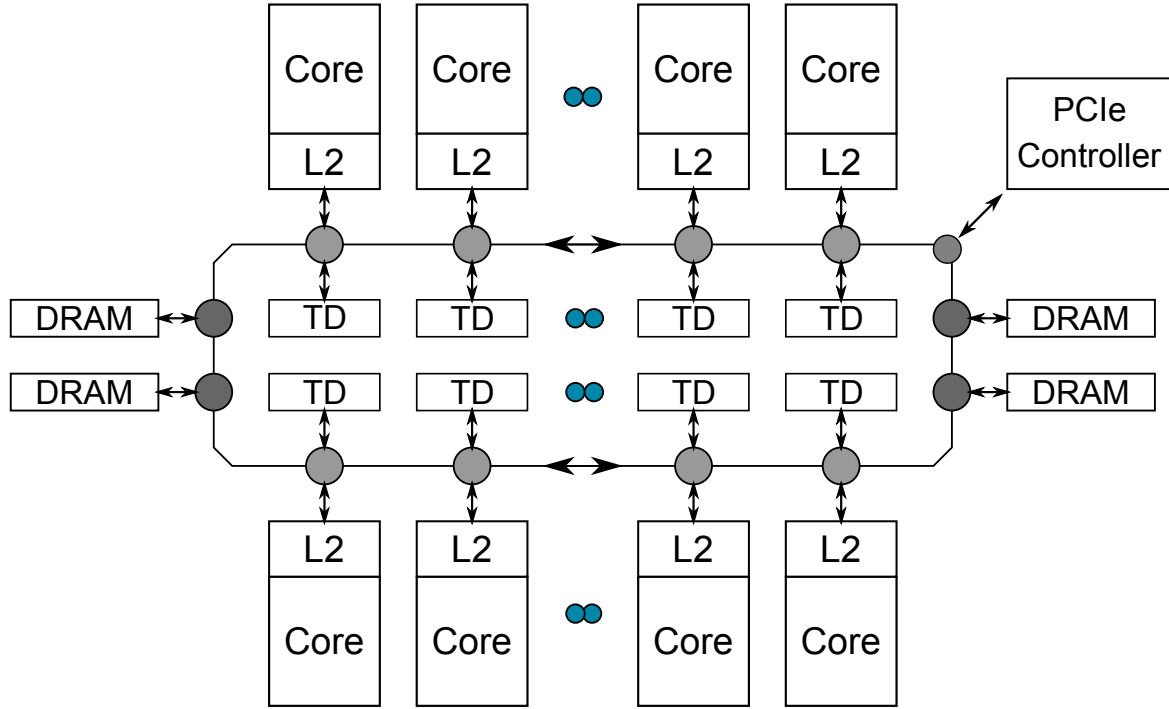
Figure 2.3: Intel Xeon Phi MIC architecture.

x87 floating point unit [21] and the Xeon Phi has a 512-bit SIMD unit in addition to the x87 pipeline. This SIMD unit is able to process 16 independent 32-bit values and eight independent 64-bit values. Additionally, fused multiply and add operations are supported. However, the unit is not compatible with existing MMX/SSE code; so, even though the Xeon Phi cards use x86 cores, they are not backward compatible with the x86 instruction set. All the cores are connected by a 512-bit ring bus and the card has up to 16 GB of on-board GDDR5 memory. The Xeon Phi cards are connected to the host computer through a PCI Express 2.0 interface.

To control the processor, a customized version of Linux is booted on the card when initialized from the host computer. The Xeon Phi supports the offloading of parts of programs from the host processor and, since the card has its own operating system, it can also work as a stand-alone system.

2.1.2   Intel IXP Network Processor

The Intel Internet eXchange Processor (IXP) Architecture [57] is an NPU. An NPU is a specialized processor designed and optimized for efficient packet handling and through-put. A typical NPU features several small processing elements optimized for pipelining and executing data plane tasks and a general-purpose processing core to execute control plane workloads, thus making this a heterogeneous multicore architecture. Intel's first generation of NPUs was called the IXP1200 [56]. This processor features a 32-bit StrongARM general-purpose core and six special-purpose cores called micro engines (μEngines). The NPU used in our research is a second-generation processor from Intel called the IXP2400 [57]. The IXP2400 cards were chosen because it was one of the easiest architectures to program. Intel continued developing their NPUs with a third generation

of processors called IXP28XX and, in 2007, sold the entire product line to Netronome.
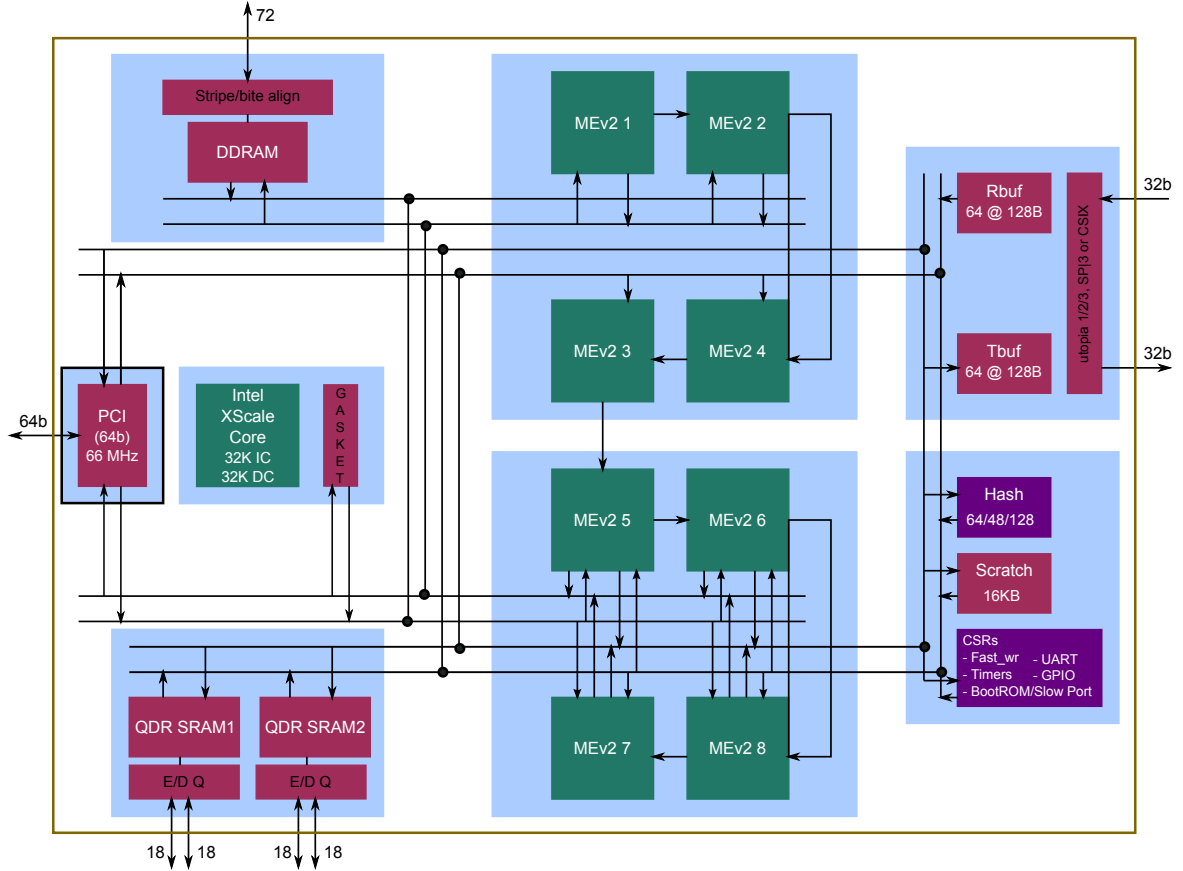
Figure 2.4: Intel IXP2400 architecture diagram.

The Intel IXP2400 architecture overview is shown in Figure 2.4. The basic elements on the chip include a 600 MHz XScale processor. The XScale is a general-purpose processor using the ARMv5 instruction set and the general-purpose processor also runs a Linux or VxWorks operating system to control the card. Additionally, the IXP2400 has eight specialized packet μEngines also running at 600 MHz. The μEngine uses a proprietary instruction set that is not compatible with the ARMv5 instruction set on the XScale. Each μEngine is capable of running four threads in hardware and the μEngines are optimized for general packet processing in the data plane (fast path). The XScale is used for the control plane (slow path). The μEngines are grouped in clusters of four cores and, within each cluster, the cores can communicate with neighbor cores via specialized registers. In normal configurations, two μEngines are reserved for low-level network receive and transmit functions using open-source software, leaving only six μEngines available for application usage.

Moreover, the IXP2400 has three kinds of memory, with different bandwidths and access times. The 256 MB of SDRAM is used for the operating system and packet store, the 8 MB of SRAM is used for metadata (e.g., packet headers), and the 16 kB of on-chip scratchpad is used for interprocess communication and synchronization between the cores. The IXP2400 is connected to the host computer and supports direct memory access (DMA) transfers with a 64-bit PCI connector and the card has three physical mini-GBIC

connector for gigabit Ethernet.

The software development kit for the IXP2400 cards includes a specialized compiler to program the μEngines. The compiler is a C compiler for the MicroC language. The IXP2400 allows data to be processed at wire speeds with very low latency. The cards can process packets with a very limited protocol stack. This allows the programmer to both update and extract information with very low processing overhead. This makes these NPUs ideal for applications such as deep packet inspection and statistics collection.

2.1.3   Nvidia Graphics Processing Units

A GPU is a specialized and dedicated hardware originally designed to render graphics on a screen. A GPU can be integrated as part of the computer's chipset, as a discrete expansion card, or integrated on the die of the main processor. Originally, the GPU was designed to render three-dimensional (3D) scenes onto a two-dimensional frame of pixels. The first generations of GPUs had a fixed rendering pipeline with very limited flexibility and programmability. Compared to a normal general-purpose processor such as the x86 architecture from Intel, GPUs have a very different architecture. On the CPU, as seen in Figure 2.5, much of the die space is used for control logics, such as out-of-order execution, cache, and branch prediction [69]. A GPU has much less control logic and more ALUs. The GPU is designed to perform the same calculations over a large number of values, which is very similar to vector processors; for example, when rendering a 1080p (Full HD) frame, about 2 million independent pixels are processed in parallel.

(a) CPU   (b) GPU
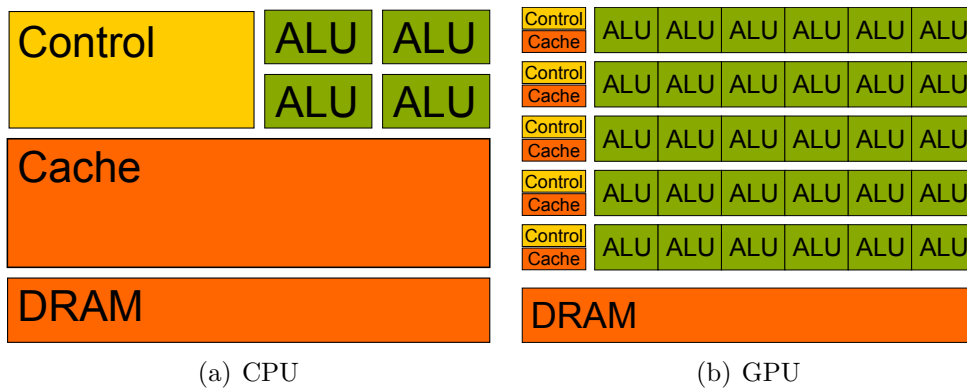
Figure 2.5: Comparison of transistor usage on a CPU and on a GPU.

The term GPU was defined in 1999, when Nvidia launched their GeForce 256 graphics adapter [133]. This was not the first 3D graphics card, but it was the first card to support hardware transform and lightning, meaning that the entire graphics pipeline now ran on the graphics adapter.

The progression of GPUs was mainly driven by the gaming industry in the beginning and the development of graphics cards was closely tied to the development of Microsoft's DirectX application programming interface (API). A new version of the API brought new generations of GPUs. In DirectX 8, programmable vertex shaders were added, which allowed the programmer to control how each vertex in the 3D scene was converted to discrete pixels in the output. DirectX 9 also added support for programmable fragment
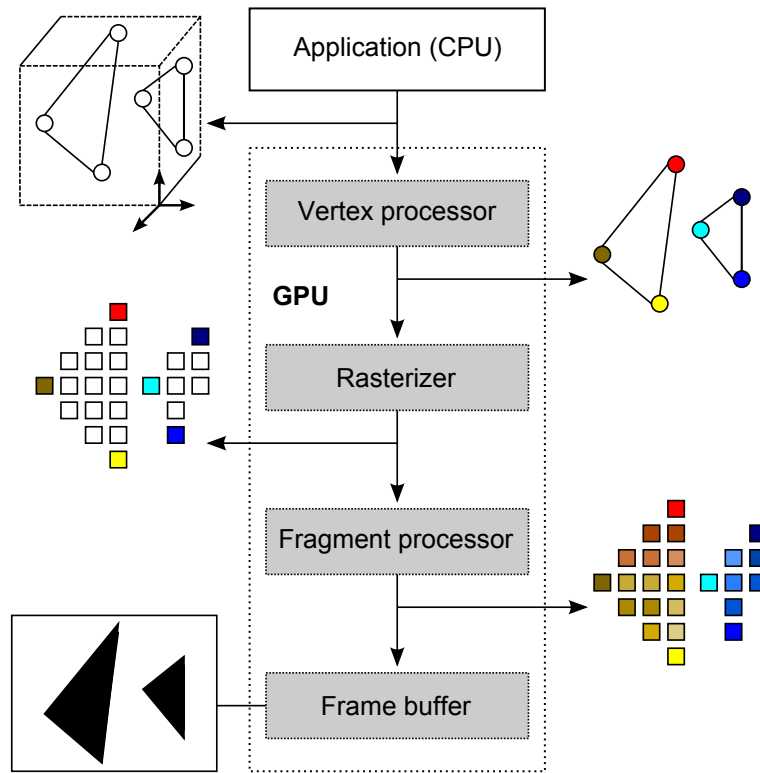
Figure 2.6: A pre-DirectX 10 graphics pipeline [43], with a programmable vertex processor and fragment processor.

shaders. An example of a DirectX 9 pipeline is shown in Figure 2.6. The fragment shaders gave the programmers direct control over pixel lightning, highlighting, translucency, and shadows. It was also possible to use these fragment and vertex shaders for general-purpose computations, given that one's problem could be mapped as a frame. To do so, data had to be stored as textures on the GPU and shader programs were executed on the data; the results had to be stored either in texture memory or in the frame buffer. With DirectX 10, the traditional 3D pipeline was replaced with a unified processing architecture [12]. Gaming was still the main driver for the GPUs, but changes in the GPU architecture made it much more suitable for general-purpose stream processing. It was also with the first DirectX 10 GPUs in 2007 that Nvidia launched CUDA, which made it possible to program GPUs with extensions to the C language [84]. Since the initial release of CUDA, both the underlying hardware and software stack have undergone both minor and major revisions, referred to as compute capability. During work on this thesis, we used three different generations of GPUs from Nvidia, an overview of which is shown in Table 2.2. During the last phase writing of this thesis, Nvidia also released a new-generation GPU called Maxwell; however, we take a more detailed look into how the third generation of CUDA-capable GPUs, codename Kepler, from Nvidia are designed and how their memory architecture works compared to that of a normal CPU.

Architecture	Codename	Compute	Released
Tesla	G80	1.0	2006
	GT200	1.3	2008
Fermi	GF100	2.0	2010
	GF110	2.1	2010
Kepler	GK104	3.0	2012
	GK110	3.5	2013
Maxwell	GM107	5.0	2014

Table 2.2: Roadmap of Nvidia GPUs used in our experiments.

The Kepler GK110 Architecture

The Kepler architecture consists of a number of processing cores clustered together in what is called a streaming multiprocessor (SMX), a shared level 2 cache, memory controllers, and a PCI Express interface to the host machine. The GPU used in the latest revision of the Kepler architecture is called GK110 [90] and this high-end chip has up to 15 active SMX clusters.

The SMX shown in Figure 2.7 on the GK110 GPU contains 256 separate cores. Of these cores, 192 cores have single-precision integer and floating point ALUs and 64 support double precision. The SMX also includes registers, instruction cache, and 64 kB of shared memory and level 1 cache, shared by all the cores on a single SMX that can be partitioned by the programmer. Each SMX also contains 32 load and store units to provide groups of threads access to DRAM in parallel. Finally, the SMX has 16 special function units (SFUs), used to calculate complex mathematical instructions such as cosines and square roots.

Scheduling on the GPU is done at two levels: High-level scheduling on the entire chip is handled by what Nvidia calls a GigaThread engine. In the documentation, it is also referred to as the Grid Management Unit [90]. This unit controls all the groups of threads executing on the GPU and can manage both CPU- and GPU-generated workloads. Low-level scheduling at the SMX level is carried out by a quad warp scheduler. An SMX schedules threads in groups of 32 parallel threads called warps. The scheduler is capable of selecting four warps in parallel and each warp can dispatch two independent instructions per cycle. All the threads in a warp have to execute the same instruction and branching is not supported. If branching should occur in the code, each of the branches must be evaluated for all the running threads.

The memory hierarchy on GPUs is different from that on CPUs. There are different types of memory with different properties and the memory is explicitly managed by the programmer; thus memory usage will often have an impact on performance. Figure 2.8 shows an overview of the memory hierarchy on the Kepler architecture. The first level in the hierarchy is at the thread level. All the cores on the SMX share a total of 65,536 32-bit registers. *Registers* are the fastest memory type on the GPU, with access times of one clock cycle. The challenge with the registers is that their number is limited and if the threads use up all the registers, the overflow data will be stored in what is called local memory. *Local memory* is private to each thread and resides in the DRAM, which is described later. The second level in the memory hierarchy involves shared memory, level 1 cache, and a
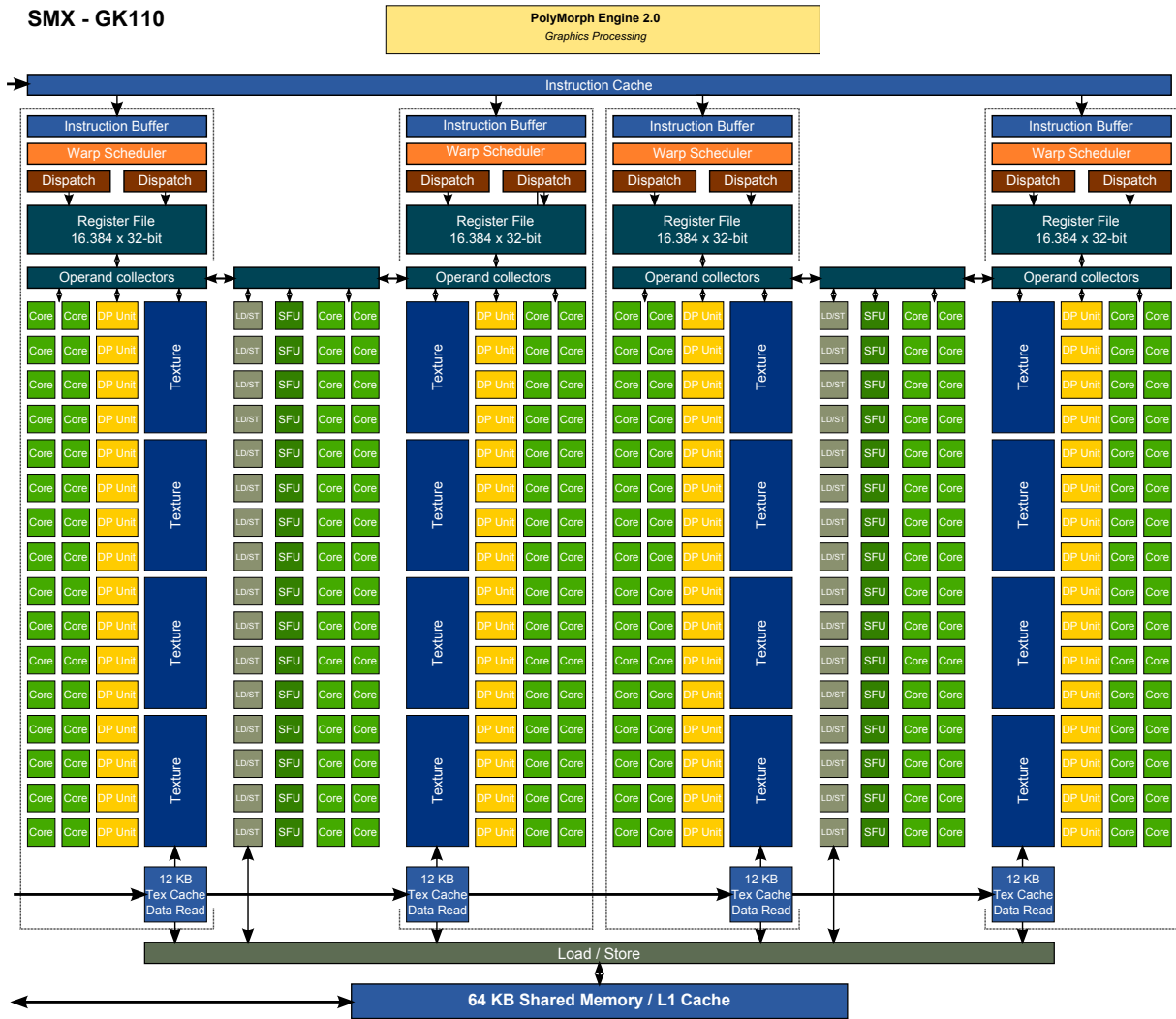
Figure 2.7: Nvidia GK110 SMX architecture [90], slightly modified.

read-only data cache. This level is also on chip within each SMX. The on-chip memory in the SMX can be dynamically partitioned by the programmer between *level 1 cache* and *shared memory*. The access time is also a single clock cycle and the memory is accessible by all threads running on the same SMX. Shared memory provides the programmer fast memory for sharing data and reducing the need for slow off-chip memory access. However, the shared memory is uniformly divided into banks and the throughput is dependent on the data layout. The Kepler architecture also introduced a 48-kB *read-only data cache*, which in previous GPU generations were only accessible by the texture units. This data cache does not have the same bank structure as the shared memory and can support full-speed unaligned memory access patterns. At the third level, the Kepler architecture has 1536 kB of *level 2 cache*. This cache is shared among all SMX units on the GPU and services all load, store, and texture requests to DRAM, enabling more efficient data sharing across the GPU. Programmers are not allowed to explicitly control the level 1 or level 2 cache. The last level in the memory hierarchy is the DRAM. The DRAM is built up of three different memory spaces: global memory, texture memory, and constant memory. With Kepler, the DRAM is off chip and is significantly larger and much slower
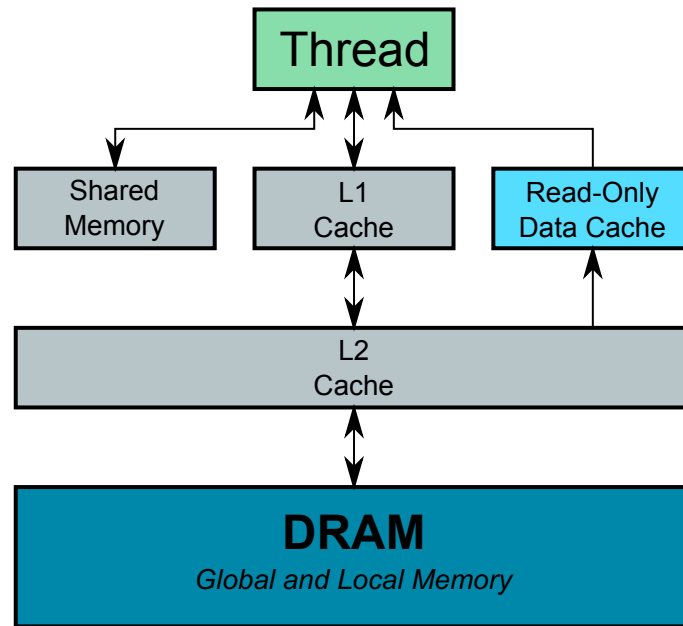
Figure 2.8: The Kepler memory hierarchy.

compared to the higher levels in the hierarchy. The DRAM is often referred to as *global memory*. Global memory can be accessed by all SMX units and can be accessed with 32 , 64-, or 128-byte transactions, given that the access address is aligned with the transaction size. The memory controller on the GPU tries to combine multiple operations into fewer and larger transactions, which is known as memory coalescing. In early GPU generations, memory coalescing was very important; however, with more advanced memory controllers that are not only optimized for graphical workloads, the demands on memory operations have been relaxed.

The two other memory spaces in DRAM are constant and texture memory. Both are read only by the GPU, meaning that they have to be allocated and written to by the host CPU. Both these memory spaces are cached and, depending on the access patterns, this can reduce access time to the data. *Constant memory* is limited to 64 kB in total and *texture memory* is limited to 48 kB per SMX unit. If the data accessed in both constant and texture memory are not in the cache, the GPU must fetch the data from DRAM with the same access time as regular global memory.

The GK110 chip is used in two different product lines: as a dedicated compute coprocessor for general-purpose GPU workloads (Nvidia Tesla) and as a graphics card (Nvidia GeForce and Quadro) for playing computer games and running other 3D applications. The chip itself is comprised of 7.1 billion transistors and has up to 2880 cores and up to 12 GB of on-board DRAM. With a typical power consumption of 250 watts, it can deliver up to 5121 tera floating point operations per second (TFLOPS) of single-precision processing power and 1707 TFLOPS of double-precision processing power.

2.1.4 STI Cell Broadband Engine

The CBE [54] is a heterogeneous multicore processor developed in cooperation by Sony, Toshiba, and IBM. The project was started by Sony in 2000 when they requested a CPU

for the new PlayStation 3 game console. The design goal of the CBE was a processor 1000 times faster than the Emotion Engine from the PlayStation 2 game console [7]. Development of the CBE started in 2001 and the first product to utilize the processor was Sony's PlayStation 3 game console. The CBE was later also used in high-performance computing, when IBM launched several blade servers featuring multiple Cell processors. The chip was also used by Toshiba in laptops and HDTVs as an accelerator for offloading media processing, such as in decoding, upscaling, and post-processing.
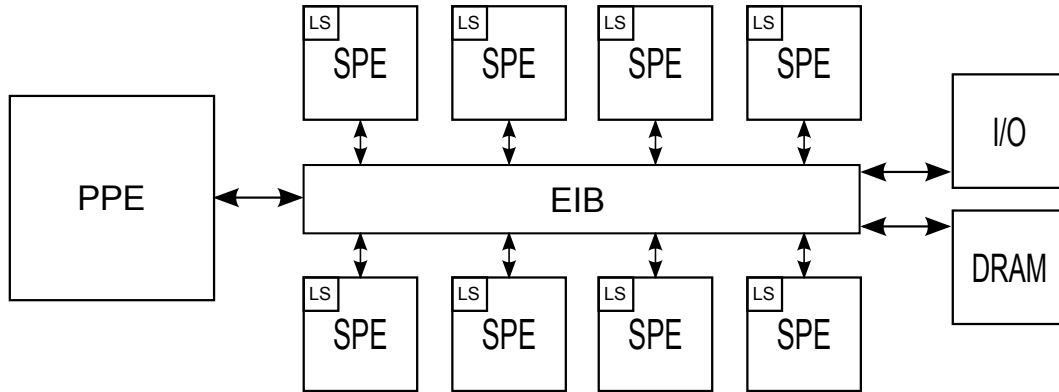
Figure 2.9: CBE architecture.

The CBE consists of several main components: a power processing element (PPE) for general-purpose processing, up to eight synergistic processing elements (SPEs) for high-throughput computations, a FlexIO interface for connecting multiple processors and devices such as network and disk controllers, and, finally, a memory controller. All the components on the chip are connected by the element interconnect bus. An overview of the CBE architecture is shown in Figure 2.9.

The general-purpose processor on the CBE, called the PPE, is based on a standard IBM PowerPC 970 [54]. This processor is from the POWER4 family and implements both the 32- and 64-bit PowerPC instruction set and is capable of executing two threads in parallel. The PPE has two levels of cache (32-kB level 1 data cache, 32-kB level 1 instruction cache, and 512-kB level 2 cache), has a simple branch prediction unit, supports virtual memory, and has a vector unit called the VMX. The vector unit on the PPE is a standard IBM AltiVec SIMD unit and it is capable of processing a 128-bit vector with either four independent 32-bit words, eight 16-bit shorts, or 16 eight-bit bytes. The VMX supports both floating point and integer values.

The specialized computational cores in the CBE are called SPEs and an overview of an SPE is shown in Figure 2.10. A single SPE contains a synergistic processor unit (SPU). The SPU has a large 128-entry 128-bit register for vector processing. The SPU is able to execute two hardware threads in parallel; even though its vector unit has much in common with the VMX unit in the PPE, they do not share the same AltiVec instruction set. The SPE also only supports a 32-bit instruction set. The SPU supports both single- and double-precision floating point values. However they are not fully compatible with the IEEE 754 standard for double precision [54]. Support for "not a number" and infinity have been removed to extend the range and numbers are truncated downward, toward zero. The memory flow controller in the SPE cannot directly access data in the main system memory; it can only access data in a small, 256-kB local storage. The local storage is
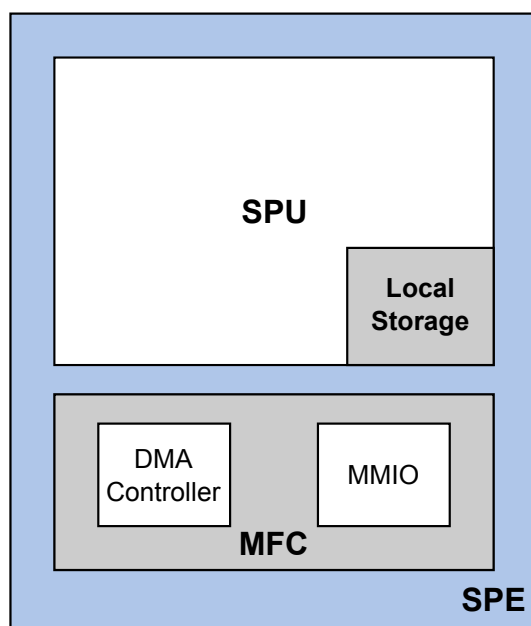
Figure 2.10: Overview of an SPE.

non-coherent and is basically a user-controlled cache, which stores both the code and data that the SPU will process. To copy data between other SPEs, the PPE, or the main memory, the SPEs must use DMA transfers. These transfers can be requested both by the PPE and the SPEs and must be set up explicitly when programming the CBE. The architecture includes hardware support for primitives such as signals, message passing, and atomic updates. There are message queues for communication between the SPEs, PPE, or internally among the SPEs and the queues can be used by either an interrupt handler or polling.

The CBEs used in our research are from the Sony PlayStation 3 game console. The CBE in a PlayStation 3 is clocked at 3.2GHz. One of the SPEs is disabled to improve the yield in the manufacturing process and a second SPE is reserved for a hypervisor, which leaves six SPEs for the user. The system has 256 MB of main memory, where about 40 MB of memory is reserved for the hypervisor. The hypervisor also blocks access to certain parts of the hardware, including the GPU and certain hardware debugging features in the CBE. Early versions of the console also supported booting Linux. In Linux, the CBE can be programmed using the standard GNU Compiler Collection to generate code for both the PPE and SPE.

2.1.5 Other Hardware Architectures

There are also several other interesting hardware architectures from a heterogeneous standpoint. In this section, we mention a couple of other architectures that we have not been able to investigate during work on this thesis and we take a brief look at their main features.

Field-Programmable Gate Array (FPGA)

A FPGA is an integrated circuit that is designed to be configured by users. To configure an FPGA, hardware description language is used. The process is similar to that used when making dedicated application-specific integrated circuits for specific applications. The FPGA contains programmable logic blocks and a reconfigurable interconnect to connect the blocks. After a schematic hardware description language design is completed, an electronic design tool must be used to generate a netlist, which describes how the logic blocks are supposed to behave and how they are connected. Finally, the design must be validated with respect to the placement and timing of the chip.

From a technical standpoint, an FPGA can be used to solve any problem that is computable. One of the challenges with FPGAs is that the programmer has to break down the problems to the logical gate level. Since all the logic gates on an FPGA work in parallel, programmers must also take into account synchronization issues. For complex workloads, another challenge is that only a limited number of logic blocks are available on an FPGA and the blocks cannot be efficiently reconfigured during execution. [73]

Modern FPGAs such as the Xilinx Zynq [135] series have started to combine traditional design with logic blocks and interconnects with an embedded ARM microprocessor. This design is often referred to as a system on a programmable chip. It makes it easier to reconfigure the chip during runtime and programmers can use the programmable part of the chip for parallel parts of their applications. We have not looked into FPGAs. The programming model is very different from that of traditional multicore systems and we do not have any experience with hardware design.

Very Long Instruction Word (VLIW) Architectures

A VLIW architecture is a type of processor architecture that is designed to take advantage of the instruction-level parallelism available in program code. The idea behind VLIW is to enable programs and compilers to explicitly specify which instructions can execute in parallel. The idea is not new, as described in Section 2.1.1. For example, x86 processors use their superscalar architecture to execute independent instructions in different parts of the processor and out-of-order execution reorders instructions to improve efficiency. The drawbacks with both these approaches is that they make the hardware more complex, resulting in larger circuits and higher power consumption. With VLIW, the processor executes operations in parallel based on a fixed schedule generated when the program is compiled. For even better efficiency, hints can be given to the compiler.

The VLIW concept was first invented in the early 1980s [36]. The first implementation of VLIW was Intel's first 64-bit processor, called the i860, released in 1989 [71]. The only VLIW processor architecture produced today is the Itanium architecture from Intel and it is used only in enterprise-class server systems. We have therefore not investigated this architecture further. Graphics processors from AMD also used VLIW [51]; however, these have since been replaced by a RISC SIMD architecture called Graphics Core Next.

NEC SX Vector Supercomputer Architecture

The SX series architecture from NEC involves a dedicated vector processor. The latest iterations of the processor are the SX-ACE and SX-9 processors [142]. Each core contains

four vector processing units. Each of the vector processing units has a 16-stage pipeline and 16 vector registers that can store 256 64-bit words. Many of the details about how the cores operate are not public. The theoretical performance of a chip with four cores is 256 GFLOPS and multiple chips can be connected in multinode systems with SMP.

The architecture is highly optimized toward vector processing and, as a result, scalar programs do not scale well on this architecture. The systems are shipped with SUPER-UX, with is a custom UNIX operating system maintained by NEC. One of the main applications for the SX architecture is to run simulations of complex climate models for meteorological use [107]. The SX architecture is not considered in this thesis, since it is only used in high-end enterprise-class servers.

2.1.6   Summary

In this section, we introduced the four heterogeneous architectures that we are going to use for our experiments in this thesis. The architectures are very different but they share some properties. Table 2.3 compares the state-of-the-market products in each of the heterogeneous architectures.

Feature	IXP	x86	Cell	GPU
General-purpose cores	1	1–16	1	0 [†]
Vector instruction set	No	256-bit AVX2	128-bit AltiVec	No [♯]
Specialized cores	8	0	8	192–2880
Instruction set	ARMv5/ μC	x86	PowerPC	PTX
SMT (multithreading)	No	Yes	Yes	No
Memory model	Shared	Shared	Exclusive	Exclusive
Cache coherency	Yes	Yes	No [‡]	No
On-chip memory	Yes	No	Yes	Yes
Off-chip memory	Yes	Yes	Yes	Yes
Memory types	SRAM/ DRAM	DRAM	DRAM	DRAM
Branch prediction	Limited	Yes	Limited	Limited
Cache hierarchy	L1	L1–L2–L3	L1–L2	L1–L2
User-controlled cache	No	No	Yes	Yes [♭]
Active development	Yes	Yes	No	Yes

[†] The GPU needs a host CPU to operate, but there is no CPU on the card.

[♯] Not exposed outside the driver.

[‡] Coherence between multiple Cell processors, but not with SPEs.

[♭] Shared memory is a cache; architecture also has caches not controlled by the programmer.

Table 2.3: Comparison of the four heterogeneous architectures.

The very different properties of the architectures makes their utilization challenging for programmers. Architectures such as the x86 use a shared memory model, where the memory management unit on the processor takes care of all the data transport between the cores and caches. However, on architectures such as the Cell and GPUs, which have

an exclusive memory model, programmers need to carefully consider the data flow of the program. Even GPUs and the IXP have multiple memory spaces available that are more suited for some operations. The properties of these memory types on the GPU are investigated further in Section 3.3.1. The number of cores in these architectures is also very different. On x86 processors and on the Cell, programmers typically need tens of threads for optimal performance, whereas on a GPU programmer need thousands and perhaps tens of thousands of threads to obtain optimal performance. Finally, all the architectures have different instruction sets, so there will be very little portability of code between the architectures. This is further explored in Chapter 3.

2.2   Hardware Abstractions and Programming Models

In Section 2.1, we described several modern heterogeneous processing architectures used in our research on processing multimedia workloads. If we look at the processing cores, memory, caches, and buses that connect them, these architectures are very different. However, the hardware abstractions and programming model used to expose the hardware to programmers do not change that often.

One example where a hardware abstraction is used to expose the hardware differently to the programmer and to the operating system is SMT. SMT is used both in x86 cores and on the PPE in the CBE. The basic idea behind SMT is that a single processing core is exposed as two or more separate cores to programmers and to the operating system. Another example in which hardware abstractions are hiding the underlying hardware is the GPUs from Nvidia described in Section 2.1.3. The last-generation Kepler GK110 GPU uses the programming model called single instruction, multiple threads (SIMT), which was already used by the first programmable Tesla G80 GPU released in 2006. The underlying hardware between these two GPUs has, however, changed considerably.

2.2.1   SMT

SMT is a technique used to improve the throughput of a superscalar CPU pipeline. When a single thread is running and stalls due to a cache miss or any other high-latency instruction, it leaves parts of the processor idle. With SMT, a single processor core is exposed to the operating system as two or more cores and the hardware tries to efficiently utilize all the resources in the superscalar pipeline, as shown in Figure 2.11.

Intel calls this Hyper-Threading Technology. It was first used in 2002 in the Pentium 4 architecture to expose a single CPU core as two virtual cores. The same technology is also used in the Intel Xeon Phi many-core architecture, whereas a single processor core is exposed as four virtual cores on the Phi. In addition to Intel, IBM is using SMT on the PPE PowerPC processing core in the CBE. SMT does not always improve performance. There are several cases in which it can actually reduce performance for both threads running on the processor core. The basic aim of a general-purpose CPU design is to run a single thread as quickly as possible. The cores are often designed with techniques such as out-of-order execution, superscalar pipelines, branch prediction, and prefetching. SMT is often considered the last resort in filling the pipeline to prevent stalls. When two threads compete for resources, they often take more time to finish than if they did not have to share any resources. On the other hand, in on some architectures, such as
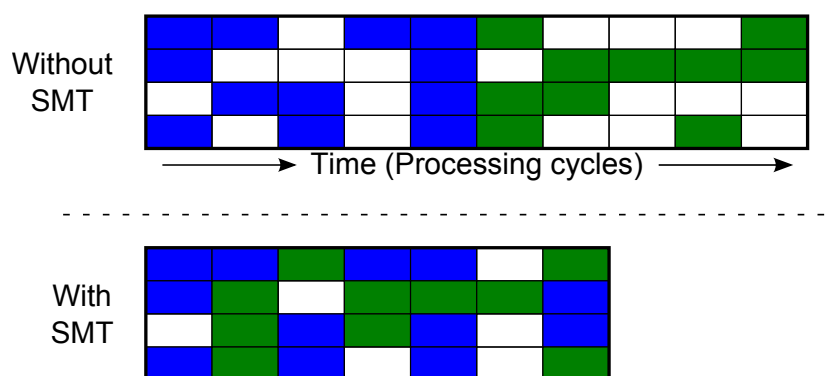
Figure 2.11: A superscalar processor design with and without SMT.

the Xeon Phi many-core architecture, many cores share the memory controller, resulting in greater memory latency. With SMT, the cores are better able to hide some of this memory latency.

2.2.2   SIMD

SIMD, described in Flynn's taxonomy [38] and shown in Figure 2.12, is a technique where multiple processing elements perform the same operation on multiple data points in parallel. SIMD is also often referred to as vector processing, since the multiple data points are often stored in a data vector. The first mainstream use of SIMD was Intel's Pentium processors with MMX extensions, launched in 1996.MMX supports 64-bit long vectors. After Intel released MMX, Motorola and IBM quickly introduced their AltiVec vector extensions for the PowerPC and POWER systems. Since its release, Intel has also improved and extended MMX, first to 128 bits with several versions of Streaming SIMD Extensions and, finally, to 256 bits with two versions of the Advanced Vector Extensions.
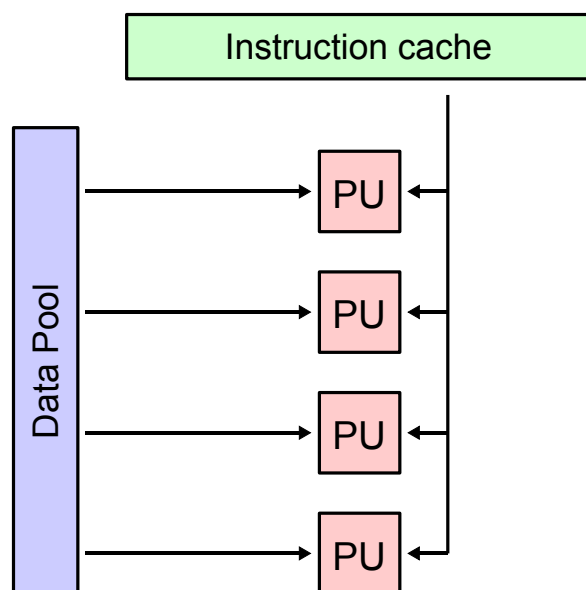
Figure 2.12: SIMD programming model.

Multimedia workloads frequently perform identical operations on large data sets, which is one of the reasons why SIMD was brought to desktop and mobile processors. In the CBE, SIMD is an essential part of the architecture. The PPE has an AltiVec SIMD unit and the SPEs work on a 128-bit vector. Although SIMD instructions became mainstream with the Pentium processor in 1996 and the adoption of the PowerPC for MacOS, their use has been and still is an art. On the Cell, SIMD instructions are used explicitly through vector extensions to C/C++, which allows basic arithmetic operations on vector data types of intrinsic values. This means that the programmer can apply a sequential programming model but needs to adapt the memory layout and algorithms to the use of SIMD vectors and operations. On the x86, programmers also have to explicitly use the SIMD instructions and, even though the compilers are able to auto-vectorize some simple data patterns, these operations generally have to be made manually.

2.2.3 SIMT

The abstraction used when programming GPUs from Nvidia is called SIMT. Nvidia first introduced this model with the CUDA processing framework, released together with the Tesla G80 GPU in 2006. CUDA uses a two-tiered threading model that maps to the architecture. Threads are bundled into groups, which are organized in a grid, as illustrated in Figure 2.13.

The global scheduler on the GPU distributes the groups to available SMXs and all the threads in a group execute on the same SMX. The program that is executed on the GPU
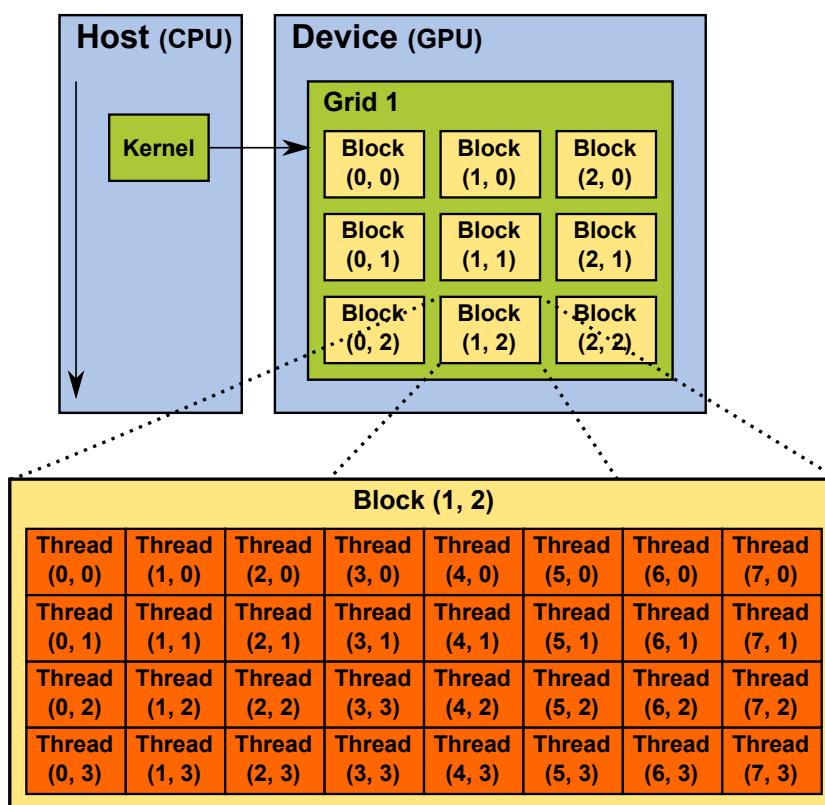
Figure 2.13: Nvidia CUDA programming model.

is called a kernel. It is up to the programmer to choose how the threads are organized. The optimal number of blocks and number of threads per block vary depending on the GPU generation used. The optimal size of these parameters also varies depending on the register space each thread in a block requires. If the register space on an SMX is exhausted, the GPU will use local memory,[1] which is located off chip, resulting in a massive increase in access time. SIMT enables code that uses only well-known intrinsic types that can be massively threaded. Low-level operating system functions schedule these threads in groups called warps (the size of a warp can be hardware specific).The control flow of the threads can diverge as in an arbitrary program. However, this will essentially serialize all the threads in the block, which will impact performance. If none of the threads in the warp diverge, all the threads will execute the same operation. The operation is then performed as a vector operation containing the data of all the threads in the block.

2.2.4   Summary

In this section, we briefly introduced the different programming models used by the architectures in this chapter. SMT is typically used by the architectures with more complex general-purpose cores (i.e., Cell and x86), to try using the execution pipelines more efficiently. For programmers and operating systems, the use of SMT is transparent, which can be challenging, since some applications have shown reduced performance when they have to share resources on the same processor cores.

Both the Cell and x86 architectures use explicit SIMD instructions. This means that the programmer can apply a sequential programming model but needs to adapt (if possible) the algorithms and memory layout to use SIMD vectors and operations. The GPUs from Nvidia use an abstraction called SIMT. Such abstractions enable programmers to write code that uses well-known intrinsic types but which are massively threaded. It is the runtime of the GPU that schedules the threads. In this model, it is possible for the threads to diverge, as in arbitrary programming, even though this will have negative effects on performance. The functionalities provided by SIMD and SIMT are very similar. In SIMD programming, vectors are used explicitly by the programmer, many of whom think in terms of sequential operations on very large operands. In SIMT programming, the programmer can think in terms of threaded operations on intrinsic data types. The SIMT concept has an interesting property: If SIMD is used, the vector width must be known to the programmer. SIMT hides this and the code can be optimized for several vector widths. Even though the functionality is similar, programmers still need to think differently when using these architectures, as demonstrated in our case studies on the different architectures in Chapters 3 and 4.

2.3   Summary

In this chapter, we introduced the heterogeneous hardware architectures and the programming models used to program them. The architectures have very different properties, which makes their utilization challenging for programmers. In the next chapter, we

---

[1]Local memory in OpenCL is the same as shared memory in CUDA.

look at case studies with simple multimedia workloads running on the different heterogeneous architectures presented in this chapter. These case studies are used to learn how to efficiently use the different architectures.

Chapter 3

Using Heterogeneous Architectures for Simple Tasks

Heterogeneous systems have recently received a lot of attention. They provide more computing power than traditional single-core systems, but their efficient use of available resources is a challenge. On some architectures, the processing cores have different strengths and weaknesses compared to desktop processors. Several different types and sizes of memory are exposed to the developer and limited architectural resources require considerations of data and code granularity.

To learn more about the properties of our heterogeneous architectures, we performed different experiments on the architectures with simple tasks related to multimedia to gain experience. By simple tasks we mean small operations, simple steps, or small parts of a larger complex pipeline. In most of the cases, only the part of the workload running on the heterogeneous architecture was optimized for performance, since we wanted to isolate only this part of the workload. The simple tasks in our investigations ranged from experiments on how to use the different memory spaces on an Nvidia graphics processing unit (GPU) most efficiently to protocol translation on the Intel IXP network processor and offloading parts of Motion JPEG (MJPEG) video encoding pipeline to the single instruction multiple data (SIMD) unit on an x86 processor, the synergistic processing element (SPE) unit on the Cell, or the cores of an Nvidia GPU.

This chapter is organized by the heterogeneous architectures. First, we take a look at the Intel IXP network processor. We then experiment with the Sony–Toshiba–IBM Cell Broadband Engine before we run tests on the x86 processor architectures. Finally, we evaluate the performance of different workloads on GPUs. In all these sections, we take a closer look at each architecture, with one or more case studies. We use these case studies to gain experience on how to efficiently use these architectures for parallel processing. However, not all the workloads have been tested on all architectures.

3.1 Intel IXP Network Processor

The Intel IXP network processor was used in the early stages of this thesis as an architecture that could explore the limits of integrated layer processing [24]. To do this, we used a protocol translation prototype. The IXP card provided early insight into how to program an asymmetric shared memory architecture and experience with video streaming.

In Section 3.1.1, we take a closer look at a case study based on our work with network protocol translation [32,33]. We use an Intel IXP2400 network processor to transparently translate RTP/UDP video streams, which was the popular way of streaming video a decade ago, to HTTP/TCP, which is the de facto solution today.

3.1.1   Case Study: Network Protocol Translation

In this section, we describe our implementation of a dynamic transport protocol translator on an Intel IXP2400 network processor. The IXP architecture was used in the early stages of the research work to gain experience with a heterogeneous architecture. Streaming services are available almost everywhere today. Major newspapers and TV stations provide on-demand and live video content. Video on-demand services are common and even personal media are frequently streamed using services such as YouTube.

Setting

The debate about the best protocols for streaming has been going on for years. Initially, streaming services on the Internet used UDP for data transfer because multimedia applications often have demands for bandwidth, reliability, and jitter that could not be offered by TCP. Today, this approach is hampered by the filters of Internet service providers (ISPs) and by firewalls in access networks and in end systems. ISPs reject UDP because it is not fair to accept it over TCP traffic and many firewalls reject UDP because it is connectionless and requires too much processing power and memory to ensure security. It is therefore fairly common to use HTTP streaming, which delivers streaming media over TCP. The disadvantage is that the end user can experience playback hiccups and quality reduction because of the probing behavior of TCP congestion management, leading to oscillating throughput and slow packet rate recovery. A sender who uses UDP would, in contrast, be able to maintain a constant desired sending rate. Servers are also expected to scale more easily when sending smooth UDP streams and avoid dealing with TCP-related processing.

To explore the benefits of both TCP and UDP, we experiment with a proxy that carries out a transparent protocol translation. This is similar to the proxy caching ISPs employ to reduce their bandwidth and we do, in fact, aim at a combined solution. There are, however, too many different sources of adaptive streaming media for end users to apply proxy caching for all of them. Instead, we aim at live protocol translation in a TCP-friendly manner that achieves high perceived quality for end users. Our prototype proxy is implemented on the Intel IXP2400 network processor and enables the server to use UDP on the server side and TCP on the client side.

Preliminary tests comparing HTTP/TCP video streaming from a web server and RTSP/RTP/UDP streaming from a Komssys video server [45] show that, in case of loss, our solution using a UDP server and a proxy later translating to TCP delivers a smoother stream at the playout rate while the TCP stream oscillates heavily.

Workload: Translating Proxy

An overview of our protocol translating proxy is shown in Figure 3.1. The client sends a GET request, which is translated to RTSP by the proxy. The proxy then generates the
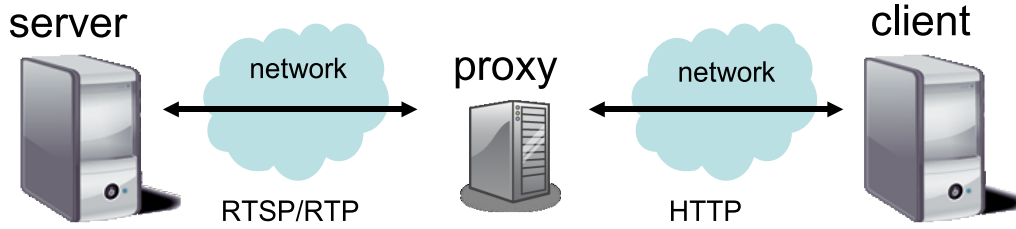
Figure 3.1: Overview of the streaming scenario.

pipeline and starts to push data. The TCP connection between the proxy and client is assumed to be fast enough. If not, the proxy will drop packages before the TCP sequence numbers have arrived. Note that both peers are unaware of each other. The server assumes the client uses UPD and vice versa.

The steps and phases of a streaming session are as follows. The client sets up a HTTP streaming session by initiating a TCP connection to the server; all packets are intercepted by the proxy and modified before being passed on to the streaming server. The proxy also forwards the TCP three-way handshake between the client and server, updating the packet with the server's port. When established, the proxy splits the TCP connection into two separate connections that allow for the individual updating of sequence numbers. The client sends a GET request for a video file. The proxy translates this into a SETUP request and sends it to the streaming server using the TCP port of the client as its proposed RTP/UDP port. If the setup is unsuccessful, the proxy will inform the client and close the connection. Otherwise, the server's response contains the confirmed RTP and RTCP ports assigned to a streaming session. The proxy sends a response with an unknown content length to the client and issues a PLAY command to the server. When received, the server starts streaming the video file, using RTP/UDP. The UDP packets are translated by the proxy as part of the HTTP response, using the source port and address matching the HTTP connection. Because the RTP and UDP headers combined are longer than a standard TCP header, the proxy can avoid the penalty of moving the video data in memory, thus permitting reuse of the same packet by padding the TCP options field with NOPs. When the connection is closed by the client during or after playback, the proxy issues a TEARDOWN request to the server to avoid flooding the network with excess RTP packets.

Implementation

Our prototype is implemented on a programmable network processor using the IXP2400 chipset [57]. The chipset is Intel's second-generation, highly programmable network processor and is designed to handle a wide range of access, edge, and core network applications. A more detailed overview of the architecture is given in Section 2.1.2.

The transport protocol translation operation is shown in Figure 3.2. The protocol translation proxy uses the XScale core and one micro engine (μEngine) application block. In addition, we use two μEngines for the receiving (RX) and the sending (TX) blocks. Incoming packets are classified by the μEngine based on the header. The RTSP and HTTP packets are queued for processing on the XScale core (control path), while the
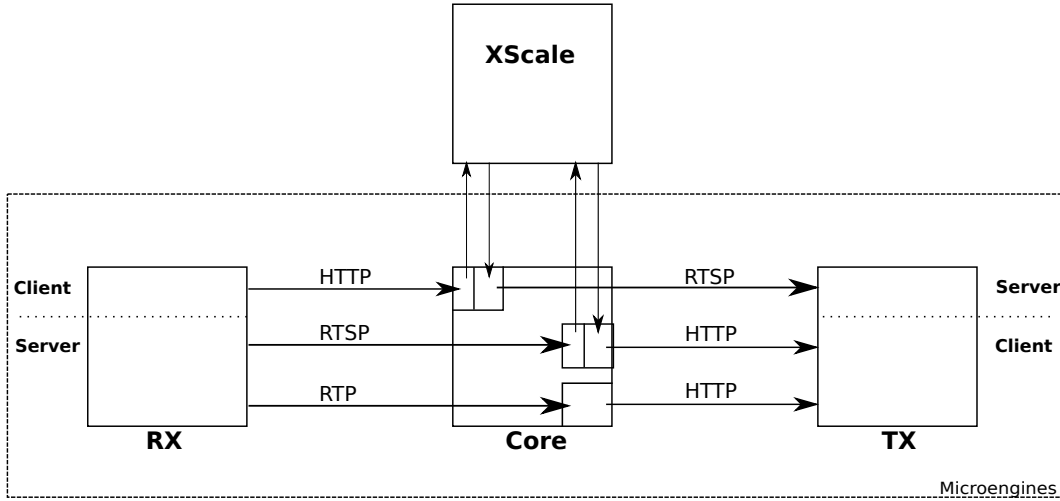
Figure 3.2: Packet flow on the Intel IXP2400.

RTP packets are handled on the μEngine (fast path). TCP acknowledgments with a zero payload size are processed on the μEngine for performance reasons.

The main task of the XScale is to set up and maintain streaming sessions, but after initialization, all video data are processed (translated and forwarded) by the μEngines. The proxy supports partial TCP/IP implementation, covering basic features. This is done, to save both time and resources on the proxy.

Experiments

We investigated the performance of our protocol translation proxy compared to plain HTTP streaming in two different settings. In the experiment shown in Figure 3.3, we induced unreliable network behavior between the streaming server and the proxy, while in the second experiment, the unreliable network connected the proxy and the client. We performed several experiments where we examined both the bandwidth and the delay while changing both the link delays (0–200 ms) and the packet drop rate (0–1%). We used a web server and an RTSP video server using RTP streaming, running on a standard Linux machine. Packets belonging to end-to-end HTTP connections made to port 8080 were forwarded by the proxy, whereas packets belonging to sessions initiated by connections made to port 80 were translated. The bandwidth was measured on the client by monitoring the packet stream with tcpdump [121]. We include only the server–proxy loss experiments in this thesis. For more details about the TCP congestion control implementation and the full evaluation, see paper I [32].

The results from the test where we introduced loss and delay between the server and the proxy are shown in Figure 3.3. The plot shows that our proxy that transparently translates from RTP/UDP to TCP achieves a mostly constant rate for the delivered stream. Sending the HTTP stream from the server, on the other hand, shows large performance drops when the loss rate and the link delay increase.

(a) HTTP streaming   (b) Protocol translation
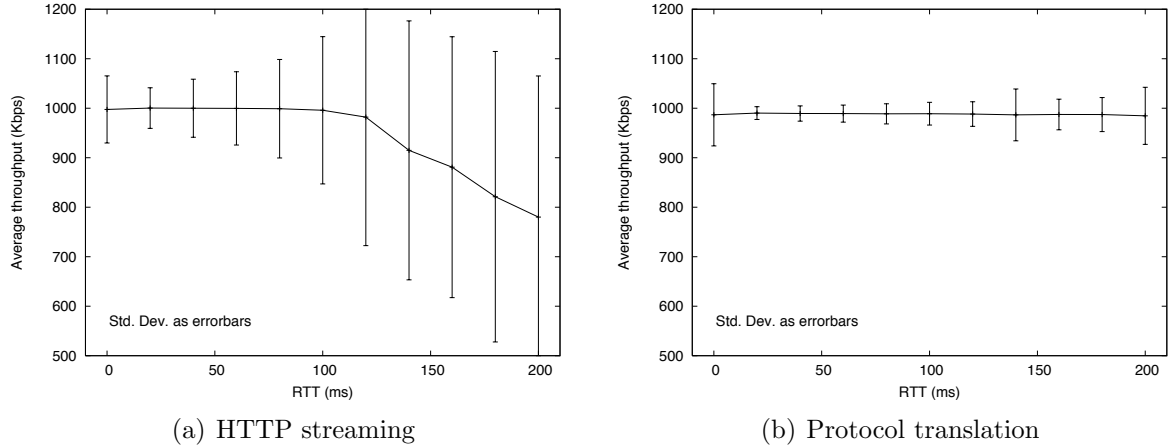
Figure 3.3: Achieved bandwidth, varying drop rate and link latency with 1% server–proxy loss.

Discussion

Even though our proxy seems to provide better, more stable bandwidths, there is a trade-off, because instead of retransmitting lost packets (and thus old data if the client does not buffer), the proxy fills the new packet with new, updated data from the server. This means that the client in our prototype does not receive all the data and artifacts may be displayed. On the other hand, in case of live and interactive streaming scenarios, delays due to retransmission may introduce dropped frames and delayed playout. This can cause video artifacts, depending on the codec used. However, this problem can be easily reduced by adding a limited buffer per stream sufficient for one retransmission on the proxy.

One issue in the context of proxies is where and how the proxy should be implemented. For this study, we chose the IXP2400 platform, since we explored the offloading capabilities of such programmable network processors earlier. With such an architecture, the network processor is suited for many similar operations and the host computer could manage the caching and persistent storage of highly popular data served by the proxy itself. However, the idea itself could also be implemented as a user-level proxy application or integrated into the kernel of an intermediate node performing packet forwarding packets at the cost of limited scalability and potentially greater latency.

Both TCP and UDP have their strengths and weaknesses. In this case study, we used a proxy that carried out transparent protocol translation to utilize the strengths of both protocols in a streaming scenario. It enabled the server to use UDP on the server side and TCP on the client side. The server gained scalability by not having to deal with TCP processing. On the client side, the TCP stream was not discarded and passed through firewalls. The experimental results show that our protocol transparent proxy achieved translation and delivers smoother streaming than HTTP streaming.

Summary

For the context of this thesis, we learned that the IXP is a complicated architecture to work with. Writing the network translation proxy requires detailed knowledge about the platform. Another observation is that, when working with a cutting-edge architecture,

the quality of the documentation and compilers can be a challenge.

3.1.2   Implications

The IXP2400 is an asymmetric multicore architecture and has processing elements with different capabilities. The XScale core is a general-purpose ARM11 core and the μEngines are specialized cores built for packet processing. This architecture is a shared memory architecture, meaning that all the cores (both XScale and μEngines) have access to the same memory. This is very convenient for the programmer when developing applications, since inter-process communication is very simple. However, there are challenges with shared memory architectures and you might end up with unnecessary transfers to prefetch local caches as a result. In addition, cache coherency protocols can consume resources on the interconnects between the processing elements, preventing efficient performance scaling.

The asymmetric nature of the IXP2400 chip can also be a disadvantage. The specialized cores often require special compilers, application programming interfaces (APIs), and tools to write applications. This was an issue with the IXP2400 chip when we conducted experiments on the platform, where lack of documentation and subpar, buggy compilers can create problems for a programmer.

Writing an application for the IXP2400 also requires the programmer's detailed knowledge about the architecture. One of the main advantages with this architecture is the ability to analyze and manipulate network packets at full line speed (1 Gbps) while adding as little latency as possible to the stream. To achieve this, programmers need to know the different memory types on the board and how to use them. The architecture also has many special features, such as a hardware thread context switch that waits until after memory fetch/store completion. A disadvantage is that one loses almost all portability with an application written for an architecture such as this. When moving to the next generation of hardware, an application might require a complete rewrite.

Revisiting with State-of-the-Market Hardware

When these experiments were conducted in 2006 and 2007, it was not possible to write a protocol translation proxy that could run on a standard Linux desktop PC at the full line speed of 1 Gbps. With recent general-purpose hardware such as the Intel Haswell architecture, it is possible to do this with software executing on the central processing unit (CPU). On the other hand, network processors have also evolved. Netronome, the company that bought the IXP technology from Intel, has continued developing both hardware and software. Their adapters are now capable of processing two 100-Gbps fiber links at line speed [82]. This is not possible on a desktop PC today.

Another possibility would be to use another heterogeneous architecture to process the network traffic. Han et al. [47] have shown that a software router implemented on an Nvidia Fermi GPU can forward data at a rate of 39 Gbps, outperforming a CPU-based software router by a factor of four. A GK110 Kepler GPU, which is the latest GPU generation, would do this even faster. However, a GPU implementation will have some of the same challenges as our IXP2400 implementation. Heterogeneous architectures provide great flexibility and performance, but often at the cost of portability.

3.2   x86 Processor Architecture

The x86 processors in modern computers are a symmetric multicore CPU with a shared memory model. In the symmetric multicore processing (SMP) model, the cores should theoretically be identical. However, many architectural details, such as the underlying hardware architecture, are hidden from programmers. We therefore want to learn more about the architecture.

The x86 architecture can have a different degree of connectivity to the other cores. Cores can be on the same chip, in the same package, in different sockets on the same machine, and in some cases even distributed over multiple machines with interconnects such as Numascale [86]. The system memory can also be segmented in such a way that access to other parts of system memory may have to traverse through other cores via the processor interconnect. In our experiments, x86 processors are also often a part of a heterogeneous architecture, with one or more GPUs connected over PCI Express as a coprocessor.

In this section, we take a closer look at three case studies. The first case study presented in Section 3.2.1 is based on our article "Tips, Tricks and Troubles: Optimizing for Cell and GPU" [112]. Even though that research paper was mainly focused on the Cell and GPUs, we also conducted experiments on the efficiency of discrete cosine transformation (DCT) algorithms and the effect of using SIMD on the x86. The second study presented in Section 3.2.2 is based on our work on multi-rate encoding with the VP8 codec [34, 35]. In these experiments, we use the shared memory model of the x86 architecture to reuse parts of the computational heavy analysis stage of the video encoder. In the final study in Section 3.2.3, we investigate the parallel execution of a game server [102], using a thread pool to execute lightweight game server -related tasks running on a multi-socket x86 SMP system.

3.2.1   Case study: Motion JPEG Encoding

We want to learn how to *think* when the multicore system at our disposal is a Cell, x86, or GPU. We aim to understand how to use the resources efficiently and point out tips, tricks, and problems as a small step toward a programming framework and a scheduler that parallelizes the same code efficiently on several architectures. Specifically, we look at effective programming for the workload-intensive yet relatively straightforward MJPEG video encoding. This task consumes many CPU cycles in the sequential DCT, quantization, and compression stages. On single-core systems, it is almost impossible to process a 1080p high-definition (HD) video in real time, so it is reasonable to apply multicore computing in this scenario.

Workload: MJPEG

The MJPEG format is widely used by webcams and other embedded systems. It is similar to video codecs such as Apple ProRes and VC-3, used for video editing and post-processing due to their flexibility and speed— hence the lack of inter-prediction between frames. As shown in Figure 3.4, the encoding process of MJPEG comprises the splitting of video frames into 8x8 macroblocks, each of which must be individually transformed to the frequency domain by forward DCT and quantized before the output
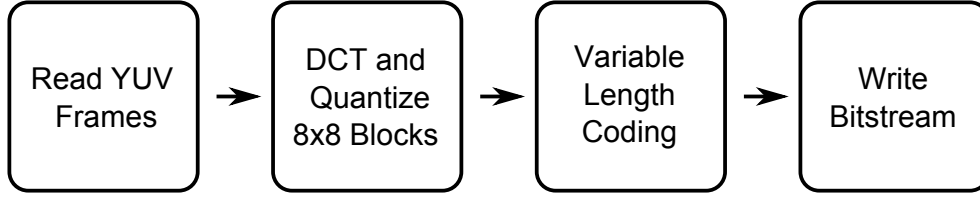
Figure 3.4: Overview of the MJPEG encoding process.

is entropy coded using variable-length coding (VLC). JPEG supports both arithmetic coding and Huffman compression for VLC and our encoder uses predefined Huffman tables for the compression of the DCT coefficients of each macroblock. The VLC step is not context adaptive and macroblocks can thus be independently compressed. The length of the resulting bitstream, however, is probably not a multiple of eight and most such macroblocks must be completely bit-shifted when the final bitstream is created.

The MJPEG format provides many layers of parallelism; starting with the many independent operations in calculating DCTs, the macroblocks can be transformed and quantized in arbitrary order and frames and color components can be encoded separately. In addition, every frame is entropy coded separately. Thus, many frames can be encoded in parallel before the resulting frame output bitstreams are merged. This provides a very fine level of granularity for parallel tasks, providing great flexibility in implementing the encoder.

The forward two-dimensional (2D) DCT function for a macroblock is defined in the JPEG standard for image component $s_{y,x}$ to output DCT coefficients $S_{v,u}$ as

$$S_{v,u} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{y,x} cos \frac{(2x+1)u\pi}{16} cos \frac{(2y+1)v\pi}{16}$$

where $C_u, C_v = \frac{1}{\sqrt{2}}$ for $u, v = 0$ and $C_u, C_v = 1$ otherwise. The equation can be directly implemented in an MJPEG encoder and is referred to as 2D plain. The algorithm can be sped up considerably by removing redundant calculations. One improved version that we label one dimensional (1D) plain uses two consecutive 1D transformations with a transposition operation in between and after. This avoids symmetries and the 1D transformation can be optimized further. One optimization uses fast DCT, the Arai–Agui–Nakajima [5], or AAN, further refined by Kovac and Ranganathan [72]. Another uses a precomputed 8x8 transformation matrix that is multiplied with the block together with the transposed transformation matrix. The matrix includes the post-scale operation and the full DCT operation can therefore be completed with just two matrix multiplications, as explained by Kabeen and Gent [15]. More algorithms for calculating DCT exist, but they were not covered in our experiments.

x86 Experiments

The first x86 experiments investigated the efficiency of choosing the correct algorithm for the platform. We implemented the different DCT algorithms as scalar single-threaded versions on an Intel Core i5-750 based on the Nehalem microarchitecture. The performance details for encoding HD video were captured using Oprofile and can be seen in Figure 3.5.
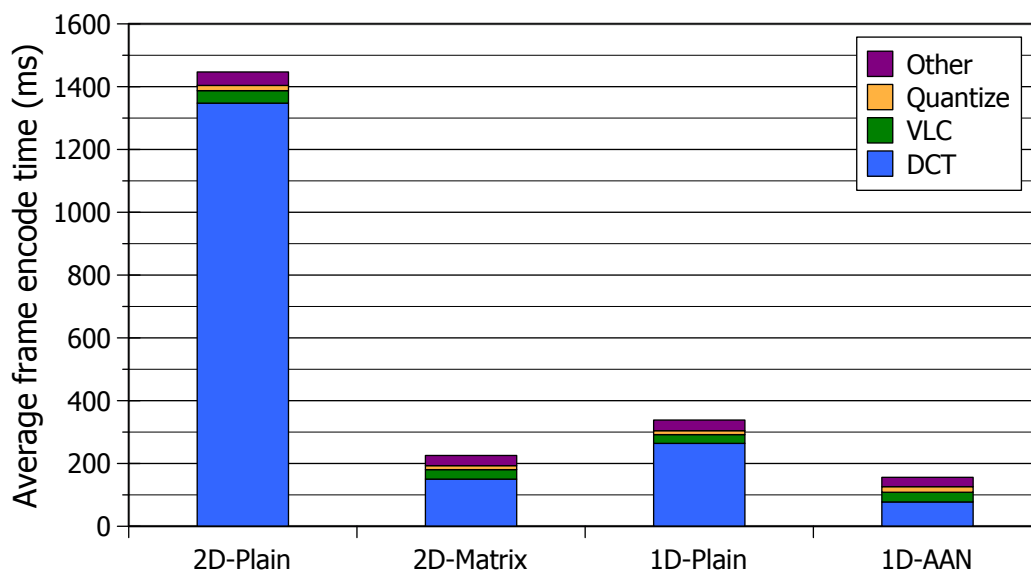
Figure 3.5: MJPEG encoding time on a single-thread x86.

The plot shows that the 1D AAN algorithm using two transposition operations was the fastest in this scenario, with the 2D matrix version second fastest. The average encoding time for a single frame using a 2D-plain arrangement is more than nine times slower than a frame encoded using 1D AAN. For all algorithms, the DCT step consumed the most CPU cycles.

Using a scalar version of the DCT is not the most efficient use of the execution pipeline on a x86 processor. We therefore took the simple DCT algorithm (2D plain) and optimized it with Streaming SIMD Extension (SSE) vector instructions. The experiments were conducted on an Intel Core i7-3720QM processor and the optimized version of the DCT algorithm used the 128-bit SSE 4.2 instruction set. In this experiment, we use the 1080p standard test sequence "tractor" to benchmark the implementation.
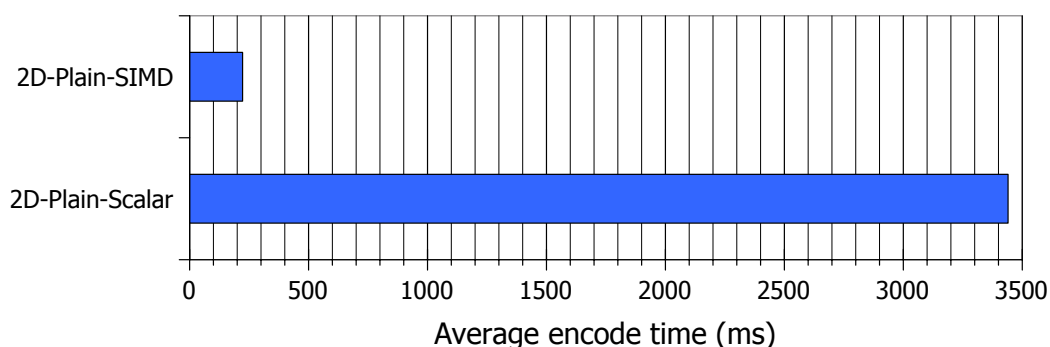
Figure 3.6: Scalar and SIMD versions with a 2D-plain arrangement on an x86.

The results are shown in Figure 3.6. The scalar version of the code that is a straightforward implementation of the 2D-plain algorithm uses around 3500 ms per HD frame. The SIMD optimized version uses only 222 ms. This implementation uses 128-bit SIMD vectors, meaning that we can process four DCT values in parallel, while the scalar version

only processes one value at a time.

Summary

These results show that even on the x86 processor architecture, it is important to both choose the right algorithm and optimize the selected algorithm for the platform. However, writing an SIMD version of the x86 code is, as we see later with Cell, a tedious process. Everything must be done by hand and, since not all x86 processors support the same SIMD instructions, you might need multiple versions of the optimization if you want code portability.

3.2.2   Case Study: Multi-Rate Video Encoding with VP8

To learn more about the importance of sharing data between multiple threads and processes when parallelizing multimedia workloads, we investigate how to use the x86 architecture for multi-rate video encoding with the VP8 codec. We use the shared memory architecture of the x86 processors to reuse the computationally expensive analysis step between multiple instances of the VP8 encoder running in different threads.

Setting

The amount of video data available on the Internet is exploding and the number of video streaming services, both live and on demand, is quickly increasing. For example, consider the rapid deployment of publicly available Internet video archives providing a wide range of content such as newscasts, movies, and educational videos. Internet users uploaded 100 hours of video to YouTube every minute in October 2014 [139]. Furthermore, all major (sports) events, such as European soccer leagues, NFL hockey, NBA basketball, and NFL football, are streamed live with only a few seconds' delay to millions of concurrent users over the Internet, supporting a wide range of devices, from mobile phones to HD displays. The number of videos streamed from such services is on the order of tens of billions per month [37, 139] and leading industry experts conjecture that traffic on mobile phone networks will soon be dominated by video content [23].

Adaptive HTTP streaming is frequently used on the Internet and is currently the de facto video delivery solution. For example, Move Networks [81] was one of the first providers of segmented adaptive HTTP streaming, later followed by major actors such as Microsoft [141], DASH [110], Apple [96], and Adobe [2]. In these systems, the bitrate (and thus video quality) can be changed dynamically to match varying bandwidths and CPU resources, providing a large advantage over non-adaptive systems, which are frequently interrupted due to buffer underruns or data loss. The video is thus encoded in multiple bitrates matching different devices and different network conditions.

Today, H.264 is the most frequently used codec. However, an emerging alternative is the simpler VP8, which is very similar to H.264's baseline profile and supposedly well suited for web streaming, with native support in major browsers, royalty-free use, and similar video quality as H.264 [95, 108]. Nevertheless, for both codecs, the challenge in the multi-rate scenario is that each version of the video requires a separate processing instance of the encoding software. This may be a challenge in live scenarios, where all the rates must be delivered in real time, and, in YouTube's case, will require an enormous

data center to maintain the upload rate. Thus, the process of encoding videos at multiple levels of quality and data rates consumes both time and resources.

Workload: The VP8 Codec

The VP8 codec [9] was originally developed by On2 Technologies as a successor to VP7 and is a modern codec for storing progressive video. After acquiring On2 Technologies in 2010, Google released VP8 as an open-source *WebM* project, a royalty-free alternative to H.264. The WebM format was later added as a supported format in the upcoming HTML5 standard.

Many of the features in the VP8 codec are heavily influenced by H.264. The VP8 codec has similar functionality as the H.264 baseline profile. One of the differences is that VP8 has an adaptive binary arithmetic coder instead of context-adaptive VLC (CAVLC). However, VP8 is not designed to be an all-purpose codec and it primarily targets web and mobile applications. This is why VP8 has omitted features such as interlacing, scalable coding, slices, and color spaces other than YUV 4:2:0. This reduces encoder and decoder complexity while retaining video quality for the most common use case, that is, making VP8 a good choice for lightweight devices with limited resources.

A VP8 frame is either of the *intra-frame* or the *inter-frame* type, corresponding to I- and P-frames in H.264, but it has no equivalent to B-frames. In addition, VP8 introduces the concept of tagging a frame as *altref* and *golden* frames, which are stored for reference in the decoder. When predicting, blocks may use regions from the immediately previous frame, from the last *golden* frame, or from the last *altref* frame.

The encoding loop of VP8 is very similar to that of H.264. The process consists of an analysis stage, which decides if intra- or inter-prediction will be used, DCT, quantization, dequantization, and inverse DCT (iDCT), followed by an in-loop deblocking filter. The result of the quantization step is entropy coded using a context-adaptive Boolean entropy coder and stored as the output bitstream. The output bitrate of the resulting video is dependent on the prediction parameters in the bitstream and quantization parameters.

Multi-Rate Encoding

The multi-rate encoder is based on the reference VP8 encoder, released as part of the WebM project [9]. Figure 3.7 shows a simplified call graph of the VP8 reference encoder. In this call graph, we can see the flow of the program, how many times a function has been called, and the percentage of execution time spent in different parts of the code. The basic flow of the entire encoder is illustrated in the upper part of Figure 3.8, with an analysis and the encoding part of the pipeline.

The *analysis* part consists of macroblock mode decision and intra/inter-prediction, which corresponds to `vp8_rd_pick_inter_mode` in Figure 3.7. The *encode* part refers to transformation, quantization, dequantization, and inverse transformation, corresponding to the functions `vp8_encode_inter*` and `vp8_encode_intra*` for the various block modes chosen. The *Output* involves entropy coding and writing the output bitstream to a file. This part of the encoder is not shown in the call graph. Profiling of the VP8 encoding process shows that during encoding of the *foreman* test sequence, over 80% of the execution time is spent in the analysis part of the code; that is, if this part can be reused for encoding operations for other rates, resource consumption can be greatly reduced. This
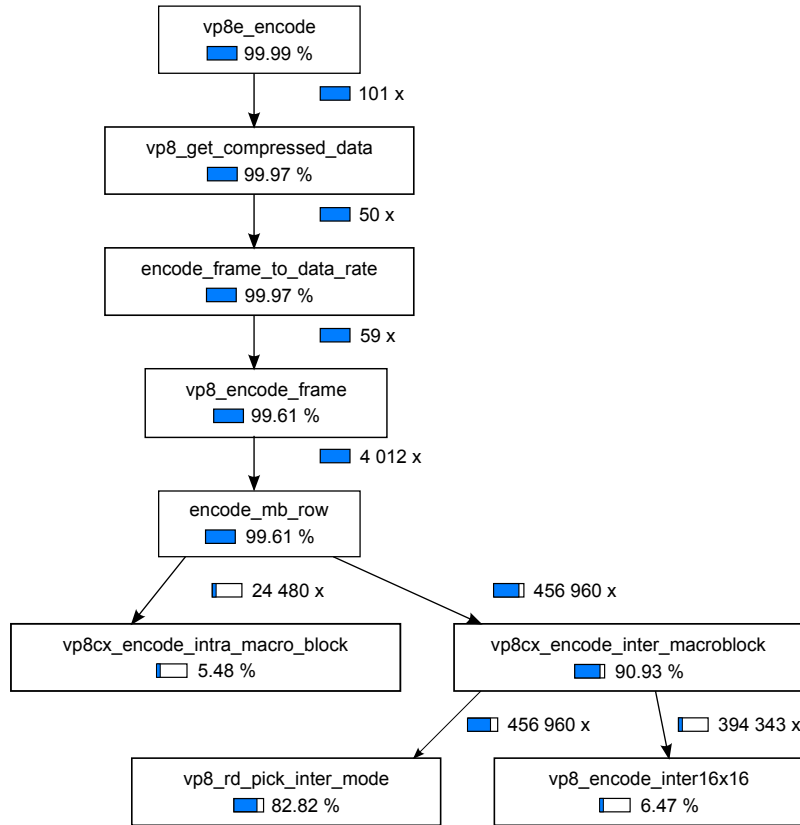
Figure 3.7: Profile of the main parts of the reference VP8 encoder.

can be done because the outputs have identical characteristics except for the bitrate and the main difference between them involves the quantization parameters. Regardless of the target bitrate, the analysis step that includes searching for motion vectors and other prediction parameters can be carried out without considering the target bitrate, trading this for prediction accuracy.

To evaluate this approach, we implemented a VP8 encoder with support for multiple outputs. We reused a single analysis step for several instances of the encoding part, as seen in Figure 3.8. This mitigates the requirements for re-doing the computationally heavy analysis step and at the same time allows the encoding instances to emit different output bitrates by varying the quantization parameters in the encoder step. The encoder starts one thread for each specified bitrate, where each thread corresponds to a separate encoding instance. The instances have identical encoding parameters, such as key frame interval and subpixel accuracy, except for the target bitrate. Since the bitrate varies, each instance must maintain its own state and reconstruction buffers. The threads are synchronized on a frame-by-frame basis, where the main encoding instance analyzes the frame before the analysis computations are made available to the other threads. This process involves macroblock mode decision and intra- and inter-prediction. The non-main encoding instances reuse these computations directly without carrying out the computationally intensive analysis steps. Most notably, vp8_rd_pick_inter_mode (Figure 3.7) is only performed by the main encoding instance. Since the VP8 encoder is not written with the intention of running multiple encoding instances in parallel, the encoder goes through
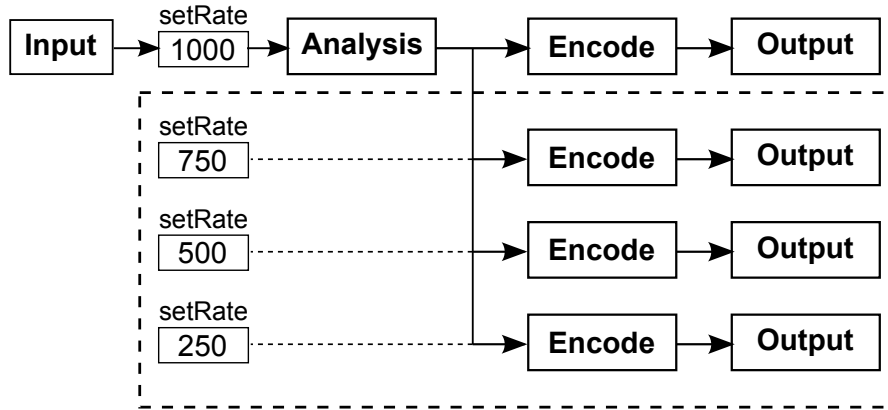
Figure 3.8: Basic flow for the multi-rate VP8 encoder.

significant changes to adapt itself to run multiple instances in parallel.

Experiments

For this case study, we include only one of the experiments that investigated encoding performance at HD resolution. We do not include quality assessment tests, prediction bitrate selection, and the analysis of encoding behavior for different content. These tests are considered out of the scope of this thesis and can be found in paper V [34].

In the HD streaming scenario, we performed experiments using the 1080p resolution test sequence *blue sky* encoded at 1500 kbps, 2000 kbps, 2500 kbps, and 3000 kbps. To measure performance, we used *time* to measure the CPU time consumed. All experiments were run on a four-core Intel Core i5-750 processor based on the Nehalem microarchitecture. This processor does not support SMT.

Figure 3.9 shows the results for the four different output rates. To see if there is a difference for the different *prediction bitrates* chosen when using the multi-rate encoder, we included one test for each prediction bitrate. These results are compared to the combined CPU time used when encoding the videos for the same rates using the reference encoder with both a single thread and multiple threads. The CPU time used in the multi-rate approach needs only 40.5% of the time it takes to encode four sequences using the reference encoder. The multi-rate approach scales further if the number of encoded streams is increased. In addition, the time spent in kernel space is far less in the multi-rate approach compared to the reference encoder and we believe this is a result of reading the source video from disk only once.

Summary

To demonstrate our idea, we implemented a prototype that reuses the most expensive operations based on a performance profile of the encoding pipeline. In particular, our multi-rate encoder reuses the analysis part consisting of macroblock mode decision and intra/inter-prediction. The experimental results indicate that we can encode the different videos at the same rates with approximately the same levels of quality compared to the VP8 reference encoder, while significantly reducing the encoding time.
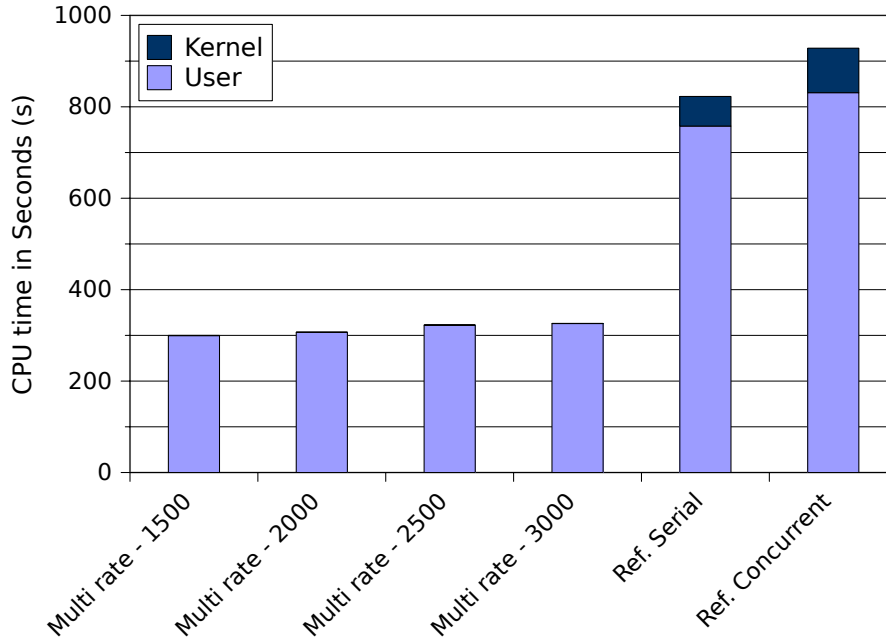
Figure 3.9: CPU time in an HD streaming scenario (blue sky).

We analyzed and performed experiments with Google's VP8 encoder, encoding different types of video at multiple rates for various scenarios. Our main contribution is that we propose a way of reusing decisions from intra- and inter-prediction in the video encoder to avoid computationally expensive steps that are redundant when encoding for multiple target bitrates of the same video object. The method can be used in any video codec, comprising an analysis and encoding step with similar structure as for H.264 and VP8. Furthermore, the method was implemented in the VP8 reference encoder as a case study and the experimental results show that the computational demands are significantly reduced at the same rates and approximately the same quality levels compared to the VP8 reference implementation; that is, for a negligible loss in quality in terms of PSNR, the processing costs can be greatly reduced.

We also learned that the shared memory architecture on the x86 is suited for sharing the data from the computationally expensive steps in the VP8 encoder. Our experiments also show that if we use multiple instances of the reference encoder, the performance is actually better if the workload is executed sequentially instead of concurrently. This is due to greater contention in both the operating system scheduler and on buses, caches, and execution resources on the CPU.

3.2.3   Case Study: Parallel Execution of a Game Server

Many multimedia workloads are massively parallel and, when such workloads are optimized on the x86 architecture, the number of threads used is an important parameter. Too many threads will result in decreased performance due to the context switching overhead in the operating system. With our game server workload, we want to investigate this threshold on the x86 architecture.

Setting

Over the last decade, online multiplayer gaming has experienced amazing growth. Providers of the popular online games must deliver reliable service to thousands of concurrent players, meeting strict processing deadlines for the players to have an acceptable quality of experience.

One major goal for large game providers is to support as many concurrent players in a game world as possible while preserving strict latency requirements for the players to have an acceptable quality of experience. The load distribution in these systems is typically achieved by partitioning game worlds into areas of interest to minimize message passing between players and allow the game world to be divided between servers. Load balancing is usually completely static, where each area has dedicated hardware. This approach is, however, limited by the distribution of players in the game world and the problem is that the distribution of players is heavily skewed, with about 30% of players in 1% of the game area [20]. To handle the most popular areas of the game world without reducing the maximum interaction distance for players, individual spatial partitions cannot be serial. The most CPU-intensive loads for a massively multiplayer online game (MMOG) server are in situations in which the players experience the most "action." Hence, the worst-case scenario for a server is when a large proportion of the players gather in a small area for high-intensity gameplay.

The traditional design of MMOG servers relies on *sharding* for further load distribution when too many players visit the same place simultaneously. Sharding involves making a new copy of an area of a game, where players in different copies are unable to interact. This approach eliminates most requirements for communication between the processes running individual shards. An example of such a design can be found in Chu et al. [22].

The industry is now experimenting with implementations that allow for greater levels of parallelization. One example is Eve Online [30], which avoids *sharding* and allows all players to potentially interact. Large-scale interactions in Eve Online are handled through an optimized database. At the local scale, however, the servers are not parallel and performance is extremely limited when too many players congregate in one area. With a lockless, relaxed atomicity state (LEARS), we take this approach even further and focus on how many players can be handled in a single game world segment. We present a model that allows for better resource utilization of multiprocessor game server systems that should not replace spatial partitioning techniques for work distribution but, rather, complement them to improve on their limitations. Furthermore, a real prototype game is used for evaluation, where captured traces are used to generate server loads. We compare multithreaded and single-threaded implementations to measure the overhead of parallelizing the implementation and to demonstrate the experienced benefits of parallelization. The change in responsiveness of different implementations with increased loads on the server is studied and we discuss how generic elements of this game design impact performance on our chosen implementation platform.

Workload: LEARS Model Game Server

Traditionally, game servers have been implemented much like game clients: based around a main loop that updates every active element in the game. These elements include, for example, player characters, non-player characters, and projectiles. The simulated world

has a list of all the active elements in the game and typically calls an update for each element. The simulated time is kept constant throughout each iteration of the loop, so that all the elements obtain updates at the same points in simulated time. Such a point in time is referred to as a *tick*. Using this method, the active element performs all its actions during the tick. Since only one element updates at a time, all actions can be performed directly. The character reads input from the network, performs updates on itself according to the input, and updates other elements with the results of its actions.

LEARS is a game server model with support for lockless, relaxed atomicity state parallel execution. The main concept is to split the game server executable into lightweight threads at the finest possible granularity. Each update of every player character, AI opponent, and projectile runs as an independent work unit.

White et al. [130] describe a model they call a *state-effect pattern*. Based on the observation that changes in a large actor-based simulation are happening *simultaneously*, the model separates read and write operations. Read operations work on a consistent previous state and all write operations are batched and executed to produce the state for the next tick. This means that the ordering of events scheduled to execute in a tick does not need to be considered or enforced. For this design, we additionally remove the requirement of batching write operations, allowing these to happen at any time during the tick. The rationale for this relaxation is found in the way traditional game servers work. In the traditional single-threaded main loop approach, every update is allowed to change any part of the simulation state at any time. In such a scenario, the state at a given time is a combination of values from two different points in time, current and previous, exactly the same situation as in the design presented here.

The second relaxation relates to the atomicity of game state updates. The fine granularity creates a need for significant communication between threads to avoid problematic lock contention. Systems where elements can only update their own state and read any state without locking [1] obviously do not work in all cases. However, game servers are not accurate simulators and, again, depending on the game design, some (internal) errors are acceptable without violating game state consistency.

The end result of our proposed design philosophy is that there is no synchronization in the server under normal running conditions. Since there are cases in which transactions are required, they can be implemented outside the LEARS event handler, running as transactions requiring locking.

Design and Implementation

In our experimental prototype implementation of the LEARS concept, the parallel approach is realized using thread pools and blocking queues. The creation and deletion of threads incur large overheads and context switching is an expensive operation. These overheads constrain a system's design, that is, threads should be kept as long as possible, and the number of threads should not grow unbounded. We use a *thread pool* pattern to work around these constraints and a thread pool executor (the Java `ThreadPoolExecutor` class) to maintain the pool of threads and a queue of tasks. When a thread is available, the executor picks a task from the queue and executes it. The thread pool system itself is not preemptive, so the thread runs each task until it is done. This means that, in contrast to normal threading, each task should be as small as possible, that is, larger units of work
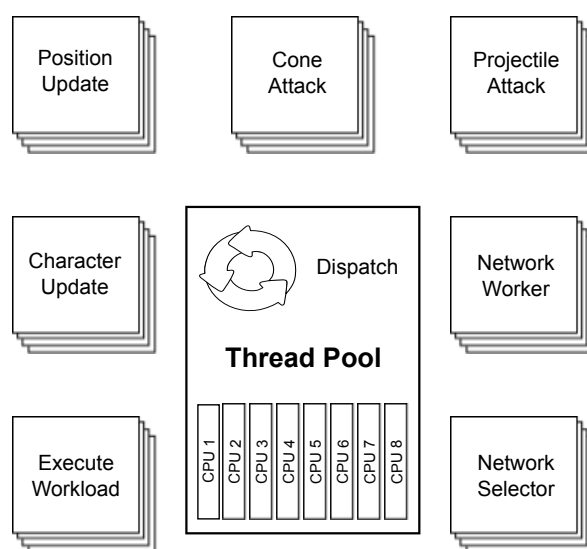
Figure 3.10: Design of a game server.

should be split up into several sub-tasks.

The thread pool is a good way to balance the number of threads when the work is split into extremely small units. When an active element is created in the virtual world, it is scheduled for execution by the thread pool executor and the active element updates its state exactly as in the single-threaded case. Furthermore, our thread pool supports the concept of delayed execution. This means that tasks can be put into the work queue for execution at a specified time in the future. When the task is finished for one time slot, it can reschedule itself for the next slot, delayed by a specified time. This allows active elements to have any lifetime, from one-shot executions to the duration of the program. It also allows different elements to be updated at different rates, depending on the game developer's requirements.

All work is executed by the same thread pool, including the slower input/output (I/O) operations. This is a consistent and clear approach, but it does mean that game updates could be stuck waiting for I/O if not enough threads are available.

The thread pool executor used as described above does not restrict which tasks are executed in parallel. All systems elements must therefore allow any of the other elements to execute concurrently.

To enable fast communication between threads with shared memory (and caches), we use *blocking queues*, using the Java `BlockingQueue` class, which implements queues that are synchronized separately at each end. This means that elements can be removed from and added to the queue simultaneously and, since each of these operations is extremely fast, the probability of blocking is low. Thus, these queues allow information to be passed between active objects. Each active object that can be influenced by others has a blocking queue of messages. During its update, it reads and processes the pending messages from its queue. Messages are processed in the order they were put in the queue. Other active elements put messages in the queue to be processed when they need to change the state of other elements in the game.

Messages in the queues can only contain relative information and not absolute values. This restriction ensures that the change is always based on updated data. For example,

if a projectile needs to tell a player character that it took damage, it should only inform the player character about the amount of damage, not the new health total. Since all changes are put in the queue and the entire queue is processed by the same work unit, all updates are based on up-to-date data.

To demonstrate LEARS, we implemented a prototype game containing all the basic elements of a full MMOG, with the exception of persistent states. The system was implemented in Java. This programming language has strong support for multithreading and has well-tested implementations of all the required components. The basic architecture of the game server is described in Figure 3.10. The thread pool size can be configured and will execute the different workloads on the CPU cores. The workloads include the processing of network messages, moving computer-controlled elements (only projectiles in this prototype), checking for collisions and hits, and sending outgoing network messages.

Experiments

In this case study, we only include the resource consumption and thread pool size experiments. We also conducted experiments on client latency. These tests are considered beyond the scope of this thesis, but the experiments can be found in paper VI [102].

To simulate realistic game client behavior, the game was run with five people playing the game with a game update frequency of 10 Hz. The network input to the server from this session was recorded with a timestamp for each message. The recorded game interactions were then played back multiple times in parallel to simulate a large number of clients. To ensure that client performance was not a bottleneck, the simulated clients were distributed among multiple physical machines. Furthermore, since an average client generates 2.6 kbps of network traffic, the 1 Gbps local network interface that was used for the experiments did not limit the performance. The game server was run on a server machine containing four dual-core AMD Opteron 8218 processors with a total of 16 GB of RAM (4 GB of RAM per socket).

We investigated resource consumption when players connected to the game server as shown in Figure 3.11. We present the results for 620 players, since this is the highest number of simultaneous players that the server could handle before a significant degradation in performance. The server was able to keep the update rate smooth, without significant spikes; CPU utilization grew while the clients were logging on and then stabilized at almost full CPU utilization for the rest of the run.

To investigate the effects of the number of threads in the thread pool, we performed an experiment where we kept the number of clients constant while varying the number of threads in the pool. A total of 700 clients were chosen, since this number slightly overloads the server. The number of threads in the pool was increased in increments of two, from two to 256. Figure 3.12 clearly shows that the system utilizes more than four cores efficiently, since the four-thread version shows significantly higher response times. At one thread per core or more, the numbers are relatively stable, with a tendency toward consistently lower response times with more available threads, up to about 40 threads. This could mean that threads are occasionally waiting for I/O operations. Since thread pools are not preemptive, such situations would lead to one core going idle if there were no other available threads. Too many threads, on the other hand, could lead to excessive context switch overhead. The results show that the average slowly increases after about
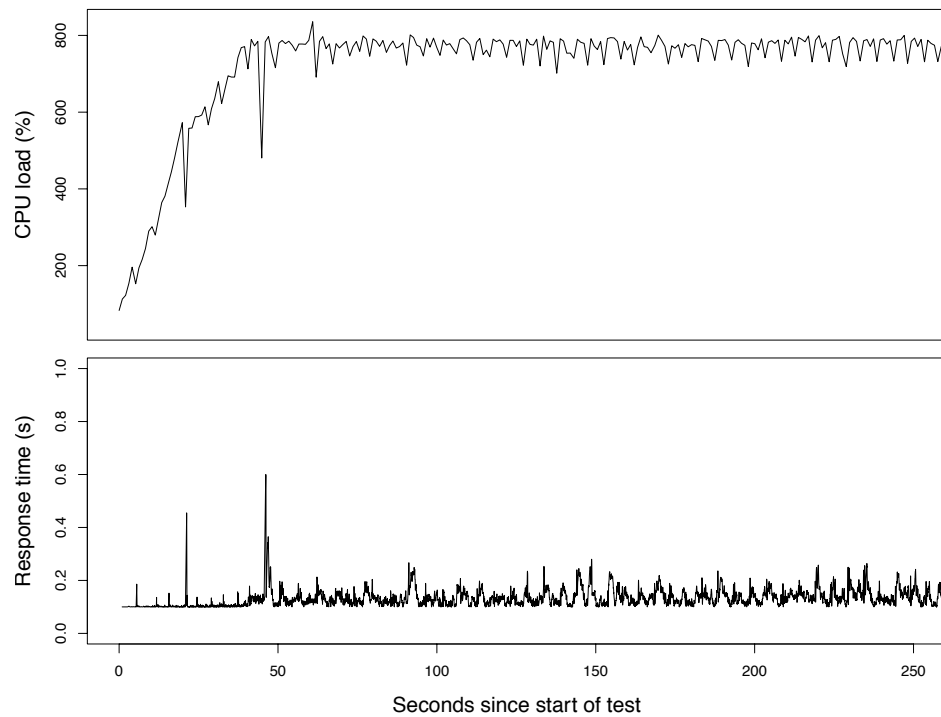
Figure 3.11: CPU load and response time for 620 concurrent clients on a multi-threaded server.
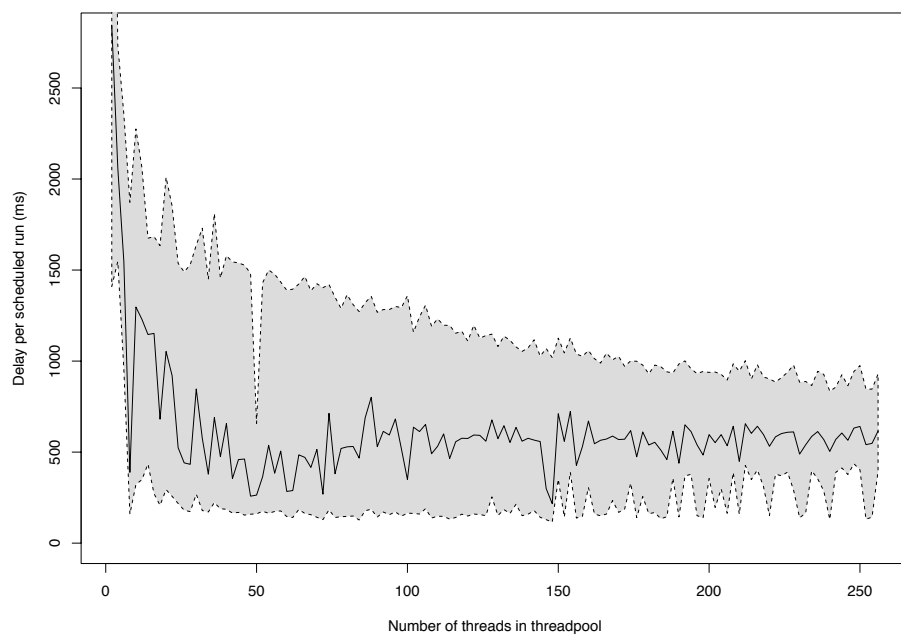
Figure 3.12: Response time for 700 concurrent clients, using various numbers of threads. The shaded area indicates the fifth to 95th percentiles.

50 threads, though the 95th percentile is still decreasing with an increasing number of threads, up to about 100. From then on, the best case worsens again, most likely due to context switching overhead.

Summary

In terms of programming techniques, we have shown that we can improve resource utilization by distributing the load across multiple CPUs in a unified memory multiprocessor system. This distribution is made possible by relaxing constraints on the ordering and atomicity of events. The system scales well, even in the case in which all players must be aware of all other players and their actions. The thread pool system balances load well between the cores and its queue-based nature means that no task is starved unless the entire system lacks resources. Message passing through the blocking queue allows objects to communicate intensively without blocking each other. Running our prototype game, we show that the eight-core server can handle a factor of two more clients before the response time becomes unacceptable.

Our results indicate that it is possible to design an "embarrassingly parallel" game server. We also observe that the implementation is able to handle a quadratic increase in in-server communication when many players interact in a game world hotspot. The experiments also show that if too many threads are added to the thread pool, performance will decrease. This is mainly due to greater contention in the operating system scheduler. We also saw the importance of balancing the number of threads with the number of CPU cores available in the system. If the size of the thread pool is too large, the delays on the game server will increase.

3.2.4   Implications

The x86 architecture is very straightforward for application development. The architecture uses a shared memory model, which means that all the processors available to an operating system are able share the memory and the processor manufacturers have implemented cache coherency protocols to make sure that the data in all the caches are updated. However, this comes at a cost. The traffic generated by these protocols can end up starving the bandwidth on the inter-core communication buses that are used for sharing data and accessing memory.

On asymmetric architectures such as the Cell, IXP, and GPUs, on often has specialized cores that are fast for specific operations. On the x86, all the cores are general purpose. However, they often have specialized functions, such as SSE/AVX units, to carry out fast vector operations, but this requires the applications to be optimized by hand.

Another challenge with the x86 when running an application with many threads is that the threads on the platform are not as lightweight as on the Cell or on GPUs and too many threads executing on too few cores will result, as we saw in Section 3.2.3, in loss of performance due to the context switching overhead. We also saw the same trends with the VP8 encoder in Section 3.2.2. Running Google's reference encoder serially provided better performance than running it concurrently.

The x86 hides a great deal of the architectural details from programmers. This makes the architecture very easy to use, but comes at the cost of performance.

Revisiting with State-of-the-Market Hardware

If we were to revisit these experiments with the latest generation of Haswell x86 processors, we would probably obtain better performance. The advantage for end users with the x86 is its backward compatibility. This means that we can just run the programs, even without recompiling them. This is not an optimal solution for hardware manufacturers. Compatibility with old instructions makes these processors' designs more complex, which increases their power consumption. However, if we want to utilize the more advanced vector instructions and other extensions (i.e., transactional memory support) that are added in new generations of processors, we still have to rewire the applications for support.

3.3 Graphics Processing Units

A GPU is an asymmetric coprocessor that is often connected to the CPU with a PCI Express bus. In some cases, it can also be integrated onto the CPU die. The GPU is a highly parallel architecture and, where an x86 processor would require tens of threads to achieve peak performance, a GPU would typically require thousands of threads. A GPU, like the Cell, has an exclusive memory model. The GPU has multiple off-chip and on-chip memory types that a programmers must use correctly to achieve the best performance. The main focus for our experiments on the GPU was first to learn how they performed in different scenarios and, later, how we could offload parallel parts of multimedia workloads.

In the following section, we take a closer look at four case studies. First, in Section 3.3.1, we look at how to use the memory correctly and the implications of not choosing the right memory space. Next, in Section 3.3.2, we look at communication patterns between the host CPU and the GPU, using a multimedia workload. In Section 3.3.3, we use the GPU to detect cheating in a multiplayer game and, finally, in Section 3.3.4, we look at the MJPEG workload that we also touched upon in the Cell and x86 portions of this chapter.

3.3.1 Case Study: GPU Memory Spaces and Access Patterns

To obtain the best possible performance when using a GPU, programmers need to be careful when it comes to resource usage. Registers per thread, occupancy on the GPU, memory placement, and access patterns are properties of a GPU kernel that are important for achieving optimal performance. As part of a master's thesis [94], we conducted experiments with the memory architecture on GPUs released in 2006 and 2008 based on the Nvidia Tesla architecture.

To gain a better understanding of how to optimize memory access, the programmer needs to be aware of how memory instructions are executed by the memory controller on the GPU. This is especially important in the case of global memory, since it is used by every thread and it is the memory space with the highest latency. The threads on an Nvidia GPU are scheduled in groups of 32 threads called warps. To make the scheduling more flexible, the memory transactions from a warp are executed on a half-warp basis. This is due to the design of the shared memory and to ease the handling of memory transactions from threads in a divergent warp. Divergence within a warp means that

threads execute different instructions, which can be due to branching in the code or idle threads.

| Thread 0 → Address 0 |
| Thread 1 → Address 4 |
| Thread 2 → Address 8 |
| Thread 3 → Address 12 |
| Thread 4 → Address 16 |
| Thread 5 → Address 20 |
| Thread 6 → Address 24 |
| Thread 7 → Address 28 |
| Thread 8 → Address 32 |
| Thread 9 → Address 36 |
| Thread 10 → Address 40 |
| Thread 11 → Address 44 |
| Thread 12 → Address 48 |
| Thread 13 → Address 52 |
| Thread 14 → Address 56 |
| Thread 15 → Address 60 |

64 byte segment

| Thread 0 → Address 0 |
| Thread 1 → Address 256 |
| Thread 2 → Address 512 |
| Thread 3 → Address 768 |
| Thread 4 → Address 1024 |
| Thread 5 → Address 1280 |
| Thread 6 → Address 1536 |
| Thread 7 → Address 1792 |
| Thread 8 → Address 2048 |
| Thread 9 → Address 2304 |
| Thread 10 → Address 2560 |
| Thread 11 → Address 2816 |
| Thread 12 → Address 3072 |
| Thread 13 → Address 3328 |
| Thread 14 → Address 3584 |
| Thread 15 → Address 3840 |

(a) Coalesced access pattern   (b) Uncoalesced access pattern

Figure 3.13: Global memory access patterns.

The half-warps executing on the GPU are most efficient when memory access from simultaneously running threads can be combined into a single memory transaction to global memory. This is known as a *coalesced* memory transaction. The half-warp must meet certain requirements to coalesce the memory transaction and these requirements are determined by the GPU's compute capability. The compute capability also affects how the transactions are issued. If the requirements are not met, this is referred to as a *uncoalesced* memory transaction.

An example of a coalesced and an uncoalesced access pattern is illustrated in Figures 3.13(a) and 3.13(b). In this example, coalesced access is achieved by having each thread access a 32-bit word in sequence within a 64-byte segment. The uncoalesced access reads values from different segments, making it impossible for the memory controller to coalesce such access.

Global Memory

Global memory is used most efficiently when all the threads of a half-warp can issue a coalesced memory transaction. The size of a memory transaction that can be executed depends on the compute capability supported by the GPU. A 64- and a 128-byte transaction can be performed on a compute capability of 1.0 and 1.1 GPU, while a compute capability of 1.2 and greater also added support for 32-byte transactions. The transaction

size is important, since global memory is considered to be partitioned into segments of 32 bytes, 64 bytes, or 128 bytes.

Constant and Texture Memory

The constant and texture memory spaces are designed for read-only data structures that have elements that reside close in memory. The memory spaces are limited in size, are read only, and are therefore not always suitable for certain applications. Both memory spaces use a caching mechanism in which an 8-kB cache is available for both texture and constant memory on each SM/SMX. If there is a cache miss, a read costs the same as a fetch from global memory, since both memory spaces are subsets of global memory. An advantage of using these read-only memory spaces is that the requirements for optimal performance are not as strict as in global memory. Threads of a warp that read texture addresses that are close together will achieve the best performance; so, mapping the read-only data to fit this alignment is considered a good optimization.

The texture and constant caches differ in the kind of locality for which they are optimized. The constant cache is as fast as reading a register, as long as all the threads in a half-warp read from the same address (data element), and the cost scales linearly with the number of different addresses that are read. The texture cache is a more flexible cache, since it does not require each thread to read the same address for full speed. However, having threads read addresses that are close to each other is recommended, since the cache is optimized for 2D spatial locality used in imaging. Texture memory is normally used for the storage of texture data used for rendering images.

Experiments and Summary

Coalesced memory access can have a large performance impact. However, few quantified results exist and the efficient usage of memory types, alignment, and access patterns remains an art. Weimer et al. [129] experimented with bank conflicts in shared memory but, to shed light on the penalties of inefficient memory type usage, further investigation is needed. We therefore performed experiments that read and wrote data to and from memory with both uncoalesced and coalesced access patterns [94] and used the Nvidia CUDA Visual Profiler to isolate the GPU times for the different kernels.
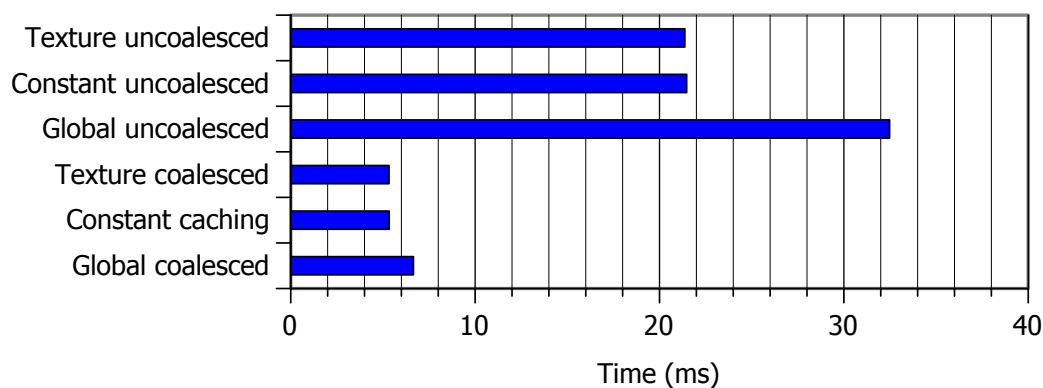
Figure 3.14: Optimization of GPU memory access.

Figure 3.14 shows that an uncoalesced access pattern increases the latency of the data transfer on the order of four times due to the increased number of memory transactions. Constant memory and texture memory are cached and the performance of their uncoalesced access is improved compared to global memory, but there is still a three-time penalty. Furthermore, the cached memory types support only read-only operations and are restricted in size. When used correctly, the performance of global memory is equal to the performance of the cached memory types. The experiment also shows that correct memory usage is imperative, even when cached memory types are used. It is also important to ensure that the memory access follows the specifications of particular GPUs because the optimal access patterns vary between GPU generations [94].

3.3.2 Case Study: Host–Device Communication Optimization

A very important aspect of using a GPU is moving the data from the host CPU into the GPU as quickly as possible. While doing this, it is also important to always have workloads ready for the GPU and not leave any cores idle. As part of a master's thesis [13], we performed experiments on a CUDA-based H.264 encoder from the National University of Defense Technology, China. The encoder is called *cuve264b* and is a port of their streaming HD H.264 encoder [134] to the CUDA architecture. In this thesis, we investigate the effects of optimizing communications between the host CPU and the GPU. For details about the H.264 video compression standard and for a full evaluation, the reader should consult the master's thesis mentioned [13].

The cuve264b encoder uses slices to help parallelize the encoding process. By dividing each frame into multiple slices, the encoder can encode each slice independently. In a snapshot of the encoder on which we conducted our experiments, the number of slices was hard-coded to 34. This version also only supports the 1080p resolution. However, 720p resolution was added later. All the tests in this experiment were conducted on a GeForce GTX 480 GPU based on the Fermi compute architecture. The CPU used in the experiments was an Intel Core i7-860 based on the Nehalem microarchitecture, with Hyper-Threading Technology (SMT) enabled.

To ensure that video frames from the host CPU are always available to the GPU, readahead was been implemented on the CPU side and makes sure that the encoder has finished reading the frames of uncompressed videos into main memory before the frame is requested by the GPU. This is one of the optimizations implemented to make sure that the cores in the GPU are never idle. By optimizing the code with readahead on the host CPU side, we were able to reduce the encoding time by around 20% [13].

CUDA Streams

Readahead was implemented by using multiple threads on the host CPU to asynchronously perform the IO operations. However, multiple threads on the host cannot perform concurrent operations on the GPU. To enable this, CUDA provides an asynchronous API called CUDA Streams [92]. This API allows us to copy data directly into the GPU's global memory and queue up multiple operations on the GPU. These operations are performed asynchronously to the host threads and the GPU is able to perform some of the operations concurrently. In early-generation GPUs from the Tesla architecture, it was

only the transfer of data between the host and device and the computations that could be overlapped. With the next generation of GPUs from the Fermi architecture, the GPU could concurrently execute up to 16 kernels while transferring a single stream. To enable asynchronous memory transfers, pinned memory must be used on the host. This enables the direct memory access (DMA) engine on the GPU to transfer the data without involving the CPU.

Experiments and Summary

To measure the effects, we implemented asynchronous transfers in the cuve264b encoder. We also implemented support for reusing memory, since pinned memory is a limited resource and slow to allocate. As we can see from Figure 3.15, the asynchronous transfers are about 10 times faster than the synchronous version. The results depend on the number of frames that have to be encoded, so, since our test sequences are of limited length, the difference would be greater for long videos, such as feature films and television shows.
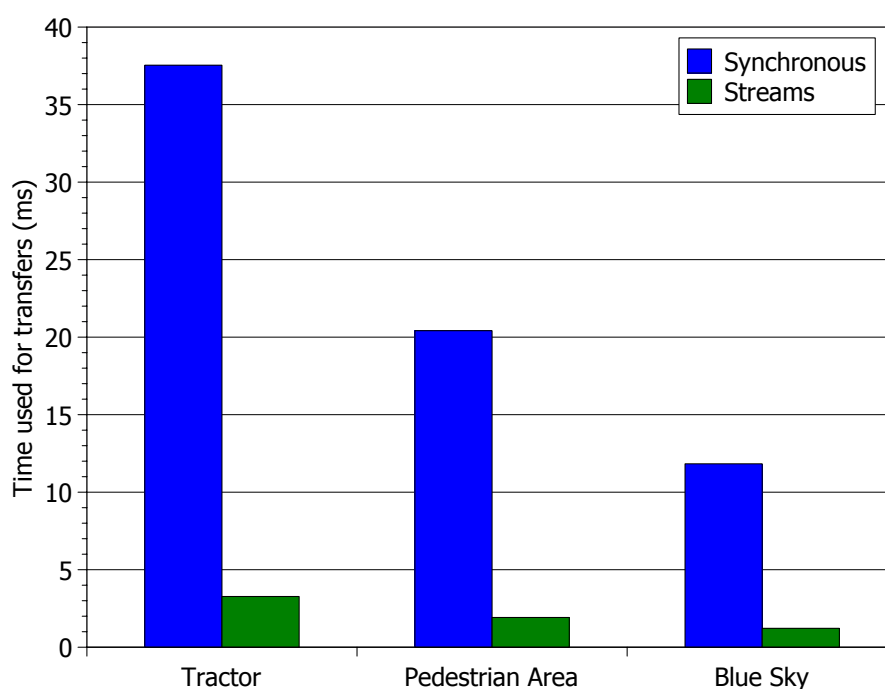
Figure 3.15: Total time spent on transfers to GPU on three different 1080p sample videos.

Our experiments show that using asynchronous transfers such as CUDA Streams to overlap the transfers with computations for large workloads such as H.264 reduces the GPU idle time and consequently also the encoding time. With newer computing devices from the Fermi architecture, the GPU is also able to schedule and execute other kinds of processing tasks concurrently, not only overlap computations from a single kernel and memory transfers.

3.3.3   Case Study: Cheat Detection

When offloading parts of a processing pipeline to a GPU, the data transfer from the host CPU will add latency to the execution. It is therefore important to make sure that the offloaded workload is large enough to overcome this transfer latency. In this section, we use a cheat detection workload to test this effect on a GPU.

Setting

On-line multiplayer gaming has experienced amazing growth over the last decade and has been accompanied by cheating as the most prominent type malicious game player behavior [137]. It is in the best interest of game service providers to eradicate cheating. However, the demand for stable service for resource-intensive games restricts the amount of resources that can be dedicated to cheat detection mechanisms.

Many on-line multiplayer games suffer from excessive cheating in one form or another. However, in many cases, cheating is hard to prove [99]. The only part of a distributed system that a game service provider can trust is the part of the system running on hardware under their control. Any other part of the system can and will most likely be exploited by a cheater.

In-game physics, aimed to increase game realism, has experienced increased popularity in many kinds of games. Most games that have implemented in-game physics use it as a major part of the gameplay experience, some even basing the entire gameplay around physics alone. In-game physics is therefore a very likely part of a game to be exploited. To address this problem, central servers or other trusted entities must ensure consistency in the movements of all the clients in the game. With our approach, the physics engine can be implemented on the server together with the cheat detection mechanisms. This solution frees resources on the game clients; however, it requires more hardware on the server side.

Adding more hardware to a system can increase its performance, but this is only a temporary solution. The hardware used in commercial game server clusters is expensive and the performance gained might only be sufficient for a short period. Because of the physical limitations halting the increase in single-threaded performance in normal CPUs, further performance increase is accomplished by adding more identical processing cores. The modern GPU is a relatively inexpensive example of such a parallel architecture. The process of adding new and faster hardware is now slowly substituted by migrating systems to parallel processors. For this change to be beneficial, serial algorithms must be parallelized.

Our goal in this case study is to use an example cheat detection workload to learn about the overhead of moving the data to the GPU. If we offload too small of a workload to a GPU, the overhead of the data transfer will be greater than the time it takes to perform the calculations on the CPU.

Workload: Cheat Detection

To show the benefits of using a GPU for cheat detection, we created a simple space race game simulation where the spacecrafts must visit virtual positions, also referred to as targets. The clients are placed randomly in the virtual world, giving some clients an

advantage, since they might be placed closer to a target. When a target is reached, the clients continue to the next target.

The simulation follows a client–server-based game architecture where all clients send their position updates to the server. This approach is chosen for the same reasons as in consumer market game development: ease of development, total control of client communication, and a centralized control point. Discrete clients are created within the simulation and communication follows the same flow that would be normal in a networked multiplayer game. Furthermore, because we wanted to design our simulation independent of wall-clock time, we used an artificial timeline based on game ticks. A tick is a theoretical time duration specified in the system's configuration.

To allow for reproducible tests, the simulation uses two different modes of operation, named the generation mode and the playback mode. The *generation mode* uses the principles of the game to determine the random placement of a given number of clients in a virtual environment. From these positions, the clients try to reach the closest target. After reaching a target, they continue to the next target, using a thruster to propel themselves. External forces, such as gravity, affect the clients. While this is happening, the server writes each client's location in the virtual environment to several files. These files are used in playback mode. Generation mode also generates movement for cheaters. The numbers of cheaters can be adjusted in generation mode. A cheater behaves in the same manner as an honest client but regularly performs unrealistic motions. *Playback mode* initializes the clients. The client state information is read from the files generated in generation mode and the states are reported to the server. The server samples the state information updates from every client, placing the samples in a sample buffer. The buffer is read by the cheat detection thread when full.

Because all clients in the game were controlled by the computer, rules were needed to determine their behavior in trying to reach a target. To reach their targets, the clients required motion planning. We did not implement any advanced motion planning algorithms. The clients knew the targets that they reached. After a target was reached, the clients continued to the closest unaccomplished target. Client movement was restricted by the physical model. Honest clients did not break the rules of the model, while cheating clients did.

In our simulation, the objects experienced both linear and angular acceleration. There was a constant gravitational pull affecting the objects, much like the gravity on Earth. All the other forces were generated by the objects themselves, using thrusters. Figure 3.16 shows an outline of a game object, with a main rear thruster and bow thrusters. Objects moved forward with the rear thruster and rotated using the bow thrusters. The size and thruster power could be modified by parameters.

The physics engine is one of the main parts of the simulation. The engine is responsible for calculating the sum of all physical forces acting on all objects and updates their positions accordingly. The physics engine is controlled by configuration parameters that allow one to change physical properties quickly, even during runtime. Game objects are registered with the physics engine, so it maintains a pool of objects to manage. Updates of the parameters of an object, such as throttle, are handled by the individual clients. Integration of the time steps from one game tick to the next is carried out by the engine. The physics engine does this by updating every game object in the object pool. The main implementation of the physics engine runs on the CPU and is only used in generation
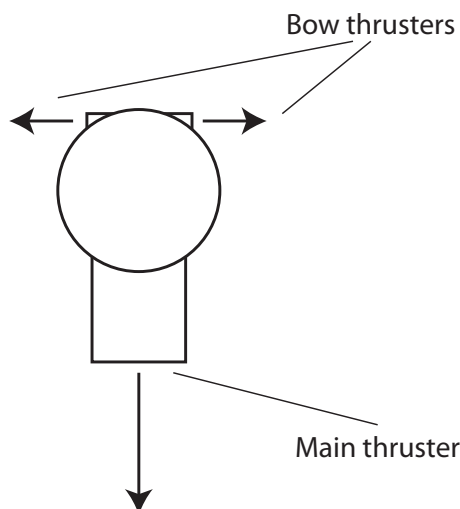
Figure 3.16: Illustration of a game object with bow thrusters in the front and the main thruster in the back.

mode. During playback mode, the cheat detection mechanisms act as a reverse physics engine; they try to determine if the position updates are valid within the current physical model.

The physical model used in this example is a simple model, with only a couple of physical effects. The most basic of these effects is *linear motion*. Basic linear motion is implemented using Newton's second law of motion:

$$\sum \mathbf{F} = m\mathbf{a} \tag{3.1}$$

This law states that the sum of all forces acting on an object is the product of the object's mass and its acceleration. The acceleration is measured by observing the change in speed over a known distance. In our game, two linear forces act on an object. The first is the acceleration applied by the game object's main thruster, as illustrated in Figure 3.16. The second is the vertical gravity, which is constant throughout the entire model. The total linear force is represented by the sum of these two vectors.

The second physical effect is *angular motion*. To allow object rotation in all dimensions, the properties of the objects in the game must be extended. Similar to the linear motion properties of distance, velocity, and acceleration, we have angular motion properties. The equations

$$\Omega = \frac{\mathrm{d}\omega}{\mathrm{d}t} \tag{3.2}$$

$$\omega = \frac{\mathrm{d}\alpha}{\mathrm{d}t} \tag{3.3}$$

where Ω is the angular displacement of an object in radians, ω is its angular velocity in radians per second, and α is its angular acceleration in radians per second squared, show these relations.

Angular motion is applied to the game objects when the bow thrusters illustrated in Figure 3.16 are used to change the course of an object. Support for collisions is

implemented in the model. However, due to lack of time, it was not implemented in the cheat detection mechanisms. It was only present in generation mode.

There are different ways to cheat in the simulation. Clients cheat by either temporarily modifying the power of their thrusters or modifying the values of their current state: their position, velocity, and rotation. If a cheating client temporarily increases the capabilities of one of its thrusters, it is able to accelerate faster, perform quicker turns. Cheaters who change their state can position themselves closer to a target or change their rotation to point toward a target. They might also increase or decrease the magnitude of their velocity vector when either dashing for a target or slowing down to avoid passing a target.

Implementation

We implemented two versions of the cheat detection mechanism. One was written for the host CPU, while the other was a CUDA version, written for the GPU device. The cheat detection mechanism on the GPU was implemented with threads. The CPU implementation was not threaded and used a basic looping structure to simulate the same behavior as the CUDA version.

The behavior of the mechanisms is illustrated in Figure 3.17. A single thread works on three consecutive game state samples for a client: Thread one (th1) works on samples s0, s1, and s2, while thread two (th2) works on samples s1, s2, and s3, and so forth. A sample is the state of the client after a tick in the artificial timeline.
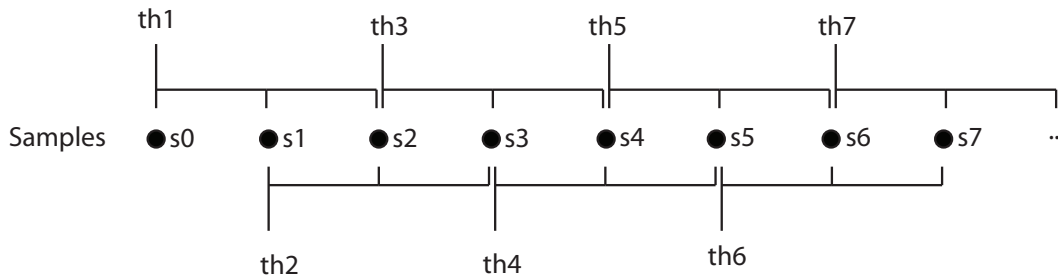
Figure 3.17: Sample reading and execution thread pattern.

A sample contains the movement of each client and a positional vector with three values, x, y, and z, as three-dimensional axes. With three samples, the threads can determine the client's acceleration as a three-dimensional vector. All external forces added by the physical model can now be subtracted by applying the calculations of the physical engine in reverse. The resulting acceleration is the result of the forces the client applied to the game object. If the thrust applied by the client is greater than the maximum thrust allowed by the game, the client is most likely a cheater.

There are two main node types in our simulation: the server and the clients. They exchange data as in real networked games. A packet is either generated by the generation mode or read from a file in playback mode by the clients once for each game tick.

The *server* reads all incoming data from the clients. When a cheater reports erroneous positional data, the cheat detection mechanisms indicate that the player's movement does not follow the rules and restrictions of the game's physical parameters.

Clients act differently depending on the execution mode. During the generation of movement files, clients write their locations and other appropriate data to file. In playback
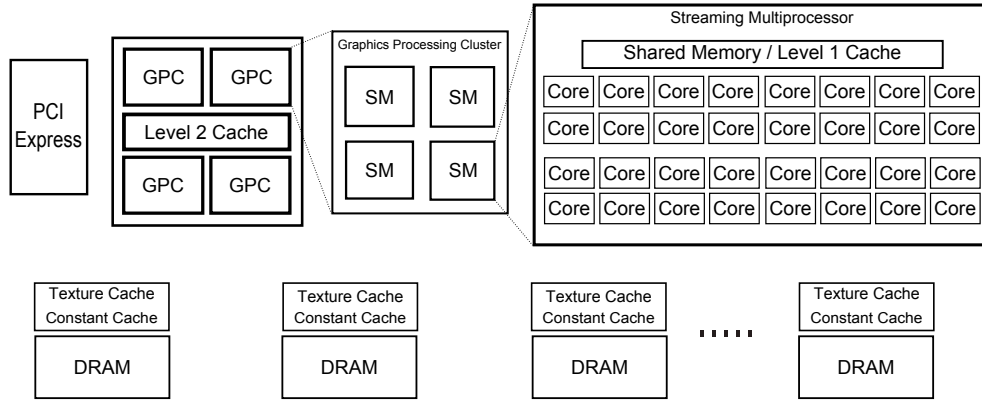
Figure 3.18: Nvidia GF100 compute architecture.

mode, clients read from the generated files and report the data written in generation mode back to the server. In this way, the system allows for reproducible tests as the test data is the same for each test run.

Experiments

In this study, we investigated both the total execution time of the cheat detection system and the total execution time spent on the cheat detection mechanisms. All tests were run on data generated in generation mode over 100 seconds of "game time." The number of clients used in the benchmarks ranges from 10 to 6000. The part of the mechanisms that runs on the GPU in these benchmarks is the reverse physics engine.

The cheat detection mechanism we tested was implemented on the following hardware: The CPU used in the benchmarks was an Intel Core i5-750 processor with 4.0 GB of RAM. The GPU was an Nvidia GeForce GTX 480 with 480 processing cores and 1.5 GB of memory. The chip used in the GeForce GTX 480 is the GF100 GPU, illustrated in Figure 3.18. This GPU is based on the Fermi compute architecture.

Figure 3.19 shows the results of the first benchmark and the total execution time of the cheat detection system. We can observe that, with a low number of clients, the CPU is faster than the GPU, due to the added latency of moving data and code to the GPU. With more than 100 clients in the game, the execution time for the CPU exceeds that of the GPU and the performance gap steadily increases up to 6000 clients, which is the maximum number of tested clients. This is due to the size of the memory on our test machines. When the number of clients increases, the cheat detection processing on the GPU scales much better than on the CPU. When the cheat detection mechanism is processed on the GPU, the CPU is relieved of these tasks and can work on other game-relevant computation.

To determine the offloading effect the GPU has on the CPU, we measured how much of the total execution time was spent on processing cheat detection mechanisms. Figure 3.20 reports the results of the second benchmark. These show that, for a small number of clients, the penalty for transferring data over the PCI Express bus to the GPU is significant, making the CPU more effective for a small number of clients. With more than 50 clients, the GPU implementation spends less time on cheat detection than the CPU implementation does. As the number of clients increases, the time spent on cheat
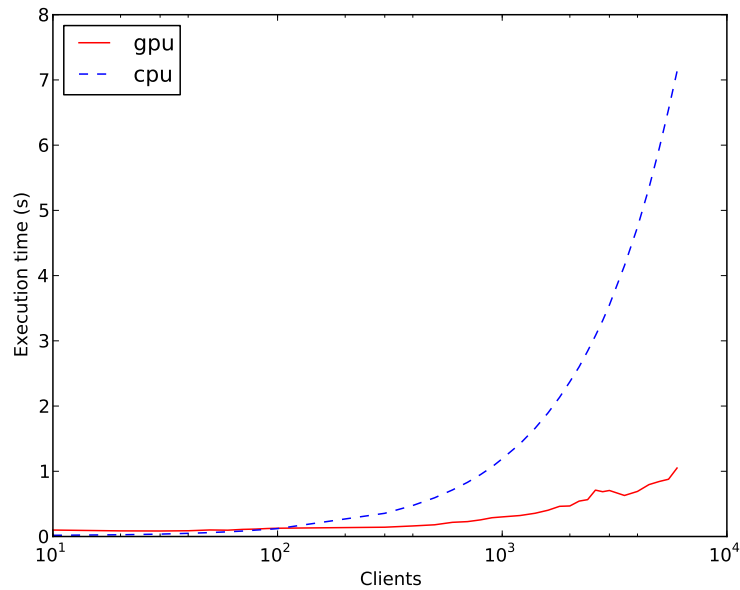
Figure 3.19: Execution time (in seconds) of the cheat detection mechanism on the GPU and the CPU.
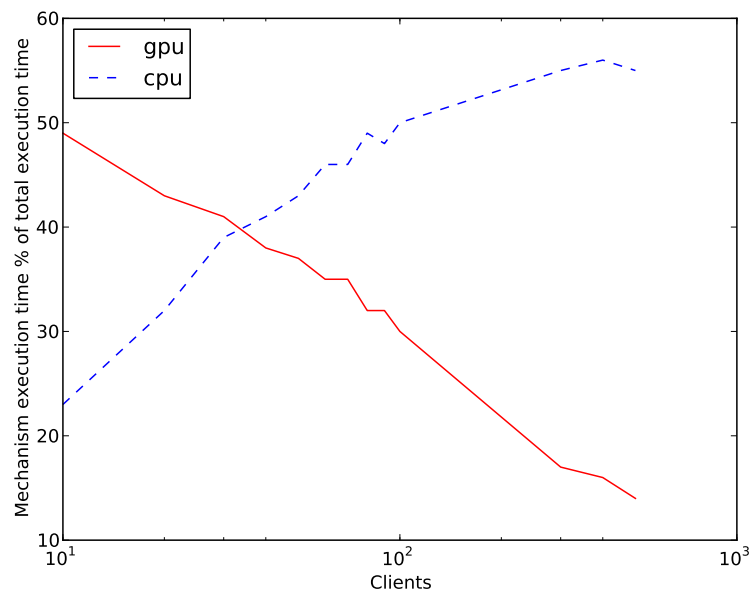
Figure 3.20: Percentage of time spent on cheat detection processing on a host, using the GPU and the CPU.

detection continues to drop to below 15% for the GPU implementation. The CPU version stabilizes around 50%. To improve the performance of the GPU implementation for a low number of clients, it is possible to buffer more samples before executing the mechanisms on the GPU.

Discussion

We have seen how the CPU and GPU implementations of our cheat detection mechanism perform differently when we increase the numbers of clients in the game. The difference between the two is smallest when the number of checks performed on the GPU is small. However, as the number of clients increases, the increase in the execution time of the CPU implementation is much steeper compared to the increase in the GPU implementation. This indicates that the GPU implementation is the more scalable of the two. This is primarily due to the GPU's highly parallel architecture. Physics operations for large numbers of clients are independent of each other. They constitute an embarrassingly parallel workload that maps well to the GPU's multithreaded architecture. Both the CPU and GPU implementations could be further optimized in further work. The CPU implementation could be extended with threading and SIMD operations and the GPU version could be extended with asynchronous transfers, optimized global memory access and the elimination of branching in the compute kernels.

The cheat detection mechanism we implemented for our system is easy to parallelize because the physics computations for clients are independent of each other. Similar systems with workloads that contain operations that can be performed simultaneously by a large number of threads can benefit from using a GPU to offload the processing. When offloading operations to a GPU, it is important to remember that the GPU is most efficient if it has enough data to process. It is also important that the offloaded tasks map well to the GPU's multithreaded architecture.

Summary

Our results show that even a simple physical model can benefit from executing the workload on a GPU. The experiments also clearly show that we need a large workload or a computationally heavy workload to benefit from a GPU. When we benchmarked our cheat detection mechanisms with too few clients, the CPU implementation was faster, because the computational load did not compensate for the latency involved with transferring the data to the GPU. Another advantage of GPU implementation is the offloading effect: While the GPU handles the cheat detection workload, the CPU can perform other tasks.

3.3.4   Case Study: MJPEG Encoding

In our MJPEG case study, we performed several experiments on the GPU architecture and a more detailed overview of the MJPEG workload can be found in Section 3.2.1. As described later for the Cell, several layouts are available for GPUs. However, because of the large number of small cores, it is not feasible to assign one frame to each core. The most time-consuming parts of the MJPEG encoding process, the DCT and quantization steps, are well suited for GPU acceleration. In addition, the VLC step can also be partly adapted.

Our experiments compared 14 different GPU implementations of the MJPEG encoder. This gives a good indication that the architectures are complex to use and that achieving high performance is not trivial. Derived from a sequential codebase, these implementations differ in terms of algorithms used, resource utilization, and coding efficiency. Figure 3.21 shows the performance results of encoding the 1080p tractor video clip in YUV
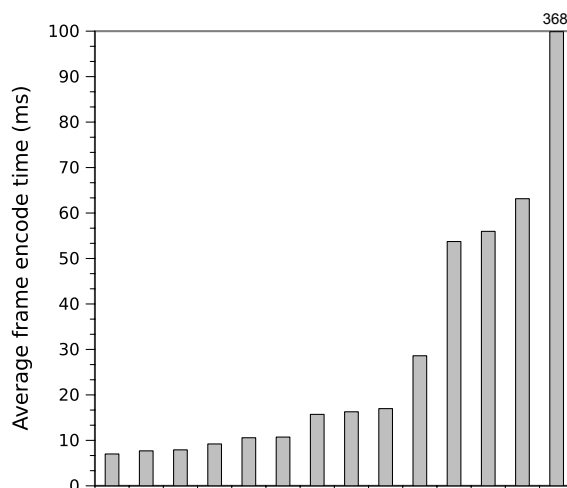
Figure 3.21: Runtime for MJPEG implementations on a GPU (GTX 280).

4:2:0. The difference between the fastest and slowest solutions is 362 ms per frame and the fastest solutions are disk I/O bound. To gain experience in what works and what does not, we examined these solutions. We did not consider coding style, but revisited algorithmic choices, inter-core data communication (memory transfers), and architecture-specific capabilities.

GPU Experiments

A GPU is a dedicated graphics rendering device and modern GPUs have a parallel structure, making them effective for carrying out general-purpose processing. Previously, shaders were used for programming, but specialized languages are now available. In this context, Nvidia released the CUDA framework with a programming language similar to ANSI C. In CUDA, the single instruction, multiple threads (SIMT) abstraction is used to handle thousands of threads.
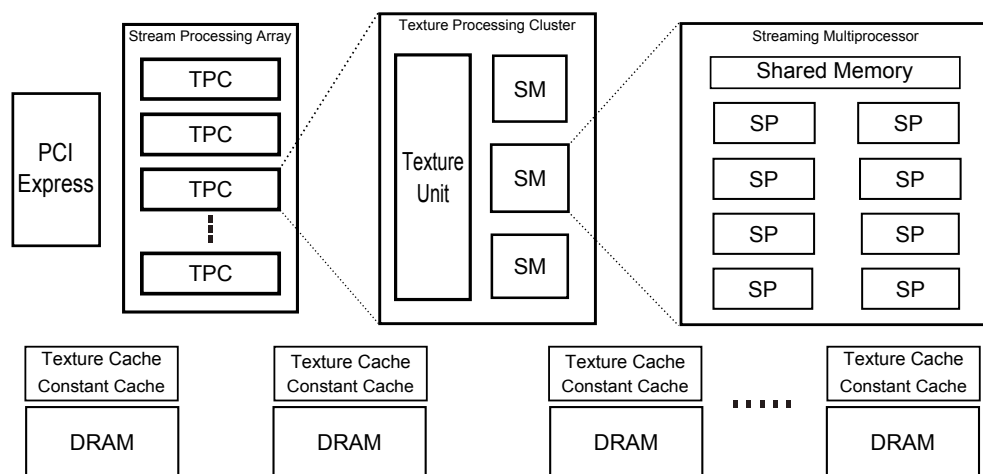
Figure 3.22: Nvidia GT200 architecture.

This case study was carried out with the first generation of programmable GPUs from
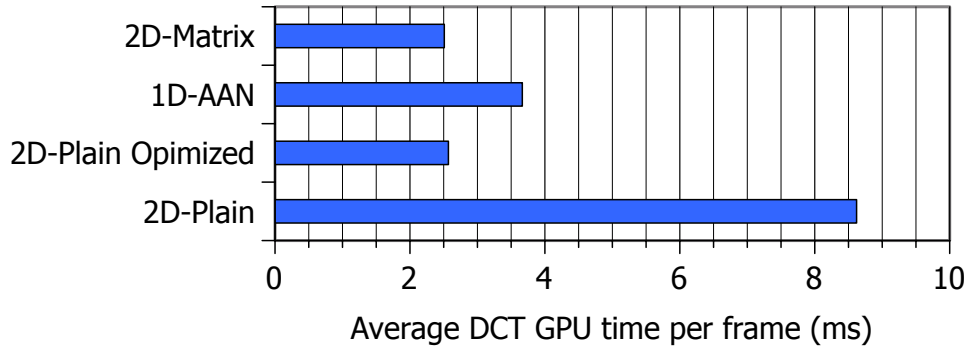
Figure 3.23: DCT performance on a GPU.

Nvidia called Tesla. The chip has the codename GT200, which is the second generation of GPUs in the Tesla architecture, released in 2008. A more detailed overview of the architecture is shown in Figure 3.22. The GT200 chip presents as a highly parallel, multithreaded, multicore processor, connected to the host computer by a PCI Express bus. The GT200 architecture contains 10 texture processing clusters with three streaming multiprocessors (SMs). A single SM contains eight stream processors (called cores in newer GPUs), which are the basic arithmetic and logic units for calculations.

To find out how memory access and other optimizations affect programs such as an MJPEG encoder, we experimented with the same DCT implementations that we used on the x86 architecture in Section 3.2.1. Our baseline DCT algorithm is the *2D-plain* algorithm. The only optimization in this implementation is that the input frames are read into cached texture memory and that the quantization tables are read into cached constant memory. Cached memory spaces improve performance compared to global memory, especially when memory access is uncoalesced. The second implementation, referred to as *2D-plain optimized*, is tuned to run efficiently using principles from the CUDA Best Practices Guide [87]. These optimizations include the use of shared memory as a buffer for pixel values when processing a macroblock, branch avoidance by using Boolean arithmetic, and manual loop unrolling. Our third implementation, the *1D AAN* algorithm, is based on the scalar implementation used on the x86. Every macroblock is processed with eight threads, that is, one thread per row of eight pixels. The input image is stored in cached texture memory and shared memory is used to temporarily store data during processing. Finally, the *2D matrix* DCT uses matrix multiplication, where each matrix element is computed by a thread. The input image is stored in cached texture memory and shared memory is used to store data during calculations.

We know from existing work that to achieve high instruction throughput, branch prevention and the correct use of flow control instructions are important. If threads on the same SM diverge, the paths are serialized, which decreases performance. Loop unrolling is beneficial on GPU kernels and can be done automatically by the compiler using pragma directives. To optimize frame exchange, asynchronous transfers between the host and GPU are used. Transferring data over the PCI Express bus is expensive and asynchronous transfers help us reuse the kernels and hide some of the PCI Express latency by transferring data in the background.

To isolate DCT performance, we used the CUDA Visual Profiler. The profiling results
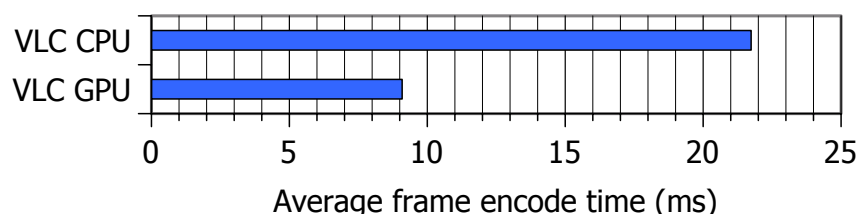
Figure 3.24: Effect of offloading VLC to the GPU.

of the different implementations are shown in Figure 3.23 and we note that the 2D-plain optimized algorithm is faster than the AAN algorithm. The 2D-plain algorithm requires significantly more computations than the others but, by optimizing it to the architecture, we obtain almost as good performance as with the 2D matrix. The AAN algorithm, which does the least number of computations, suffers from the low number of threads per macroblock. A low number of threads per SM can result in stalling, where all the threads are waiting for data from memory, which should be avoided.

The experiment shows that, for architectures with vast computational capabilities, writing a good implementation of an algorithm adapted for the underlying hardware can be as important as the algorithm's theoretical complexity.

The last GPU experiment considers entropy coding on the GPU. VLC can be offloaded to the GPU by assigning a thread to each macroblock in a frame to compress the coefficients and then store the bitstream of each macroblock and its length in global memory. The output of each macroblock's bitstream can then be merged either on the host or by using atomic OR on the GPU. For these experiments, we chose the former, since the host is responsible for the I/O and must traverse the bitstream anyway. Figure 3.24 shows the results of an experiment that compares MJPEG with AAN DCT, with VLC performed on the host and on the GPU, respectively. We doubled the encoding performance when running VLC on the GPU. In this particular case, offloading VLC was faster than running on the host. It is worth noting that by running VLC on the GPU, the entropy coding scales together with the rest of the encoder with the resources available on the GPU. This means than if the encoder runs on a machine with a slower host CPU or a faster GPU, it will still scale.

Discussion

The GPU architecture is different from the x86 architecture used for MJPEG in Section 3.2.1. Some algorithms may be more suited than others. This can clearly be seen in our experiments with DCT, where the AAN algorithm performed best on the x86, but did not achieve the highest throughput on the GPU. This was because of the relatively low number of threads per macroblock for the AAN algorithm, which must perform the 1D DCT operation (one row of pixels within a macroblock) as a single thread. This is only one example of achieving a shorter computation time through increased parallelity at the price of a higher, sub-optimal total number of operations.

Porting the encoder to the GPU in a straightforward manner without significant optimizations for the architecture yields very good offloading performance compared to native x86. This indicates that the GPU is easy to use but, to reap the full potential of the ar-

chitecture, one must have a deep level of understanding.

Summary

We see that heterogeneous multicore architectures provide the resources required for real-
time multimedia processing. However, achieving high performance is not trivial. In
general, there are similarities between the architectures, but the way of thinking must
be substantially different. The different architectures have different capabilities that
must be taken into account, both when choosing a specific algorithm and when mak-
ing implementation-specific decisions. A great deal of trust is put into the compilers of
development frameworks and new languages such as OpenCL, which are supposed to be
a "recompile-only" solution. However, to tune performance, the application must still be
hand optimized for different versions of the GPUs and x86.

3.3.5   Implications

Working with a GPU is different from working with a x86. Compared to the SIMD model,
the SIMT model seems easier for programmers to grasp. With SIMT, programmers have
to think in terms of threaded operations on intrinsic data. However, programming a
GPU has some challenges, as we saw in the case studies. First, as seen in the memory
tests, the GPU has an exclusive memory model and it is important for the programmer
to fully understand the access pattern of the GPU kernel to use the correct memory
layout and memory space. If the data pattern generates too many uncoalesced memory
accesses, it might be an idea to see if it is possible to use one of the cached memory
types. In the host–device optimization experiments, we learned that transfers from the
host CPU to the GPU over the PCI Express bus can be a slow process and it is very
important to try to overlap transfers with computations by using one of the asynchronous
APIs available to the programmers. The cheat detection studies show that computational
workloads such as physics calculations scale very well on a GPU. However, with too few
calculations, the overhead of transferring data from the CPU to the GPU is too large—
hence, the importance of efficient transfers between the host CPU and the GPU. Finally,
the MJPEG experiments on the GPU show that algorithms that are efficient on a CPU,
such as the AAN fast DCT algorithm, might not yield the best performance on a GPU. On
the GPU, the naive 2D DCT algorithm with optimizations for the architecture (with both
branching and shared memory bank conflicts removed and the correct memory layout)
performed better that the AAN fast DCT.

Revisiting with State-of-the-Market Hardware

If we were to revisit these experiments with a state-of-the-market GPU from the Kepler
family, all the experiments would have been able to run; however, to obtain optimal
performance, we would have to revisit some of the optimizations. The memory controllers
on modern GPUs such as the Kepler are more advanced and they will now detect more
access patterns, facilitating coalesced memory accesses. Nevertheless, the selection of
the different memory spaces, such as shared memory and texture memory, still has to
be done by the programmer. Later versions of CUDA also open up the possibility of
what they call managed memory [92] between the host CPU and GPU, that is, page

faults are used to trigger transfers. The most optimal solution is still to manually use the asynchronous API to transfer the data. Modern GPUs now also have larger caches that, as on the x86 architecture, cannot be managed by the programmer. In many cases, this can also speed up the computations. Another challenge of the Kepler architecture is that, since the number of processors per SMX has been increased, compensating for decreasing the clock frequency requires more active threads to obtain the same performance. The numbers of registers per SMX has not been increased, meaning that each thread has less register space, thus making it easier to run out of space. This is not a fatal problem for the applications, since the threads have a private memory space in global memory. However, it will have negative effects on performance. Using the latest generation of GPUs will therefore, in most cases, provide better performance, since the number of cores has increased many times. However, in some cases, performance can decrease and, in many cases, the applications would have the same efficiency as on the older architectures.

3.4   Cell Broadband Engine

The Cell Broadband Engine is one of the asymmetric architectures we used for experiments with an exclusive memory model. The primary application for the Cell was as the main processor in Sony's PlayStation 3 gaming console, so the processor was designed with multimedia workloads in mind. The focus of the Cell experiments was to try to learn how programmers need to think to efficiently utilize the platform.

In the following section, we take a closer look at a case study based on our paper "Tips, Tricks and Troubles: Optimizing for Cell and GPU" [112]. Our analysis of 14 different MJPEG implementations indicates that there exists great potential for optimizing performance with the Cell architecture, but there are also many pitfalls to avoid.

The Cell Broadband Engine was developed by Sony Computer Entertainment, Toshiba, and IBM. The central components in the Cell are a power processing element (PPE) containing a general-purpose 64-bit PowerPC RISC core and eight specialized synergistic processing elements (SPEs). A more detailed overview of the architecture can be found in Section 2.1.4.

3.4.1   Case Study: MJPEG Encoding

In our MJPEG case study, we conducted several experiments on the Cell architecture. A more detailed overview of the MJPEG workload can be found in Section 3.2.1. As for the GPU, several layouts are available for the Cell. However, because of the small number of more capable cores (SPEs), it is feasible to assign one frame to each core. The most time-consuming parts of the MJPEG encoding process, the DCT and quantization steps, are well suited for Cell acceleration. In addition, the VLC step can be adapted.

We also compared 14 different implementations on the Cell. The results indicate that the Cell is also a complex architecture to use and that achieving high performance is not trivial. Figure 3.25 shows performance results for encoding the 1080p tractor video clip in YUV 4:2:0. The difference between the fastest and slowest solution is 1869 ms and the fastest solutions were disk I/O bound. To gain experience of what works and what does not, we examined these solutions using the same criteria as with the GPU implementations. In general, we found that the Cell architecture has great potential, but
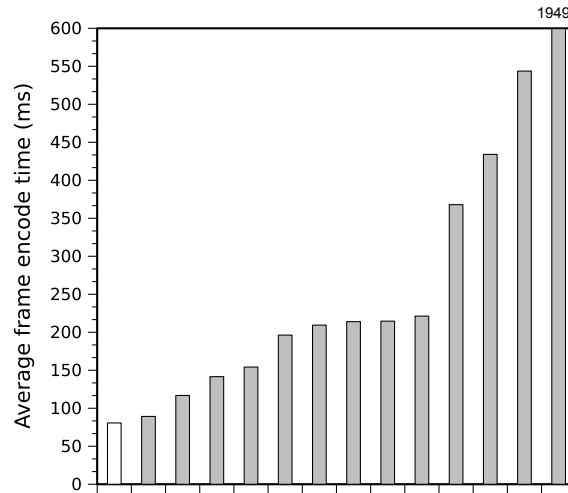
Figure 3.25: Runtime for MJPEG implementations on the Cell on a PlayStation 3 (six SPEs).

also many possible pitfalls, both when choosing specific algorithms and in implementation-specific decisions.

Cell Broadband Engine Experiments

By learning from the design choices of the implementations in Figure 3.25, we designed experiments to investigate how performance improvements are achieved on the Cell. All the experiments encoded HD video (1920x1080, 4:2:0) from raw YUV frames found in the *tractor* video test sequence. However, we used only the first frame of the sequence and encoded it 1000 times in each experiment to overcome the disk I/O limit. This became apparent at the highest level of encoding performance, since we did not have a high-bandwidth video source available. All programs were compiled with the highest level of compiler optimizations using GCC for Cell. The Cell experiments were tested on a QS22 blade server (with eight SPEs; the results in Figure 3.25 were for a PlayStation 3 with six SPEs)

Considering the *embarrassingly parallel* parts of MJPEG video encoding, a number of different layouts are available to map the different steps of the encoding process to the Cell. Because of the amount of work, the DCT and quantization steps should be executed on SPEs, but the entropy coding step can also run in parallel between complete frames. Thus, given that a few frames of encoding delay are acceptable, the approach we consider best is to process full frames on each SPE, with every SPE running DCT and quantization of a full frame. This minimizes synchronization between cores and allows us to perform VLC on the SPEs.

Regardless of the placement of the encoding steps, it is important to avoid idle cores. We resolve this situation by adding a frame queue between the frame reader and the DCT step and another queue between the DCT and VLC steps. Since a frame is processed in full by a single Cell processor, the AAN algorithm is well suited. It can be implemented in a straightforward manner to run on SPEs, with VLC coding on the PPE. We tested the same algorithm optimized with SPE intrinsics for vector processing (SIMD), resulting in
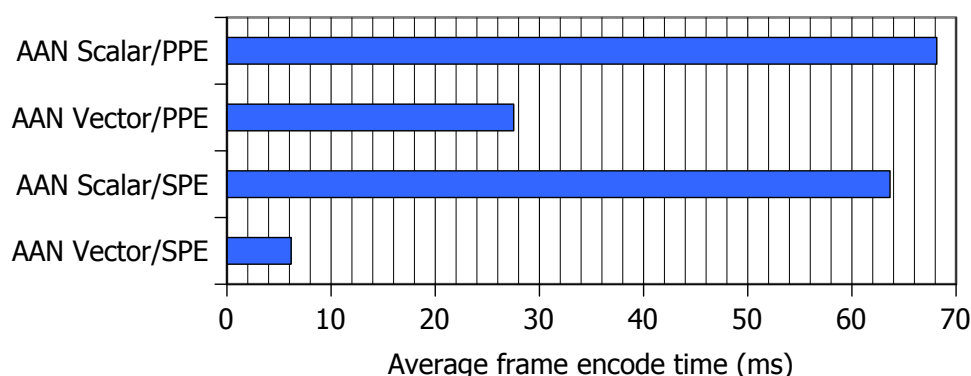
Figure 3.26: Encoding performance on the Cell with different implementations of the AAN algorithm and VLC placement.
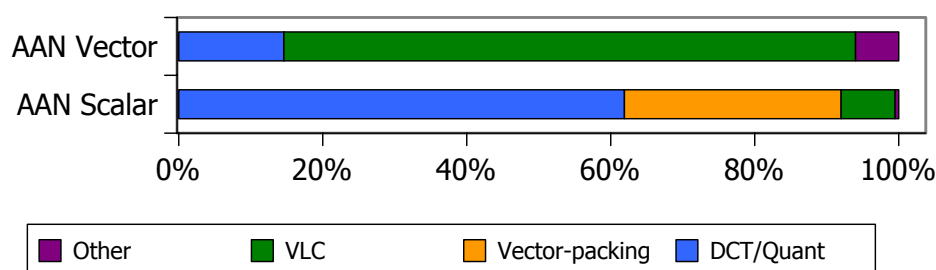
Figure 3.27: SPE utilization using scalar or vector DCT.

double encoding throughput, which can be seen in Figure 3.26 (scalar and vector PPE). Another experiment involved moving the VLC step to the SPEs, offloading the PPE. This approach left the PPE with only the task of reading and writing files to disk, in addition to dispatching jobs to SPEs. To do so, the luma and chroma blocks of the frames had to be transformed and quantized in interleaved order, that is, two rows of luma and a single row of both chroma channels. The results show that the previous encoding speed was limited by the VLC, as shown in Figure 3.26 (scalar and vector SPEs).

To gain some insight into SPE utilization, we collected a trace (using PDTR, part of the IBM software development kit for Cell) showing how much time is spent on the encoding parts. Figure 3.27 shows the SPE utilization when encoding HD frames for the scalar and vector SPEs from Figure 3.26. This distinction is necessary because the compiler does not generate SIMD code, requiring the programmer to hand-code SIMD intrinsics to achieve high throughput. The scalar version uses about four times more SPE time to perform the DCT and quantization steps for a frame than the vector version does and an additional 30% of the total SPE time to pack and unpack scalar data into vectors for SIMD operations. Our vectorized AAN implementation is nearly eight times faster than the scalar version.

With the vector version of DCT and quantization, the VLC coding uses about 80% of each SPE. This can possibly be optimized further, but we did not find time to pursue this.

The Cell experiments demonstrate the necessary level of fine-grained tuning to obtain high performance on this architecture. In particular, correctly implementing an algo-

rithm using vector intrinsics is imperative. Of the 14 implementations for the Cell in Figure 3.25, only one offloaded VLC to the SPEs, but this was the second fastest implementation. The fastest implementation vectorized the DCT and quantization and the vector/SPE implementation in Figure 3.26 is a combination of the two. One reason why only one implementation offloaded the VLC may be that it is unintuitive. An additional communication and shift step is required in parallelizing VLC because the lack of arbitrary bit shifting of large fields on the Cell as well as the GPU prevents a direct port from the sequential codes. Another reason may stem from the dominance of the DCT step in early profiles and the awkward process of later gathering profiling data on multicore systems. The hard part is to know what is best in advance, especially because moving an optimized piece of code from one system to another can be significant work and may even require rewriting the program entirely. It is therefore good practice to structure programs in such a way that parts are loosely coupled. In that way, they can be both replaced and moved to other processors with minimal effort.

When comparing the 14 Cell implementations of the encoder shown in Figure 3.25 to find out what differentiates the fastest from the medium-speed implementations, we found some distinguishing features, the most prominent one being not exploiting the SPE's SIMD capabilities, but also in the areas of memory transfer and job distribution. Uneven workload distribution and lack of proper frame queuing resulted in idle cores. Additionally, some implementations suffered from small, often unconcealed DMA operations that left SPEs in a stalled state, waiting for the memory transfer to complete. It is evident that many pitfalls need to be avoided when writing programs for the Cell architecture and we have only touched upon a few of them. Some of these are obvious, but not all and achieving acceptable performance from a program running on the Cell architecture may require multiple iterations, restructuring, and even rewrites.

Discussion

Heterogeneous architectures such as the Cell provide large amounts of processing power, with encoding throughputs of 480 MB/s on the 1080p tractor video clip. Thus, real-time MJPEG HD encoding may be no problem. However, an analysis of the many implementations of MJPEG available and our additional testing show that it is important to use the right concepts and abstractions and that there may be large differences in the way a programmer must think.

Deciding efficiently the granularity at which data should be partitioned is very hard *a priori*. One approach is to try to design the programs in such a way that the cores are seldom idle or stall. In practice, however, multiple iterations may be necessary to determine the best approach.

Similar to data partitioning, efficient code partitioning is hard to carry out in advance. A rule of thumb is to write modular code to allow the parts to be moved to other cores. In addition, fine granularity is beneficial, since small modules can be merged again and also be executed repeatedly with low overhead. Offloading is by itself advantageous, since resources on the main processor become available for other tasks. It also improves the scalability of the program with new generations of hardware. In our MJPEG implementation on the Cell, we found that offloading DCT/quantization and VLC coding was advantageous in terms of performance, but offloading may not always provide higher

throughput.

Summary

The programming models used on the Cell and the GPU require two different ways of thinking in parallel. The approach of the Cell is very similar to multithreaded programming on the x86, with the exception of shared memory. The SPEs are used as regular cores with explicit caches and the vector units on the SPEs require careful data structure consideration to achieve peak performance. The GPU model of programming is much more rigid, with a static grid used for blocks of threads and synchronization only through barriers. This hides the architecture complexity and provides a simpler concept to grasp for programmers. This notion is also strengthened by the better average GPU throughput of the implementations in Figures 3.21 and 3.25. However, to achieve the highest possible performance, the programmer must also understand the nitty-gritty details of the architecture to avoid pitfalls such as warp divergence and uncoalesced memory access.

Heterogeneous multicore architectures such as the Cell and GPUs may provide the resources required for real-time multimedia processing. However, achieving high performance is not trivial and to learn how to think and use resources efficiently, we experimentally evaluated several issues to discover tricks and problems. Generally, there are similarities, but the way of thinking must be substantially different, not only compared to an x86 architecture but also between the Cell and the GPUs. The different architectures have different capabilities that must be taken into account when both choosing a specific algorithm and making implementation-specific decisions.

The encoding throughput achieved on the two architectures was surprisingly similar. Although, the engineering effort to accomplish this throughput was much greater on the Cell, this was mainly due to the tedious process of writing an SIMD version of the encoder.

3.4.2 Implications

We learned that when working with an architecture such as the Cell, which has multiple vector processors (SIMD), it is imperative to vectorize the application, even though this can be a very tedious process. Our experiments have shown that the code on SPEs has to go through vector packing if it is not vectorized. The exclusive memory architecture of the Cell also provides very good performance, but at the cost of complexity.

With the Cell, it is also very important to choose the correct granularity for the architecture. The SPEs have only 256 kB of local storage, for both code and data, and multiple attempts are often required to find an optimal solution. Our MJPEG workload also showed the importance of finding an algorithm that is optimal for the architecture.

It is very important to consider data movement on the chip itself. Since the Cell has an exclusive memory model, the programmer has to consider the DMA transfers to move data between main memory and the local storage on the SPEs. To obtain optimal performance, often one has to implement double-buffering schemes to make sure that one can overlap memory transfers and computations.

Revisiting with State-of-the-Market Hardware

Unfortunately, IBM discontinued development of the Cell architecture and the CPU in the PlayStation 4 uses x86 cores together with a GPU.

It is possible to use multicore x86 processors instead of the Cell. These architectures use a shared memory model instead of exclusive memory, so it is simpler for the programmer. The x86 cores also have SIMD units. The closest x86 core compared to the Cell is the Intel Xeon Phi many-core processor, with up to 61 simple x86 cores with a shared memory model and 512-bit vector units per core. The shared memory model makes the programmer's life simpler; however, the SIMD version still has to be written mainly by hand and the AltiVec SIMD code from the Cell is not portable, so a new version must be written for the Xeon Phi. Several of these considerations for the Cell are valid for GPUs; however, the architectures are very different, and both the granularity and numbers of threads need to be different.

3.5 Architecture Comparison

In this chapter, we saw multiple case studies on four heterogeneous multicore architectures. These platforms are in many ways very different; however, they all have heterogeneous processing resources. Our experiments have revealed that, in all these architectures, code placement, code partitioning, and data locality have a huge effect on performance.

The Intel IXP was used at the very start of our investigations. It is a shared memory platform (i.e., all the cores can share memory), with cores with different capabilities. As on the GPUs, the IXP has multiple memory types with different properties, such as SRAM for storing packet headers and DRAM for storing packet payloads. However, the IXP differs greatly from the other three architectures. Since the IXP was built to process network traffic, it has very limited floating point support and is limited to manipulating network traffic. The IXP lives on today as a dedicated network flow processing fast fiber links at line speed for applications such as deep packet inspection. Programming the IXP was also challenging, since compilers and documentation were somewhat lacking; however, when these experiments were conducted in 2007, their performance was impressive and most state-of-the-art desktop computers in 2007 were not able to process multiple 1-Gbps network streams at line speed.

The three remaining architectures—the Cell, x86, and GPUs—are more suited for processing multimedia workloads. They all have optimizations for carrying out fast floating point operations. The GPU and Cell are built for floating point operations and the x86 architecture has been constantly extended with better floating point support since the x87 floating point coprocessor was integrated in the 80486 CPUs and since the first vector unit, called MMX, was added to the Pentium CPUs. One property that differentiates the Cell and GPUs with the x86 architecture is support for a shared memory architecture on the x86. This makes parallel programming much easier for the developer. However, it has also proven to be a challenge when scaling the number of threads that share the memory space: With more cores and threads, more traffic is also required on the CPU interconnect to make sure that no parts of the cache are dirty. Another advantage and challenge with x86 cores compared to the Cell and GPUs is that the cores on the x86 are considered fat cores, meaning that they have many features, such as branch prediction,

prefetching from both caches and memory, and multithreading support. This takes up many transistors, making the CPU designs very complex, which is a challenge with regard to power consumption in systems.

The Cell and GPUs have several things in common: They both have an exclusive memory model, where the programmer is responsible for all the allocations. The SIMD programming model on the Cell is the most extreme in this respect, where the programmer must manage the transfer of code and data between main memory and local storage on the SPEs with DMA operations. The local storage is like a user-managed cache and, since each SPE has only 256 kB, it is a very limited resource. On modern GPU architectures, the caches cannot be managed by the programmer. However, the GPU has multiple memory types—on chip, off chip, and cached—which can be used for different parts of the processing. Selecting the correct memory space can sometimes be challenging. Another important factor for both the Cell and the GPU is the efficient transfer of data from main memory on the general-purpose core (the PPE on the Cell and the CPU on the GPU) and into the processing cores. To do this efficiently, asynchronous APIs such as CUDA Streams and double buffering should be used. One important detail of how the Cell and the GPU differ is in the programming model, as we saw in our MJPEG experiments (Sections 3.4.1 and 3.3.4). On the Cell, we use an SIMD model, where the programmer must use vector extensions and adapt the memory layout and algorithms to the use of SIMD vectors and operations. Nvidia uses an abstraction called SIMT on their GPUs. SIMT allows for code that uses only well-known intrinsic types but that can be massively threaded. The functionalities provided by SIMD and SIMT are very similar. However, our experience is that it is much more straightforward to port the program to the GPU and, even without significant optimizations, the GPU architecture yielded very good offloading performance compared to the native x86 architecture. To reap the full potential of the GPU architecture, one must still have a thorough understanding of the architecture, just as with the Cell. The "nail in the coffin" for the Cell architecture is, however, the fact that Sony, Toshiba, and IBM have decided not to continue its development. IBM will instead use GPUs for massively parallel workloads.

With our experiments on simple workloads, we learned that, of the four architectures we experimented with, the GPU and x86 architectures are the most promising. The GPU currently also needs a CPU as a host to run the operating system and manage the data flow. The combination of a x86 processor and a GPU is also a true heterogeneous multicore architecture; that is, the CPU has a few fat cores that are fast with operations that are not very well suited for parallelization and the GPU has many "simple cores" that are very fast at carrying out simple massively parallel operations.

3.6 Summary

In this chapter, we investigated several simple multimedia workloads running on four different heterogeneous architectures. Of the four architectures we evaluated, we are moving forward with the GPU and x86 multicore. The Cell has been discontinued and the IXP network processor is limited to network processing. In the next chapter, we take a closer look at a more complex multimedia workload, with a pipeline with different workloads, executing on a single system in real time, where we have to optimize for both the CPU and the GPU.

Chapter 4

Using Heterogeneous Architectures for Complex Workloads

The previous chapter described our experiments with offloading simple workloads to a heterogeneous architecture. In these experiments, we mainly carried out our optimizations on just the target architecture. Our experiments also showed that good architectural knowledge about the target architecture is essential when optimizing programs. For the x86 architecture, it is important to use threading and to adjust the number of threads to the number of cores available, that is, too many threads will result in reduced performance, and it is also important to use single instruction multiple data (SIMD) units on processors where possible. On graphic processor units (GPUs), we have seen that it is important to use a memory space that is optimized for one's access pattern, as well as to make sure to transfer the data as efficiently as possible to the GPUs and, finally, to try to prevent branching in any code running on the GPU. We decided to focus on x86 processors and GPUs, since the Cell has been discontinued and the IXP network processor is better suited for video processing.

In this chapter, we investigate a more complex workload based on our research on systems for real-time sports analysis [114]. The complex workload is defined in this thesis as a video stitching pipeline, optimized for multiple heterogeneous architectures, which in our case is an x86 processor and a GPU. The workload also has real-time demands, meaning that the pipeline needs to deliver a new video frame every 33 ms to produce video at 30 frames per second (fps).

This chapter is organized as follows: First, in Section 4.1, we introduce our scenario and the non–real-time prototype we implemented. Then, in Section 4.2, we take a closer look at the enhancements and optimizations to make it run in real time on a single machine. The system presented is a large system with many contributors and we focus on architectural optimizations carried out to run the system in real time. We also test how different parameters and heterogeneous architectures affect the performance of the prototype system.

4.1   Bagadus Sports Analysis System

Sports analysis has become a huge industry and a large number of (elite) sports clubs study their athletes' performance, spending great amounts of money. This analysis is

conducted either manually or using one of many analytics tools. In soccer, several systems enable trainers and coaches to analyze the gameplay to improve performance. For instance, at Interplay-sports [62], videostreams are manually analyzed and annotated using a soccer ontology classification scheme. ProZone [100] automates some of the manual annotation process with video analysis software. In particular, it quantifies player movement patterns and characteristics such as athlete speed, velocity, and position and has been successfully used, for example, at Old Trafford in Manchester and Reebok Stadium in Bolton [106] in the United Kingdom. Similarly, STATS' SportVU tracking technology [111] uses videocameras to collect the players' positioning data within the playing field in real time. This information is further compiled into player statistics and performance. Camargus [17] provides a very nice video technology infrastructure but lacks other analytics tools. As an alternative to video analysis, which is often inaccurate and resource hungry, both Cairo's VIS.TRACK [16] and ZXY Sport Tracking [146] systems use global positioning and radio-based systems to capture the performance measurements of athletes. Thus, these systems can present player statistics, including speed profiles, accumulated distances, fatigue, fitness graphs, and coverage maps, in many different ways, such as with charts, three-dimensional graphics, and animations.

To improve game analytics, video that replays real-game events has become increasingly important. However, the integration of player statistics systems and video systems still requires a large amount of manual labor. For example, events tagged by coaches or other human expert annotators must be manually extracted from the videos, often requiring hours of work in front of the computer. Furthermore, connecting the player statistics to the video also requires manual work. One recent example is the Muithu system [67], which integrates coach annotations with related video sequences, but the video must be manually transferred and mapped to the game timeline.

As the above examples show, several tools for soccer analysis exist. However, to the best of our knowledge, no system exists that fully integrates all the features stated above. In this respect, earlier we presented [46] and demonstrated [105] a system called Bagadus. This system integrates a camera array video capture system with the ZXY Sport Tracking system for player statistics and a system for human expert annotation. Bagadus allows the game analytics to automatically play back a tagged game event or extract a video of events from the statistical player data, for example, all sprints at a given speed. Using the exact player positions provided by sensors, a trainer can also follow individuals or groups of players, with the videos presented either by using a stitched panorama view or by switching cameras. Our earlier work [46,105] demonstrated the integrated concept but did not address all operations, such as the generation of the panoramic video, in real time. We now present enhancements providing live, real-time analysis and video playback by using algorithms to enhance image quality, parallel processing, and offloading to heterogeneous architectures units such as GPUs. Our prototype was deployed at Alfheim Stadium (Tromsø IL, Norway) and we use a dataset captured at a Norwegian premier league game for our experiments.

4.1.1   Bagadus: The Basic Idea

Interest in sports analysis systems has increased greatly recently and sports analytics are predicted to be a real game changer, that is, "statistics keep changing the way sports are
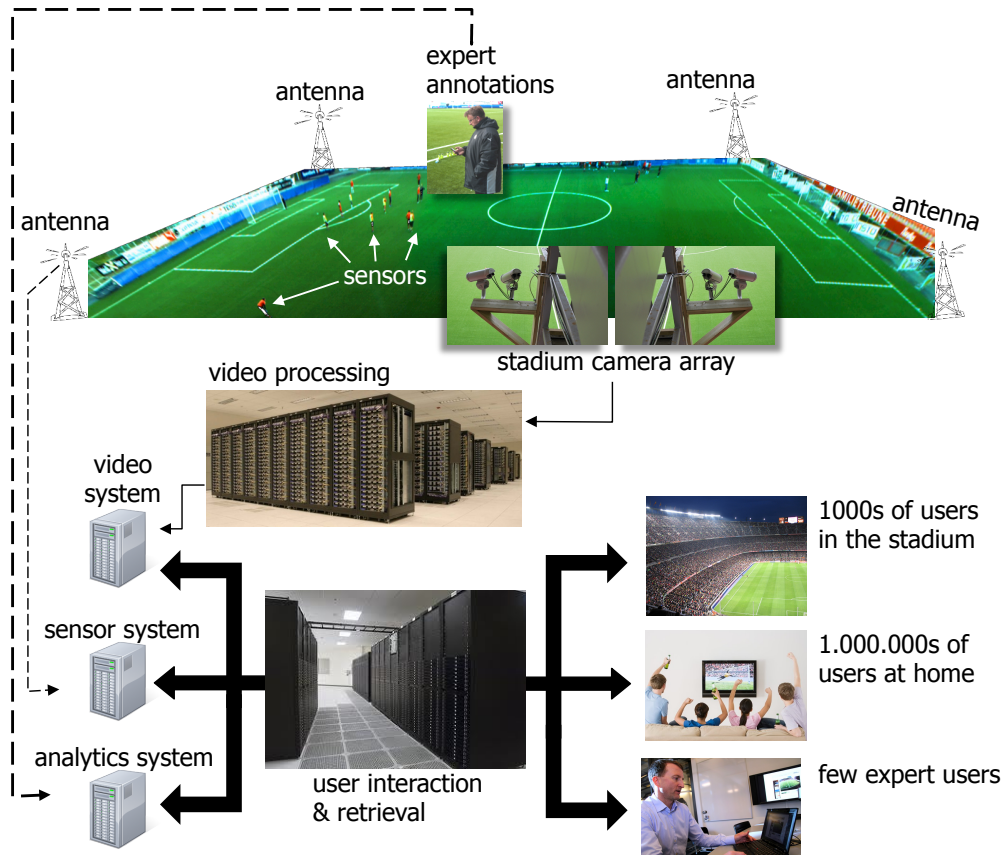
Figure 4.1: Overall sports analysis system architecture.

played—and changing minds in the industry" [29]. As described above, several systems exist, some already providing game statistics, player movements, video highlights, and so forth, since a long time. However, to a large degree, the existing systems are offline systems and require a great deal of manual work to integrate information from various computer systems and expert sport analytics. In this respect, *Bagadus* is a prototype that aims to fully integrate existing systems and enable the real-time presentation of sport events. Our system was built in cooperation with the Tromsø IL soccer club and the ZXY Sport Tracking company. A brief overview of the architecture and interaction of the different components is given in Figure 4.1. The Bagadus system is divided into three different subsystems, all of which are integrated in our soccer analysis application.

The *video* subsystem consists of multiple small shutter-synchronized cameras that record high-resolution video of the soccer field. They cover the full field with sufficient overlap to identify common features necessary for camera calibration and image stitching. Furthermore, the video subsystem supports two different playback options. The first allows playback of video that switches between streams from the different cameras, either by manually selecting a camera or automatically following players based on sensor information. The second option plays back a panorama video stitched from the different camera feeds. The cameras are calibrated in their fixed positions and the captured video is all processed and stored using a capture–debarrel–rotate–stitch–encode–store pipeline. In the offline mode, Bagadus allows a user to zoom in on and mark a player(s) in the

retrieved video on the fly, but this feature is not yet supported in the live mode used during the game.

To identify and follow players on the field, we use a *tracking* (sensor) subsystem. Tracking people through camera arrays has been an active research topic since several years. The accuracy of such systems has improved greatly, but there are still errors. Therefore, for stadium sports, an interesting approach is to use sensors on players to capture their exact positions. ZXY Sport Tracking [146] employs such a sensor-based solution to provide player position information. Bagadus uses this position information to track players or groups of players in single camera views, stitched views, or zoomed-in modes.

The third component of Bagadus is an *analytics* subsystem. Coaches have long analyzed games to improve their own team's gameplay and to understand that of their opponents. Traditionally, this was done by taking notes using pen and paper, either during the game or while watching hours of video. Some clubs even hire one person per player to describe the player's performance. To reduce the manual labor, we implemented a subsystem that equips team members with a tablet (or even a mobile phone) with which they could register predefined events quickly with the press of a button or provide textual annotation. In Bagadus, the registered events are stored in an analytics database and can later be extracted automatically and shown along with a video of the event.

The tracking and analytics subsystem does not have the same processing requirements as the video subsystem does and these are therefore not presented in this thesis. More details about the tracking and analytics subsystem in Bagadus can be found in paper VIII [114].

4.1.2   Video Subsystem

To record high-resolution video of the entire soccer field, we installed a camera array using small industry cameras that together cover the entire field. The video subsystem then extracts, processes, and delivers video events based on given time intervals, player positions, and so forth.

There are two versions of the video subsystem: one non–real-time system, which is presented in this section, and one live real-time system optimized with heterogeneous architectures. This system is presented in Section 4.2.

Both video subsystems support two different playback modes. The first mode allows the user to play video from the individual cameras by manually selecting a camera or by automatically following players. The second mode plays back a panoramic video stitched from the four camera feeds. The non–real-time system plays back recorded video stored on disk and, because of the processing times, the video will not be available before the match is finished. The live system, on the other hand, supports playing back video directly from the cameras and the events are available in real time.

Camera Setup

To record high-resolution video of the entire soccer field, we installed a camera array consisting of four Basler industry cameras with a 1/3-inch image sensor supporting 30 fps and a resolution of 1280×960. The cameras were synchronized by an external trigger

signal to enable a video stitching process that produces a panoramic video picture. For a minimal installation, the cameras are mounted close to the middle line under the roof covering the spectator area, that is, approximately 10 meters from the side line and 10 meters above the ground. With a 3.5 mm wide-angle lens, each camera covered a field of view of about 68 degrees; that is, all four cameras covered the full field with sufficient overlap to identify common features necessary for camera calibration and stitching (see Figure 4.2).
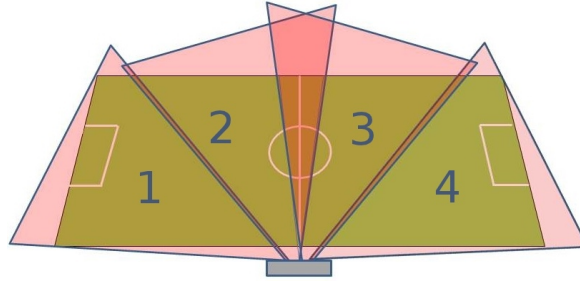
Figure 4.2: Camera setup at Alfheim stadium.

The cameras were managed using our own library, called Northlight, developed for the Verdione project [126], to manage frame synchronization, storage, encoding, and so on. We ran the system on a single computer with an Intel Core i7-3930K at 3.2 GHz, with 16 GB of memory. Northlight integrates the software development kit provided by Basler for the cameras, video encoding using x264, and color space conversion using FFmpeg.

Stitching

Tracking game events over multiple cameras is a nice feature, but in many situations a complete view of the field is desirable. In addition to camera selection functionality, we therefore generated a panoramic picture by combining images from multiple trigger-synchronized cameras. The cameras were calibrated in their fixed positions using a classical chessboard pattern [143], and the stitching operation required a more complex processing pipeline. We used alternative implementations with respect to what to store and process offline, but generally we had to 1) correct the images for lens distortion in the outer parts of the frame due to the fish-eye lens effect, 2) rotate and morph the images into panoramic perspective due to different positions covering different areas of the field, 3) correct the image brightness due to light differences, and 4) stitch the video images into a panoramic image. Figure 4.3 shows the process of combining four warped camera images into a single large panoramic image. The highlighted areas in the figure are the regions of camera overlap.

After the initial steps, the overlapping areas between the frames were used to stitch the four videos into a panoramic picture before storing it to disk. We first tried the open-source solutions given by the computer vision library OpenCV, which are based on the automatic panoramic image stitcher of Brown et al. [14]; that is, we used the auto-stitcher functions using planar, cylindrical, and spherical projections. Our analysis shows that none of the OpenCV implementations are perfect, due to large execution times and varying image quality and resolutions [46]. The fastest algorithm is the spherical projection, but it
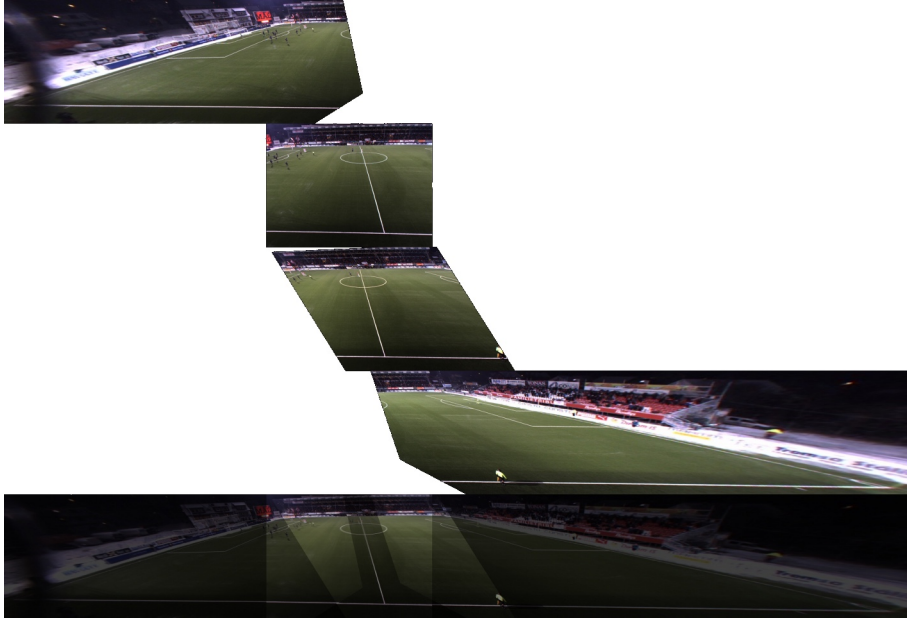
Figure 4.3: The stitching process. Each image from the four different frames is warped and combined into a panorama.

has severe barreling effects and the execution time is 1746 ms per frame, far above our real-time goal. Therefore, a different approach, homography stitching [48], was selected.

Non–Real-Time Processing-Loop Implementation

As a first proof-of-concept prototype [46], we implemented the stitching operation as a single-threaded sequential processing loop, as shown in Figure 4.4, that is, processing one frame per loop iteration.
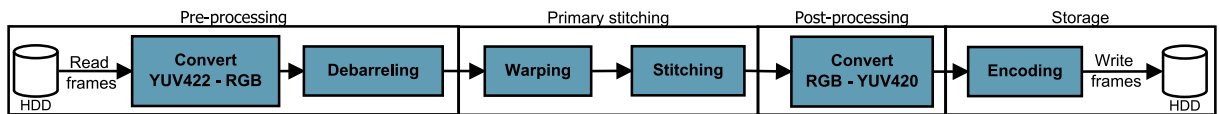
Figure 4.4: The non–real-time Bagadus stitching pipeline.

As seen in the figure, the process consists of four main parts: one pre-processing part, which reads video frames from either disk or the cameras; one part that converts the video from YUV to RGB, which is used by the rest of the pipeline; debarreling, to remove any barrel distortion from the cameras; and primary stitching. This system version used the OpenCV debarreling functions and the primary stitching part used the homography-based stitching algorithm to stitch the four individual camera frames into a 7000×960 panoramic frame. As we can see from Figure 4.5, this last part is the most resource-demanding aspect of the system. After the stitching, post-processing is responsible for converting the video back from RGB to YUV due to the x264 video encoder's lack of support for RGB. The reason for using the RGB color space is that we use OpenCV components, which are written for RGB. The single-threaded loop means that all the

steps are performed sequentially for one set of frames before the next set of frames is processed.
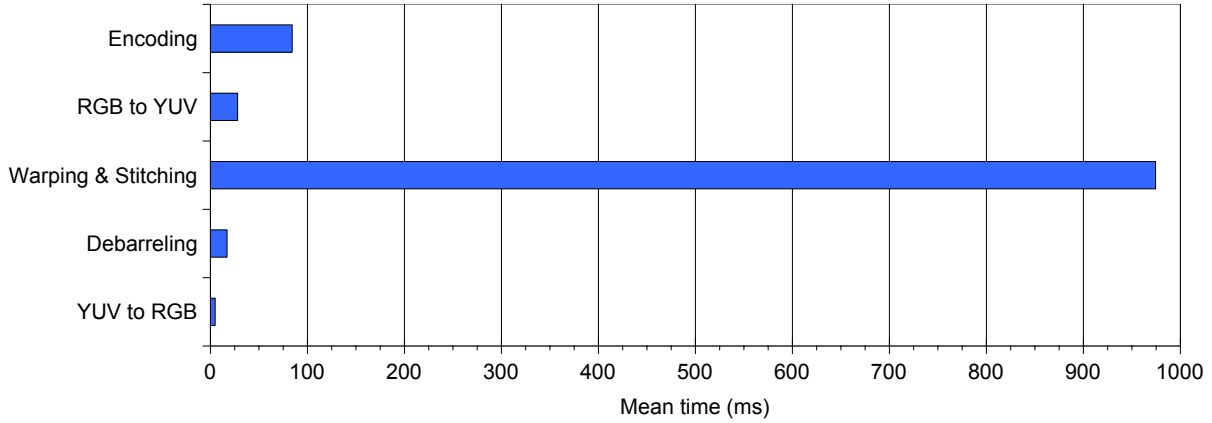
Figure 4.5: Frame processing time in the non–real-time Bagadus stitching pipeline.

The system's performance is presented in Figure 4.5 and the total execution time per panoramic frame exceeds 1100 ms, on average. To meet our 30-fps requirement, our next approach, optimized for heterogeneous architectures and presented in Section 4.2, improves performance by parallelizing and offloading several steps onto a GPU.

4.2   The Real-Time Bagadus Video Pipeline

In this section, we investigate the optimized real-time pipeline shown in Figure 4.6. We analyze the different modules in the pipeline and perform simple benchmark tests to compare the central processing unit (CPU) and GPU performance on the main modules (background subtraction, color correction, stitching, and conversion) running on the GPU. There are two main parts to the real-time pipeline: one part running on the CPU and the other running on a GPU using the CUDA framework. To implement the pipeline's real-time properties, we have to benchmark and load-balance the components with components running on the CPU and others on the GPU. We have to optimize the transfers between the CPU and GPU and try to eliminate any unnecessary transfers.

The experiments in this section were performed on an Intel Core i7-3930K six-core processor with Hyper-Threading enabled, based on the Sandy Bridge-E architecture. The machine had 32 GB of RAM and an Nvidia GeForce GTX 680 GPU based on the Kepler GK104 architecture.

The Controller Module

The single-threaded controller runs on the CPU and is responsible for initializing the pipeline, synchronizing the different modules, handling global errors and dropped frames, and transferring data between the different modules. After initialization, it waits for and receives the next set of frames from the camera reader (CamReader) module (see below). Next, it controls the transfer of data from the output buffers of module N to the input buffers of module $N + 1$. This is done primarily through pointer swapping to avoid
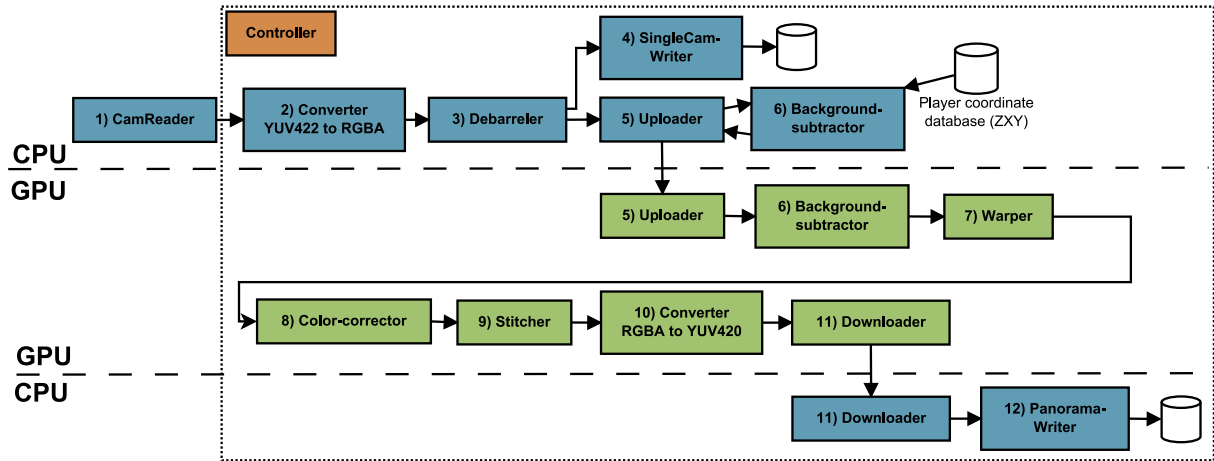
Figure 4.6: The real-time panoramic video stitching pipeline.

unnecessary memory transfers, but with memory copies as an alternative. It then signals all modules to process the new input and waits for them to finish processing. Next, the controller continues looping by waiting for the next set of frames from the reader. Another important task of the controller is to check the execution speed. If an earlier step in the pipeline runs too slowly and one or more frames from the cameras has been lost, the controller will tell the modules in the pipeline to skip the delayed or dropped frame and reuse the previous frame.

The CamReader Module

The CamReader module is responsible for retrieving frames from the Ethernet cameras. It runs on the CPU and consists of one dedicated reader thread per camera. Each of the threads waits for the next frame and then writes the retrieved frame to an output buffer, overwriting the previous frame. The cameras provide a single frame in YUV 4:2:2 format and the CamReader's frame retrieval rate determines the real-time threshold for the rest of the pipeline. As described in Section 4.1.2, camera shutter synchronization is controlled by an external trigger box and, in our current configuration, the cameras deliver a frame rate of 30 fps; that is, the real-time threshold and CamReader processing time are thus 33 ms.

The Converter Module

The CamReader module outputs frames in YUV 4:2:2 format. However, the stitching pipeline requires RGBA internally for processing and the system therefore converts frames from YUV 4:2:2 to RGBA. This process is handled by the Converter module, using *FFm-peg* and *swscale*. The processing time for these conversions on the CPU, as seen later in Figure 4.13, is well below the real-time requirement, so this operation can run as a single thread. Conversion is an embarrassingly parallel operation that can also be carried out efficiently on a GPU; however, the transfer of data to and from the GPU for a single operation would add too much latency to the system.

The Debarreler Module

Due to the wide-angle lenses used in our cameras to capture the entire field, the images delivered suffer from barrel distortion, which needs to be corrected. We found the performance of the existing debarreling implementation in the old stitching pipeline to perform fast enough. The Debarreler module is therefore still based on OpenCV's debarreling function, which is optimized for execution on CPUs with SIMD, using nearest-neighbor interpolation, and executes as a dedicated thread per camera.

The SingleCamWriter Module

In addition to storing the stitched panoramic video, we also want to store the video from the separate cameras. This storage operation is carried out by the SingleCamWriter module, which runs as a dedicated thread per camera. As noted by Halvorsen et al. [46], storing the videos as raw data proves impractical due to the size of the uncompressed raw data. The different CamWriter modules (here SingleCamWriter) therefore encode and compress frames into three-second H.264 files, which has proven to be very efficient. Due to the use of H.264, every SingleCamWriter thread starts by converting from RGBA to YUV 4:2:0, which is the input format required by the x264 encoder. The threads then encode the frames and write the results to disk. There are not many efficient H.264 encoders that can run on GPUs without dedicated hardware encoding blocks; therefore, we did not consider moving this step to the GPU.

The Uploader Module

Due to the great potential of parallelizing the panoramic workload and the high computing power of modern GPUs, large parts of our pipeline run on a GPU. We therefore need to transfer data from the CPU to the GPU, a task performed by the Uploader module. In addition, the Uploader module is also responsible for executing the CPU part of the BackgroundSubtractor module (see Section 4.2). The Uploader module consists of a single CPU thread that first runs the player pixel lookup creation needed by the background subtractor. Next, it transfers the current RGBA frames and the corresponding player pixel maps from the CPU to the GPU. This is done by the use of double buffering and asynchronous transfers (CUDA Streams). We use one stream for each camera and a stream for the pixel maps for the background subtractor.

The BackgroundSubtractor Module

Background subtraction is the process of determining which pixels of a video belong to the foreground and which belong to the background. The BackgroundSubtractor module, running on the GPU, generates a foreground mask (for moving objects such as players) that is later used in the Stitcher module to avoid seams through the players. Our background subtractor can run like traditional systems searching the entire image for foreground objects. However, we can also exploit information gained by the tight integration with the player sensor system. Through the sensor system, we know the player coordinates that can be used to improve both the performance and precision of the module. By first retrieving player coordinates for a frame, we can then create a player pixel lookup map, where we set only the players' pixels, including a safety margin, to one. The

creation of these lookup maps is executed on the CPU as part of the Uploader module. The background subtractor on the GPU then uses this lookup map to process only pixels close to a player, which reduces the GPU kernel processing times from 811,793 ms to 327,576 ms, on average, on a GeForce GTX 680. When run in a pipelined fashion, the processing delay caused by the lookup map creation is also eliminated. The sensor system coordinates are retrieved by a dedicated slave thread that continuously polls the sensor system database for new samples.
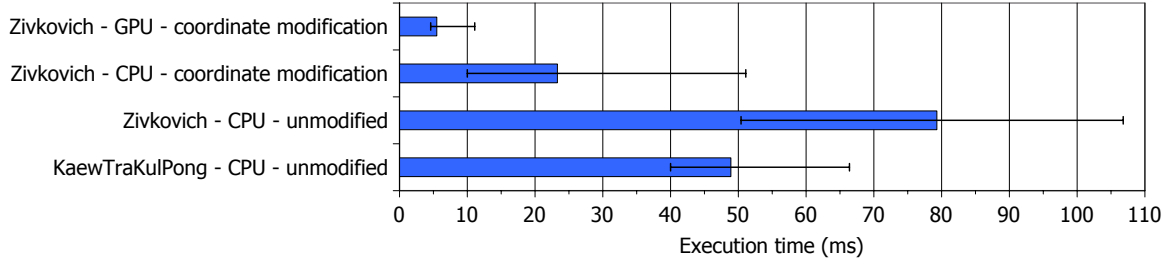
Figure 4.7: Execution time of alternative algorithms for the BackgroundSubtractor module (single camera stream).

Even though we enhanced the background subtraction with sensor data input, there are several implementation alternatives. When determining which algorithm to implement, we evaluated two alternatives: Zivkovic's [144,145] and Kaewtrakulpong and Bowden's [68]. Even though the CPU implementation was slower (see Figure 4.7), Zivkovic's method provided the best visual results and was therefore selected for further modification. Furthermore, the Zivkovic algorithm proved to be fast enough when modified with input from the sensor system data. The GPU implementation, based on Zivkovic's [98], proved to be even faster and the final performance numbers for a single camera stream are shown in Figure 4.7. A visual comparison of the unmodified Zivkovic implementation and the sensor system-modified version is shown in Figure 4.8, where the sensor coordinate modification reduces noise, as seen in the upper parts of the figures.

(a) Unmodified Zivkovic approach.

(b) Player sensor data modification of Zivkovic's approach.

Figure 4.8: Background subtraction comparison.

The Warper Module

The Warper module is responsible for warping the camera frames to fit the stitched panorama image. By warping we mean twisting, rotating, and skewing the images to fit the common panoramic plane. As we saw from the old pipeline, this is necessary because the stitcher assumes that its input images are perfectly warped and aligned to be stitched to a large panorama. Executing on the GPU, the Warper also warps the foreground masks provided by the BackgroundSubtractor module. This is because the Stitcher module will later use the masks and therefore expects them to fit perfectly to the corresponding warped camera frames. Here, we use the Nvidia Performance Primitives (NPP) library [89] for optimized implementation.

The Color-Corrector Module

When recording frames from several different cameras pointing in different directions, it is nearly impossible to calibrate the cameras to output the exact same colors due to the different lighting conditions. This means that, to generate the best panoramic videos, we need to correct the colors of all the frames to remove disparities. In our panorama pipeline, this is done by the Color corrector module running on the GPU.
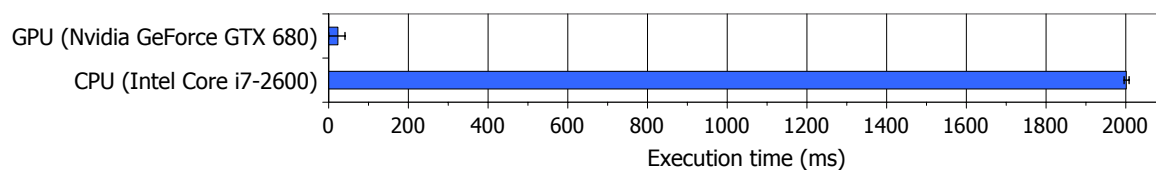
Figure 4.9: Execution time of color correction.

We choose to carry out the color correction after warping the images. The reason for this is that locating the overlapping regions is easier with aligned images and the overlap is also needed when stitching the images together. This algorithm is executed on the GPU, enabling fast color correction within our pipeline. The implementation is based on the algorithm presented by Xiong and Pulli [136], but with minor modifications to optimize for the GPU. We calculate the color differences between the images for every single set of frames delivered from the cameras. We color-corrected each image in sequence, meaning that each image was corrected according to the overlapping frame to the left. The algorithm implemented is easy to parallelize and does not use pixel-to-pixel mapping, which makes it well suited for our scenario. Figure 4.9 compares running the algorithm on the CPU and on a GPU. The CPU version could be further optimized with SIMD; however, the GPU implementation would still be much faster.

The Stitcher Module

As in the old non–real-time pipeline, we use a homography-based stitcher where we simply create seams between the overlapping camera frames and then copy pixels from the images based on these seams. These frames need to follow the same homography, which is why they have to be warped two steps back in the pipeline. In our old pipeline, we used static cuts for seams, which meant that a fixed rectangular area from each frame was copied

directly to the output frame. Static cut panoramas are faster but can introduce graphical errors in the seam area, especially when there is movement in the scene, as illustrated in Figure 4.10(a).

(a) The original fixed cut stitching with a straight vertical seam.

(b) The new dynamic stitching with color correction.
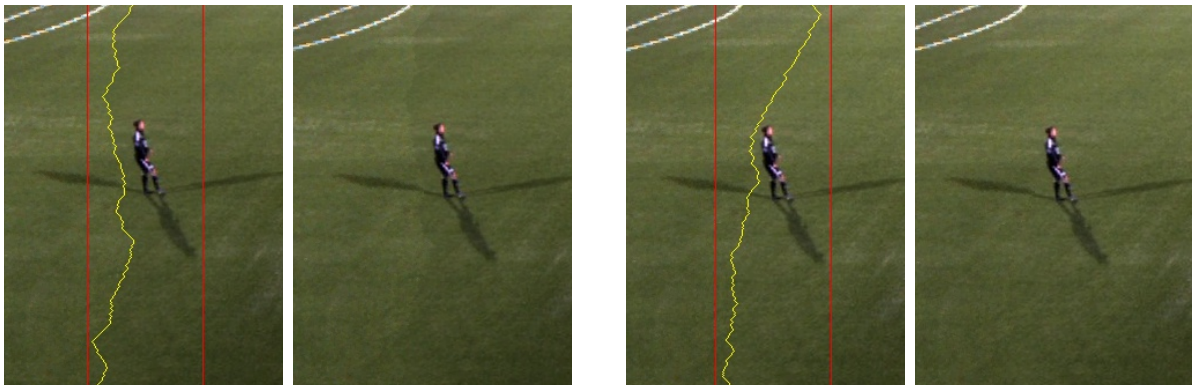
(c) Dynamic stitching with **no** color correction. The left image shows the seam search area between the red lines and the seam in yellow. The right image clearly shows the seam going outside the player, but there are still color differences.

(d) Dynamic stitching **with** color correction. The left image shows the seam search area between the red lines and the seam in yellow. The right image has no visible seam and no color differences.

Figure 4.10: Stitcher comparison, improving the visual quality with dynamic seams and color correction.

To make a seam with a better visual result, we therefore introduced a dynamic cut stitcher instead of the old static cut. The dynamic cut stitcher creates seams by first creating a rectangle of adjustable width over the static seam area. Then, it treats all pixels within the seam area as graph nodes. The graph is directed from the bottom to the top in such a way that each pixel points to the three adjacent pixels above (the left- and right-most pixels only point to the two pixels available). These edges' weights are calculated by using a custom function that compares the absolute color differences between the corresponding pixels in each of the two frames we are trying to stitch. The weight function also checks the foreground masks from the BGS module to see if any player is in the pixel and, if so, it adds a large weight to the node. In effect, both these

steps create edges between nodes where the colors differ and the players present have much larger weights. We then run the Dijkstra graph algorithm [28] on the graph to create a minimal cost route from the start of the offset at the bottom of the image to the end at the top. Since our path is directed upward, we can only move up or diagonally from each node and we only obtain one node per horizontal position. By looping through the path, we therefore obtain our new cut offsets by adding the node's horizontal position to the base offset. An illustration of how the final seam looks is shown in Figure 4.10(b), where the seams without and with color correction are shown in Figures 4.10(c) and 4.10(d).
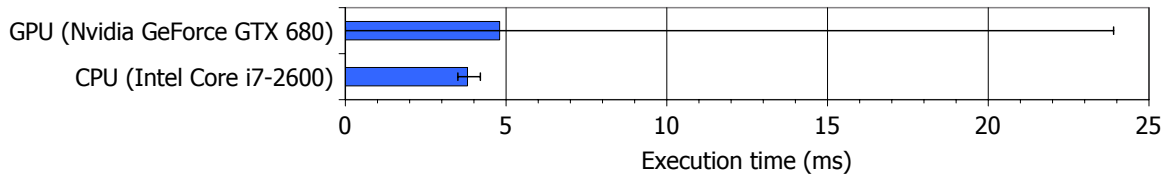
Figure 4.11: Execution time for dynamic stitching.

The timings for the dynamic stitching module are shown in Figure 4.11. The CPU version is currently slightly faster than our GPU version, since this algorithm is more serial than the other image processing algorithms. Searches and branches are also more efficient on traditional CPUs, but further optimization of the CUDA code could improve this GPU performance; however, this is not needed, since we are well within the real-time requirements. The performance difference between the GPU and CPU versions is also not large enough to justify moving the module to the CPU, which would also add delay by transferring the data over the PCI Express bus.

The YuvConverter Module

Before storing the stitched panoramic frames, we need to convert back from RGBA to YUV 4:2:0 for the H.264 encoder, just as in the SingleCamWriter module. However, due to the size of the output panorama, this conversion is not fast enough on the CPU, even with the highly optimized *swscale* library that uses SIMD. This module is therefore implemented on the GPU. For the GPU version, we based the module on a function from Nvidia's NPP [89]. The NPP contains several conversion primitives, but no direct conversion from RGBA to YUV 4:2:0. The GPU-based version therefore first uses the NPP to convert from RGBA to YUV 4:4:4 and we wrote a small CUDA kernel to carry out the final conversion from YUV 4:4:4 to YUV 4:2:0.
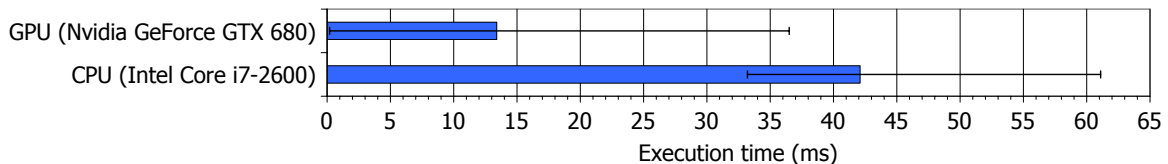
Figure 4.12: Execution time for conversion from RGBA to YUV 4:2:0.

Figure 4.12 compares the performance of the CPU-based implementation with that of the optimized GPU-based version. These results show that the GPU version, even with a two-step conversion, is over twice as fast as the CPU version.
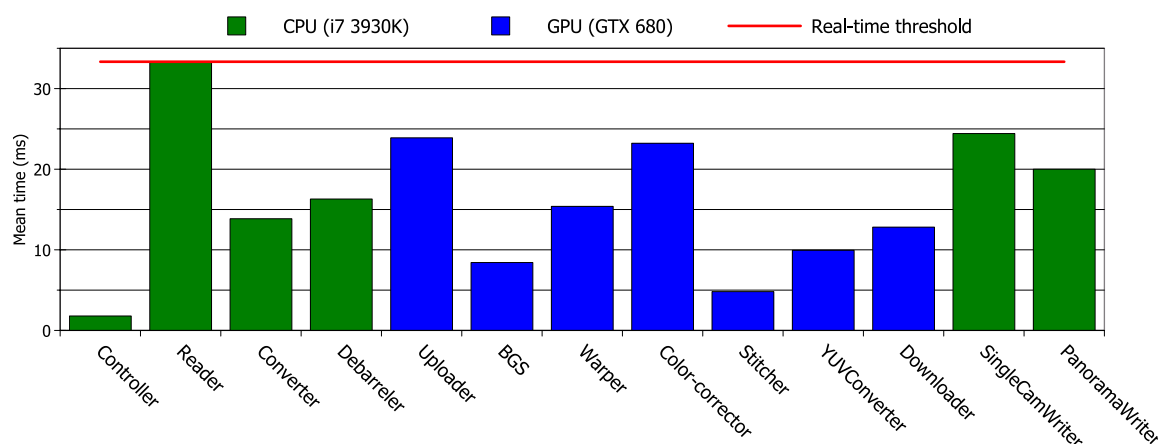
Figure 4.13: Improved real-time pipeline performance: module overview with default setup.

The Downloader Module

Before we can write the stitched panorama frames to disk, we need to transfer them back to the CPU, which is carried out by the Downloader module. It runs as a single CPU thread that synchronously copies a frame to the CPU. We could have implemented the Downloader module as an asynchronous transfer with double buffering, like the Uploader, but since the performance, as shown in Figure 4.13, is very good, this is left as a further optimization.

The PanoramaWriter Module

The last module, executing on the CPU, is the Writer module, which writes the panoramic frames to disk. The conversion from RGBA to YUV was already done on the GPU, so the only steps the PanoramaWriter needs to follow are to first encode the input frame to H.264 and then write the result to disk as three-second H.264 video files.

4.2.1　Performance Analysis

To evaluate the performance of our pipeline, we used an off-the-shelf PC with an Intel Core i7-3930K processor and an Nvidia GeForce GTX 680 GPU. We benchmarked each individual component and the pipeline as a whole, capturing, processing, and storing 1000 frames from the cameras.

In the non–real-time pipeline [46], the main bottleneck was panorama creation (warping and stitching). This operation alone used *974 ms per frame.* As shown by the performance breakdown into individual components in Figure 4.13, the new pipeline was greatly improved. Note that all the individual components run concurrently in real time on the same set of hardware. All of these, however, add up to times far longer than 33 ms. The reason why the pipeline still runs in real time is because several frames are processed in parallel. Note that all CUDA kernels are executed at the same time on a single GPU, so the performance of the GPU modules is affected by that of the other GPU modules. On earlier GPUs from the Tesla architecture (e.g., the GTX 280), where different CUDA
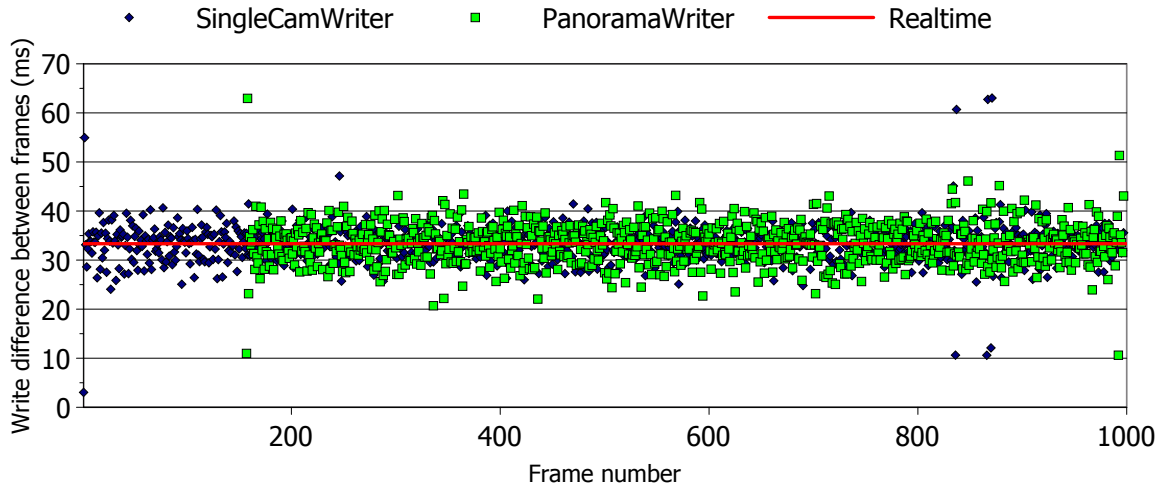
(a) Pipeline write differences (showing the times for 1000 frames).
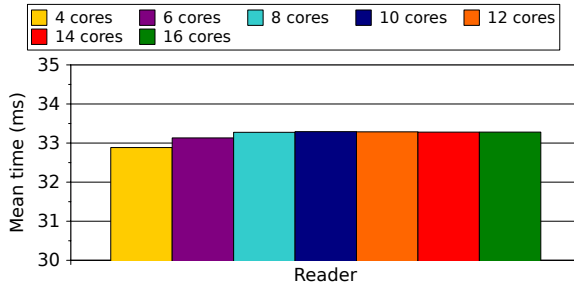
(b) Core count scalability.
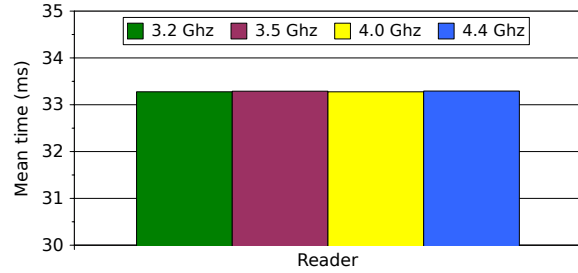
(c) Core frequency scalability.

Figure 4.14: Inter-departure times of frames when running the entire pipeline. In a real-time scenario, the output rate should follow the input rate (given here by the trigger box) of 30 fps (33 ms).

kernels were serialized, this was not possible. However, the Fermi architecture (GTX 480 and above) introduced concurrent CUDA kernel execution [88]. Thus, since the Controller module schedules the other modules according to an input rate of 30 fps, the resources are sufficient for real-time execution.

For the pipeline to function in real time, the output rate should follow the input rate, that is, deliver all output frames (both for four single cameras and for one panorama) at 30 fps. Thus, to give an idea of how often a frame is written to file, Figure 4.14 shows individual and average frame inter-departure rates. The figures show the time differences between consecutive writes for the generated panorama, as well as for the individual camera streams. Operating system calls, interrupts, and disk accesses most likely cause small spikes in the write times (as seen in the scatter plot in Figure 4.14(a), but as long as the average times are equal to the real-time threshold, the pipeline can be considered to run in real time. As shown in Figures 4.14(b) and 4.14(c), the average frame inter-arrival time (Reader) is equal to the average frame inter-departure time (both SingleCamWriter and PanoramaWriter). This is also the case when testing other CPU frequencies and numbers of available cores. Thus, the pipeline runs in real time.

As stated above and seen in Figure 4.14(a), there is a small latency in the panorama

pipeline compared to writing the single cameras immediately. The 33 ms are due to the camera frame rate of 30 fps, meaning that even though a module may finish before the threshold time, the Controller module will make it wait until the next set of frames arrives before signaling it to re-execute.

We added a five-second input buffer to the pipeline, because the sensor system has a latency of at least three seconds before the data are ready for use, and we added a two-second buffer for safety and GPU processing. This means that the end-to-end time from when a picture is recorded by the camera until it is stored on disk is 5.33 seconds per frame, on average.

4.2.2   Discussion

The first non–real-time prototype aimed at full integration at the system level, rather than optimization for performance. However, the challenge with the real-time pipeline has been increased by aiming at running the system in real time on low-cost, off-the-shelf hardware.

The new real-time capability also enables future enhancements with respect to functionality. For example, several systems have already demonstrated their ability to serve available panoramic video to the masses [53, 83] and, by generating the panoramic video live, enables the audience to mark and follow particular players and events. We can also use this information to create video playlists [66] automatically, providing a video summary of extracted events.
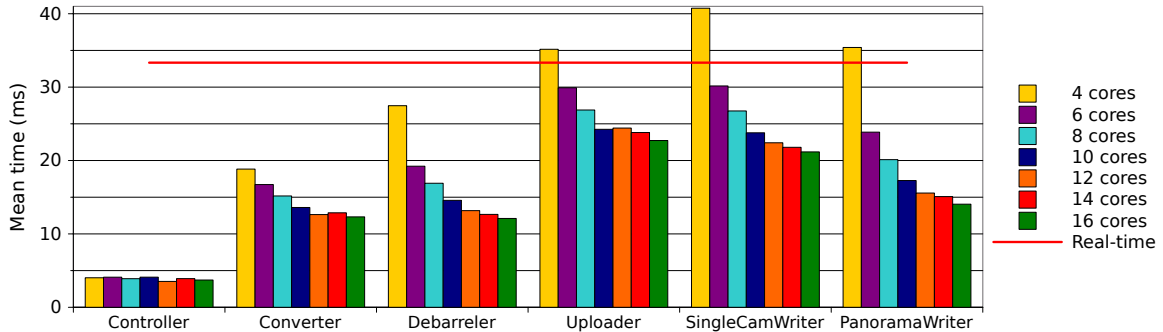
Figure 4.15: Core count scalability.

Due to limited availability of resources during these experiments, we were not able to test our system with more cameras or with higher-resolution cameras. However, to still obtain an impression of the scalability capabilities of our pipeline, we performed several benchmark tests, changing the number of available cores and, the processor clock frequency, and experimented with GPUs from different architectures and with different computing resources.

Figure 4.15[1] shows the results changing the number of available cores that can process the many concurrent threads in the CPU part of the pipeline (Figure 4.14(b) shows that the pipeline is still in real time). As we can observe from the figure, every component

---

[1]Note that this experiment was run on a machine with more available cores (16), each at a lower clock frequency (2.0 GHz), compared to the machine installed at the stadium, which was used for all the other tests.

runs in real time, using more than four cores and the pipeline as a whole using eight or more cores. Furthermore, the CPU pipeline contains a large but configurable number of threads (86 in the current setup) and, due to the many threads of the embarrassingly parallel workload, the pipeline seems to scale well with the number of available cores.
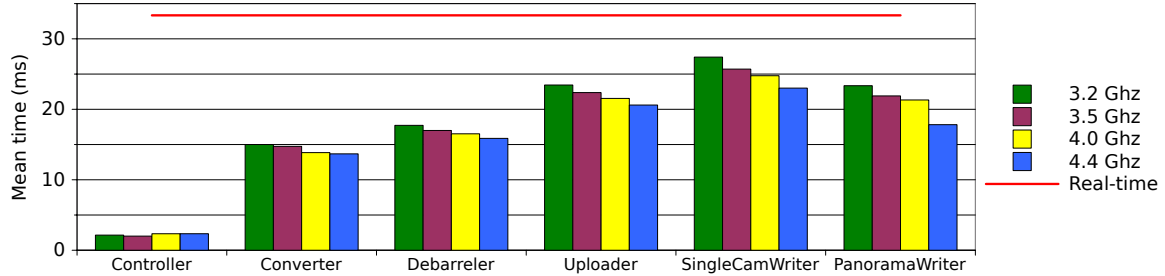
Figure 4.16: CPU frequency scalability.

Similar conclusions can be drawn from Figure 4.16, where the processing time is reduced with a higher processor clock frequency; that is, the pipeline already runs in real time at 3.2 GHz and scaling is almost linear with CPU frequency (Figure 4.14(c) shows that the pipeline is still in real time). The H.264 encoder scales especially well when scaling the CPU frequency.
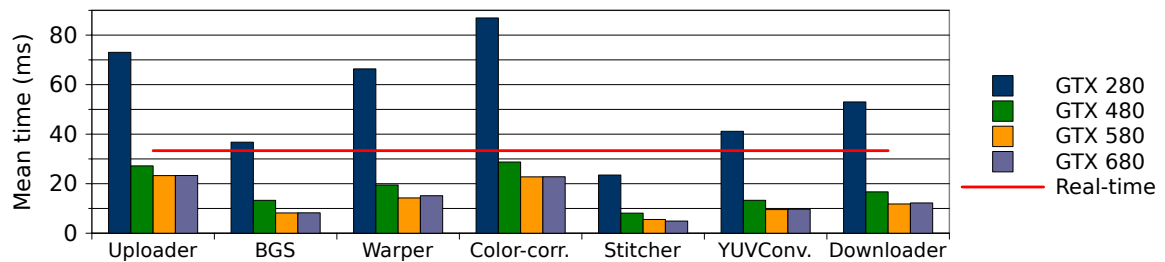
Figure 4.17: GPU comparison.

With respect to the GPU part of the pipeline, Figure 4.17 plots the processing times using different GPUs. The high-end GPUs GTX 480 and above (based on the Fermi architecture) all achieve real-time performance in the current setup. The GTX 280, based on the Tesla architecture, does not support the concurrent CUDA kernel execution introduced with the Fermi architecture [88] and performance is therefore lower than in real time, since the kernels executing on the GPU must be serialized. As expected, more powerful GPUs reduce the processing time. This is shown for the GTX 580 results. The GTX 580 is based on the same Fermi architecture as the GTX 480; however, it has 512 cores versus 480 cores and the cores clocked at a higher frequency. We can also see that the GTX 680 GPU, which is based on the newer Kepler architecture, performs better than the GTX 580 in some cases and has slightly lower performance in other cases. This is due to the fact that we developed and optimized this pipeline for the Fermi architecture, not the Kepler architecture.

Looking at the bandwidth used on the PCI Express bus between the CPU and GPU, we only use a small portion of the pipeline. Our Uploader module takes 737 MB/s and the downloader takes 291 MB/s. The theoretical bandwidth of a 16-lane PCI Express 3.0 link

is 16 GB/s. For now, one GPU fulfills our real-time requirement. We did therefore not experiment with multiple GPUs, but the GPU processing power can easily be increased by adding multiple cards. Profiling of the modules running on the GPU showed that the pipeline uses seven kernels running concurrently on the GPU. These seven kernels have an average compute utilization of 14.8% on the latest-generation GPU based on the Kepler GK110 architecture. Thus, based on these results, we believe that our pipeline can be scaled up to both higher numbers of cameras and higher-resolution cameras.

4.3   Summary

Where people earlier used a huge amount of time to analyze games manually, Bagadus is an integrated system that automatically manages the required operations and video synchronization. For example, in online mode, Bagadus receives expert-annotated events from the analytics team and enables immediate playback during a game or a practice session. To enable this feature, we distributed the workload of the video subsystem on both the CPU and GPU. Our experiments show that the pipeline can run in real time on a low-cost six-core machine with a commodity GPU. To achieve this, each component in the pipeline was optimized for the target architecture, both as a standalone component and as a part of a pipeline. We had to carefully consider which of the components we wanted to run on the GPU and the CPU. Some of the components needed input from the network, some needed to write to storage, and some components needed input from other parts of the system, such as the tracking subsystem. This meant that modules such as the camera readers and writers had to run on the CPU.

An important step in the optimizations was the benchmarking of all the separate modules together as a complete pipeline. One of the lessons we learned was that, even though the standalone module is faster on the CPU, we had to keep this part of the pipeline on the GPU, since the next module in the pipeline had to be executed on the GPU. The CPU version of this module could not meet our real-time requirements. Moving only the dynamic stitcher module to the CPU for processing would also add extra latency to the system because of the extra transfer over the PCI Express bus. The same lesson also goes the other way: In some cases (i.e., with the Debarreling module) the workload might be very well suited for offloading to the GPU; however, the next step in the pipeline would require processing on the CPU.

Another lesson learned during our experiments was that the real-time pipeline uses seven kernels running concurrently on the GPU and all seven kernels combined had a average compute utilization of just 14.8% on the GPU. This means that we could potentially share the GPU with other workloads. When programming a complex workload such as the video subsystem of the Bagadus system, we had to have detailed knowledge about both the GPU and CPU architectures in our target system. Several iterations of optimizations are often required to make a complex workload such as the Bagadus video pipeline run in real time. This process involves much trial and error. If we were to change the hardware used for our system, which components to run on the GPU and on the CPU might have to be redetermined.

To make this process easier, we therefore need a framework that is aware of an application's real-time requirements so that, with the help of instrumented runs of the application, the framework can move resources between the GPU and CPU for optimal

execution. Ideally, we would just want the programmer to express the maximum level of parallelism in the program code and have the framework generate multiple versions for any heterogeneous architectures that are supported. The framework would also be able to dynamically adapt the pipeline when the hardware resources changed. A processing framework would also be able to utilize the resources on the GPU more efficiently. Our experiments show that the average compute utilization of the GPU when executing all our seven kernels was 14.8%. If the real-time requirements were met, the framework would be able to schedule a greater workload on the GPU.

The Bagadus prototype presented here supports four 1K cameras and runs on a single computer. The next-generation camera setup use five 2K cameras. To support this setup, the Bagadus video pipeline was extended to run on multiple machines. This was implemented after the completion of this thesis. The latest version of the Bagadus pipeline [40,80,97] runs in real time on multiple machines connected with a PCI Express-based interconnect.

In the next chapter, we introduce a programming language and framework designed for the real-time processing of multimedia workloads on heterogeneous architectures. This system is designed to enable complex workloads such as Bagadus to run in real time on distributed systems with heterogeneous architectures without manually tweaking the system.

Chapter 5

The P2G Framework and the Future

In the previous chapter, we saw that complex multimedia workloads with real-time constraints are well suited for heterogeneous architectures. However, we observed that many optimizations were required to make sure that the processing time of all the components was less than the real-time threshold and that the workloads were executed on the optimal architecture for the particular workload or task.

In this chapter, we take a take a closer look at a framework called P2G that we designed for the real-time processing of multimedia workloads on heterogeneous architectures. The goal of this framework is to automate the parallelization of the program code and provide programmers a unified programming abstraction for writing multimedia workloads for heterogeneous architectures. This framework is a work in progress. At the conclusion of this thesis, a simple prototype was running on a single multicore machine with a shared memory architecture. We used two simple multimedia workloads to test the feasibility of our system.

This chapter is organized as follows: First, in Section 5.1, we summarize some of the challenges we observed in previous chapters. In Section 5.2, we present ideas for designing the framework and take a closer look at other frameworks for distributed processing. In Section 5.3, we present related work on other processing frameworks. Next, in Section 5.4, we present the architecture and programming model of our P2G framework and evaluate two simple multimedia workloads with the prototype implementation. Finally, in Section 5.5, we discuss the future vision of our framework.

5.1 Summary of Challenges

In Chapters 3 and 4, we experimented with heterogeneous architectures for simple and complex multimedia workloads. We observed that the architectures can efficiently process the workloads. However, challenges remain, especially for the programmer.

We saw that many of our simple workloads had to be optimized for the architectures to run efficiently. On the x86 and the Cell, single instruction multiple data (SIMD) programming is recommended to process more data per cycle. We also saw the importance, on the x86 architecture, of adapting the number of threads used by the workload to the number of cores in the system. If too many threads are used, performance suffers because of scheduling overhead. If the workload easily scales too many threads, another possibility is to use a graphics processing unit (GPU). When using a GPU, the programmer has to

be aware that it takes time to transfer data to and from the GPU, so the workload has to be large enough to compensate for the transfer overhead. There are also several potential pitfalls of not using the memory on the GPU correctly and not optimizing the transfers from the host central processing unit (CPU) to the GPU.

Finally, when programmers want to process more complex multimedia workloads with real-time constraints that execute on multiple heterogeneous architectures, a great deal of profiling, tweaking, and optimization is required to make sure that the right components run on the right architecture.

5.2   Design Ideas for a New Processing Framework

Our work with both simple and complex multimedia workloads led to several observations and challenges with optimizing parts of programs and entire pipelines for different heterogeneous architectures. We want to use some of these ideas when designing our framework for processing multimedia workloads on heterogeneous architectures.

One of the first observations when working with the x86, Cell, and GPUs was that programmers tend to prefer the single instruction, multiple threads (SIMT) abstraction used on GPUs when they have small kernels that execute on the same data over the more rigid SIMD abstraction found on the x86 and the Cell. The main advantage with SIMT is that the programmer can think in terms of scalar code when writing programs. All the architectures we worked with use a C-like programming language, so we want to keep C as the programming language when designing the framework.

Furthermore, moving data between the different architectures is a challenge for many programmers. It is therefore important to design the framework in such a way that it takes care of data transfers. We also want to present the data as arrays to programmers, because this is a familiar data representation when working with multimedia workloads. For scheduling multimedia workloads, we want to use the two-level scheduling that we experimented with on a very simple scheduling simulator [115]. Here, a high-level scheduler (HLS) has global control of all the resources available and a low-level scheduler executes the workloads and time slicing on the different processing cores.

It is important for the framework to track data dependencies in the pipelines. If the dependency tracker is efficient and fine grained, the framework will be able to expose both task parallelism and pipeline parallelism in the programs execution. Dependency tracking is also important for the framework when moving data between different processing cores. To efficiently carry out fine-grained dependency tracking, programmers must write their applications in such way that they express as much parallelism as possible.

5.3   Existing Processing Frameworks

A great deal of research aims at solving challenges with parallel and distributed processing of large quantities of data. Two of the most popular frameworks for distributed processing are Google's MapReduce [27] and Microsoft's Dryad [63]. In addition, we have System S from IBM [41], PigLatin from Yahoo [93], Cosmos [85], Scope [18], and DryadLINQ [140].

MapReduce uses a data-parallel model and is based on keys and values. There are several implementations of MapReduce for multicore processors [103], clusters [4], GPUs [49],

and even the Cell Broadband Engine [26]. To process data relations among heterogeneous data more efficiently, which is not supported by the original MapReduce model, Map–Reduce–Merge [138] was introduced. The Oivos project [124] addresses the same issues but, in addition, the system provides a more expressive, declarative programming model. Reducing the layering overhead of the software running on the top of MapReduce is the goal of Cogset [125], where the architecture of the processing is changed to increase performance.

The Dryad, Cosmos, and System S frameworks have several properties in common. All three use directed graphs to model communication between the processing stages and execute them on a cluster. System S also supports cycles in the graphs. However, since all of these systems are closed source, many details are unknown. Compared to the data-parallel MapReduce, which is one of the most cited paradigms for expressing parallel workloads, both Dryad and System S use a task-parallel model.

A limitation of MapReduce, Dryad, and Cosmos is their inability to model interactive algorithms. The rigid semantics of MapReduce does not map well to all types of problems and workloads [138], which in many cases may lead to decreased performance and unnaturally expressed solutions [127]. An alternative frameworks to MapReduce is the Khan process network (KPN). KPNs support arbitrary communication graphs with cycles and are deterministic. However, not many general-purpose KPN implementations exist. Some known implementations include the Sesame project [123], YAPI [70], and the Nornir framework [128]. These frameworks have several benefits, but for application developers the KPN model has challenges. One of the main challenges is that the communication channels between the processes must be specified manually and, in an environment without a shared memory model, distributed deadlock detection must be implemented.

An alternative is a framework based on a process network paradigm, such as StreamIt [44]. Here, we have a language and runtime for the implementation of streaming programs that are described by a graph with computational blocks, called filters, that has a single input and output. The filters can be combined in loops and fork/join patterns but must provide bounds on the number of messages produced and consumed, making a StreamIt graph a synchronous data flow process network [74]. The framework supports multiple machines and processors. However, this must be specified at compilation time.

Processing and developing distributed multimedia applications is more complex than for traditional sequential applications. Multimedia workloads often have strict requirements and deadlines. Iterative processing is also essential for live multimedia workloads, such as Bagadus. Thus, all existing frameworks have shortcomings that are hard to address and the traditional batch processing frameworks simply come up short in our multimedia scenario. In the next section, we describe the design ideas and a basic implementation of our new framework for real-time multimedia processing.

5.4   The P2G Framework

The basic idea behind the P2G framework comes from the observation that most of the frameworks for distributed processing lack support for real-time multimedia workloads. The frameworks often also sacrifice task and data parallelism. With data parallelism,

multiple processing cores perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple processing cores to perform different operations in parallel.

Many of the existing processing frameworks optimize for either task or data parallelism, but not both. This means that they can potentially limit the ability to express the parallelism of a given workload. For example, MapReduce and its related approaches provide considerable support for parallelization but restrict runtime processing to data parallelism [39]. Functional languages such as Haskell [52], Erlang [6], and the event-based Specification and Description Language [65], map well to task parallelism. In these languages, programs are expressed as communicating processes either through event distribution or message passing. This makes it challenging to express data parallelism without specifying a fixed number of communication channels.

For multimedia workloads, the Nornir framework improves on some of the shortcomings of the traditional batch processing frameworks, such as Dryad and MapReduce. KPNs are deterministic; each execution of a process network produces the same output given the same input. KPNs also support arbitrary communication graphs (with cycles/iterations), while MapReduce and Dryad restrict application developers to a parallel pipeline structure and directed acyclic graphs (DAGs). However, Nornir is only task parallel and data parallelism must be explicitly added by the programmer. Furthermore, as a framework for heterogeneous multicore architectures, Nornir still has challenges. For example, the message-passing communication channels, with exactly one sender and one receiver, are modeled as infinite first in, first out queues. In real-life heterogeneous architectures, however, queue length is limited by available memory. A heterogeneous and distributed Nornir implementation would therefore require a distributed deadlock detection algorithm. Another issue is the complex programming model. The KPN model requires the application developer to specify the communication channels between the processes manually.

P2G builds on some of the knowledge gained from Nornir and we address the requirements from multimedia workloads, with inherent support for deadlines. A particularly desirable feature of processing multimedia workloads is the automatic combination of task and data parallelism. Intra-frame prediction in H.264 or VP8, for example, introduces many dependencies between the sub-blocks of a frame and, together with other overlapping processing stages, these operations have great potential to benefit from both types of parallelism. Multimedia algorithms are iterative and exhibit many pipeline parallel opportunities. It is hard to exploit them, because an intrinsic knowledge of fine-grained dependencies is required and it is difficult to structure programs in such a way that pipeline parallelism can be used. Thies et al. [122] wrote an analysis tool for finding parallel pipeline opportunities by evaluating memory accesses, assuming stable behavior. They evaluated their system on multimedia algorithms and significantly increased parallelism by utilizing the complex dependencies found. In the P2G framework, application developers model data and task dependencies explicitly, which enables the runtime system to automatically detect and take full advantage of all parallel opportunities without manual intervention.

The main source of non-determinism in the other languages and frameworks lies in the arbitrary order of read and write operations from and to memory. This source of non-determinism behavior can be removed by using a strict write-once semantics for writing to memory [8]. Several languages take advantage of the concept of single assignment, including Haskell [52] and Erlang [6]. This enables the schedulers to determine when

code depending on a memory cell is runnable. This is a key concept that we adopted for P2G. While write-once semantics are well suited for a scheduler's dependency analysis, it is not straightforward to think about multimedia algorithms in the functional terms of Erlang and Haskell. Multimedia algorithms tend to be formulated in terms of iterations of sequential transformation steps. They act on multidimensional data arrays (e.g., the pixels in a picture) and frequently provide very intuitive data partitioning opportunities (e.g., 8*8 pixel macroblocks in a picture). Prominent examples are the computation-heavy MPEG-4 Advanced Video Coding encoding [64] and scale-invariant feature transform [75] pipelines. Both are also examples of algorithms whose subsequent steps provide data decomposition opportunities at different granularities and along different dimensions of input data. Consequently, P2G should allow programmers to think in terms of fields without losing write-once semantics. In this proof-of-concept implementation of P2G, we use multidimensional arrays to implement fields.

The ability to carry out flexible partitioning requires the processing of clearly distinct data units without side effects. The idea in P2G is to use *kernels* as in stream processing [44, 92]. This is the same paradigm used by GPUs and, when we experimented with simple workloads in Chapter 3, the preferred paradigm of programmers. A kernel describes the transformation of multidimensional fields of data. When such a transformation is formulated as a loop of equal steps, the field should instead be partitioned and the kernel instantiated to achieve data-parallel execution. Each of these data partitions and tasks can then be scheduled independently by the schedulers, which analyze dependencies and guarantee a fully deterministic output, independent of order, due to the write-once semantics of fields.

Together these observations determine four basic ideas for the design of P2G:

- The use of *multidimensional fields* as the central concept for storing data in P2G to achieve straightforward implementations of simple and complex multimedia algorithms.

- The use of *kernels* that process slices of fields to achieve data decomposition.

- The use of *write-once semantics* to such fields to achieve deterministic behavior.

- The use of *dependency analysis at runtime* at a granularity finer than entire fields to achieve task decomposition along with data decomposition.

The P2G framework is designed to be language independent; however, for this prototype, we defined a C-like language that captures many of P2G's central concepts. As such, the P2G language was inspired by several other languages. Cray's Chapel [19] language antedates many of P2G's features in a more complete manner. However, P2G adds write-once semantics and support for multimedia workloads. Furthermore, P2G programs consist of interchangeable language elements where the programmer formulates data dependencies with fetch and store statements between implicitly instantiated kernels, (currently) written in C and C++. The biggest deviation from most other modern language designs is that the P2G kernel language makes both message passing and parallelism implicit and allows users to think in terms of sequential data transformations. Furthermore, the P2G concept supports deadlines, which allows for scheduling decisions

such as termination, branching, and the use of alternative code paths based on runtime observations.

P2G allows programmers to focus on data transformations in a sequential manner while simultaneously providing enough information to dynamically adapt the data and task parallelization. The fields in P2G look mostly like global multidimensional arrays in C, although their representation in memory may differ. They do not have to be placed contiguously in the memory of a single node; the fields can even be distributed across multiple execution nodes.

5.4.1   Architecture
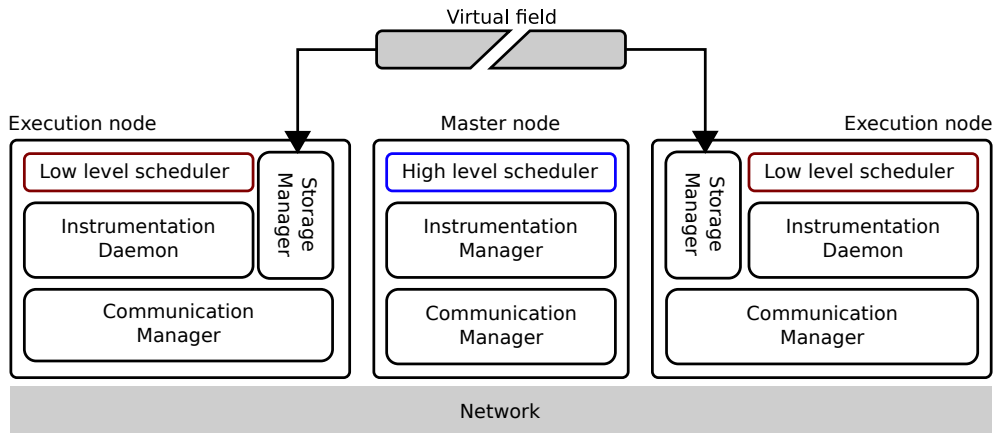
Figure 5.1: Overview of the architecture in the P2G system.

The basic architecture of the P2G framework can be seen in Figure 5.1. The P2G architecture consists of a *master node* and an arbitrary number of *execution nodes*. Each of the execution nodes reports its local topology (a graph of multicore and single-core CPUs and GPUs, connected by various kinds of buses and other networks) to the master node, which combines this information into a global topology of available resources. This global topology can be updated during runtime, as execution nodes are dynamically added and removed to accommodate for changes in the global load.

To maximize throughput, P2G uses the two-level scheduling approach we investigated in 2008 [115]. On the master node, we have an HLS and, on the execution node(s), we use a low-level scheduler (LLS). The HLS can analyze a workload's store and fetch statements, from which it can generate an intermediate implicit static dependency graph. An example of such a graph is shown in Figure 5.2(a), where the edges connecting two kernels through a field can be merged, circumventing the need for a vertex representing the field, which is shown in Figure 5.2(b). From the intermediate graph, the HLS can then derive the final implicit static dependency graph shown in Figure 5.2(b). The HLS will then use a graph partitioning [50] or search-based [42] algorithm to partition the workload into a suitable number of components that can be distributed to and run on the resources available in the topology. Using instrumentation data collected from the nodes executing the workload, we can weight the final graph with this profiling data during runtime. The weighted final graph can then be repartitioned, with the intent of

(a) Intermediate implicit static dependency graph

(b) Final implicit static dependency graph
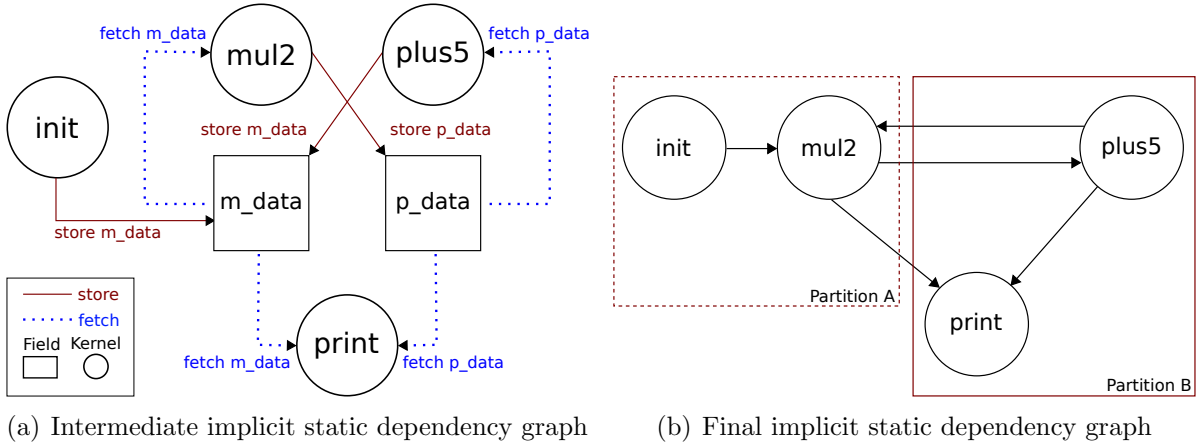
Figure 5.2: Dependency graphs in the P2G system.

improving system throughput, accommodating for changes in the global load or adapting to changes in available resources.

Given a partial workload such as *partition A* in Figure 5.2(b), an LLS at an execution node is responsible for local scheduling decisions. Figure 5.4 shows how the LLS can combine tasks and data to minimize overhead introduced by P2G and take advantage of specialized hardware, such as GPUs and other coprocessors.

The distribution of data, reporting, and other communication patterns in P2G is carried out through an event-based, distributed publish–subscribe model. Dependencies between components in a multimedia workload are deterministically derived from the code and the HLSs' partitioning decisions, resulting in direct communication. As such, the P2G framework relies on a combination of HLS and LLS instrumentation data and the global topology to make the best use of the performance of several heterogeneous processing architectures in a distributed system.

5.4.2 Programming Model

The programming model used in the P2G framework has two main concepts: the *implicit static dependency graph* shown in Figures 5.2(a) and 5.2(b) and the *dynamically created directed acyclic dependency graph* (DC-DAG) illustrated in Figure 5.4. A *kernel language* was also implemented to make it easier for programmers to develop workloads using P2G. An example workload written in the P2G kernel language is shown in Figure 5.5). The initial C++ version of the workload is shown in Figure 5.3.

The P2G version consists of two primary kernels: *mul2* and *plus5*. These two kernels form a pipeline where *mul2* first multiplies a value by two and stores the data, which *plus5* then fetches and increases by five; *mul2* then fetches the data stored by *plus5*; and so on. The *print* kernel runs orthogonally to these two kernels and fetches and writes the data they produced to *cout*. In combination, these three kernels form a cycle. The kernel *init* runs only once and writes initial data for *mul2* to consume. The kernels operate on two one-dimensional, five-element fields. The print kernel writes {*10, 11, 12, 13, 14*}, {*20, 22, 24, 26, 28*} for the first *age* and {*25, 27, 29, 31, 33*}, {*50, 54, 58, 62, 66*} for the second. Since there is no termination condition, this program runs indefinitely.

```cpp
void print( int *in, int number )
{
    for( int i = 0; i < number; ++i ) {
        std::cout << in[i] << " ";
    }
    std::cout << std::endl;
}

void mul2(int *in, int *out, int number)
{
    for (int i = 0; i < number; ++i)
        out[i] = in[i] * 2;
}

void plus5(int *in, int *out, int number)
{
    for (int i = 0; i < number; ++i)
        out[i] = in[i] + 5;
}

int main()
{
    int m_data[5] = { 10, 11, 12, 13, 14 };
    int p_data[5];

    int number = sizeof(m_data) / sizeof(m_data[0]);

    while( true )
    {
        mul2(m_data, p_data, number);

        print( m_data, number );
        print( p_data, number );

        plus5(p_data, m_data, number);
    }
    return 0;
}
```

Figure 5.3: Initial C++ version of a mul/sum example.
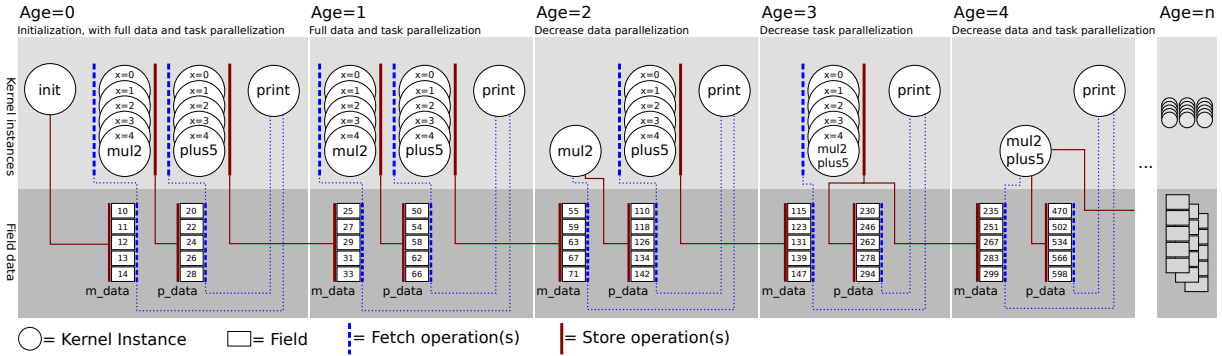
Figure 5.4: Dynamically created directed acyclic dependency graph (DC-DAG).

Dependency Graphs

The intermediate implicit static dependency graph can be seen in Figure 5.2(a) and is extracted from the *fetch* and *store* statements in a kernel. These statements are used by kernels to interact with fields. This intermediate graph can be further refined by merging the edges of kernels linked through a field vertex, resulting in a final implicit static dependency graph, as depicted in Figure 5.2(b). This final graph can serve as input to the HLS, which can use it to determine how best to partition the workload given a global topology. In this proof-of-concept implementation of P2G, dependency analysis is only carried out during runtime.

The graph can be further weighted using instrumentation data during runtime to serve as input for repartitioning. These weighted graphs can also serve as input in static offline analysis. For example, it could be used as input to a simulator to best determine how to initially configure a workload. During runtime, the intermediate implicit static dependency graph is expanded to form a dynamically created directed acyclic dependency graph, as shown in Figure 5.4. This expansion from a cyclic graph to a DAG occurs as a result of our write-once semantics. As such, we can see how P2G is designed to unroll loops without introducing implicit barriers between iterations. We call each such unrolled loop an *Age*. In Figure 5.4, we see how the LLS can then use the DC-DAG to combine both tasks and data to reduce overhead introduced by the P2G framework.

When moving from *Age=1* to *Age=2*, we can see that the LLS made a decision to reduce data parallelity. In P2G, kernels fetch slices of data and, initially, *mul2* was defined to work on each single field entry in parallel, but in *Age=2* the LLS decreased the granularity of the fetch statement to encompass the entire field. The LLS could also split the field in two, leading to two kernel instances of *mul2*, working on disparate sets of the field. Moving from *Age=2* to *Age=3*, we see that the LLS made the decision to decrease task parallelity. This is possible because *mul2* and *plus5* effectively form a pipeline, information that is available from the static graphs. By combining these two tasks, the individual store operations of the tasks are deferred until the data have been fully processed by each task. If the *print* kernel is not present, storage to the intermediate field *m_data* could be circumvented in entirety. In the final step, moving from *Age=3* to *Age=4*, we can see that a decision to decrease both task and data parallelity was made. This effectively renders this single kernel instance into a classical *for loop*, working on each data element of the field, with each task (*mul2*, *plus5*) performed sequentially on the

Field definitions:

0	int32[] m_data age;
1	int32[] p_data age;

Kernel definitions:

0	init:
1	local int32[] values;
2	
3	%{
4	int i = 0;
5	for(;i < 5; ++i)
6	{
7	put(values, i+10, i);
8	}
9	%}
10	
11	store m_data(0) = values;
12	
13	

0	mul2:
1	age a;
2	index x;
3	local int32 value;
4	
5	fetch value = m_data(a)[x];
6	
7	%{
8	value *= 2;
9	%}
10	
11	store p_data(a)[1] = value;
12	
13	

0	plus5:
1	age a;
2	index x;
3	local int32 value;
4	
5	fetch value = p_data(a)[x];
6	
7	%{
8	value += 5;
9	%}
10	
11	store m_data(a+1)[x] = value;
12	
13	
14	
15	

0	print:
1	age a;
2	local int32[] m, p;
3	
4	fetch m = m_data(a);
5	fetch p = p_data(a);
6	
7	%{
8	for(int i=0; i < extent(m, 0);)
9	cout << get(m, ++i) << " ";
10	cout << endl;
11	
12	for(int i=0; i < extent(p, 0);)
13	cout << get(p, ++i) << " ";
14	cout << endl;
15	%}

Figure 5.5: Kernel and field definitions.

data.

The P2G framework can make these runtime adjustments to data and task parallelism dynamically based on the resources available at the time.

Kernel Language

In the current prototype of the framework, P2G is exposed to the developer through the *kernel language* (BNF grammar of the kernel language can be found in Appendix A). An implementation of a simple workload in kernel language is outlined in Figure 5.5. The language can be replaced easily. However, it exposes the basic functions of the design. The most important parts are the kernel and field definitions, which describe the code and interaction patterns in P2G.

The main purpose of a kernel definition is to describe the required interaction of a kernel instance with an arbitrary number of fields (holding the application data) through

the fetch and store statements. A field in P2G serves as an interaction point for kernel definitions, as shown in Figure 5.2(a).

An important aspect of a multimedia workload such as the Bagadus video pipeline is the ability to express deadlines. It is unnecessary to stitch a panorama if the playback has moved past that point in the video. We therefore implemented language support for expressing deadlines. In principle, a deadline gives the application developer the option of defining a global timer. This timer can then be polled and updated from within a kernel definition. Given a condition based on a deadline, a timeout can occur and an alternate code path can be executed. Such an alternate code path is executed by storing to a different field then in the primary path, leading to new dependencies and new behavior. In this proof-of-concept implementation, we did not implement support for timers; however, we are currently reevaluating the concept of timers.

Fields in P2G have several properties, including a type and a dimensionality. An important property mentioned earlier is aging. Aging allows kernels to be iterative while maintaining write-once semantics in such cyclic execution. Aging enables unique storage to the same position in a field several times, as long as the age increases for each store operation, as shown in Figure 5.4). In essence, this adds an extra dimension to the field and makes it possible to accommodate iterative algorithms. It is also important to note that a field is not connected to a single execution node; it can be distributed across multiple execution nodes, as shown in Figure 5.1).

When defining the interaction between kernels and fields, the programmer is encouraged to express the finest possible granularity of kernel definition and, likewise, the most precise slices possible for the kernel within the field. The reason is because it provides the LLS more control over the granularity of task and data decomposition. With instrumentation data, the framework can reduce scheduling overhead by combining several instances of a kernel that process different data or several instances of different kernels that process data in sequence, as shown in Figure 5.4). The scheduler in P2G makes its decisions based on instrumentation data and the implicit static dependency graph.

Runtime

We can now extrapolate the concept of kernel definitions to kernel instances. A kernel instance is the unit of code that is executed during runtime and the number of kernel instances executed in parallel for a given kernel definition will depend on the fetch statements.

A kernel instance works on an arbitrary number of slices of fields, depending on the number of fetch statements in the kernel definition. If we look at the example in Figures 5.4 and 5.5, we can see how the *mul2* kernel, given its *fetch* statement on *m_data* with *age=a* and *index=x*, fetches only a single element of the data. Thus, since the *m_data* field consists of five data elements, this means that P2G can execute a maximum of x kernel instances simultaneously per age, yielding *a*x mul2* kernel instances.

P2G also supports the automatic resizing of fields. This is shown in the kernel definition of the *print* kernel in Figure 5.5. Initially, the extents of *m_data* and *p_data* are not defined. With each iteration of the *for* loop in *init*, the local field *values* is resized locally. This leads to a resizing of the global field *m_data* when *values* is stored to it. These extents are then propagated to the respective fields impacted by this resizing, such

as *p_data*.

It is important to note that a kernel instance is only dispatched when all its dependencies are fulfilled, that is, all the data it fetches have been stored to the respective fields and elements. Figures 5.4 and 5.5 also show that *mul2* stores its result to *p_data* with *age=a* and *index=x*. This means that once *mul2* has stored its results to *p_data* with *index=2* and *age=0*, the kernel instance of *plus5* with the fetch statement *fetch(0)[2]* can be dispatched. With our write-one semantics, each kernel instance is only dispatched once.

5.4.3   Prototype

A prototype implementation of the basic concepts of the P2G framework was implemented. The prototype consisted of a compiler for the kernel language and a runtime that could execute P2G programs on the x86 multicore architecture with a shared memory model running a Linux operating system.

Compiler

Programs written for the P2G framework are designed to be platform independent and have native blocks of code written in C or C++. Heterogeneous systems are specifically targeted; however, many of these require a custom compiler for the native blocks, such as Nvidia's NVCC compiler for GPUs with CUDA. For the prototype, instead of generating binaries directly, we decided to compile P2G programs into C++ files, which can be further compiled and linked with native code blocks. This approach provides us with less control of the resulting object code, but we gain both the flexibility and optimizations of the native compilers, resulting in a lightweight P2G compiler.

Runtime

Our prototype implementation of P2G features a basic execution node, with support for multidimensional fields, instrumentation, the implicit resizing of fields, and parallel execution of kernel instances on a single machine, using the implicit dependency graph formed by kernel definitions. Support for deadline expressions was not implemented.

The target architecture for this prototype is a single machine with a shared memory multicore x86 architecture. The system was designed as a push-based system using event subscriptions on field operations. Kernel instances are executed in parallel and produce events on *store* statements, which could require resizing operations. A kernel subscribes to events related to the fields that it depends on, that is, fields referenced by the kernel's *fetch* statements. When such a storage event is detected, the runtime finds all *new* valid combinations of age and index variables that can be processed as a result of the *store* statement and places these in a per-kernel ready queue. This means that the ready queues always contain the maximum number of parallel instances that can be executed at any time, limited only by unfulfilled data dependencies.

The prototype uses a simple LLS that consists of a dependency analyzer and kernel instance dispatcher. The dependency analyzer uses the implicit dependency graph to add new kernel instances to a ready queue, which can later be executed by the worker threads. Dependencies are analyzed in a dedicated thread that handles events emitted

from running kernel instances that notifies upon *store* and *resize* operations performed on fields.

When executed, kernel instances are dispatched from the ready queue. They are scheduled in an order that prefers the execution of kernel instances with a lower age value (older kernel instances). This ensures that no runnable kernel instance is starved by others that have no *fetch* statements.

5.4.4 Workloads

To test the initial idea and the prototype, we developed a few simple workloads commonly used in multimedia processing. The P2G kernel language is able to expose both the data and task parallelism of the workloads to the P2G system, so that the runtime is able to adapt the execution of the programs to suit the target architecture.

K-Means Clustering
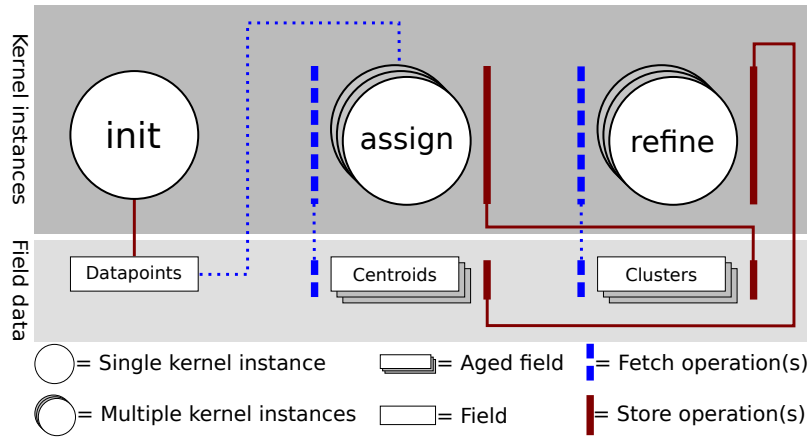
Figure 5.6: Overview of the K-means clustering algorithm.

K-Means clustering is an iterative algorithm for cluster analysis that aims to partition n data points into k clusters, where each data point belongs to the cluster with the nearest mean. The basic structure of the workload is shown in Figure 5.6. Our implementation in P2G consists of an *init* kernel, which generates n data points and *stores* them to the data points field. Next, it randomly selects k of these data points as the initial means and *stores* them to the centroids field. Then, the *assign* kernel *fetches* a slice of data, a single data point per *kernel instance*, the last calculated centroids, and *stores* this data point in the cluster of the closest centroids using a Euclidean distance calculation. Finally, the *refine* kernel *fetches* a cluster, calculates its new mean, and *stores* this information in the centroids field. The kernel definitions of *assign* and *refine* will form a loop that gradually leads to a convergence in centroids, at which point the k-means algorithm is completed.

Motion JPEG

The Motion JPEG (MJPEG) workload is based on the same algorithms and code used to test simple workloads on the x86 architecture, GPUs, and the Cell in Section 3.2.1. The

MJPEG format provides many layers of parallelism and is well suited for illustrating the potential of the framework. We focus on optimizing the discrete cosine transform (DCT) and quantization part, as this is the most computationally intensive part of the codec.

The *read + splitYUV* kernel reads the input video in YUV format and stores the data in three global fields: *yInput*, *uInput*, and *vInput*. In this workload, the three YUV components can be processed independently of each other and this property is exploited by creating three kernels: *yDCT*, *uDCT*, and *vDCT*. In Figure 5.7, we see that the respective DCT kernels are dependent on one of these fields.
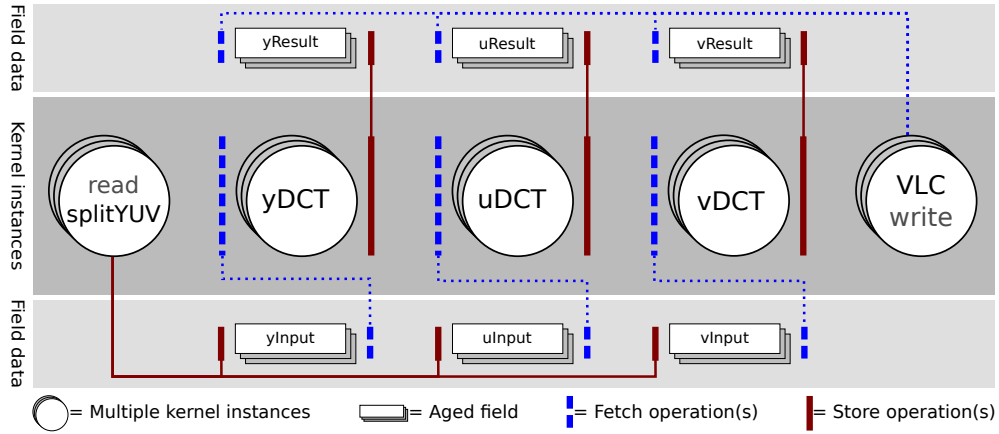
Figure 5.7: Overview of the MJPEG encoding process

The MJPEG encoding process splits the video frames into 8*8 macroblocks. The CIF resolution of 352*288 pixels per frame used in our tests will generate 1584 macroblocks of Y (luminance) data, each with 64-pixel values. The 4:2:2 chroma sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the discrete cosine transformed macroblock into the global result fields *yResult*, *uResult*, and *vResult*. Finally, the *VLC + write* kernel stores the MJPEG bitstream to disk.

5.4.5   Evaluation

We ran experiments with the *K*-means and MJPEG workloads, as described in Section 5.4.4. Each test was run on the following x86 multicore architectures, with the number of worker threads ranging from one to eight:

- One Intel Core i7-860 based on the Nehalem architecture running at 2.8 GHz, with four cores and Hyper-Threading (simultaneous multithreading) enabled.

- Four AMD Opteron 8218 processors based on the K8 architecture running at 2.6 GHz, with two cores per processor, for a total of eight cores.

In addition, we performed micro-benchmark tests for both workloads. These summarize the number of kernel instances dispatched per kernel definition, dispatch overhead, and time spent in the kernel code.

K-Means Clustering

The K-means workload is run with K=100, using a randomly generated data set with
2000 data points. The K-means algorithm does not run until convergence and we defined
a breakpoint after 10 iterations. Without any breakpoint, the algorithm's convergence is
undefined and, as such, we introduce this condition to ensure that we achieve relatively
stable running times.
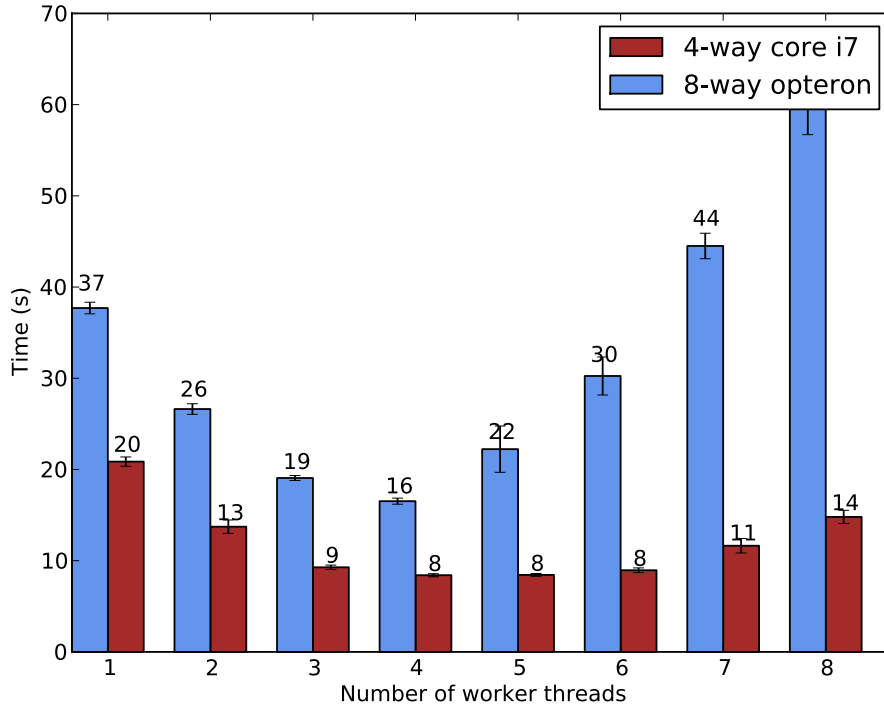
Figure 5.8: Workload execution time for K-means.

As seen in Figure 5.8, the K-means workload scales well to four worker threads.
After this, the execution time increases with the number of worker threads. This can be
explained by the fine granularity of the *assign* kernel definition. This leads to the serial
dependency analyzer becoming a bottleneck in the system. As discussed in Section 5.4.2,
this condition could be alleviated by decreasing the granularity of data parallelism, in
effect leading to each kernel instance of *assign* working on larger slices of data. By doing
so, we would increase the ratio of time spent in kernel code compared to dispatch time
and reduce the workload of the dependency analyzer.

The two different test machines behave somewhat differently, in that the Opteron
suffers more than the Core i7 when the dependency analyzer saturates a core. The Core
i7 is able to increase the frequency of a single core to mitigate serial bottlenecks and the
memory architectures of Intel processors are generally more efficient than AMD processors.
We think this is why the Core i7 suffers less under the limitations dictated by Amdahl's
law.

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 μs	9,829.00 μs
assign	2,024,251	4.07 μs	6.95 μs
refine	1000	3.21 μs	92.91 μs
print	11	1.09 μs	379.36 μs

Table 5.1: Micro-benchmarks of K-means in P2G.

MJPEG

The MJPEG workload is run on 50 frames of the standard *Foreman* test sequence encoded in *CIF* resolution.
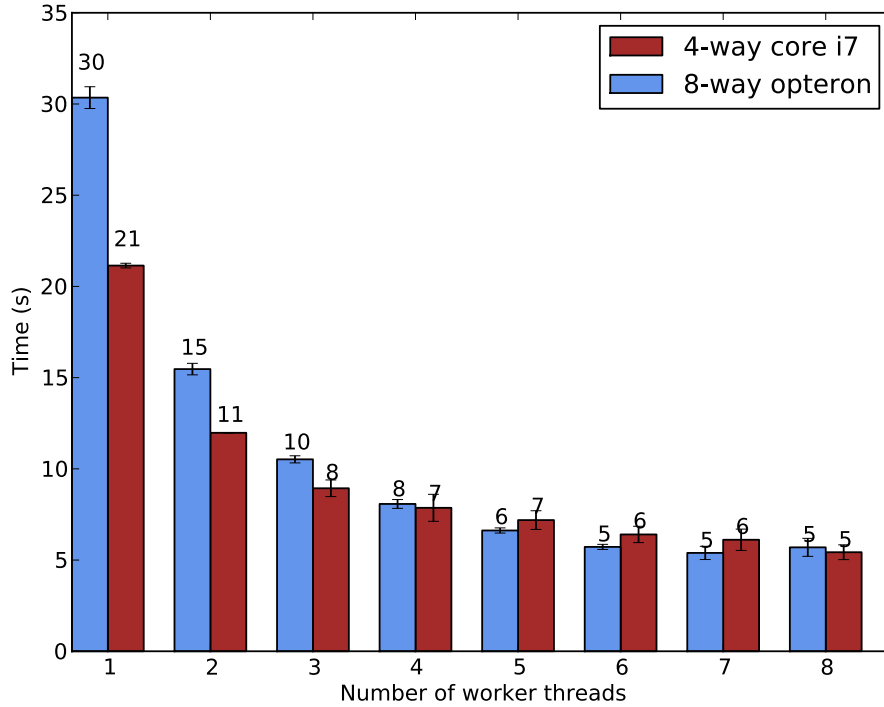
Figure 5.9: Workload execution times for MJPEG.

As we can observe in Figure 5.9, P2G is able to scale close to linearly with its available resources. In P2G, the dependency analyzer of the LLS runs in a dedicated thread. This affects the execution time when moving from seven to eight worker threads, where the eighth thread shares resources with the dependency analyzer.

A native single-threaded version of the MJPEG encoder on which the P2G version is based has a running time of 30 seconds on the Opteron machine and 19 seconds on the Core i7 machine.

From the micro-benchmarks in Table 5.2, we can see that time spent in kernel code is considerably greater compared to the dispatch overhead for the kernel definitions. The

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 μs	18.00 μs
read/splityuv	51	35.50 μs	1,641.57 μs
yDCT	80,784	3.07 μs	170.30 μs
uDCT	20,196	3.14 μs	170.24 μs
vDCT	20,196	3.15 μs	170.58 μs
VLC/write	51	3.09 μs	2,160.71 μs

Table 5.2: Micro-benchmarks of MJPEG encoding in P2G.

dispatch time includes the allocation or reallocation of fields as part of the timing oper-
ation. As a result, *init* and *read/splitYUV* have considerably higher dispatch times then
the **DCT* operations. We can also see that the majority of the CPU time is spent in the
kernel instances of *yDCT, uDCT*, and *vDCT*, which is the most computationally intensive
part of this workload. This indicates that decreasing the data and task granularity, as
discussed in Section 5.4.2, has little impact on system throughput. This is because the
majority of the time is already spent in kernel code.

5.4.6   Summary

With the P2G framework, we proposed a new flexible system for the automatic, parallel
real-time processing of multimedia workloads. We encourage the programmer to specify
parallelism in as fine a granularity as possible along the axes of task and data decomposi-
tion. Using our kernel language, this decomposition is expressed through kernel definitions
and fetch and store statements on fields.

Given a workload that uses our kernel language and is compiled for execution in P2G,
this workload can be partitioned by the HLS in a P2G master node, which then distributes
the partitions to the P2G execution nodes, which will execute the tasks. Execution nodes
can consist of heterogeneous resources. The LLS at the execution nodes will adapt the
workload to run optimally, using the resources available. Feedback from the instrumen-
tation module at the execution node can lead to workload repartitioning.

We implemented a prototype execution node capable of running on a shared memory
multicore architecture. The results from our experiments running on this prototype ex-
ecution node show the potential of our ideas. However, many features—such as support
for agglomeration, distribution, timers, heterogeneous architectures, and so on—have yet
to be added.

5.5   The Future

Our prototype implementation of P2G is a proof-of-concept implementation of the P2G
execution node with support for shared memory x86 multicore architecture. However,
several features have yet to be implemented in the execution node, two of these being
timers and agglomeration. Timers are important for enforcing soft deadlines in multi-
media workloads and agglomeration is important to adapt the level of parallelism to the
heterogeneous architecture we want to use. A kernel running on a GPU would require

many more threads than a kernel running on the CPU. This issue is apparent in the
K-means workload evaluated in Section 5.4.5.

The master node has also not yet been implemented. One of the reasons for targeting
the x86 multicore architecture in our prototype of the framework is the shared memory
property. When we add more machines or heterogeneous architectures with an exclusive
memory model, we need a distributed name service to manage message passing between
machines and processing nodes.

One of the next steps for the P2G framework would be to add support for using GPUs
on the same machine as the x86 multicore architecture. This would not require any master
node, since the GPUs on a single machine are managed by the host CPU.

Chapter 6

Papers and Author's Contributions

6.1 Overview of Research Papers

The research conducted during the PhD period focused on system support for multimedia. My papers addressed a large set of challenges, ranging from scheduling mechanisms for heterogeneous architectures to the optimization of video codecs. This research was a cooperative effort and the contribution of this thesis is its investigation of homogenous and heterogeneous computing platforms and infrastructures, an understanding of the developer's ability to exploit their performance, and the development of improvements for processing multimedia workloads. Nine of our papers were chosen to document this research effort and are included as the main contribution of this thesis. Five of these papers [32,34,102,112,117] look at processing simple workloads with heterogeneous architectures. Two of the papers [113,114] investigate a complex multimedia workload. The knowledge gained in these two Chapters is used in the previous chapter in the design of the P2G framework [31,115].

Although the other papers [35,46,76,77,116], posters, and demonstrations [10,11,33, 101,105,120,132] we wrote are related to multimedia systems, we limited the thesis to those nine papers that were central in forming our understanding of multicore programming and the development of P2G from the developer's perspective. These papers are presented chronologically in the following.

6.2 Paper I: Transparent Protocol Translation for Streaming

Abstract   The transport of streaming media data over TCP is hindered by TCP's probing behavior, which results in the rapid reduction and slow recovery of packet rates. On the other hand, UDP has been criticized for being unfair to TCP connections and it is therefore often blocked out of access networks. In this paper, we try to benefit from a combined approach using a proxy that transparently translates the transport protocol. We translate HTTP requests by the client transparently into RTSP requests and translate the corresponding RTP/UDP/AVP stream into the corresponding HTTP response. This enables the server to use UDP on the server side and TCP on the client side. This is beneficial for the server side, which scales to a higher load when it does not have to deal with

TCP. On the client side, streaming over TCP has the advantage that connections can be established from the client side and data streams can pass through firewalls. Preliminary tests demonstrate that our protocol translation delivers a smoother stream compared to HTTP streaming, where the TCP bandwidth oscillates heavily.

Lessons learned   This paper is written as a network research paper, using the Intel IXP network processor to translate a network protocol in real time. However, the experience gained while working with the IXP architecture gave us experience in working with a heterogeneous architecture with a shared memory model. The heterogeneous elements on the IXP also have different instruction sets and compilers, which we can also find in other heterogeneous architectures.

Author's contributions   Stensland contributed to the evaluation of the experiments. Espeland and Lunde carried out the design and implementation of the proxy and the experimental setup. The paper was written in collaboration with all the other authors.

6.3   Paper II: Evaluation of Multi-Core Scheduling Mechanisms for Heterogeneous Processing Architectures

Abstract   General-purpose central processing units (CPUs) with multiple cores are established products and new heterogeneous technology such as the Cell Broadband Engine and general-purpose graphic processing units (GPUs) bring an even higher degree of true multiprocessing to the market. However, the means for utilizing the processing power are immature. Current tools typically assume that the exclusive use of these resources is sufficient, but this assumption will soon be invalid because of interest in using their processing power for general-purpose tasks. Among the applications that can benefit from such technology is transcoding support for distributed media applications, where remote participants join and leave dynamically. Transcoding consists of several clearly separated processing operations that consume a great deal of resources, such that individual processing units are unable to handle all the operations of a session of arbitrary size. The individual operations can then be distributed over several processing units and data must be moved between them according to the dependencies between operations. Many multiprocessor scheduling approaches exist but, to the best of our knowledge, the challenge remains to find mechanisms that can schedule dynamic workloads of communicating operations while taking both the processing and communication requirements into account. For such applications, we believe that feasible scheduling can be performed at two levels, that is, divided into the task of placing a job onto a processing unit and the task of multitasking time slices within a single processing unit. We implemented some simple

high-level scheduling mechanisms and simulated a videoconferencing scenario running on topologies inspired by existing systems from Intel, AMD, IBM, and Nvidia. Our results show the importance of using an efficient high-level scheduler.

Lessons learned   In this paper, we designed a simple simulator that simulated the high-level scheduling aspect of a multimedia workload on different multicore architectures. We showed that a two-level scheduling approach, where the high level scheduler places jobs onto a processing core and the low-level scheduler takes on the job of time slicing within a single processing unit. Our results show the importance of using an efficient high-level scheduler. Lessons learned from this paper were later used when designing the P2G framework.

Author's contributions   Stensland contributed significantly to the design of the event-driven simulator. He also designed the workloads for the simulator and performed the experiments for the paper. Stensland also contributed to writing the paper.

6.4 Paper III: Tips, Tricks and Troubles: Optimizing for Cell and GPU

Abstract   When used efficiently, modern multicore architectures, such as the Cell and GPUs, provide the processing power required by resource-hungry multimedia workloads. However, the diversity of resources exposed to the programmer intrinsically requires very different mindsets to efficiently utilize these resources—not only compared to an x86 architecture, but also between the Cell and the GPUs. In this context, our analysis of 14 different Motion JPEG (MJPEG) implementations indicate great potential for optimizing performance, but there are also many pitfalls to avoid. By experimentally evaluating algorithmic choices and inter-core data communication (memory transfers) and architecture-specific capabilities, such as instruction sets, we present tips, tricks, and problems with respect to the efficient utilization of available resources.

Lessons learned   In the third paper, we analyze 14 different implementations of a multimedia workload from a graduate level course we teach on two different heterogeneous architectures. We learned that heterogeneous architectures such as the Cell and GPUs are suitable for processing real-time multimedia workloads such as MJPEG video encoding. However, it is not trivial for programmers to achieve high performance on either of these architectures. There are similarities between the Cell and GPUs, but the way programmers need to think is substantially different, not only compared to the x86 architecture but also between the Cell and the GPU. The Cell uses a single instruction multiple data (SIMD) programming model, which seems harder to grasp compared to the

SIMT abstraction used by the GPUs. All three architectures have different architectural capabilities that must be taken into account when choosing the algorithms to use.

Author's contributions   Stensland contributed significantly to the design, implementation, and evaluation of this work. Together with Espeland, he designed the experiments and evaluated the results. The paper was written in collaboration with the other authors.

Published in   The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2010), ACM, 2010.

DOI   10.1145/1806565.1806585

6.5   Paper IV: Cheat Detection Processing: A GPU versus CPU Comparison

Abstract   In modern online multiplayer games, game providers have been struggling to keep up with the many different types of cheating. Cheat detection is a task that requires great computational resources. Advances made within the field of heterogeneous computing architectures, such as in GPUs, have given developers easier access to considerably more computational resources, enabling a new approach to resolving this issue.

In this paper, we developed a small game simulator that includes a customizable physics engine and a cheat detection mechanism that checks the physical model used by the game. To make sure that the mechanisms are fair to all players, they are executed on the server side of the game system. We investigate the advantages of implementing physics cheat detection mechanisms on a GPU using the Nvidia CUDA framework and we compare the GPU implementation of the cheat detection mechanism with a CPU implementation. The results obtained from the simulations show that offloading the cheat detection mechanisms to the GPU reduces the time spent on cheat detection, enabling the servers to support larger numbers of clients.

Lessons learned   In the fourth paper, we used a cheat detection mechanism implemented on a GPU to learn about the effect of offloading workload from the CPU. The results shows that, even with a simple physical model, the GPU is able to outperform the CPU. However, we also observed that, with a low number of clients, the CPU implementation is faster that the GPU implementation. This is due to the latency cost of transferring the workload from CPU memory to the GPU for processing.

Author's contributions   Stensland designed the experiments for the paper. He also used a Fermi generation GPU for the paper's experiments and analyzed their data. The prototype was designed and implemented as part of Myrseth's master's thesis, for which Stensland was the main supervisor. The paper text was mainly written by Stensland, with contributions from the other authors.

6.6   Paper V: Reducing Processing Demands for Multi-Rate Video Encoding: Implementation and Evaluation

Abstract   Segmented adaptive HTTP streaming has become the de facto standard for
video delivery over the Internet for its ability to scale video quality to available network re-
sources. Each video is encoded at multiple levels of quality, that is, running the expensive
encoding process for each quality layer. However, these operations consume a great deal
of both time and resources and, in this paper, the authors propose a system for reusing
redundant steps in a video encoder to improve the multilayer encoding pipeline. The idea
is to have multiple outputs for each of the target bitrates and quality levels, where the
intermediate processing steps share and reuse the computationally heavy analysis. A pro-
totype has been implemented using the VP8 reference encoder and experimental results
show that, for both low- and high-resolution videos, the proposed method can significantly
reduce processing demands and time when encoding the different quality layers.

Lessons learned   The fifth paper looks at the possibility of reusing the computationally
intensive analysis part of a video encoder. We learned that the shared memory architecture
in x86 multicore processors greatly facilitates the sharing of data between multiple threads.
When running the reference VP8 encoder on four videos, we can see that serial encoding
performs better that running four instances concurrently. This shows the negative effects
of running too many threads on too few cores: The scheduling overhead in the operating
system increases and contention arises between the execution resources on the CPU.

Author's contributions   The multi-rate prototype was designed and implemented as
part of Finstad's master's thesis, of which Stensland was the main supervisor. Espeland
contributed with the basic idea of multi-rate encoding. The experiments were evaluated
by Stensland and Espeland. The text was also mainly written by Espeland and Stensland,
with input from the other authors.

6.7 Paper VI: LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition

Abstract Supporting thousands of interacting players in a virtual world poses huge processing challenges. Work that addresses the challenge utilizes a variety of spatial partitioning algorithms to distribute the load. If, however, a large number of players need to interact tightly across an area of the game world, spatial partitioning cannot subdivide this area without incurring massive communication costs, latency, or inconsistency. Scaling such areas to the largest number of players possible is a major challenge of game engines. Deviating from earlier thinking, we apply parallelism on multicore architectures to increase scalability. In this paper, we evaluate the design and implementation of our game server architecture, called a lockless, relaxed-atomicity state, or LEARS, which allows for lock-free parallel processing of a single spatial partition by considering every game cycle an atomic tick. Our prototype is evaluated using traces from live game sessions where we measure the server response time for all objects that require timely updates. We also measure how the response time for the multithreaded implementation varies with the number of threads used. Our results show that the challenge of scaling up a game server can be an embarrassingly parallel problem.

Lessons learned The sixth paper describes an architecture to scale a game server workload. We showed that resource utilization can be improved by distributing the load over the shared memory architecture of x86 multicore CPUs. However, we learned the importance of balancing the number of threads executing on the physical CPU. If too many threads are executed concurrently, the performance will start to degrade because of the increased context switching overhead. This is the opposite case from that of a GPU, where thousands of threads are required for good performance.

Author's contributions Stensland contributed to the discussion and evaluation of the game server with respect to scaling on the x86 multicore architecture. He also contributed with architectural knowledge about the evaluation system. Stensland also contributed to writing the paper.

6.8 Paper VII: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Abstract The computational demands of multimedia data processing are steadily increasing as consumers call for progressively more complex and intelligent multimedia

services. New multicore hardware architectures provide the required resources, but writing parallel, distributed applications remains a labor-intensive task compared to their sequential counterpart. For this reason, Google and Microsoft implemented their respective processing frameworks, MapReduce and Dryad, since they allow the developer to think sequentially yet benefit from parallel and distributed execution. An inherent limitation in the design of these processing frameworks is their inability to express arbitrarily complex workloads. The dependency graphs of the frameworks are often limited to directed acyclic graphs or even predetermined stages. This is particularly problematic for video encoding and other algorithms that depend on iterative execution. With the Nornir runtime system for parallel programs, which is a Kahn process network implementation, we addressed and resolved several of these limitations. However, it is more difficult to use than other frameworks are due to its complex programming model. In this paper, we build on the knowledge gained from Nornir and present a new framework, called P2G, designed specifically for developing and processing distributed real-time multimedia data. P2G supports arbitrarily complex dependency graphs with cycles, branches, and deadlines and provides both data parallelism and task parallelism. The framework is implemented to scale transparently with available (heterogeneous) resources, a concept familiar from the cloud computing paradigm. We implemented a (interchangeable) P2G model to ease development. In this paper, we present a proof-of-concept implementation of a P2G execution node and some experimental examples using complex workloads, such as MJPEG and K-means clustering. The results show that the P2G system is a feasible approach to multimedia processing.

Lessons learned In the seventh paper, we present a prototype and framework for the distributed real-time processing of multimedia workloads. We also implemented and evaluated two simple multimedia workloads to verify that multimedia workloads such as K-means and MJPEG can be expressed in the framework. Our experiments shows that our prototype is able to scale performance with the available resources in the system, as long as there is a large enough workload per instance.

Author's contributions Stensland contributed to the design and implementation of the workloads used to benchmark the framework and to the ideas behind the P2G framework. Espeland and Beskow designed, implemented, and micro-benchmarked the framework. The paper and discussions were written in collaboration with all of the authors.

6.9 Paper VIII: Bagadus: An Integrated Real-Time System for Soccer Analytics

Abstract   The importance of winning has increased the role of performance analysis in the sports industry and underscores how statistics and technology keep changing the way sports are played. Thus, this is a growing area of interest, both from a computer system view, to manage the technical challenges, and from a sports performance view, to aid the development of athletes. In this respect, Bagadus is a *real-time* prototype of a sports analytics application using soccer as a case study. Bagadus integrates a sensor system, a soccer analytics annotation system, and a video processing system using a videocamera array. A prototype was recently installed at Alfheim Stadium in Norway and, in this paper, we describe how the system can be used in real time to play back events. The system supports both stitched panoramic video and camera switching modes and creates video summaries based on queries to the sensor system. Moreover, we evaluate the system from a systems point of view, benchmarking different approaches, algorithms, and trade-offs, and show how the system runs in real time.

Lessons learned   The eighth paper describes the integration of the three different sub-systems in the Bagadus sports analysis system. The paper focuses on the optimization of the video system, where we optimize the video pipeline for both an x86 multicore and a GPU with the goal of running the system in real time on a single machine. The experiments shows that we are able to process video from four cameras, stitch the video to a panorama, and use video processing algorithms to enhance the quality of this panoramic video.

Author's contributions   Stensland contributed to the design and evaluation of the real-time video pipeline presented in this paper. He also analyzed the experimental results from the GK110 GPU presented in this paper and provided insight into the heterogeneous architecture used. Stensland was also a supervisor for all the master's students involved in this project. The paper was mainly written by Stensland, with contributions from the other authors.

6.10 Paper IX: Processing Panorama Video in Real-Time

Abstract   There are many scenarios in which a high-resolution, wide field of view video is useful. Such panoramic video may be generated using camera arrays, where the feeds from multiple cameras pointing at different parts of the captured area are stitched together. However, processing the different steps of a panoramic video pipeline in real time is

challenging due to the high data rates and the stringent timeliness requirements. In our research, we use panoramic video in a sports analysis system called Bagadus. This system was deployed at Alfheim stadium in Tromsø, Norway, and, due to live usage, the video events had to be generated in real time. In this paper, we describe our real-time panoramic system built using a low-cost CCD HD videocamera array. We describe how we implemented different components and evaluated alternatives. The performance results from experiments run on commodity hardware with and without coprocessors, such as GPUs, demonstrate that the entire pipeline is able to run in real time.

Lessons learned The ninth paper undertakes a more detailed analysis of the video system in the Bagadus sports analysis system. The paper focuses on how we achieved real-time performance by optimizing the video pipeline for both CPU and GPU execution. Finally, we evaluated the performance of the complete video pipeline on different heterogeneous architectures and machine setups. We learned that a system such as Bagadus, with real-time requirements, requires a GPU that can execute workloads concurrently. The GPU utilization for a system such as Bagadus is also fairly low and the GPU can be shared with other workloads.

Author's contributions Stensland contributed to the design and evaluation of the real-time video pipeline presented in this paper. He also provided insight into the different heterogeneous architectures and the machine setup used to evaluate the video pipeline. Stensland was also a supervisor for all the master's students involved in this project. The paper was mainly written by Stensland, with contributions from the other authors.

6.11 Supervised Master's Students

Student: Håvard Espeland

Title: Investigation of parallel programming on heterogeneous multiprocessors

Summary: This thesis investigates different parallelization strategies and performance for real-world problems using two heterogeneous architectures, the Intel IXP2400 architecture and the Cell Broadband Engine. The tests show promising throughput for some applications and the thesis proposes a scheme for offloading computationally intensive parts of an application.

Student: Alexander Ottesen

Title: Efficient parallelization techniques for applications running on GPUs using the CUDA framework

Summary: This thesis investigates the GPU architecture and processing capabilities of the first generation of Nvidia GPUs with support for the CUDA framework. We investigate how CUDA applications can share the GPU resource and see what challenges are connected with concurrent applications executing on the GPU.

Student: Magne Eimot

Title: Offloading an encrypted user space file system on GPUs

Summary: Modern computers often have powerful GPUs and an important use of this technology is to assist the main CPU with computationally intensive tasks. We investigate the challenges of using GPUs to offload the encryption operations from the main CPU.

Student: Martin Øinæs Myrseth

Title: Cheat detection in on-line multi-player games using graphics processing units

Summary: This thesis investigates the benefits of using GPUs for cheat detection mechanisms. We develop a framework for a game simulator that includes a simple customizable physical engine and a cheat detection mechanism. The results shows that, in addition to being faster, the GPU mechanism allows the CPU to perform other game-relevant tasks while the mechanism is executing.

Student: Espen Angell Kristiansen

Title: Dynamic adaption and distribution of binaries to heterogeneous architectures

Summary: Real-time multimedia workloads require extensive processing power. Here, we develop the foundation for network distribution in P2G and suggest a viable solution for the execution of workloads on heterogeneous multicore architectures.

Student: Dag Haavi Finstad

Title: Multi-rate VP8 video encoding

Summary: This thesis addresses the resource consumption issues of encoding multiple videos by proposing a method for reusing redundant steps in a video encoder, emitting multiple outputs with various bitrates and quality levels.

Student: Magnus Funder Halldal

Title: Exploring computational capabilities of GPUs using H.264 prediction algorithms

Summary: We explore the processing power of two generations of GPUs by implementing H.264 prediction algorithms. We implement motion vector search and motion vector prediction on the GPU.

Student: Kristoffer Egil Bonarjee

Title: Investigating host–device communication in a GPU-based H.264 encoder

Summary: This thesis investigates the performance pitfalls of an H.264 encoder written for GPUs. More specifically, we look into the interaction between the host CPU and the GPU. We do not focus on optimizing the GPU code but, rather, on how the execution and communication are handled by the CPU. Given the large amount of manual labor required to optimize the GPU code, it is easy to neglect the CPU part of accelerated applications.

Student: Simen Særgrov

Title: Bagadus: Next generation sports analysis and multimedia platform using camera array and sensor network

Summary: Bagadus, a system that integrates a sensor system, soccer analytics, and video processing with OpenCV on a camera array, is presented. A proof-of-concept prototype is implemented based on the system installed at Alfheim stadium in Norway.

Student: Espen Oldeide Helgedagsrud

Title: Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching

Summary: In this thesis, the Bagadus system is rewritten with real-time processing of the video as one of the main goals. This thesis focuses on the implementation of dynamic stitching and offloading this operation to a GPU.

Student: Marius Tennøe

Title: Efficient implementation and processing of a real-time panorama video pipeline with emphasis on background subtraction

Summary: In this thesis, the Bagadus system is rewritten with real-time processing of the video as one of the main goals. This thesis focuses on different implementations of background subtraction and offloading those algorithms to a GPU.

Student: Mikkel Næss

Title: Efficient implementation and processing of a real-time panorama video pipeline with emphasis on color correction

Summary: In this thesis, the Bagadus system is rewritten with real-time processing of the video as one of the main goals. This thesis focuses on the implementation of the color correction module and offloading this operation to a GPU.

Student: Ragnar Langseth

Title: Implementation of a distributed real-time video panorama pipeline for creating high quality virtual views

Summary: The Bagadus video pipeline with an updated camera array is redesigned with distributed processing in mind. Features such as HDR and the demosaicing of raw Bayer data from the new cameras are added to the GPU pipeline in Bagadus. A virtual camera is also extracted from the panoramic video.

Student: Vegard Aalbu

Title: MovieCutter: A system to make personalized video summaries from archived video content

Summary: This thesis investigates how adaptive streaming can be used to create a montage from events in a video archive. With metadata such as subtitles and chapters, users can search and generate customized video playlists from movies.

Student: Sigurd Ljødal

Title: Implementation of a real-time distributed video processing pipeline

Summary: The Bagadus video pipeline with an updated camera array is redesigned with distributed processing in mind. To do so, we use a PCI Express-based interconnect from Dolphin Interconnect Solutions and a low-level application programming interface for message passing called SISCI. We also investigate different techniques of moving data as efficiently as possible from cameras connected to a capture machine to GPU memory on the processing machine.

Student: Martin Alexander Wilhelmsen

Title: Real-time interactive cloud applications

Summary: This thesis investigates commodity hardware H.264 encoders and uses the NVENC hardware encoder found on modern GPUs from Nvidia to offload encoding in the Bagadus pipeline. We also implement support for streaming in Quake III to test the feasibility of using the hardware encoder in cloud gaming.

6.12   Other Publications

Several other papers were published in conferences during the PhD period. We did not include all the papers to limit the scope of this thesis. Instead, we provide a short summary of their contributions.

Disk input/output (I/O) We worked on I/O performance optimizations by improving file tree traversal performance by scheduling in user space [76, 77]. The technique proposed in these papers orders directory tree requests by logical block order on the physical disk. This optimization significantly improves the performance of file tree traversal operations; however, this is not possible in kernel space, since too few I/O operations are issued at a time for the scheduler to react efficiently. With a dirty file system, we were able to obtain up to four times the performance compared to that of a normal file tree traversal. This work clearly demonstrates the advantage of having several levels of schedulers and it can be adapted to the scheduling approach

used by the P2G framework, where a high-level scheduler issues the workload to the different processing cores and low-level schedulers on the different processing cores carry out time slicing.

Fault Tolerant Routing We implemented dynamic fault-tolerant routing in an scalable coherent interconnect (SCI) network [116]. We implemented support for dynamic fault tolerance in an SCI network on hardware produced by Dolphin Interconnect Solutions. By dynamic fault tolerance, we mean that the interconnection network reroutes affected packets around a fault in the network, while the rest of the network remains fully functional. Our implementation focuses on a two-dimensional torus topology and is compatible with the existing hardware and software stack. The routing algorithm is tested in clusters with real hardware and our tests show that distributed databases such as MySQL are able to run uninterrupted while the network reacts to faults.

Chapter 7

Conclusion

Processing multimedia workloads with heterogeneous architectures is not a trivial task. The abstractions to program these architectures can be different and programmers often have to manually tune and optimize their applications on multiple heterogeneous architectures to achieve the desired performance. In many cases, these optimizations are not portable; for example, if the hardware is changed, the applications have to be optimized or even rewritten for the new architecture. Several languages and processing frameworks exist; however, they are typically designed to support batch processing and making them support real-time multimedia workloads is not a trivial task. Our main research question, stated in Section 1.2, states that we want to investigate how to develop and process multimedia workloads for modern heterogeneous multicore architectures. In this thesis, we addressed this issue from a low-level standpoint, learning about the behavior of heterogeneous architectures with simple multimedia workloads and how to use multiple heterogeneous architectures for a complex pipeline with several multimedia workloads. We also addressed the problem statement from a high-level standpoint, where we presented the design and evaluated the prototype of P2G, a framework for processing multimedia workloads on heterogeneous architectures.

7.1 Summary

In this thesis, we looked at heterogeneous multicore architectures and their ability to process multimedia workloads. First, we selected four different architectures. To learn more about their behavior, we first conducted several case studies with simple multimedia workloads, where we only performed optimizations for one architecture. The first architecture we experimented with was the Intel IXP network processor. This architecture was used for experiments involving network protocol translation [32]. The next architecture was the x86 processor architecture. The first case study was the efficient implementation of Motion JPEG (MJPEG) encoding [112]. We also conducted case studies on using multiple x86 cores for multi-rate video encoding [34] and running a multithreaded game server prototype [102]. For the GPU architecture, we carried out case studies on the memory architecture of GPUs [94], optimization of host–device communication [13], and cheat detection [117], and we also revisited the MJPEG workload with both a GPU [112] and Cell architecture.

The knowledge obtained from investigating the simple multimedia workloads was used

to investigate a more complex multimedia workload, namely, the video processing pipeline of the Bagadus soccer analysis system [114]. We optimized the workload for multiple heterogeneous architectures to achieve the real-time capture, pre-processing, stitching, and encoding of a panoramic videostream from the soccer stadium on a single commodity gaming computer [113].

Using the knowledge gained from processing both simple and complex multimedia workloads on heterogeneous systems and also from our evaluation of multi-core scheduling mechanisms [115], we proposed a programming language and framework that exposes the parallelization opportunities of multimedia workloads for a runtime that allows for efficient execution on the available heterogeneous hardware [31]. A running prototype of this system, called P2G, running on a single machine with x86 multicore processors was developed together with two simple multimedia workloads. These workloads were used as proof of concept to show that we can express and run multimedia workloads in our framework.

7.2 Concluding Remarks

In the beginning of this thesis, we asked how a programmer efficiently develops multimedia workloads for modern heterogeneous multicore architectures. To answer this question, we further decomposed the question into three steps.

To learn about the behavior of heterogeneous architectures, we selected four architectures and looked at case studies with simple multimedia workloads optimized for only one architecture. From our evaluations, we observed that, for all the architectures, it is important to select algorithms that are suited to the architecture. To obtain optimal performance, especially on the Cell and x86, programmers must use architecture-specific vector extensions and rewrite their programs to use single instruction multiple data (SIMD) intrinsics. On the x86, we also noted the importance of balancing the number of threads used by the workloads to the available number of cores in the system. Too many threads executing on too few cores results in decreased performance due to contention and context switching overhead. On the GPU architecture, we noted the importance of using the correct memory space and also the importance of efficiently moving the data to and from the GPU. If the workload that is offloaded to the GPU is too small, performance can decrease compared to that of a CPU implementation. Our MJPEG experiments also suggest that programmers prefer the single instruction, multiple threads (SIMT) programming model exposed by the GPU compared to the SIMD model exposed by the Cell and the vector unit on the x86.

Next, we selected the two most promising heterogeneous architectures from our evaluation of simple multimedia workloads and evaluated a complex multimedia workload. One of the main requirements for this workload was for it to run in real time. We accomplished this by optimizing the workload for both heterogeneous architectures. The complex workload was the video subsystem of the Bagadus sports analysis system. Here, we learned that, in addition to making every module run separately in real time, we also had to make sure all the modules were running together in real time, as a pipeline. This required a great deal of manual tuning to decide which parts of the pipeline had to run on the CPU and which parts could be offloaded to a GPU. One of our observations was that the GPU's overall utilization was fairly low (only 14.8%). This finding, together

with the manual labor required to optimize the pipeline to run in real time, highlighted the importance of having a processing framework supporting real-time multimedia workloads to fully utilize available resources and ease the development of future cross-platform systems.

Using all the knowledge gained from our studies of simple and complex workloads, we designed and proposed a framework for processing real-time multimedia workloads on heterogeneous architectures, called P2G. P2G allows the programmer to use a programming model similar to the SIMT model used in CUDA and to express as much parallelism as possible in the code. The framework will then use the workload's data, task, and pipeline parallelism to optimize the granularity of the programs, either at compilation time or at runtime. The framework is designed to support distributed processing and uses a two-level scheduling approach. A high-level scheduler maps the workloads to processing nodes and a low-level scheduler manages the time slices of the available processing cores in a node. The fundamental ideas of the P2G framework were implemented in a prototype of the framework running on an x86 multicore architecture. To test the prototype implementation, we used two simple multimedia workloads. Developing workloads in the P2G Kernel Language is effective compared to working with low-level abstractions such as SIMD intrinsics and threads. Kernel Language provides good abstractions and helps developers to express both data parallelism and task parallelism.

Even though we did not implement all the concepts in the P2G framework, we showed that we are able to express multimedia workloads in the P2G programming model and to scale performance when more processing resources are added to the system. However, a great deal of work remains before the processing efficiencies of multimedia workloads in the P2G framework are anywhere near what can be achieved for workloads written natively for the architecture. We believe we have made significant contributions in expressing workloads and designing a framework that supports some of the different heterogeneous architectures available today.

7.3 Future Work

Several areas have potential for further work and we highlight a few potential next steps.

- An interesting heterogeneous architecture that is not explored in this thesis is the Intel Xeon Phi many-core processor [21]. This coprocessor uses many simple x86 cores on a shared memory architecture with a 512-bit vector unit per core. We did not test any simple multimedia workloads on this architecture. It would therefore be very interesting to see how it performs compared to a GPU, as well as investigate how programmers have to think to program multimedia workloads for this architecture.

- The P2G prototype presented in this thesis runs on a single machine with x86 multicore processors and a shared memory architecture. An interesting research opportunity could be to rewrite and extend P2G to take advantage of GPUs using the Nvidia CUDA framework or the Intel Xeon Phi many-core architecture.

- In the current version of P2G, only the execution node is implemented. Another potential area for further work is to extend P2G to run on multiple machines. This

is crucial for working with large data sets. The distribution of the framework would also introduce a new level of complexity in the scheduling and the design of efficient two-level schedulers presents several interesting research opportunities.

- The video pipeline of the Bagadus sports analysis system can run on a single x86 multicore machine with a high-end Nvidia GPU. If the P2G framework is extended with support for GPUs, it would be interesting to port parts of the Bagadus video pipeline to P2G. The feedback-based scheduling using instrumentation in P2G and the support for real-time workloads should be able to adapt and distribute the Bagadus video workload automatically on the CPU and GPU and make it run in real time, given sufficient processing resources in the system without the manual tuning required today.

It is also possible for new hardware to be developed that will open up interesting research topics within this field.

Bibliography

[1] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium - IPDPS 2004*, page 72, 2004.

[2] Adobe. HTTP Dynamic Streaming. `http://www.adobe.com/mena_en/products/hds-dynamic-streaming.html` [Online. Last accessed: July 2014], 2010.

[3] Advanced Micro Devices. AMD Demonstrates Worlds First X86 Dual-Core Processor. `http://www.amd.com/us/press-releases/Pages/Press_Release_89872.aspx` [Online. Last accessed: January 2014], 2004.

[4] Apache. Hadoop. `http://hadoop.apache.org` [Online. Last accessed: July 2010], 2010.

[5] Y. Arai, T. Agui, and M. Nakajima. A Fast DCT-SQ Scheme for Images. *Transactions of IEICE*, 71(11):1095–1097, 1988.

[6] J. Armstrong. A history of Erlang. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pages 6:1–6:26, 2007.

[7] Ars Technica. A technical overview of the Emotion Engine. `http://arstechnica.com/gadgets/2000/02/ee/` [Online. Last accessed: July 2014], 2000.

[8] R. S. N. Arvind, R. S. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632, 1989.

[9] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. VP8 Data Format and Decoding Guide. RFC 6386 (Informational), November 2011.

[10] P. B. Beskow, H. Espeland, H. K. Stensland, P. N. Olsen, S. Kristoffersen, E. A. Kristiansen, C. Griwodz, and P. Halvorsen. Distributed Real-Time Processing of Multimedia Data with the P2G Framework, 2011. Poster presented at ACM Eurosys 2011.

[11] P. B. Beskow, H. K. Stensland, H. Espeland, E. A. Kristiansen, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen. Processing of multimedia data using the P2G framework. In *Proceedings of the 19th ACM International Conference on Multimedia - MM '11*, pages 819–820. ACM, 2011.

[12] D. Blythe. The Direct3D 10 system. *ACM Transactions on Graphics*, 25(3):724, 2006.

[13] K. E. Bonarjee. *Investigating host-device communication in a GPU-based H.264 encoder*. Master thesis, University of Oslo, 2012.

[14] M. Brown and D. G. Lowe. Automatic Panoramic Image Stitching using Invariant Features. *International Journal of Computer Vision*, 74(1):59–73, August 2007.

[15] K. Cabeen and P. Gent. Image Compression and the discrete Cosine Transform. In *Math 45*. College of the Redwoods, 1998.

[16] Cairos Technologies. VIS.TRACK. `http://www.cairos.com/unternehmen/vistrack.php` [Online. Last accessed: October 2013], 2013.

[17] Camargus. Premium Stadium Video Technology Inrastructure. `http://www.camargus.com/` [Online. Last accessed: October 2013].

[18] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, August 2008.

[19] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 23(3), 2007.

[20] K. Chen and C. Lei. Network game design: hints and implications of player interaction. In *Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games NetGames '06*, 2006.

[21] G. Chrysos. Intel Xeon Phi coprocessor (codename Knights Corner). In *Proceedings of the 24th Hot Chips Symposium - HotChip 2012*.

[22] H. S. Chu. Building a simple yet powerful MMO game architecture. `http://www.ibm.com/developerworks/architecture/library/ar-powerup1/` [Online. Last accessed: June 2010], 2008.

[23] Cisco Systems Inc. Visual Networking Index. `http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html` [Online. Last accessed: June 2010], 2010.

[24] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review*, 20(4):200–208, 1990.

[25] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[26] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell BE Architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

[27] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2004.

[28] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[29] P. Dizikes. Sports analytics: a real game-changer. `http://web.mit.edu/newsoffice/2013/sloan-sports-analytics-conference-2013-0304.html` [Online. Last accessed: October 2013], 2013.

[30] B. Drain. EVE Evolved: EVE Online's server model. `http://massively.joystiq.com/2008/09/28/eve-evolved-eve-onlines-server-model/` [Online. Last accessed: June 2010], 2008.

[31] H. Espeland, P. B. Beskow, H. K. Stensland, P. N. Olsen, S. Kristoffersen, C. Griwodz, and P. Halvorsen. P2G: A Framework for Distributed Real-Time Processing of Multimedia Data. In *Proceedings of the 40th International Conference on Parallel Processing Workshops - ICPPW 2011*, pages 416–426. IEEE, 2011.

[32] H. Espeland, C. H. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen. Transparent protocol translation for streaming. In *Proceedings of the 15th ACM International Conference on Multimedia - MM '07*, pages 771–774. ACM, 2007.

[33] H. Espeland, C. H. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen. Transparent protocol translation and load balancing on a network processor in a media streaming scenario. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08*, pages 129–130. ACM, 2008.

[34] H. Espeland, H. K. Stensland, D. H. Finstad, and P. Halvorsen. Reducing Processing Demands for Multi-Rate Video Encoding. *International Journal of Multimedia Data Engineering and Management*, 3(2):1–19, 2012.

[35] D. H. Finstad, H. K. Stensland, H. Espeland, and P. Halvorsen. Improved Multi-Rate Video Encoding. In *Proceedings of the IEEE International Symposium on Multimedia - ISM 2011*, pages 293–300, 2011.

[36] J. A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *Proceedings of the 10th annual International Symposium on Computer Architecture - ISCA '83*, pages 140–150. ACM, 1983.

[37] S. L. Flosi. comScore Releases April 2010 U.S. Online Video Rankings, 2010. Press Release.

[38] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[39] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[40] V. R. Gaddam, R. Langseth, H. K. Stensland, P. Gurdjos, V. Charvillat, C. Griwodz, D. Johansen, and P. Halvorsen. Be your own cameraman: real-time support for zooming and panning into stored and live panoramic video. In *Proceedings of the 5th Annual ACM Conference on Multimedia Systems - MMSys 2014*, pages 168–171. ACM, 2014.

[41] B. Gedik, H. Andrade, K. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data - SIGMOD 2008*, pages 1123–1134. ACM, 2008.

[42] F. Glover. Tabu search Part II. *ORSA journal on Computing*, 2(1):4–32, 1990.

[43] N. Goodnight and G. Humphreys. Computation on Programmable Graphics Hardware. *IEEE Computer Graphics and Applications*, 25(5):12–15, 2005.

[44] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS-XII*, pages 151–162. ACM, 2006.

[45] C. Griwodz and M. Zink. KOM(S) Streaming System. `http://komssys.sourceforge.net/html/index.html` [Online. Last accessed: September 2014], 2001.

[46] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen. BAGADUS: An Integrated System for Arena Sports Analytics A Soccer Case Study. In *Proceedings of the 4th Annual ACM Conference on Multimedia Systems - MMSys 2013*, pages 48–59, 2013.

[47] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.

[48] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.

[49] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269. ACM, 2008.

[50] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *ORSA Journal on Computing*, 26(12):1519–1534, 2000.

[51] M. Houston. Anatomy of AMD's TeraScale Graphics Engine, 2008. Slides of a talk given at SIGGRAPH 2008.

[52] P. H. J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pages 12:1–12:55, 2007.

[53] K. Huguenin, A. Kermarrec, K. Kloudas, and F. Taiani. Content and Geographical Locality in User-Generated Content Sharing Systems. In *Proceedings of the 22nd International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV 2012*, pages 77–82, 2012.

[54] IBM. Cell Broadband Engine Architecture. Technical report, 2007.

[55] IBM, Sony, and Toshiba. *Cell Broadband Engine Programming Handbook*. IBM, 2008.

[56] Intel Corporation. Intel IXP1200 Network Processor Family: Hardware Reference Manual. Technical report, 2001.

[57] Intel Corporation. Intel IXP2400 Network Processor Family: Hardware Reference Manual. Technical report, 2003.

[58] Intel Corporation. Tick Tock Model. `http://www.intel.com/content/www/us/en/silicon-innovations/intel-tick-tock-model-general.html` [Online. Last accessed: February 2014], 2007.

[59] Intel Corporation. An Introduction to the Intel QuickPath Interconnect. Technical report, 2009.

[60] Intel Corporation. Intel Hyper-Threading Technology. `http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html` [Online. Last accessed: January 2012], 2012.

[61] Intel Corporation. Intel Quick Sync Video. `http://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html` [Online. Last accessed: January 2014], 2013.

[62] Interplay Sports. The Ultimate Video Analysis and Scouting Software. `http://www.interplay-sports.com/` [Online. Last accessed: October 2013], 2013.

[63] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 59–72. ACM, 2007.

[64] ISO/IEC 14496-10:2003. *Information Technology - Coding of audio-visual objects - Part 10: Advanced Video Coding*. ISO/IEC, 2003.

[65] ITU-T Z.100. *Specification and Description Language (SDL)*. ITU, 2007.

[66] D. Johansen, T. Endestad, H. Riiser, C. Griwidz, P. Halvorsen, H. Johansen, T. Aarflot, J. Hurley, Å. Kvalnes, C. Gurrin, S. Zav, B. Olstad, and E. Aaberg. DAVVI: a prototype for the next generation multimedia entertainment platform. In *Proceedings of the 17th ACM International Conference on Multimedia - MM '09*, pages 989–990. ACM, 2009.

[67] D. Johansen, M. Stenhaug, R. B. A. Hansen, A. Christensen, and P. M. Høgmo. Muithu: Smaller Footprint, Potentially Larger Imprint. In *Proceedings of the 7th IEEE International Conference on Digital Information Management - ICDIM 2012*, pages 205–214, 2012.

[68] P. Kaewtrakulpong and R. Bowden. An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection. In *Proceedings of the 2nd European Workshop on Advanced Video Based Surveillance Systems - AVBS '01*, pages 135–144, 2001.

[69] D. B. Kirk and W. H. Wen-mei. *Programming massively parallel processors: a hands-on approach.* Newnes, 2012.

[70] E. A. Kock, G. Essink, W. J. M. Smits, and P. Wolf. YAPI: Application modeling for signal processing systems. In *Proceeding of the 37th Design Automation Conference - DAC'2000*, pages 402–405. ACM, 2000.

[71] L. Kohn and N. Margulis. Introducing the Intel i860 64-bit microprocessor. *IEEE Micro*, 9(4):15–30, 1989.

[72] M. Kovac and N. Ranganathan. JAGUAR: A Fully Pipelined VLSI Architecture for JPEG Image Compression Standard. *Proceedings of the IEEE*, 83(2):247–258, 1995.

[73] I. Kuon, R. Tessier, and J. Rose. FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2007.

[74] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[75] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[76] C. H. Lunde, H. Espeland, H. K. Stensland, and P. Halvorsen. Improving file tree traversal performance by scheduling I/O operations in user space. In *Proceedings of the 28th IEEE International Performance Computing and Communications Conference - IPCCC 2009*, pages 145–152. IEEE, 2009.

[77] C. H. Lunde, H. Espeland, H. K. Stensland, A. Petlund, and P. Halvorsen. Improving disk I/O performance on Linux. In *UpTimes - Proceedings of Linux-Kongress and OpenSolaris Developer Conference*, 2009.

[78] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip Terascale Processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis - SC 2008*, pages 1–11. IEEE, 2008.

[79] C. Mims. Why CPUs Aren't Getting Any Faster. `http://www.technologyreview.com/view/421186/why-cpus-arent-getting-any-faster/` [Online. Last accessed: July 2014], 2010.

[80] A. Mortensen, V. R. Gaddam, H. K. Stensland, C. Griwodz, D. Johansen, and P. Halvorsen. Automatic event extraction and video summaries from soccer games. In *Proceedings of the 5th Annual ACM Conference on Multimedia Systems - MMSys 2014*, pages 176–179. ACM, 2014.

[81] Move Networks. Internet Television: Challenges and Opportunities. Technical report, Move Networks, Inc., 2008.

[82] Netronome. FlowNICs. `http://www.netronome.com/product/flownics/` [Online. Last accessed: July 2014], 2014.

[83] O. A. Niamut, R. Kaiser, G. Kienast, A. Kochale, J. Spille, O. Schreer, J. R. Hidalgo, J. Macq, and B. Shirley. Towards A Format-agnostic Approach for Production, Delivery and Rendering of Immersive Media. In *Proceedings of the 4th Annual ACM Conference on Multimedia Systems - MMSys 2013*, pages 249–260, 2013.

[84] J. Nickolls and W. J. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56–69, 2010.

[85] C. Nicolaou. An architecture for real-time multimedia communication systems. *IEEE Journal on Selected Areas in Communications*, 8(3):391–400, 1990.

[86] Numascale. NumaConnect - the Technology. `https://numascale.com/` [Online. Last accessed: September 2014], 2012.

[87] Nvidia. CUDA C Best Practices Guide. `http://docs.nvidia.com/pdf/CUDA_C_Best_Practices_Guide.pdf` [Online. Last accessed: September 2014], 2009.

[88] Nvidia. Nvidia's Next Generation CUDA Compute Architecture: Fermi. Technical report, 2010.

[89] Nvidia. NVIDIA Performance Primitives. `https://developer.nvidia.com/npp` [Online. Last accessed: March 2012], 2011.

[90] Nvidia. Nvidia's Next Generation CUDA Compute Architecture: Kepler GK110. Technical report, 2012.

[91] Nvidia. Tegra 4 Processors. `http://www.nvidia.com/object/tegra-4-processor.html` [Online. Last accessed: December 2013], 2013.

[92] Nvidia. CUDA C Programming Guide, version 6.0. `http://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf` [Online. Last accessed: September 2014], 2014.

[93] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data - SIGMOD 2008*, pages 1099–1110. ACM, 2008.

[94] A. Ottesen. Efficient parallelisation techniques for applications running on GPUs using the CUDA framework. Master thesis, University of Oslo, Norway, 2009.

[95] J. Ozer. First Look: H.264 and VP8 Compared. `http://www.streamingmedia.com/articles/editorial/featured-articles/first-look-h.264-and-vp8-compared-67266.aspx` [Online. Last accessed: June 2010], 2010.

[96] R. Pantos, J. Batson, D. Biderman, B. May, and A. Tseng. HTTP Live Streaming. `http://tools.ietf.org/html/draft-pantos-http-live-streaming-04` [Online. Last accessed: October 2013], 2010.

[97] S. A. Pettersen, P. Halvorsen, D. Johansen, H. Johansen, V. Berg-Johansen, V. R. Gaddam, A. Mortensen, R. Langseth, C. Griwodz, and H. K. Stensland. Soccer video and player position dataset. In *Proceedings of the 5th Annual ACM Conference on Multimedia Systems - MMSys 2014*, pages 18–23. ACM, 2014.

[98] V. Pham, P. Vo, and V. T. Hung. GPU implementation of extended gaussian mixture model for background subtraction. In *Proceedings of the IEEE RIVF International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future - RIVF 2010*, pages 1–4. IEEE, 2010.

[99] M. Pritchard. How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It. `http://www.gamasutra.com/view/feature/3149/how_to_hurt_the_hackers_the_scoop_.php` [Online. Last accessed: May 2009], 2000.

[100] Prozone. Prozone Sports – Introducing Prozone Performance Analysis Products. `http://www.prozonesports.com/subsector/football/` [Online. Last accessed: October 2014].

[101] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz. A demonstration of a lockless, relaxed atomicity state parallel game server (LEARS). In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games - NetGames 2011*, pages 1–3. IEEE, 2011.

[102] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz. LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition. In *Proceedings of the 41st International Conference on Parallel Processing Workshops - ICPPW 2012*, pages 382–389. IEEE, 2012.

[103] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceesing of the IEEE 13th International Symposium on High Performance Computer Architecture - HPCA 2007*, pages 13–24. IEEE, 2007.

[104] Real World Technologies. Intels Haswell CPU Microarchitecture. `http://www.realworldtech.com/haswell-cpu/` [Online. Last accessed: February 2014], 2012.

[105] S. Sægrov, A. Eichhorn, J. Emerslund, H. K. Stensland, C. Griwodz, D. Johansen, and P. Halvorsen. BAGADUS: An Integrated System for Soccer Analysis (Demo). In *Proceedings of the ACM/IEEE International Conference on Distributed Smart Cameras - ICDSC 2012*, 2012.

[106] V. D. Salvo, A. Collins, B. McNeill, and M. Cardinale. Validation of Prozone: A new video-based performance analysis system. *International Journal of Performance Analysis in Sport (serial online)*, 6(1):108–119, 2006.

[107] T. Sato. The earth simulator: Roles and impacts. *Nuclear Physics B - Proceedings Supplements*, 129-130:102–108, 2004.

[108] P. Seeling, F. H. P. Fitzek, G. Ertli, A. Pulipaka, and M. Reisslein. Video network traffic and quality comparison of VP8 and H.264 SVC. In *Proceedings of the 3rd Workshop on Mobile Video Delivery - MoViD 2010*, pages 33–38, 2010.

[109] L. Seiler, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, and J. Sugerman. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):18:1–18:15, 2008.

[110] Iraj Sodagar. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE MultiMedia*, 18(4):62–67, 2011.

[111] Stats Technology. STATS — SportVU — Football/Soccer. `http://www.sportvu.com/football.asp` [Online. Last accessed: October 2013], 2013.

[112] H. K. Stensland, H. Espeland, C. Griwodz, and P. Halvorsen. Tips, tricks and troubles: optimizing for Cell and GPU. In *Proceedings of the 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '10*, pages 75–80. ACM, 2010.

[113] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, C. Griwodz, P. Halvorsen, and D. Johansen. Processing Panorama Video in Real-Time. *International Journal of Semantic Computing (IJSC)*, 8(2):209–227, 2014.

[114] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, Ø. Landsverk, C. Griwodz, P. Halvorsen, M. Stenhaug, and D. Johansen. Bagadus: An integrated real-time system for soccer analytics. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 10(1s):1–21, 2014.

[115] H. K. Stensland, C. Griwodz, and P. Halvorsen. Evaluation of multi-core scheduling mechanisms for heterogeneous processing architectures. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08*, pages 33–38. ACM, 2008.

[116] H. K. Stensland, O. Lysne, R. Nordstrøm, and H. Kohmann. Making an SCI fabric dynamically fault tolerant. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing - IPDPS 2008*, pages 1–8. IEEE, 2008.

[117] H. K. Stensland, M. Ø. Myrseth, C. Griwodz, and P. Halvorsen. Cheat detection processing: A GPU versus CPU comparison. In *Proceedings of the 9th IEEE Annual Workshop on Network and Systems Support for Games - NetGames 2010*, pages 1–6. IEEE, 2010.

[118] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[119] M. Tennøe, E. O. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, V. R. Gaddam, D. Johansen, C. Griwodz, and P. Halvorsen. Efficient Implementation and Processing of a Real-Time Panorama Video Pipeline. In *Proceedings of the IEEE International Symposium on Multimedia - ISM 2013*, pages 76–83. IEEE, 2013.

[120] M. Tennøe, E. O. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, P. Halvorsen, and C. Griwidz. Realtime Panorama Video Processing Using NVIDIA GPUs, 2013. Poster presented at Nvidia GPU Technology Conference 2013.

[121] The TCPdump and libpcap project. `http://www.tcpdump.org/` [Online. Last accessed: May 2013], 2013.

[122] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369. IEEE, 2007.

[123] M. Thompson and A. Pimentel. Towards Multi-application Workload Modeling in Sesame for System-Level Design Space Exploration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 222–232. Springer Berlin / Heidelberg, 2007.

[124] S. V. Valvåg and D. Johansen. Oivos: Simple and Efficient Distributed Data Processing. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications - HPCC'08*, pages 113–122, 2008.

[125] S. V. Valvåg and D. Johansen. Cogset: A Unified Engine for Reliable Storage and Parallel Processing. In *Proceeding of the 6th IFIP International Conference on Network and Parallel Computing - NPC'08*, pages 174–181, 2009.

[126] Verdione Project. Verdione - Technology for mixed-reality stages. `http://verdione.org/` [Online. Last accessed: October 2014], 2011.

[127] Z. Vrba, P. Halvorsen, C. Griwodz, and P. B. Beskow. Kahn Process Networks are a Flexible Alternative to MapReduce. *Proceeding of the IEEE International Conference on High Performance Computing and Communications - HPCC 2009*, pages 154–162, 2009.

[128] Z. Vrba, P. Halvorsen, C. Griwodz, P. B. Beskow, and D. Johansen. The Nornir Runtime System for Parallel Programs Using Kahn Process Networks. In *Proceedings of the 6th IFIP International Conference on Network and Parallel Computing - ICNPC 2009*, pages 1–8. IEEE, 2009.

[129] W. Weimer, M. Boyer, and K. Skadron. Automated Dynamic Analysis of CUDA Programs. In *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems - STMCS 2008*, 2008.

[130] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proceedings of the IEEE International Conference on Management of Data - SIGMOD 2007*, pages 31–42, 2007.

[131] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, 2003.

[132] M. A. Wilhelmsen, H. K. Stensland, V. R. Gaddam, P. Halvorsen, and C. Griwodz. Performance and Application of the NVIDIA NVENC H.264 Encoder, 2014. Poster presented at Nvidia GPU Technology Conference 2014.

[133] E. Wu and Y. Liu. Emerging technology about GPGPU. In *Proceedings of the IEEE Asia Pacific Conference on Circuits and Systems - APCCAS 2008*, pages 618–622. IEEE, 2008.

[134] N. Wu, M. Wen, W. Wu, J. Ren, H. Su, C. Xun, and C. Zhang. Streaming HD H.264 encoder on programmable processors. In *Proceedings of the 17th ACM International Conference on Multimedia - MM '09*, pages 371–380. ACM, 2009.

[135] Xilinx. Zynq-7000 All Programmable SoC First Generation Architecture. Technical report, 2013.

[136] Y. Xiong and K. Pulli. Color correction for mobile panorama imaging. In *Proceedings of the First International Conference on Internet Multimedia Computing and Service - ICIMCS '09*, pages 219–226, 2009.

[137] J. Yan and B. Randell. A systematic classification of cheating in online games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system Support for Games - NetGames '05*, pages 1–9, 2005.

[138] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data - SIGMOD 2007*, pages 1029–1040. ACM, 2007.

[139] YouTube. Statistics - YouTube. `https://www.youtube.com/yt/press/statistics.html` [Online. Last accessed: October 2014], 2014.

[140] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language, 2008.

[141] A. Zambelli. Smooth Streaming Technical Overview. `http://www.iis.net/learn/media/on-demand-smooth-streaming/smooth-streaming-technical-overview` [Online. Last accessed: October 2014], 2009.

[142] T. Zeiser, G. Hager, and G. Wellein. The world's fastest CPU and SMP node: Some performance results from the NEC SX-9. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium - IPDPS 2009*, pages 1–8. IEEE, 2009.

[143] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *Proceeding of the IEEE International Conference on Computer Vision - ICCV 1999*, pages 666–673, 1999.

[144] Z. Zivkovic. Improved adaptive Gaussian mixture model for background subtraction. In *Proceedings of the 17th IEEE International Conference on Pattern Recognition - ICPR 2004*, pages 28 – 31 Vol.2, 2004.

[145] Z. Zivkovic and F. van der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recognition Letters*, 27(7):773–780, 2006.

[146] ZXY. ZXY Sport Tracking. `http://www.zxy.no/` [Online. Last accessed: January 2014], 2013.

Part II

Research Papers

Paper I: Transparent Protocol Translation for Streaming

Transparent Protocol Translation for Streaming

Håvard Espeland[1], Carl Henrik Lunde[1], Håkon Kvale Stensland[1,2], Carsten Griwodz[1,2],
Pål Halvorsen[1,2]

[1]IFI, University of Oslo, Norway   [2]Simula Research Laboratory, Norway

{haavares, chlunde, haakonks, griff, paalh}@ifi.uio.no

ABSTRACT

The transport of streaming media data over TCP is hindered by TCP's probing behavior that results in the rapid reduction and slow recovery of the packet rates. On the other side, UDP has been criticized for being unfair against TCP connections, and it is therefore often blocked out in the access networks. In this paper, we try to benefit from a combined approach using a proxy that transparently performs transport protocol translation. We translate HTTP requests by the client transparently into RTSP requests, and translate the corresponding RTP/UDP/AVP stream into the corresponding HTTP response. This enables the server to use UDP on the server side and TCP on the client side. This is beneficial for the server side that scales to a higher load when it doesn't have to deal with TCP. On the client side, streaming over TCP has the advantage that connections can be established from the client side, and data streams are passed through firewalls. Preliminary tests demonstrate that our protocol translation delivers a smoother stream compared to HTTP-streaming where the TCP bandwidth oscillates heavily.

Categories and Subject Descriptors

D.4.4 [**OPERATING SYSTEMS**]: Communications Management—*Network communication*

General Terms

Measurement, Performance

1. INTRODUCTION

Streaming services are today almost everywhere available. Major newspapers and TV stations make on-demand and live audio/video (A/V) content available, video-on-demand services are becoming common and even personal media are frequently streamed using services like pod-casting or uploading to streaming sites such as YouTube.

The discussion about the best protocols for streaming has been going on for years. Initially, streaming services on the Internet used UDP for data transfer because multimedia applications often have demands for bandwidth, reliability and jitter than could not be offered by TCP. Today, this approach is impeded with filters in Internet service providers (ISPs), by firewalls in access networks and on end-systems. ISPs reject UDP because it is not fair against TCP traffic, many firewalls reject UDP because it is connectionless and requires too much processing power and memory to ensure security. It is therefore fairly common to use HTTP-streaming, which delivers streaming media over TCP. The disadvantage is that the end-user can experience playback hiccups and quality reductions because of the probing behavior of TCP, leading to oscillating throughput and slow recovery of the packet rate. A sender that uses UDP would, in contrast to this, be able to maintain a desired constant sending rate. Servers are also expected to scale more easily when sending smooth UDP streams and avoid dealing with TCP-related processing.

To explore the benefits of both TCP and UDP, we experiment with a proxy that performs a transparent protocol translation. This is similar to the use of proxy caching that ISPs employ to reduce their bandwidth, and we do in fact aim at a combined solution. There are, however, too many different sources for adaptive streaming media that end-users can retrieve data from to apply proxy caching for all of them. Instead, we aim at live protocol translation in a TCP-friendly manner that achieves a high perceived quality to end-users. Our prototype proxy is implemented on an Intel IXP2400 network processor and enables the server to use UDP at the server side and TCP at the client side.

We have earlier shown the benefits of combining the use of TFRC in the backbone with the use of TCP in access networks [1]. In the experiments presented in that paper, we used course-grained scalable video (scalable MPEG (SPEG) [4]) which makes it possible to adapt to variations in the packet rate. To follow up on this idea, we describe in this paper our IXP2400 implementation of a dynamic transport protocol translator. Preliminary tests comparing HTTP video streaming from a web-server and RTSP/RTP-streaming from the komssys video server show that, in case of some loss, our solution using a UDP server and a proxy later translating to TCP delivers a smoother stream at play out rate while the TCP stream oscillates heavily.

2. RELATED WORK

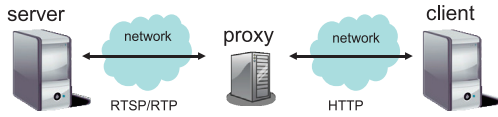Proxy servers have been used for improved delivery of

Figure 1: System overview

streaming media in numerous earlier works. Their tasks include caching, multicast, filtering, transcoding, traffic shaping and prioritizing. In this paper, we want to draw attention to issues that occur when a proxy is used to translate transport protocols in such a way that TCP-friendly transports mechanisms can be used in backbone networks and TCP can be used in access networks to deliver streaming video through firewalls. Krasic et al. argue that the most natural choice for TCP-friendly traffic is using TCP itself [3]. While we agree in principle, their priority progress streaming approach requires a large amount of buffering to hide TCP throughput variations. In particular, this smoothing buffer is required to hide the rate-halving and recovery time in TCP's normal approach of probing for bandwidth which grows proportionally with the round-trip time. To avoid this large buffering requirement at the proxy, we would prefer an approach that maintains a more stable packet rate at the original sender. The survey of [7] shows that TFRC is a reasonably good representative of the TCP-friendly mechanisms for unicast communication. Therefore, we have chosen this mechanism for the following investigation.

With respect to the protocol translation that we describe here, we do not know of much existing work, but the idea is similar to the multicast-to-unicast translation [6]. We have also seen voice-over-IP proxies translating between UDP and TCP. In these examples, a packet is translated from one type to another to match the various parts of the system, and we here look at how such an operation performs in the media streaming scenario.

3. TRANSLATING PROXY

An overview of our protocol translating proxy is shown in figure 1. The client and server communicates by the proxy, which transparently translates between HTTP and RTSP/RTP. Both peers are unaware of each other.

The steps and phases of a streaming session follows. The client tries to set up a HTTP streaming session, by initiating a TCP connection to the server. All packets are intercepted by the proxy, and modified before passing it on to the streaming server. The proxy also forwards the TCP 3-way handshake between client and server, updating the packet with the server's port. When established, the proxy splits the TCP connection into two separate connections that allow for individual updating of sequence numbers. The client sends a GET request for a video file. The proxy translates this into a SETUP request and sends it to the streaming server using the TCP port of the client as its proposed RTP/UDP port. If the setup is unsuccessful, the proxy will inform the client and close the connections. Otherwise, the server's response contains the confirmed RTP and RTCP ports assigned to a streaming session. The proxy sends a response with an unknown content length to the client and issues a PLAY command to the server. When received, the server starts streaming the video file using RTP/UDP. The UDP packets are translated by the proxy as part of the
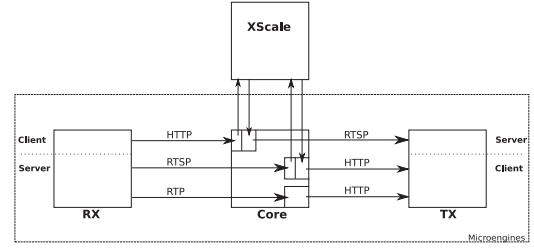

Figure 2: Packet flow on the IXP2400

HTTP response, using the source port and address matching the HTTP connection. Because the RTP and UDP headers combined are longer than a standard TCP header, the proxy can avoid the penalty of moving the video data in memory, thus permitting reuse of the same packet by padding the TCP options field with NOPs. When the connection is closed by the client during or after playback, the proxy issues a TEARDOWN request to the server to avoid flooding the network with excess RTP packets.

4. IMPLEMENTATION

Our prototype is implemented on a programmable network processor using the IXP2400 chipset [5], which is designed to handle a wide range of access, edge and core applications. The basic features include a 600 MHz XScale core running Linux, eight 600 MHz special packet processors called micro-engines (μEngines), several types of memory and different controllers and busses. With respect to the different CPUs, the XScale is typically used for the control plane (slow path) while μEngines perform general packet processing in the data plane (fast path).

The transport protocol translation operation[1] is shown in figure 2. The protocol translation proxy uses the XScale core and one μEngine application block. In addition, we use two μEngines for the receiving (RX) and the sending (TX) blocks. Incoming packets are classified by the μEngine based on the header. RTSP and HTTP packets are enqueued for processing on the XScale core (control path) while the handling of RTP packets is performed on the μEngine (fast path). TCP acknowledgements with zero payload size are processed on the μEngine for performance reasons.

The main task of the XScale is to set up and maintain streaming sessions, but after the initialization, all video data is processed (translated and forwarded) by the μEngine. The proxy supports a partial TCP/IP implementation, covering only basic features. This is done to save both time and resources on the proxy.

To be fair with competing TCP streams, we implemented congestion control for the client loss experiment. TFRC [2] computation is used to determine the bandwidth available for streaming from the server. TFRC is a specification for best effort flows competing for bandwidth, designed to be reasonable fair to other TCP flows. The outgoing bandwidth is limited by the following formula:

$$X = \frac{s}{R * \sqrt{2 * b * \frac{p}{3}} + (t_{RTO} * 3 * \sqrt{3 * b * \frac{p}{8}} * p * (1 + 32 * p^2))}$$

---

[1] Our proxy also performs proxying of normal RTSP sessions and transparent load balancing between streaming servers, but this is outside of the scope of this paper. We also have unused resources (μEngines) enabling more functionality.
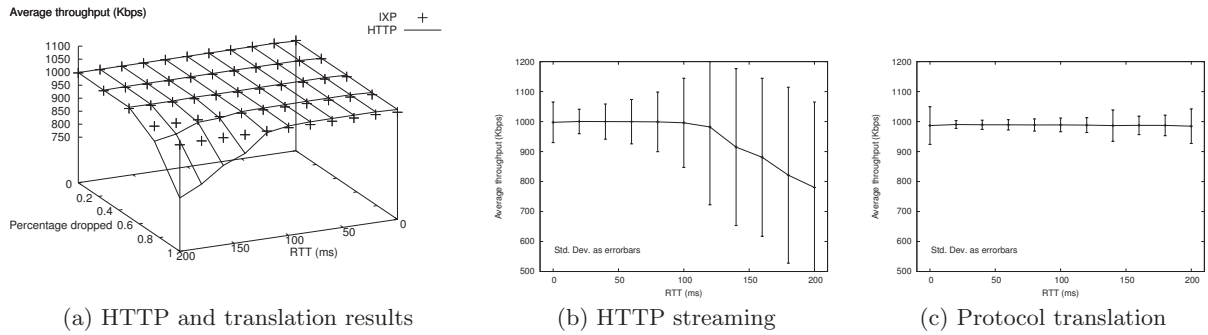
(a) HTTP and translation results   (b) HTTP streaming   (c) Protocol translation

Figure 3: Achieved bandwidth varying drop rate and link latency with 1% server-proxy loss

where X is the transmit rate in bytes per second, s is the packet size in bytes, R is the RTT in seconds, b is the number of packets ACKed by a single TCP acknowledgment, p is the loss event rate (0-1.0), and t_{RTO} is the TCP retransmission timeout. The formula is calculated on a μEngine using fixed point arithmetic. Packets arriving at a rate exceeding the TFRC calculated threshold are dropped.

We are aware that this kind of dropping has different effects on the user-perceived quality than sender-side adaptation. We have only made preliminary investigations on the matter and leave it for future work. In that investigation, we will also consider the effect of buffering for at most 1 RTT.

5. EXPERIMENTS AND RESULTS

We investigated the performance of our protocol translation proxy compared to plain HTTP-streaming in two different settings. In the first experiment, we induced unreliable network behavior between the streaming server and the proxy, while in the second experiment, the unreliable network connected proxy and client. We performed several experiments where we examined both the bandwidth and the delay while changing both the link delays (0 - 200 ms) and the packet drop rate (0 - 1 %). We used a web-server and an RTSP video server using RTP streaming, running on a standard Linux machine. Packets belonging to end-to-end HTTP connections made to port 8080 were forwarded by the proxy whereas packets belonging to sessions initiated by connection made to port 80 were translated. The bandwidth was measured on the client by monitoring the packet stream with tcpdump.

5.1 Server-Proxy Losses

The results from the test where we introduced loss and delay between server and proxy are shown in figure 3. Figure 3(a) shows a 3D plot where we look at the latency that we achieved for the different combinations of loss and link delays. Additionally, figures 3(b) and 3(c) show the respective results for the HTTP and protocol translation scenarios when keeping the loss rate constant at 1% (keeping the link delay constant gives similar results). The plots show that our proxy that translates transparently from RTP/UDP to TCP achieves a mostly constant rate for the delivered stream. Sending the HTTP stream from the server, on the other hand, shows large performance drops when the loss rate and the link delay increase. From figures 3(b) and 3(c),
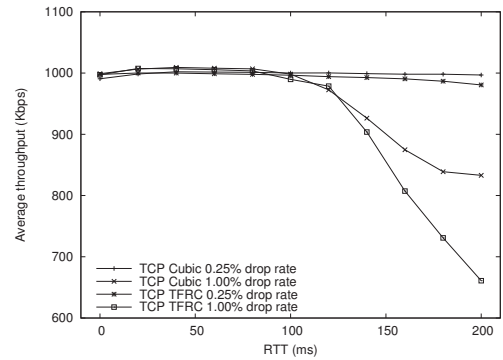

Figure 4: TCP cubic congestion vs. TFRC

we see also that the translation provides a smoother stream whereas the bandwidth oscillates heavily using TCP end-to-end.

5.2 Proxy-Client Losses

In the second experiment, loss and delay are introduced between the proxy and the client, and the data rate is limited according to TFRC measuring RTT and packet loss during the transfer. Furthermore, packets are not buffered on the network card, meaning that the traffic exceeding the calculated rate of TFRC are dropped and that TCP retransmissions contains only data with zero values.

In figure 4, we first show the average throughput of streaming video from a web-server using cubic TCP congestion control compared with our TCP implementation using TFRC. As expected, the TFRC implementation behaves similar (fair) to normal TCP congestion control with a slightly more pessimistic approach. Moreover, figure 5 is a plot of the received packets' interarrival time. This shows that the delay variation of normal TCP congestion control increases with the drop rate, while TFRC is less affected. Thus, we see again that that our proxy gives a stream without large variations whereas the bandwidth oscillates heavily using TCP throughout the path.

6. DISCUSSION

Even though our proxy seems to give better, more stable bandwidths, there is a trade-off, because instead of retransmitting lost packet (and thus old data if the client does not buffer), the proxy fills the new packet with new updated data from the server. This means that the client in our pro-
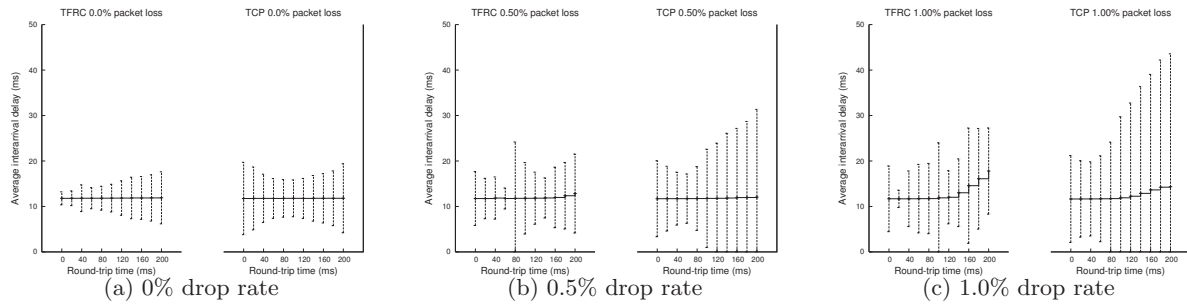
773

Figure 5: Average interarrival delay and variation with proxy-client loss

totype does not receive all data, and some artifacts may be displayed. On the other hand, in case of live and interactive streaming scenarios, delays due to retransmission may introduce dropped frames and delayed play out. This can cause video artifacts, depending on the codec used. However, this problem can easily be reduced by adding a limited buffer per stream sufficient for one retransmission on the proxy.

One issue in the context of proxies is where and how it should be implemented. For this study, we have chosen the IXP2400 platform as we earlier have explored the offloading capabilities of such programmable network processors. Using such an architecture, the network processor is suited for many similar operations, and the host computer could manage the caching and persistent storage of highly popular data served from the proxy itself. However, the idea itself could also be implemented as a user-level proxy application or integrated into the kernel of an intermediate node performing packet forwarding.

The main advantage of the scheme proposed in this paper is a lower variation in bandwidth and interarrival times in an unreliable network compared to normal TCP. It also combines some of the benefits of HTTP streaming (firewall traversal, client player support) with the performance of RTP streaming. The price of this is uncontrolled loss of data packets that may impact the perceived video quality more strongly than hiccups.

HTTP streaming may perform well in a scenario where a stored multimedia object is streamed to a high capacity end-system. Here, a large buffer may add a small, but acceptable, delay to conceal losses and oscillating resource availability. However, in the case where the receiver is a small device like a mobile phone or a PDA with a limited amount of resources, or in an interactive scenario like conferencing applications where there is no time to buffer, our protocol translation mechanisms could be very useful.

The server-proxy losses test can be related to a case where the camera on a mobile phone is used for streaming. Mobile devices are usually connected to unreliable networks with high RTT. The proxy-client losses test can be related to a traditional video conference scenario.

In the experiment, we compare a normal web-server streaming video with a RTP server (komssys) to a client by encapsulating the video data in HTTP packets on a IXP network card close to the video server. The former setup runs a simple web-server on Linux, limiting the average bandwidth from user-space to the video's bit rate.

Using RTP/UDP from the server through the backbone to a proxy is also an advantage for the resource utilization. RTP/UDP packets reduce memory usage, CPU usage and

overhead in the network compared to TCP. This combined with the possibility of sending a single RTP/UDP stream to the proxy, and make the proxy do separation and adaptation of the stream to each client can reduce the load in the backbone. Therefore the proxy should be placed as close to the clients as possible, e.g. in the ISP's access network, or in a mobile provider's network.

7. CONCLUSION

Both TCP and UDP have their strengths and weaknesses. In this paper, we use a proxy that performs transparent protocol translation to utilize the strengths of both protocols in a streaming scenario. It enables the server to use UDP on the server side and TCP on the client side. The server gains scalability by not having to deal with TCP processing. On the client side, the TCP stream is not discarded and passes through firewalls. The experimental results show that our protocol transparent proxy achieves translation and delivers smoother streaming than HTTP-streaming.

8. REFERENCES

[1] GRIWODZ, C., FIKSDAL, S., AND HALVORSEN, P. Translating scalable video streams from wide-area to access networks. *Campus Wide Information Systems 21*, 5 (2004), 205–210.

[2] HANDLEY, M., FLOYD, S., PADHYE, J., AND WIDMER, J. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448 (Proposed Standard), Jan. 2003.

[3] KRASIC, B., AND WALPOLE, J. Priority-progress streaming for quality-adaptive multimedia. In *Proceedings of the ACM Multimedia Doctoral Symposium* (Oct. 2001).

[4] KRASIC, C., WALPOLE, J., AND FENG, W.-C. Quality-adaptive media streaming by priority drop. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (2003), pp. 112–121.

[5] INTEL CORPORATION. Intel IXP2400 network processor datasheet, Feb. 2004.

[6] PARNES, P., SYNNES, K., AND SCHEFSTRÖM, D. Lightweight application level multicast tunneling using mtunnel. *Computer Communication 21*, 515 (1998), 1295–1301.

[7] WIDMER, J., DENDA, R., AND MAUVE, M. A survey on TCP-friendly congestion control. *Special Issue of the IEEE Network Magazine "Control of Best Effort Traffic" 15* (Feb. 2001), 28–37.

Paper II: Evaluation of Multi-Core Scheduling Mechanisms for Heterogeneous Processing Architectures

Evaluation of Multi-Core Scheduling Mechanisms for Heterogeneous Processing Architectures

Håkon Kvale Stensland[1], Carsten Griwodz[1,2], Pål Halvorsen[1,2]
[1]Simula Research Laboratory, Norway
[2]Department of Informatics, University of Oslo, Norway
{haakonks, griff, paalh}@simula.no

ABSTRACT

General-purpose CPUs with multiple cores are established products, and new heterogeneous technology like the Cell broadband engine and general-purpose GPUs bring an even higher degree of true multi-processing into the market. However, means for utilizing the processing power is immature. Current tools typically assume that exclusive use of these resources is sufficient, but this assumption will soon be invalid because the interest in using their processing power for general-purpose tasks. Among the applications that can benefit from such technology is transcoding support for distributed media applications, where remote participants join and leave dynamically. Transcoding consists of several clearly separated processing operations that consume a lot of resources, such that individual processing units are unable to handle all operations of a session of arbitrary size. The individual operations can then be distributed over several processing units, and data must be moved between them according to the dependencies between operations. Many multi-processor scheduling approaches exist, but to the best of our knowledge, a challenge is still to find mechanisms that can schedule dynamic workloads of communicating operations while taking both the processing and communication requirements into account. For such applications, we believe that feasible scheduling can be performed in two levels, i.e., divided into the task of placing a job onto a processing unit and the task of multitasking time-slices within a single processing unit. We have implemented some simple high-level scheduling mechanisms and simulated a video conferencing scenario running on topologies inspired by existing systems from Intel, AMD, IBM and nVidia. Our results show the importance of using an efficient high-level scheduler.

1. INTRODUCTION

Multi-processor and multi-core systems are quickly becoming mainstream computing resources. Dual-core general-purpose CPUs are established products, but systems including the Cell broadband engine (BE) and general-purpose GPUs bring an even higher degree of true multi-processing into the market. Making use of this parallel processing capacity, however, is still in the earlier stages. Current tools that assist in developing for many-core systems like the Cell and GPUs assume typically that the parallel computing resources can be used exclusively for a single task. We expect that this will become an invalid assumption because the increasing commoditization of parallel processing hardware and an increasing interest in using it for general-purpose tasks. Among the applications that can benefit strongly from this hardware are computing-heavy media processing applications.

Examples of this are multi-party video conferences where participants dynamically join and leave the session, personalized video streaming services and free-viewpoint 3D environments. In the video conferencing scenario, multiple streams have to be merged, adapted, and so on. Contributors and consumers join and leave conferences at arbitrary times and use heterogeneous devices. The necessary media processing operations can then consume much processing and bandwidth resources. Each individual operation can possibly be handled by an individual core without missing any deadlines. However, individual processing units are unable to handle the operations that are necessary for a conference of arbitrary size. Operations must then be distributed over several processing units, and data must be moved between them, all while staying within the application-defined deadlines. The media processing applications are time-dependent and cyclic depending on individual contributors' frame and sampling rates. Because the increasing heterogeneity of end systems, the resource consumption of media processing operations varies widely, as does the amount of data that is needed for communication between processing steps. The challenge is to then find a system scheduler for such dynamically changing workloads that fulfills the processing requirements of the application and at the same time avoids congestion on the interconnects between the processing units. Many multi-processor scheduling approaches exist, but we are not aware of efforts that address the dependencies of long-lived, cyclic, inter-job dependent processing operations that require time-critical communication with neighboring processing units. Our envisioned applications require that a feasible scheduler takes both capacity of processing units and the bandwidth of interconnects between them into account. We believe that feasible scheduling can be performed in two levels, i.e., divided into the task of placing a job onto a processing unit and the task of multitasking time-slices within a single process-

ing unit. Our application provides a considerable amount of long-term knowledge for the long time-scale that can be fed to a high-level scheduler that can reserve resources within each timeslice, while the low-level scheduler for processing units and interconnects can act independently.

In this paper, we look at the interaction between interconnection topology and workload patterns using simple high-level scheduling mechanisms. The topologies are inspired by the interconnection topologies of the Opteron, the Core 2, the Cell BE and nVIDIA's latest GPUs. The workloads are modeled to reflect a video conferencing scenario where the size of each job is varied under the capacity limit of the single cores and links. Our results show the importance of using an efficient high-level scheduler in order to reduce the overall resource consumption. In particular, means to cluster jobs on cores in close proximity are promising because the bandwidth consumption is usually reduced.

The rest of this paper is organized as follows. The next section describes our example scenario. In section 3, we look at our hierarchical scheduling approach. We describe our simulator in section 4 and present some results in section 5. In section 6, we present some related work before we summarize and conclude in section 7.

2. EXAMPLE SCENARIO

Large-scale media processing applications are coming, and we use here video conferencing systems as a motivating example. Such conferences have multiple senders and receivers as for example possible today using equipment from vendors like Polycom, Tandberg or Lifesize. In this scenario, every conference participant should be able to receive A/V content from all the other participants. Only active participants will produce A/V data that must be processed, merged and sent to all the others. The setup allows a large degree of sharing in the processing graph, but requires expensive individual processing as well. Furthermore, depending on the equipment used by each participant, each stream may have different characteristics. Thus, we end up with a scenario similar to the one as depicted in figure 1 where we cannot assume equally sized jobs.
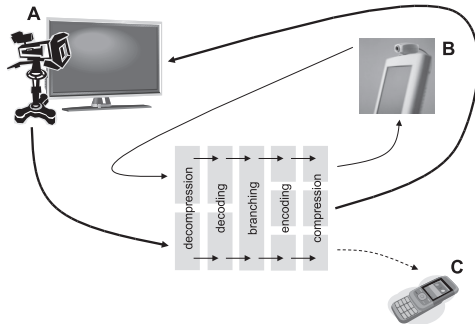

Figure 1: Example scenario: video conferencing

In the example in figure 1, the conference has three participants. The first (A) has equipment that produce and receive HD data. The second user (B) produce and receive SD data whereas the last participant (C) is only following (listening) the conference using a mobile phone displaying only video in CIF resolution. In this situation, each stream could be processed according to the layers in figure 1: 1) decompression, 2) decoding, 3) branching (e.g., to filter and

adapt the data to the available resources and the receiving devices), 4) encoding and 5) compression. In addition, the system must handle dynamics in terms of users joining or leaving the session, users becoming active and oscillating resource availability in other parts of the system. Finally, each system often manages many concurrent sessions further complicating the task of finding available resources.

The processing resources required for handling these real-time tasks often (at least when the conferences are large) exceed the processing capacity provided by a single processing unit. Consequently, the imposed workload must be distributed to a number of processing units for parallel execution, for example using multi-core processors, multi-chip computers, processing clusters or distributed nodes over the Internet. However, parallel processing of a large number of real-time tasks introduces challenges related to orchestrating and multiplexing the resource usage in a timely manner. We are therefore addressing the challenge of finding feasible dynamic schedulers for a given set of (possibly heterogeneous) processing units and their corresponding links (like busses or network links).

3. TWO-LEVEL SCHEDULING

In the application scenarios that we imagine here, long-lived dynamic sessions of media operations must be mapped to resources of interconnected processing units. The data flows that are processed are sent from remote machines and are subject to packet loss and jitter. We can therefore not assume that a fine-grained, detailed reservation mechanism at cycle granularity is reasonable for the scheduled processing units. Rather, long-term, high-level decisions must be made to allocate resources of processing units, such that they are available for arriving data with high likelihood. To separate this from accurate low-level scheduling that is concerned with the allocation of resource shares, we consider two-level scheduling as a reasonable option.

First, a high-level scheduler acting on the timescale of signaling protocols must select processing units for tasks in such a way that each task has its required resources and that it can communicate with those jobs from which it receives streaming data and with those to which it forwards streaming data. This scheduler must know profiling information of processing stages and know approximately the processing demands of the expected workload. It must then choose processing units taking both remaining processing capacity and remaining interconnect bandwidth into account. When the jobs have been placed on a processing unit with available capacity, a low-level algorithm working on the timescale of operating system time slices must handle the individual jobs on this unit in order to find a schedule where each job meets its deadline.

3.1 Mapping media operations

The first step of the scheduler in our scenario is to perform the mapping of media operations to the processing units. Here, a decision is necessary for every join and leave operation, i.e., every change in the workload. We approach this as a timeless problem. At this level, we ignore the necessity to schedule units, and instead reason about capacities. Future work must also take the difference between timeslice cycle length and preemptable workload blocks into account. For each media operation, only the resource consumption is considered, resulting in a timeless flow model.

Likewise, the load that can be handled by a processing unit, the bandwidth between processing units and the communication requirements between operations are expressed by a single capacity value. The high level scheduler's task is then to find a mapping that neither violates the capacity limits of any processing unit nor any communication channel.

Migration of media operations between processing units is in principle possible in our scenario. However, we consider migration highly undesirable. There is no particular need for avoiding it in symmetric multiprocessing systems; however, most of the topologies that we investigate are either NUMA or non-shared memory architectures that would require active movement of the operations' state between the processing units. This would be very costly in terms of the interconnection bandwidth that is required for moving the complete processing state that is stored in one processing unit to the new processing unit. Additionally, migration takes time and may cause disruption of streams, and in a worst case, break the dependency if interconnection bandwidth is exhausted during migration, resulting in a complete rescheduling. Consequently, media operations are here mapped and pinned to processing units by the high level scheduler.

3.2   Managing timeslices

The scheduling of tasks at a particular processing unit is handled by a low level scheduler. The low level scheduler must support the flow abstraction, and the worst case performance and overhead must be known. This provides the capacity value available to the high level scheduler. For the low-level scheduler, media operations are independent of each other. This implies, that communication between tasks and the resources consumed for inter-task communication are not considered. Each task has a period and in every period a certain amount of resources is consumed.

Nevertheless, at this level, old basic schedulers such as Rate Monotonic, Earliest Deadline First, Heits, AQUA, RT Upcalls, SMART, etc. are useful schedulers, and earlier work (e.g., [7]) has shown that these schedulers are able to find a schedule for each individual processing unit as long as there in total is enough available resources (which is an admittance check performed by the high-level scheduler). We are therefore currently planning for independent single-core real-time schedulers to solve our need for low-level scheduling, but do not consider it any further in this paper.

4.   SIMULATOR

To understand the interaction between interconnection topologies, different workload patterns and high-level scheduling algorithms, we have written a simulator to evaluate the schedulability of dynamic workloads. It allows us to look at metrics like bandwidth consumption and scheduling failure rates in a controlled environment, and we here introduce the topologies, workloads and algorithms that we have used.

4.1   Topologies

Current multi-core architectures take a variety of approaches for memory handling and interconnection between cores. We are currently not considering memory at all, but assume the need for data forwarding from one processing node the next. This is an approach that is not typically used in the current Intel architecture, but rather typical for specialized media processors, and also applicable for NUMA architec-

tures. Interconnects come as switch, bus, or point-to-point approaches. We model all of them as switches and point-to-point links at this time. Furthermore, the architectures that we use in the simulations in this paper are inspired by the topologies of the Intel Core 2, AMD's Opteron, the STI Cell BE and nVIDIA's GPUs (see figure 2). Since we intend to examine the effects of topologies and not real-world processor capabilities, we have provide all models with the same total processing capacity. The processing capacity is however, distributed over different numbers of processing units (heterogeneous in case of the Cell-inspired topology). Similarly, the bandwidth of interconnects is modeled in abstract performance numbers according to their role in the multi-core architecture that inspired each topology.

4.2   Workloads

The workload is specified as graphs of interconnected media processing operations which represent the need for processing capacity. Bandwidth requirements between media processing operations represent the consumed interconnect bandwidth that is required when neighboring processing stages in the graph are not processed by the same processing unit. Figure 1 shows a very simple example for such a processing graph, and where for example the initial conference members are users (A) and (B), and the conference is later dynamically extended to accommodate user (C). Moreover, to test various scenarios, we used a workload generator with the ability to create dynamically changing media processing demands with well-known parameters. The workload generator creates conferences with negative exponentially distributed duration, and with either Poisson-distributed inter-arrival times or at a constant number of concurrent conferences. Within each conference, uniformly distributed sets of participants arrive and depart according to a Poisson process, where each of the media operations claims a processing power and bandwidth from a uniform distribution up to a given maximum value.

4.3   Scheduling Algorithms

Developing high-level scheduling algorithms for arbitrary inter-dependent workloads is a future goal, along with real-world implementations. A one-for-all algorithm appear to be out of the question, and there is no reason why a real-world implementation should be able to adapt dynamically to wide ranges of different hardware. In any case, this is out of the scope of this paper. Here, we use strategies that are inspired by packing strategies for passive operating system resources, but extended with functionality to check link availability. The selected results use the following strategies:
First Fit (FF): Assign every processing unit an index. For every media operation that requires a certain amount of processing capacity, start searching at the processing unit with index 0 for available processing capacity. If the unit with index 0 does not have sufficient resources, use a breadth-first-search (BFS) approach through the topological neighbors of the unit, until a unit with sufficient capacity is found, or until the scheduling fails.
Next Fit (NF): Similar to FF, but instead of starting at index 0 (first processing unit) for every new job, NF starts its search (in BFS order) from the node where the previous media operation was successfully scheduled. If subsequent media operations are interconnected, this can achieve a high degree of packing and saves interconnection bandwidth.
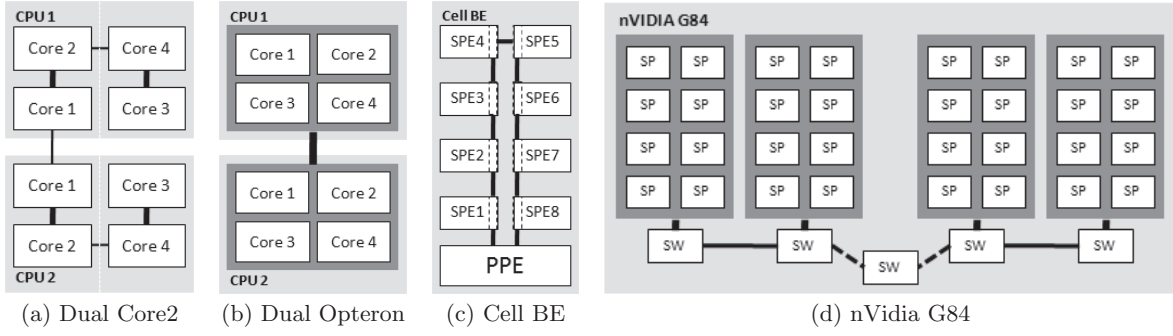
(a) Dual Core2   (b) Dual Opteron   (c) Cell BE   (d) nVidia G84

Figure 2: Tested processor topologies

Random Start (RS): In contrast to NF, RS keeps separate indices for each conference to start the search for appropriate processing units for newly arriving media operations belonging to that conference. This is meant to achieve better clustering and thus, less interconnection bandwidth consumption than NF. For a newly starting conference, a random processing unit is chosen.

Worst Start (WS): WS is similar to RS, but instead of randomly finding a processing unit for a newly starting conference, WS starts at the processing unit with the highest remaining free capacity (thus the name worst fit). This is done in the hope that the highest initial packing can be achieved for the new session.

We have also tested several others like plain first-, best-, worst-fit, etc., known from old memory management systems placing data elements in memory, but they do not take bandwidth into account so we have only used these as basis and benchmarks.

5. RESULTS

This section presents the results from our simulations including some selected plots and our main findings. Using the workload generator mentioned above, we have generated several workloads over the same set of parameters to be able to extract statistics. We have varied the core and link load and the number of concurrent conference sessions and participants. The same workloads have been used for all algorithms on all topologies.

5.1 Scheduling Ability

We first look at how well different means help to find a schedule. As expected, the failure rates increase for all algorithms if either of the processing or communication cost increase (see figure 3 for a representative example). Furthermore, in figure 4, we compare the tested algorithms with respect to the failure rate. If we do not use BFS but a random ordering of the processing unit, and apply FF and NF as they are known from memory management systems that are not concerned with bandwidth, the performance drops significantly. There is usually also a failure rate reduction if we try to place related and dependent jobs in close proximity in order to minimize communication, as done by NF in contrast to only finding the first available as in FF. However, in our set of algorithms, we see the largest performance gain if we search for a suitable place (least loaded node) to start a new session like in RS and WS. These approaches are beneficial because they allow dynamically joining media operations to be packed on processing units that are topologically close to
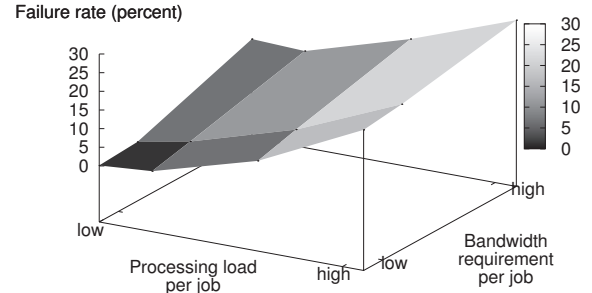

Figure 3: Failure rate versus processing and communication costs (FF on nVidia G84)

other jobs in the same session. When resource fragmentation grows, this saves considerable interconnect bandwidth.

5.2 Bandwidth Consumption

One other important metric in finding a good schedule is a measure for the communication required between processing units. Such communication both takes time and consumes bandwidth. A good schedule for a time-dependent scenario should often try to pack jobs on a cluster of processing cores to minimize communication. To see the bandwidth required for the found schedules, we monitored the communication needs during all the simulations. Figures 5 and 6 show representative examples where the workload is executed on Cell- and DualOpteron-based topologies. The plots show the consumed bandwidth on the time line and also a plot on which times the algorithm failed to find a schedule for a new job.

In general, always starting a search on the same node (as FF) effectively packs data on a small area of the topology. This is fine as long as there is enough processing power, but as the load increases, the system benefits from a more distributed clustering between sessions, i.e., packing each session on different places. A better approach is therefore to start every new jobs where the last job in the same session was placed (as NF). Furthermore, if the starting point is varied for each new session the results are increased further, where the best results are achieved by searching for the least loaded node with WS. As a consequence of a reduced bandwidth consumption, we see again that the number of failed
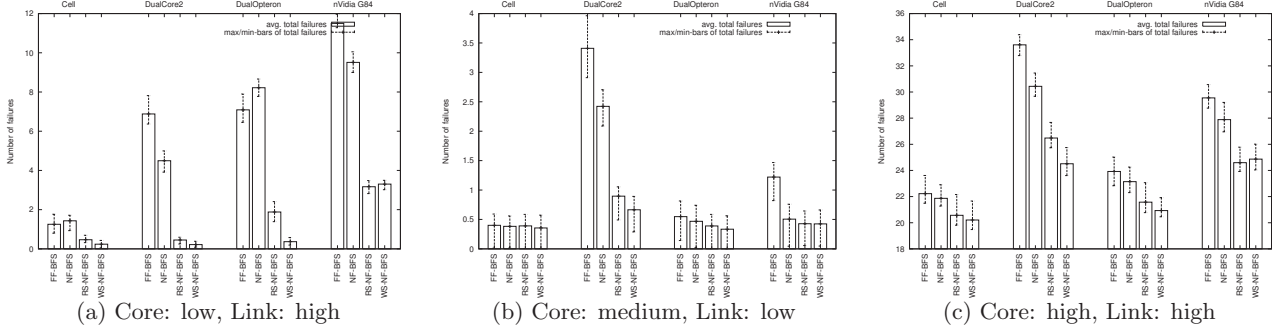
| (a) Core: low, Link: high | (b) Core: medium, Link: low | (c) Core: high, Link: high |

Figure 4: Scheduling failure rate for different algorithms on different topologies when varying the load

| (a) FF | (b) NF | (c) RS | (d) WS |
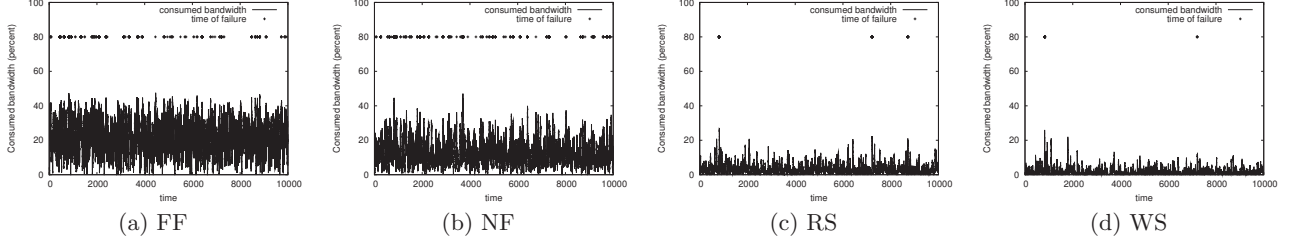
Figure 5: Consumed bandwidth resources on a Cell topology.

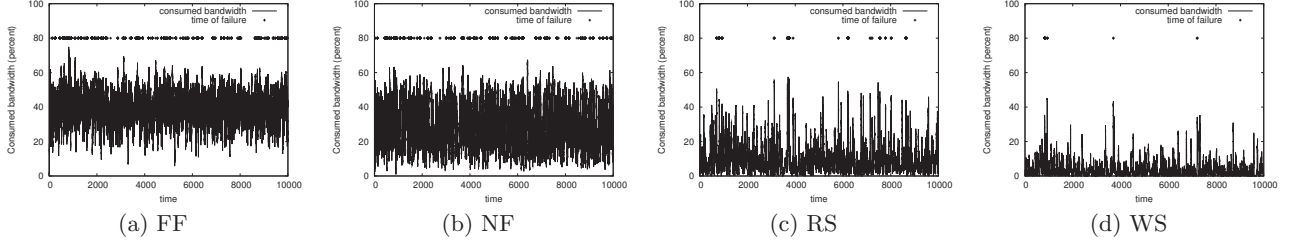| (a) FF | (b) NF | (c) RS | (d) WS |

Figure 6: Consumed bandwidth resources on a DualOpteron topology.

schedules are reduced using RS and WS.

5.3 Discussion

There are several metrics that are important with respect to resource scheduling. In order to scale and support as many inter-dependent jobs (conference participants) as possible, important metrics include scheduling ability and efficient resource utilization. We have here shown that there are large differences for different kind of scheduling means, and there are also some differences based on the topology characteristics they are used on. In this respect, our results show that mechanisms like performing job clustering, using BFS and finding a high capacity place to start a new session, are promising because the bandwidth consumption is usually reduced. This is especially important if bandwidth is a scarce resource or the workload is time-critical due to added communication delay.

Other interesting approaches in terms of compacting the workload is to analyze the workload and try to make larger computing blocks ahead of running the scheduler and then recursively split on the cheapest link is no schedule is found. This is efficient as long as there are much available processing capacity, but when the load increase and the large

blocks must be re-divided, this approach increase the probability that large blocks will be placed further from each other, i.e., consuming bandwidth on several links. Thus, in our test, the failure rate on a loaded system is greatly increased meaning that more logic must catch and avoid the long-distance placement of neighboring jobs.

In this preliminary study, we have focused on scheduling ability and bandwidth consumption only. However, there are other metrics that are relevant. For example, the time to find a schedule or scheduling overhead will in some scenarios be important, but they will be implementation and architecture dependent and are thus not considered in the simulator. Load-balancing properties of the algorithm is also in some cases important, but in the conferencing scenario discussed here, the reduction of bandwidth consumption and load balancing is contradictory in terms of link overhead, and we therefore here look at the ability to compact the workload to save bandwidth.

In our work on efficient scheduling algorithms for inter-job dependent workloads, we aim for a Linux implementation. Here, it will be important to also integrate the low- and high-level schedulers for example to exchange information about available resources. However, as a first step, we look

6. RELATED WORK

Multi-core scheduling attracts a considerable amount of research that aims at load-balancing, cache coherency, and the like. Approaches found in the literature that combine mapping to processing units and scheduling of tasks on the processing nodes in one stage, such as PFair [6], can schedule based on processing capacity alone, since they rely on a unified shared memory architecture. Our problem requires a coordinated scheduling of processing capacity and interconnection bandwidth that is due to a lack of a shared memory-assumption. Perfect scheduling under these conditions is NP-hard, and alternative heuristics must be explored. One of the classical scheduling methods are greedy algorithms. An example of a greedy algorithm can be found in [1]. Here, the tasks of a parallel application are mapped onto different processors based on a the expected run-time of the job. This algorithm does not take job dependencies into account. Another approach is based on the representation of dependent parts of an application as a directed acyclic graph (DAG). DAG can represent both processing and communication demands. Kwok et al. [5] have made a survey of several classes of scheduling algorithms for allocating workload DAGs to a network of processors, e.g., highest level first with estimated times, linear clustering, task duplication and mapping heuristic. DAG-algorithms are also often used to schedule workloads in Grids and several approaches have been suggested in [2]. These solutions are not directly applicable to our case because the do not handle dynamic workloads without full reconfiguration. Another type of algorithms are genetic algorithms [4]. These algorithms are focused around deterministic scheduling problems, meaning that all information about the task and their relations to each other are known in advance. On asymmetric multi-core processors there have been several approaches for soft real-time scheduling. In [3], a soft real-time scheduler for Linux has been implemented. Dependencies and communication between the real-time jobs have however not been considered.

7. CONCLUSION

As a step towards better scheduling support in the operating system for inter-dependent jobs on possibly heterogeneous multi-processors without uniform shared memory, we have examined the effects of topologies on the schedulability of dynamic workloads that consist of interdependent long-lived media operations.

We have motivated the distinction between high-level schedulers that act on the timescale of application scheduling and assign processing resources to long-living media operations, and low-level schedulers that managed computing cycles on processing units locally based on assignments of the high-level scheduler and otherwise without global knowledge.

We found that high-level schedulers that try to achieve compact assignment of operations that belong to the same media session to processing units is generally beneficial (as expected), but we have observed extremely strong improvements in topologies that rely on a bandwidth-constraint switched architecture such as the Core 2 or nVidia-GPU inspired topologies. These effects appear to be fairly independent of whether the switches connect few high-capacity processing units or many low-capacity units. On the other hand, we can observe that even the simplest tested scheduling algorithm (first fit) performs quite efficiently on the Cell-inspired ring topology with asymmetric nodes.

In various other tests, we observed also a variety of other effects of scheduler features. One of the more interesting observations was that it is not generally feasible to schedule media operations of a session in larger blocks. This forces the scheduler in many cases to place the block rather far away from earlier parts of the session, consuming excessive interconnection bandwidth and finally, exhausting it. We could also observe that choosing a suitable first node for newly starting sessions has a major effect on performance. An alternative scheduler might generally optimize for minimal bandwidth consumption and processing capacity only as a computational bound.

After these initial observations, we will use our simulator to evaluate a variety of high-level schedulers. Concurrently, we are working on programming tools for Cell and nVidia that circumvent the provided frameworks, which are unable to schedule non-exclusive workloads, as well as porting of media operations to these environments. Given such tools, we will be able to investigate the interaction of high and low-level schedulers experimentally. We find that a final step in this development must integrate this scheduling with the operating system scheduler.

8. REFERENCES

[1] Armstrong, R., Hensgen, D., and Kidd, T. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Heterogeneous Computing Workshop (HCW)* (Mar. 1998), pp. 79–87.

[2] Batista, D. M., da Fonseca, N. L. S., and Miyazawa, F. K. A set of schedulers for grid networks. In *ACM Symposium on Applied Computing (SAC)* (New York, NY, USA, 2007), ACM, pp. 209–213.

[3] Calandrino, J. M., Baumberger, D., Li, T., Hahn, S., and Anderson, J. H. Soft real-time scheduling on performance asymmetric multicore platforms. In *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)* (2007), pp. 101–112.

[4] Chockalingam, T., and Arunkumar, S. Genetic algorithm based heuristics for the mapping problem. *Comput. Oper. Res. 22*, 1 (1995), 55–64.

[5] Kwok, Y.-K., and Ahmad, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys 31*, 4 (1999), 406–471.

[6] Moir, M., and Ramamurthy, S. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *IEEE Real-Time Systems Symposium (RTSS)* (1999), pp. 294–303.

[7] Wolf, L. C., Burke, W., and Vogt, C. Evaluation of a cpu scheduling mechanism for multimedia systems. *Software - Practice and Experience 26*, 4 (april 1996), 375–398.

Paper III: Tips, Tricks and Troubles: Optimizing for Cell and GPU

Tips, Tricks and Troubles: Optimizing for Cell and GPU

Håkon Kvale Stensland, Håvard Espeland, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory, Norway
Department of Informatics, University of Oslo, Norway
{haakonks, haavares, griff, paalh}@simula.no

ABSTRACT

When used efficiently, modern multicore architectures, such as Cell and GPUs, provide the processing power required by resource demanding multimedia workloads. However, the diversity of resources exposed to the programmers, intrinsically requires specific mindsets for efficiently utilizing these resources - not only compared to an x86 architecture, but also between the Cell and the GPUs. In this context, our analysis of 14 different Motion-JPEG implementations indicates that there exists a large potential for optimizing performance, but there are also many pitfalls to avoid. By experimentally evaluating algorithmic choices, inter-core data communication (memory transfers) and architecture-specific capabilities, such as instruction sets, we present tips, tricks and troubles with respect to efficient utilization of the available resources.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUE**]: Concurrent Programming—*Parallel programming*

General Terms

Measurement, Performance

1. INTRODUCTION

Heterogeneous systems like the STI Cell Broadband Engine (Cell) and PCs with Nvidia graphical processing units (GPUs) have recently received a lot of attention. They provide more computing power than traditional single-core systems, but it is a challenge to use the available resources efficiently. Processing cores have different strengths and weaknesses than desktop processors, the use of several different types and sizes of memory is exposed to the developer, and limited architectual resources require considerations concerning data and code granularity.

We want to learn how to *think* when the multicore system at our disposal is a Cell or a GPU. We aim to understand

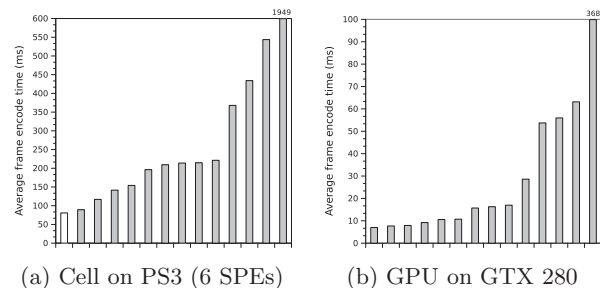| (a) Cell on PS3 (6 SPEs) | (b) GPU on GTX 280 |

Figure 1: Runtime for MJPEG implementations.

how to use the resources efficiently, and point out tips, tricks and troubles, as a small step towards a programming framework and a scheduler that parallelizes the same code efficiently on several architectures. Specifically, we have looked at effective programming for the workload-intensive yet relatively straight-forward Motion-JPEG (MJPEG) video encoding task. Here, a lot of CPU cycles are consumed in the sequential discrete cosine transformation (DCT), quantization and compression stages. On single core systems, it is almost impossible to process a 1080p high definition video in real-time, so it is reasonable to apply multicore computing in this scenario.

Our comparison of 14 different implementations on both Cell and GPU gives a good indication that the two considered architectures are complex to use, and that achieving high performance is not trivial. Derived from a sequential codebase, these multicore implementations differ in terms of algorithms used, resource utilization and coding efficiency. Figure 1 shows performance results for encoding the "tractor" video clip[1] in 4:2:0 HD. The differences between the fastest and slowest solution are 1869 ms and 362 ms per frame on Cell and GPU, respectively, and it is worth noting that the fastest solutions were disk I/O-bound. To gain experience of what works and what does not, we have examined these solutions. We have not considered coding style, but revisited algorithmic choices, inter-core data communication (memory transfers) and use of architecture-specific capabilities.

In general, we found that these architectures have large potentials, but also many possible pitfalls, both when choosing specific algorithms and for implementation-specific decisions. The way of thinking cross-platform is substantially different, making it an art to use them efficiently.

---

[1] Available at ftp://ftp.ldv.e-technik.tu-muenchen.de/ dist/test_sequences/1080p/tractor.yuv

2. BACKGROUND

2.1 SIMD and SIMT

Multimedia applications frequently perform identical operations on large data sets. This has been exploited by bringing the concept of SIMD (single instruction, multiple data) to desktop CPUs, as well as the Cell, where a SIMD instruction operates on a short vector of data, e.g., 128-bits for the Cell SPE. Although SIMD instructions have become mainstream with the earliest Pentium processors and the adoption of PowerPC for MacOS, it has remained an art to use them. On the Cell, SIMD instructions are used explicitly through the vector extensions to C/C++, which allow basic arithmetic operations on vector data types of intrinsic values. It means that the programmer can apply a sequential programming model, but needs to adapt memory layout and algorithms to the use of SIMD vectors and operations.

Nvidia uses an abstraction called SIMT (single-instruction, multiple thread). SIMT enables code that uses only well-known intrinsic types but that can be massively threaded. The runtime system of the GPU schedules these threads in groups (called warps) whose optimal size is hardware-specific. The control flow of such threads can diverge like in an arbitrary program, but this will essentially serialize all threads of the block. If it does not diverge and all threads in a group execute the same operation or no operation at all in a step, then this operation is performed as a vector operation containing the data of all threads in the block.

The functionality that is provided by SIMD and SIMT is very similar. In SIMD programming, vectors are used explicitly by the programmer, who may think in terms of sequential operations on very large operands. In SIMT programming, the programmer can think in terms of threaded operations on intrinsic data types.

2.2 STI Cell Broadband Engine

The Cell Broadband Engine is developed by Sony Computer Entertainment, Toshiba and IBM. As shown in Figure 2, the central components are a Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPE) connected by the Element Interconnect Bus (EIB). The PPE contains a general purpose 64-bit PowerPC RISC core, capable of executing two simultaneous hardware threads. The main purpose of the PPE is to control the SPEs, run an operating system and manage system resources. It also includes a standard Altivec-compatible SIMD unit. An SPE contains a Synergistic Processing Unit and a Memory Flow controller. It works on a small (256KB) very fast memory, known as the local storage, which is used both for code and data without any segmentation. The Memory Flow Controller is used to transfer data between the system memory and local storage using explicit DMA transfers, which can be issued both from the SPE and PPE.
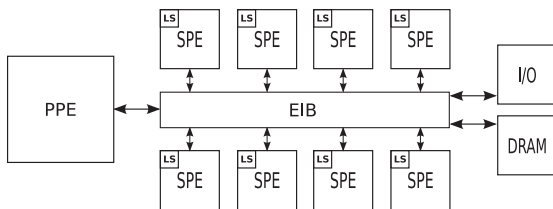

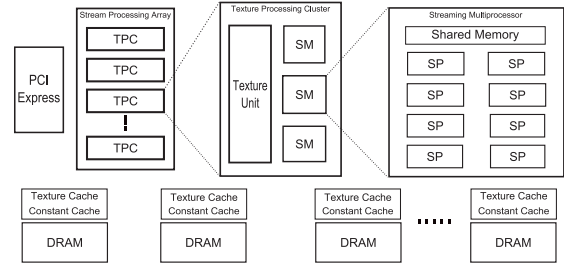Figure 2: Cell Broadband Engine Architecture

Figure 3: Nvidia GT200 Architecture

2.3 Nvidia Graphics Processing Units

A GPU is a dedicated graphics rendering device, and modern GPUs have a parallel structure, making them effective for doing general-purpose processing. Previously, shaders were used for programming, but specialized languages are now available. In this context, Nvidia has released the CUDA framework with a programming language similar to ANSI C. In CUDA, the SIMT abstraction is used for handling thousands of threads.

The latest generation available from Nvidia (GT200) is shown in Figure 3. The GT200 chip is presented to the programmer as a highly parallel, multi-threaded, multi-core processor - connected to the host computer by a PCI Express bus. The GT200 architecture contains 10 texture processing clusters (TPC) with 3 streaming multiprocessors (SM). A single SM contains 8 stream processors (SP) which are the basic ALUs for doing calculations. GPUs have other memory hierarchies than an x86 processor. Several types of memory with different properties are available. An application (*kernel*) has exclusive control over the memory. Each thread has a private local memory, and the threads running on the same stream multiprocessor (SM) have access to a shared memory. Two additional read-only memory spaces called constant and texture are available to all threads. Finally, there is the global memory that can be accessed by all threads. Global memory is not cached, and it is important that the programmer ensures that running threads perform *coalesced* memory accesses. Such a coalesced memory access requires that the threads' accesses occur in a regular pattern and creates one large access from several small ones. Memory accesses that cannot be combined are called uncoalesced.

3. EXPERIMENTS

By learning from the design choices of the implementations in Figure 1, we designed experiments to investigate how performance improvements were achieved on both Cell and GPU. We wanted to quantify the impact of design decisions on these architectures.

All experiments encode HD video (1920x1080, 4:2:0) from raw YUV frames found in the *tractor* test sequence. However, we used only the first frame of the sequence and encode it 1000 times in each experiment to overcome the disk I/O bottleneck limit. This becomes apparent at the highest level of encoding performance since we did not have a high bandwidth video source available. All programs have been compiled with the highest level of compiler optimizations using gcc and nvcc, respectively, for Cell and GPU. The Cell experiments have been tested on a QS22 bladeserver (8 SPEs, the results from Figure 1 were on a PS3 with 6 SPEs) and the GPU experiments on a GeForce GTX 280 card.
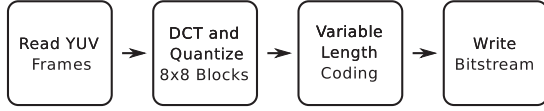
164

Figure 4: Overview of the MJPEG encoding process

3.1 Motion JPEG Encoding

The MJPEG format is widely used by webcams and other embedded systems. It is similar to videocodecs such as Apple ProRes and VC-3, used for video editing and postprocessing due to their flexibilty and speed, hence the lack of inter-prediction between frames. As shown in Figure 4, the encoding process of MJPEG comprises splitting the video frames in 8x8 macroblocks, each of which must be individually transformed to the frequency domain by forward discrete cosine transform (DCT) and quantized before the output is entropy coded using variable-length coding (VLC). JPEG supports both arithmetic coding and Huffman compression for VLC, our encoder uses predefined Huffman tables for compression of the DCT coefficients of each macroblock. The VLC step is not context adaptative, and macroblocks can thus be compressed independently. The length of the resulting bitstream, however, is probably not a multiple of eight, and most such blocks must be bit-shifted completely when the final bitstream is created.

The MJPEG format provides many layers of parallelism; Starting with the many independent operations of calulating DCT, the macroblocks can be transformed and quantized in arbitrary order, also frames and color components can be encoded separately. In addition, every frame is entropy-coded separately. Thus, many frames can be encoded in parallel before merging the resulting frame output bitstreams. This gives a very fine-level granularity of parallel tasks, providing great flexibility in how to implement the encoder. It is worth noting that many problems have much tighter data dependencies than we observe in the MJPEG case, but the general ideas for optimizing individual parts pointed out in this paper stand regardless of whether the problem is limited by dependencies or not.

The forward 2D DCT function for a macroblock is defined in the JPEG standard for image component $s_{y,x}$ to output DCT coefficients $S_{v,u}$ as

$$S_{v,u} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{y,x} cos\frac{(2x+1)u\pi}{16} cos\frac{(2y+1)v\pi}{16}$$

where $C_u, C_v = \frac{1}{\sqrt{2}}$ for $u, v = 0$ and $C_u, C_v = 1$ otherwise. The equation can be directly implemented in an MJPEG encoder and is referred to as 2D-plain. The algorithm can be sped up considerably by removing redundant calculations. One improved version that we label 1D-plain uses two consecutive 1D transformations with a transpose operation in between and after. This avoids symmetries, and the 1D transformation can be optimized further. One optimization uses the AAN algorithm, originally proposed by Arai et al. [1] and further refined by Kovac and Ranganathan [5]. Another uses a precomputed 8x8 transformation matrix that is multiplied with the block together with the transposed transformation matrix. The matrix includes the postscale operation, and the full DCT operation can therefore be completed with just two matrix multiplications, as explained by Kabeen and Gent [2].

More algorithms for calculating DCT exist, but they are not covered here. We have implemented the different DCT algorithms as scalar single-threaded versions on x86 (Intel Core i5 750). The performance details for encoding HD video were captured using oprofile and can be seen in Figure 5. The plot shows that the 1D-AAN algorithm using two transpose operations was the fastest in this scenario, with the 2D-matrix version as number two. The average encoding time for a single frame using 2D-plain is more than 9 times that of a frame encoded using 1D-AAN. For all algorithms, the DCT step consumed most CPU cycles.

3.2 Cell Broadband Engine Experiments

Considering the *embarrassingly parallel* parts of MJPEG video encoding, a number of different layouts is available for mapping the different steps of the encoding process to the Cell. Because of the amount of work, the DCT and quantization steps should be executed on SPEs, but also the entropy coding step can run in parallel between complete frames. Thus, given that a few frames of encoding delay are acceptable, the approach we consider best is to process full frames on each SPE with every SPE running DCT and quantization of a full frame. This minimizes synchronization between cores, and allows us to perform VLC on the SPEs.

Regardless of the placement of the encoding steps, it is important to avoid idle cores. We solved this by adding a frame queue between the frame reader and the DCT step, and another queue between the DCT and VLC steps. Since a frame is processed in full by a single processor, the AAN algorithm is well suited for the Cell. It can be implemented in a straight-forward manner for running on SPEs, with VLC coding placed on the PPE. We tested the same algorithm optimized with SPE intrinsics for vector processing (SIMD) resulting in double encoding throughput, which can be seen in Figure 6 (Scalar- and Vector/PPE).

Another experiment involved moving the VLC step to the SPEs, offloading the PPE. This approach left the PPE with only the task of reading and writing files to disk in addition to dispatching jobs to SPEs. To be able to do this, the luma and chroma blocks of the frames had to be transformed and quantized in interleaved order, i.e., two rows of luma and a single row of both chroma channels. The results show that
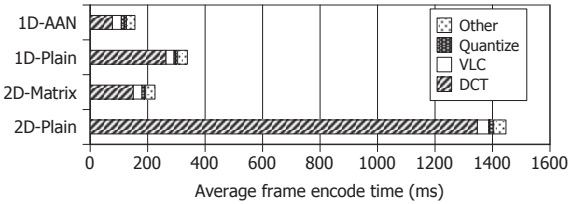

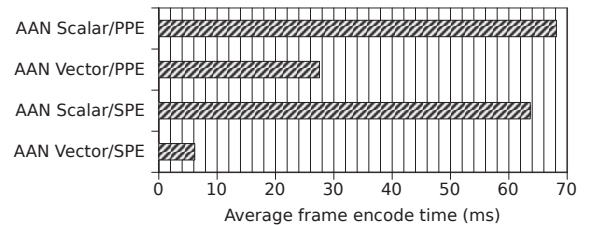**Figure 5: MJPEG encode time on single thread x86**

Figure 6: Encoding performance on Cell with different implementations of AAN and VLC placement

77

165

the previous encoding speed was limited by the VLC as can be seen in Figure 6 (Scalar- and Vector/SPE).

To get some insight into SPE utilization, we collected a trace (using pdtr, part of IBM SDK for Cell) showing how much time is spent on the encoding parts. Figure 7 shows the SPE utilization when encoding HD frames for the Scalar- and Vector/SPE from Figure 6. This distinction is necessary because the compiler does not generate SIMD code, requiring the programmer to hand-code SIMD intrinsics to obtain high throughput. The scalar version uses about four times more SPE time to perform the DCT and quantization steps for a frame than the vector version, and additionally 30% of the total SPE time to pack and unpack scalar data into vectors for SIMD operations. Our vectorized AAN implementation is nearly eight times faster than the scalar version.

With the vector version of DCT and quantization, the VLC coding uses about 80 % of each SPE. This can possibly be optimized further, but we did not find time to pursue this.

The Cell experiments demonstrate the necessary level of fine-grained tuning to get high performance on this architecture. In particular, correctly implementing an algorithm using vector intrinsics is imperative. Of the 14 implementations for Cell in Figure 1, only one offloaded VLC to the SPEs, but this was the second fastest implementation. The fastest implementation vectorized the DCT and quantization, and the Vector/SPE implementation in Figure 6 is a combination of these two. One reason why only one implementation offloaded the VLC may be that it is unintuitive. An additional communication and shift step is required in parallelizing VLC because the lack of arbitrary bit-shifting of large fields on Cell as well as GPU prevents a direct port from the sequential codes. Another reason may stem from the dominance of the DCT step in early profiles, as seen in Figure 5, and the awkward process of gathering profiling data on multicore systems later on. The hard part is to know what is best in advance, especially because moving an optimized piece of code from one system to another can be significant work, and may even require rewriting the program entirely. It is therefore good practice to structure programs in such a way that parts are coupled loosely. In that way, they can both be replaced and moved to other processors with minimal effort.

When comparing the 14 Cell implementations of the encoder shown in Figure 1 to find out what differentiates the fastest from the medium speed implementations, we found some distinguising features: The most prominent one being not exploiting the SPE's SIMD capabilities, but also in the areas of memory transfers and job distribution. Uneven workload distribution and lack of proper frame queuing resulted in idle cores. Additionally, some implementations suffered from small, often unconcealed, DMA operations that left SPEs in a stalled state waiting for the memory transfer to complete. It is evident that many pitfalls need to be avoided when writing programs for the Cell architecture, and we have only touched upon a few of them. Some of these are
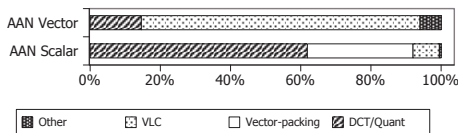

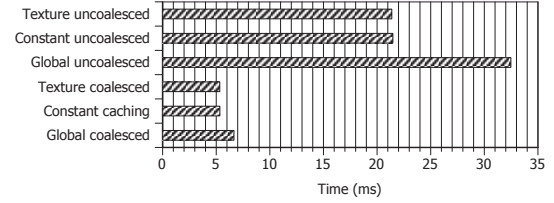**Figure 7: SPE utilization using scalar or vector DCT**

Figure 8: Optimization of GPU memory accesses

obvious, but not all, and to get acceptable performance out of a program running on the Cell architecture may require multiple iterations, restructuring and even rewrites.

3.3   GPU Experiments

As for the Cell, several layouts are available for GPUs. However, because of the large number of small cores, it is not feasible to assign one frame to each core. The most time-consuming parts of the MJPEG encoding process, the DCT and quantization steps, are well suited for GPU acceleration. In addition, the VLC step can also be partly adapted.

Coalesced memory accesses are known to have large performance impacts. However, few quantified results exist, and efficient usage of memory types, alignment and access patterns remains an art. Weimer et al. [11] experimented with bank conflicts in shared memory, but to shed light on the penalties of inefficient memory type usage, further investigation is needed. We therefore performed experiments that read and write data to and from memory with both uncoalesced and coalesced access patterns [7], and used the Nvidia CUDA Visual Profiler to isolate the GPU-time for the different kernels.

Figure 8 shows that an uncoalesced access pattern decreases throughput in the order of four times due to the increased number of memory transactions. Constant and texture memory are cached, and the performance for uncoalesced accesses to them is improved compared to global memory, but there is still a three-time penalty. Furthermore, the cached memory types support only read-only operations and are restricted in size. When used correctly, the performance of global memory is equal to the performance of the cached memory types. The experiment also shows that correct memory usage is imperative even when cached memory types are used. It is also important to make sure the memory accesses are correct according to the specifications of particular GPUs because the optimal access patterns vary between GPU generations.

To find out how memory accesses and other optimizations affect programs like a MJPEG encoder, we experimented with different DCT implementations. Our baseline DCT algorithm is the 2D-plain algorithm. The only optimizations in this implementation are that the input frames are read into cached texture memory and that the quantization tables are read into cached constant memory. As we observed in Figure 8, cached memory spaces improve performance compared to global memory, especially when memory accesses are uncoalesced. The second implementation, referred to as 2D-plain optimized, is tuned to run efficiently using principles from the CUDA Best Practices Guide [6]. These optimizations include the use of shared memory as a buffer for pixel values when processing a macroblock, branch avoidance by using boolean arithmetics and manual loop unrolling. Our third implementation, the 1D-AAN algorithm, is based upon the scalar implementation
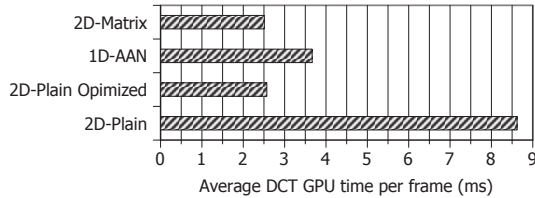
Figure 9: DCT performance on GPU

used on the Cell. Every macroblock is processed with eight threads, one thread per row of eight pixels. The input image is stored in cached texture memory, shared memory is used for temporarily storing data during processing. Finally, the 2D-matrix DCT using matrix multiplications where each matrix element is computed by a thread. The input image is stored in cached texture memory, and shared memory is used for storing data during calculations.

We know from existing work that to achieve high instruction throughput, branch prevention and the correct use of flow control instructions are important. If threads on the same SM diverge, the paths are serialized which decreases performance. Loop unrolling is beneficial on GPU kernels and can be done automatically by the compiler using pragma directives. To optimize frame exchange, asynchronous transfers between the host and GPU were used. Transferring data over the PCI Express bus is expensive, and asynchronous transfers help us reuse the kernels and hides some of the PCI Express latency by transferring data in the background.

To isolate the DCT performance, we used the CUDA Visual Profiler. The profiling results of the different implementations can be seen in Figure 9, and we can observe that the 2D-plain optimized algorithm is faster than AAN. The 2D-plain algorithm requires significantly more computations than the others, but by correctly implementing it, we get almost as good performance as with the 2D-matrix. The AAN algorithm, which does the least amount of computations, suffers from the low number of threads per macroblock. A low number of threads per SM can result in stalling, where all the threads are waiting for data from memory, which should be avoided.

This experiment shows that for architectures with vast computational capabilities, writing a good implementation of an algorithm adapted for the underlying hardware can be as important as the theoretical complexity of an algorithm.
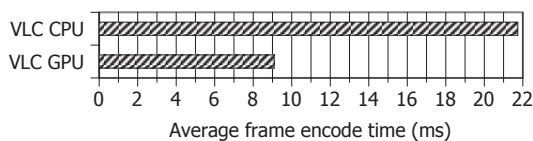

Figure 10: Effect of offloading VLC to the GPU

The last GPU experiment considers entropy coding on the GPU. As for the Cell, VLC can be offloaded to the GPU by assigning a thread to each macroblock in a frame to compress the coefficients and then store the bitstream of each macroblock and its length in global memory. The output of each macroblock's bitstream can then be merged either on the host, or by using atomic OR on the GPU. For the experiments here, we chose the former since the host is responsible for the I/O and must traverse the bitstream anyway. Figure 10 shows the results of an experiment that compares

MJPEG with AAN DCT with VLC performed on the host and on the GPU, respectively. We achieved a doubling of the encoding performance when running VLC on the GPU. In this particular case offloading VLC was faster than running on the host. It is worth noting that by running VLC on the GPU, the entropy coding scales together with the rest of the encoder with the resources available on the GPU. This means than if the encoder runs on a machine with a slower host CPU or faster GPU, the encoder will still scale.

4. DISCUSSION

Heterogeneous architectures like Cell and GPU provide large amounts of processing power, and achieving encoding throughputs of 480 MB/s and 465 MB/s, respectively, real-time MJPEG HD encoding may be no problem. However, an analysis of the many implementations of MJPEG available and our additional testing show that it is important to use the right concepts and abstractions, and that there may be large differences in the way a programmer must think.

The architectures of GPU and Cell are very different, and in this respect, some algorithms may be more suited than others. This can be seen in the experiments, where the AAN algorithm for DCT calculation performed best on both x86 and Cell, but did not achieve the highest throughput on GPU. This was because of the relatively low number of threads per macroblock for the AAN algorithm, which must perform the 1D DCT operation (one row of pixels within a macroblock) as a single thread. This is only one example of achieving a shorter computation time through increased parallelity at the price of a higher, sub-optimal total number of operations.

The programming models used on Cell and GPU mandate two different ways of thinking parallel. The approach of Cell is very similar to multi-threaded programming on x86, with the exception of shared memory. The SPEs are used as regular cores with explicit caches, and the vector units on the SPEs require careful data structure consideration to achieve peak performance. The GPU model of programming is much more rigid, with a static grid used for blocks of threads, and only synchronization through barriers. This hides the architecture complexity, and is therefore a simpler concept to grasp for some programmers. This notion is also strengthened by the better average GPU throughput of the implementations in Figure 1. However, to get the highest possible performance, the programmer must also understand the nitty details of the architecture to avoid pitfalls like warp divergence and uncoalesced memory accesses.

Deciding at which granularity the data should be partitioned is very hard to do correct *a priori*. The best granularity for a given problem differs with the architecture and even different models of the same architecture. One approach towards accomplishing this is to try to design the programs in such a way that the cores are seldom idle or stall. In practice, however, multiple iterations may be necessary to determine the best approach.

Similar to data partitioning, code partitioning is hard to do correctly in advance. In general, a rule of thumb is to write modular code to allow moving the parts to other cores. Also, a fine granularity is beneficial, since small modules can be merged again, and also be executed repeatedly with small overhead. Offloading is by itself advantageous as resources on the main processor become available for other tasks. It also improves scalability of the program with new

generations of hardware. In our MJPEG implementations, we found that offloading DCT/quantization and VLC coding was advantageous in terms of performance on both Cell and GPU, but it may not always be the case that offloading provides higher throughput.

The encoding throughput achieved on the two architectures was surprisingly similar. Although, the engineering effort for accomplishing this throughput was much higher on the Cell. This was mainly caused by the tedious process of writing a SIMD version of the encoder. Porting the encoder to the GPU in a straight-forward manner without significant optimizations for the architecture yielded a very good offloading performance compared to native x86. This indicates that the GPU is easier to use, but to reap the full potential of the architecture, one must have the same deep level of understanding as with the Cell architecture.

5. RELATED WORK

Heterogeneous multi-core platforms like the Cell and GPUs have attracted a considerable amount of research that aims at optimizing specific applications for the different architectures such as [9] and [4]. However, little work has been done to compare general optimization details of different heterogeneous architectures. Amesfoort et al. [10] have evaluated different multicore platforms for data-intensive kernels. The platforms are evaluated in terms of application performance, programming effort and cost. Colic et al. [3] look at the application of optimizing motion estimation on GPUs and quantify impact of design choices. The workload investigated in this paper is different from the workload we benchmark in our experiments, but they show a similar trend as our GPU experiments. They also conclude that elegant solutions are not easily achievable, and that it takes time, practice and experience to reap the full potential of the architecture. Petrini et al. [8] implement a communication-heavy radiation transport problem on Cell. They conclude that it is a good approach to think about problems in terms of five dimensions and partitioning them into: process parallelism at a very large scale, thread-level parallelism that handles inner loops, data-streaming parallelism that double-buffers data for each loop, vector parallelism that uses SIMD functions within a loop, and pipeline parallelism that overlaps data access with computations by threading. From our MJPEG implementations we observed that programmers had difficulties thinking parallel in two dimensions. This level of multi-dimensional considerations strengthens our statement that intrinsic knowledge of the system is essential to reap full performance of heterogeneous architectures.

6. CONCLUSION

Heterogeneous, multicore architectures like Cell and GPUs may provide the resources required for real-time multimedia processing. However, achieving high performance is not trivial, and in order to learn how to think and use the resources efficiently, we have experimentally evaluated several issues to find the tricks and troubles.

In general, there are some similarities, but the way of thinking must be substantially different - not only compared to an x86 architecture, but also between the Cell and the GPUs. The different architectures have different capabilities that must be taken into account both when choosing a specific algorithm and making implementation-specific deci-

sions. A lot of trust is put on the compilers of development frameworks and new languages like Open CL, which are supposed to be a "recompile-only" solution. However, to tune performance, the application must still be hand-optimized for different versions of the GPUs and Cells available.

7. REFERENCES

[1] ARAI, Y., AGUI, T., AND NAKAJIMA, M. A fast dct-sq scheme for images. *Transactions of IEICE E71*, 11 (1988).

[2] CABEEN, K., AND GENT, P. Image compression and the discrete cosine transform. In *Math 45*, College of the Redwoods.

[3] COLIC, A., KALVA, H., AND FURHT, B. Exploring nvidia-cuda for video coding. In *ACM SIGMM conference on Multimedia systems (MMSys)* (2010), ACM, pp. 13–22.

[4] CURRY, M., SKJELLUM, A., WARD, H., AND BRIGHTWELL, R. Accelerating reed-solomon coding in raid systems with gpus. In *International Parallel and Distributed Processing Symposium (IPDPS)* (April 2008), IEEE, pp. 1–6.

[5] KOVAC, M., AND RANGANATHAN, N. JAGUAR: A fully pipelined VLSI architecture for JPEG image compression standard. *Proceedings of the IEEE 83*, 2 (1995).

[6] NVIDIA. Nvidia cuda c programming best practices guide 2.3, 2009.

[7] OTTESEN, A. Efficient parallelisation techniques for applications running on gpus using the cuda framework. Master's thesis, Department of Informatics, University of Oslo, Norway, May 2009.

[8] PETRINI, F., FOSSUMA, G., FERNANDEZ, J., VARBANESCU, A. L., KISTLER, M., AND PERRONE, M. Multicore surprises: Lessons learned from optimizing Sweep3D on the Cell Broadband Engine. In *International Parallel and Distributed Processing Symposium (IPDPS)* (March 2007), IEEE, pp. 1–10.

[9] SACHDEVA, V., KISTLER, M., SPEIGHT, E., AND TZENG, T.-H. K. Exploring the viability of the Cell Broadband Engine for bioinformatics applications. *Parallel Computing 34*, 11, 616–626.

[10] VAN AMSESFOORT, A., VARBANESCU, A., SIPS, H. J., AND VAN NIEUWPOORT, R. Evaluating multi-core platforms for hpc data-intensive kernels. In *ACM Conference on Computing Frontiers (ICCF)* (2009).

[11] WEIMER, W., BOYER, M., AND SKADRON, K. Automated dynamic analysis of cuda programs. In *Third Workshop on software Tools for MultiCore Systems (STMCS)* (2008).

Paper IV: Cheat Detection Processing: A GPU versus CPU Comparison

Cheat Detection Processing:
A GPU versus CPU Comparison

Håkon Kvale Stensland, Martin Øinæs Myrseth, Carsten Griwodz, Pål Halvorsen
Simula Research Laboratory, Norway and Department of Informatics, University of Oslo, Norway
Email: {haakonks, martinom, griff, paalh}@simula.no

Abstract—In modern online multi-player games, game providers are struggling to keep up with the many different types of cheating. Cheat detection is a task that requires a lot of computational resources. Advances made within the field of heterogeneous computing architectures, such as graphics processing units (GPUs), have given developers easier access to considerably more computational resources, enabling a new approach to solving this issue.

In this paper, we have developed a small game simulator that includes a customizable physics engine and a cheat detection mechanism that checks the physical model used by the game. To make sure that the mechanisms are fair to all players, they are executed on the server side of the game system. We investigate the advantages of implementing physics cheat detection mechanisms on a GPU using the Nvidia CUDA framework, and we compare the GPU implementation of the cheat detection mechanism with a CPU implementation. The results obtained from the simulations show that offloading the cheat detection mechanisms to the GPU reduces the time spent on cheat detection, enabling the servers to support a larger number of clients.

I. Introduction

On-line multi-player gaming has experienced an amazing growth over the last decade. It goes along with cheating as the most prominent case of malicious behavior performed by game players [11]. It is therefore in the best interest of game service providers to eradicate cheating. However, the demand for a stable service for resource intensive games restricts the amount of resources that can be dedicated to cheat detection mechanisms.

Many on-line multi-player games suffer from excessive cheating in one form or another. However, in many cases, the existence of cheating is hard to prove [6]. The only part of a distributed system that a game service provider can trust is the part of the system running on hardware under their control. Any other part of the system can and will most likely be exploited by a cheater. As of now, no existing framework manages to eliminate all kinds of cheating, so game developers are forced to either create their own mechanisms or use a selection of existing solutions to cover the aspects of a game that a cheater might exploit.

In-game physics, aimed to increase game realism, experiences increased popularity in many kinds of games. Most games that have implemented in-game physics use it as a major part of the game-play experience, some even base the entire game-play around physics alone. In-game physics is therefore a very likely part of a game to be exploited. To solve this problem, central servers or other trusted entities must ensure consistency in the movements of all the clients in the game. With our approach the physics engine can be implemented on the server together with the cheat detection mechanisms. This solution frees resources on the game clients. However, it requires more hardware at the server side.

Adding more hardware to a system can increase its performance, but this serves only as a temporary solution. The hardware used in commercial game server clusters is expensive, and the performance gained might only be sufficient for a short period of time. Because of the physical limitations halting the single-threaded performance increase in normal CPUs, further performance increase is accomplished by adding more identical processing cores. The modern GPU is a relatively inexpensive example of such a parallel architecture. This forces developers to think differently. The process of adding new and faster hardware is now slowly substituted by migrating systems to parallel processors. For this change to be beneficial, serial algorithms must be parallelized.

Our goal in this paper is to determine if graphics processing units (GPUs) can handle cheat detection mechanisms in a client-server based game system. We investigate also how a GPU implementation might scale when compared to the same mechanisms running on normal CPUs. The results of our benchmarks show that the GPU scales better than a CPU when processing cheat detection mechanisms. Offloading cheat detection to a GPU also frees server resources.

II. Background

One of the main challenges in implementing cheat detection and prevention mechanisms in games is the consumption of valuable computational resources by the execution of these detection mechanisms. Game developers strive to create games with the latest features in the fields of graphical effects, in-game physics, etc. These features require already most of the resources available in a computer, both on the server and the client side of a system. A mechanism for cheat detection is only usable if its impact on the application is small, both with regard to performance demands and modifications to the existing infrastructure.

A. Classification of Cheating

Through the evolution of on-line multi-player games, cheating has emerged as a serious problem for game providers. Cheating can ruin in-game economics, turn honest players into cheating players and in the worst case, lead to players

abandoning the game [4]. The diversity of the games being played on the Internet allows for several means of cheating as each genre of games have their own unique characteristics and vulnerabilities. The first important step towards a cheat-free game is to examine and determine which forms of cheating are most likely to be attempted.

An early review of the existence of cheating and its prevention was performed by Matt Pritchard [6]. The paper, aimed at the game development industry, mentions concrete examples of games which have experienced problems with cheating, different game communication models and how cheating applies to these models. The paper also presents several ideas on solving different cheating cases. However, cheating problems were largely investigated and dealt with on a case-by-case basis until Yan and Randell [11] presented an extensive list of different categories and with it, a taxonomy of on-line game cheating. This is a three-dimensional taxonomy based on what are the underlying vulnerabilities, the cheating consequences and the cheating principals. The taxonomy is thorough, but unstructured, so GauthierDickey et al [3] present a more structured taxonomy by categorizing cheats in the layer in which they occur. Continuing from this work, Webb and Soh [9] present an updated review and classification of cheating in networked computer games based on the same categories defined by [3]:

- *Game level cheats* are achieved by breaking the rules or misusing features of the game. Game level cheats do not require any modifications to the game client or the general infrastructure.
- *Application level cheats* include modifications to the code of the game or the operating system. A common form of application level cheats are reflex enhancers and farming bots. Both give the cheater an unfair advantage by boosting such as the accuracy of the aim or allow for automation of certain tasks to let the cheater gain resources while not even playing the game.
- *Protocol level cheats* are changes to the protocol of a game like changing packet contents or delaying packets. Fixed delay cheats are based on introducing a delay before sending packets from the cheater. This delay appears only as latency for the other players and the central server. The delay can allow the cheater to examine all updates received from the other players before choosing an appropriate action based on the acquired knowledge.
- *Infrastructure level cheats* involve modifications and manipulations of game dependent pieces of infrastructure, i.e., modifications to driver, libraries, hardware, network, etc. Information exposure cheats can examine broadcasted network traffic to give additional information to a cheater.

B. Existing Cheat Detection Mechanisms

Because of the many existing forms of cheating, there are many attempted solutions to prevent cheating, both within academics and the game industry. Different types of cheats apply to different types of games, therefore some of the solutions approach different problems with different communication models. Some are designed for client-server systems, others for P2P systems and some are usable with whichever communication model.

There have been several papers suggesting cheat detection systems. In [12], the authors propose a statistical approach to cheat detection based on a dynamic Bayesian network approach. The proposed detection framework relies solely on the game state, and the proposed solution is designed to run on the server side to prevent hacks and tampering. Their experiments show that they are able to effectively detect cheaters and that the false positive rate is low. However, the system needs to be trained to detect specific cheats.

A different approach presented by Feng et al. [2] examines an approach for cheat detection that is based on the use of stealth measurements via tamper-resistant hardware. This is a client side modification, and the authors' solution utilizes the Intel Active Management Technology platform to access contents in the physical memory. They present a range of measurements supported by the hardware that might detect the methods used by hackers to compromise games. The challenges with this system are that specialized hardware is required on all clients and that users might have privacy issues since this approach requires full access to the physical memory on their clients.

There are also several anti-cheating systems that have been put to mainstream use. Three of the most notable are VAC [8], PunkBuster [1] and Warden [10]. The similarity between the three mentioned anti-cheat systems is that they are separate programs that examine programs running alongside the game being played. They inspect the main memory of the computer, searching for programs altering or reading the memory used by the game. Valve reported that over 10.000 cheating players of "Counter-Strike: Source" were caught within a single week in late 2006 by running cheating software [7]. However, one issue with these solutions is that they run in software on the game clients, and the anti-cheating systems are therefore vulnerable for hacks and tampering.

Rather than attempting to solve many forms of cheating, we investigate ways to effectively implement cheat detection mechanisms in games. We focus on parallel hardware and how it can be used to make the impact of a cheat detection mechanism on the game system as transparent as possible. The cheats that can be exposed by our cheat detection mechanism could be application, protocol and infrastructure level cheats, but in particular such cheats that involve modifications to the client application and network packets to improve game physics properties. Cheats have also been discovered where clients increase or decrease the internal clock speed of their processors, increasing simulation speeds so that objects may accelerate faster. These kinds of cheats can also be discovered.

C. Nvidia Graphics Processing Units

A GPU is a dedicated graphics rendering device. Modern GPUs have a parallel structure, making them effective for general-purpose processing. Previously, shaders were used for
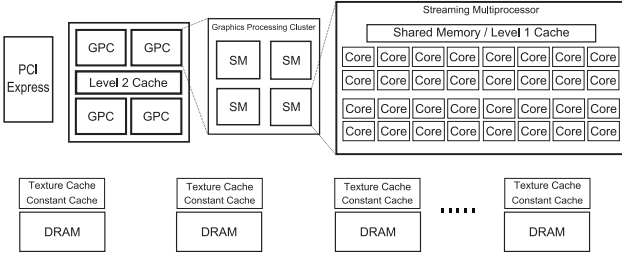
Figure 1. Nvidia GF100 Compute Architecture

general-purpose programming, but specialized languages are now available. Nvidia has released the CUDA framework with a programming language similar to ANSI C.

The latest generation of GPUs available from Nvidia, illustrated in Figure 1, is the GF100. This generation is often referred to as the Fermi compute architecture. The GF100 chip is presented to the programmer as a highly parallel, multi-threaded, multi-core processor. The GF100 architecture contains up to 512 simple processing cores [5].

GPUs have other memory hierarchies than an x86 processor. Several types of memory with different properties are available to the programmer. Each thread has some private local memory, and the threads running on the same stream multiprocessor (SM) have access to some shared memory. Two additional read-only memory spaces called constant and texture are available to all threads. Finally, there is global memory that can be accessed by all threads. The GF100 architecture also introduces an L1 cache and a unified L2 cache for all operations to global and texture memory.

III. EXAMPLE GAME

To show the benefits of using a GPU for cheat detection, we created a simple space race game simulation, where the spacecrafts must visit virtual positions, also referred to as targets. The clients are placed randomly in the virtual world, giving some clients an advantage as they might be placed closer to a target. When a target is reached, the clients continue to the next target. Figure 2 shows a GUI representation of player objects in the game.

The simulation follows a client-server based game architecture, where all clients send their position updates to the server. This approach is chosen for the same reasons as in consumer market game development: ease of development, total control of client communication and a centralized control point. Discrete clients are created within the simulation, and communication follows the same flow that would be normal in a networked multi-player game. Furthermore, because we wanted to design our simulation independent of wallclock time, we used an artificial timeline based on game ticks. A tick is a theoretical time duration specified in the configuration of the system.

To allow reproducible tests, the simulation uses two different modes of operation named generation mode and playback mode. The *generation mode* uses the principles of the game to determine random placement of a given number of clients in
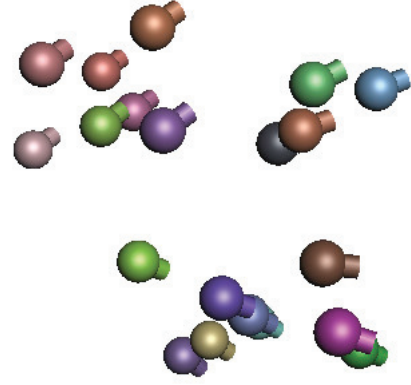

Figure 2. Screenshot of the graphical representation of player objects in the virtual environment.

a virtual environment. From these positions, the clients try to reach the closest target. After a target is reached, they continue to the next target. They use a thruster to propel themselves around. External forces, such as gravity, affect the clients. While this is happening, the server writes each client's location in the virtual environment to several files. These files are used in playback mode. The generation mode generates also movement for cheaters. The numbers of cheaters can be adjusted in generation mode. A cheater behaves in the same manner as an honest client, but regularly performs unrealistic motions. *Playback mode* initializes the clients. The client state information is read from the files generated in generation mode, and the states are reported to the server. The server samples the state information updates from every client, putting the samples in a sample buffer. The buffer is read by the cheat detection thread when full.

Because all clients in the game are controlled by the computer, some rules must be set their behaviour in trying to reach a target. To reach their targets, the clients require motion planning. We have not implemented any advanced motion planning algorithms for this paper. The clients know the targets that they have reached. After a target is reached, the client continues to the closest unaccomplished target. The movement of a client is restricted by the physical model. Honest clients do not break the rules of the model, while cheating clients do.

In our simulation, the objects experience both linear and angular acceleration. There is a constant gravitational pull, affecting the objects, much like the gravity on Earth. All the other forces are generated by the objects themselves using thrusters. Figure 3 shows an outline of a game object, with a main rear thruster and bow thrusters. Objects move forward with the rear thruster and rotate using the bow thrusters. The size and thruster power can be modified by parameters.

The physics engine is one of the main parts of the simulation. The engine is responsible for calculating the sum of all physical forces acting on all objects and updates their positions accordingly. The physics engine is controlled by configuration parameters that allow for changing physical properties quickly,
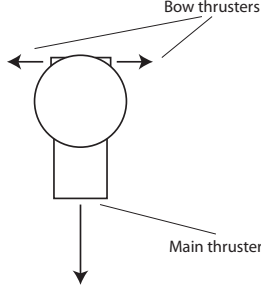
Figure 3. Illustration of a game object with bow thrusters in the front and the main thruster at the back.

even during runtime. Game objects are registered with the physics engine, so it maintains a pool of objects to manage. Updates of the parameters of an object, such as throttle, are handled by the individual clients. The integrations of the time steps from a game tick to the next are done by the engine. The physics engine does this by updating every game object in the object pool. The main implementation of the physics engine runs on the CPU and is only used during generation mode. During playback mode, the cheat detection mechanisms act as a reverse physics engine. They try to determine if the position updates are valid within the current physical model.

The physical model used in this example is a simple model, with only a couple of physical effects. The most basic of these effects is *linear motion*. Basic linear motion is implemented using Newton's second law of motion as shown in equation 1:

$$\sum \mathbf{F} = m\mathbf{a} \qquad (1)$$

The law states that the sum of all forces acting on an object is the product of the mass and its acceleration. The acceleration is measured by looking at the change in speed over a known distance. In our game, there are two linear forces acting on an object. The first is the acceleration applied by the game object's main thruster as illustrated by figure 3. The second is the vertical gravity that is constant in the entire model. The total linear force is represented by the sum of these two vectors.

The second physical effect is *angular motion*. To allow object rotation in all dimensions, the properties of the objects in the game must be extended. Similar to the linear motion properties of distance, velocity and acceleration, we have angular motion properties. The equations 2 and 3

$$\Omega = \frac{d\omega}{dt} \qquad (2)$$

$$\omega = \frac{d\alpha}{dt} \qquad (3)$$

where Ω is the angular displacement of an object in radians, ω is the angular velocity in radians per second, and α is the angular acceleration in radians per second squared, show how these relate to each other.

Angular motion is applied to the game objects when the bow thrusters illustrated in figure 3 are used to change the

course of a object. Support for collisions is implemented in the model. However, due to the lack of time, it has not been implemented in the cheat detection mechanisms. It is only present in generation mode.

There are different ways to perform a cheat in the simulation. Clients cheat either by modifying the power of their thruster temporarily or by modifying the values of their current state: their position, velocity and rotation. If a cheating client temporarily increases the thrust capabilities of one of its thrusters, it is able to accelerate faster in a direction to perform quicker turns or pick up speed faster. Cheaters who change their state can position themselves closer to a target or change their rotation to point towards a target. They might also increase or decrease the magnitude of their velocity vector when either dashing for a target or slowing down to prevent passing a target.

IV. IMPLEMENTATION

We have implemented two versions of the cheat detection mechanism. One is written for the host CPU, while the other is a CUDA version, written for the GPU device. The cheat detection mechanism on the GPU is implemented with threads. The CPU implementation is not threaded and uses a basic looping structure to simulate the same behavior as the CUDA version.

The behavior of the mechanisms is illustrated in figure 4. A single thread works on three consecutive game state samples for a client, thread one (th1) works on sample s0, s1 and s2, while thread two (th2) works on sample s1, s2 and s3 etc. A sample is the state of the client after a tick in the artificial timeline.
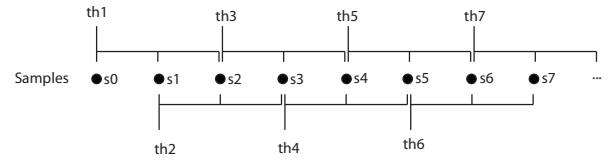
Figure 4. Sample reading and execution pattern of the threads.

A sample contains the movement of each client and a positional vector with three values: x, y and z according to the three-dimensional axes. With three samples the threads can find the acceleration of the client as a three-dimensional vector. All external forces added by the physical model can now be subtracted by applying the calculations of the physical engine in reverse. The resulting acceleration is the result of the forces the client has applied to the game object. If the thrust applied by the client is greater than the maximum thrust allowed by the game, the client is most likely a cheater.

There are two main node types in our simulation; the server and the clients. They exchange data as in real networked games. A packet is either generated by the generation mode or read from file in playback mode by the clients once for each game tick.

The *server* reads all incoming data from the clients. When a cheater reports erroneous positional data, the cheat detection
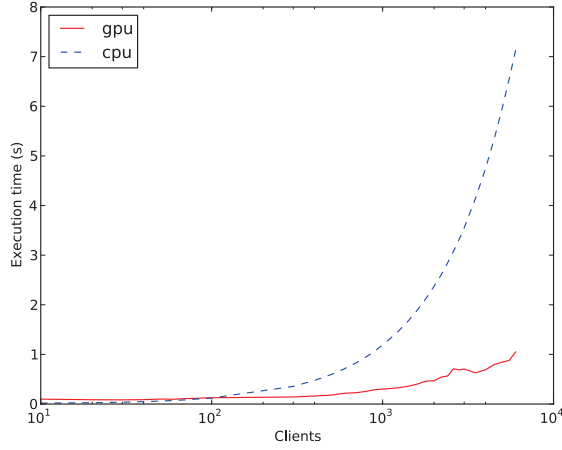
Figure 5. Execution time (in seconds) of the cheat detection mechanism on the GPU and the CPU.

Figure 6. Percent of time spent on cheat detection processing on host using the GPU and the CPU.

mechanisms indicates that the movement of the player does not follow the rules and restrictions of physical parameters of the game.

Clients act differently depending on the execution mode. During the generation of movement files, clients write their locations and other appropriate data to file. In playback mode, clients read from the generated files and report the data written in generation mode back to the server. In this way, the system allows for reproducible tests as the test data is the same for each test run.

V. EVALUATION

In this section, we describe the performance of our solution by presenting the experimental results. We investigate both the total execution time of the cheat detection system and the total execution time spent on the cheat detection mechanisms. All tests were run on data generated in generator mode over 100 seconds of "game time". The number of clients used in the benchmarks ranges from 10 to 6000. The part of the mechanisms that runs on the GPU in these benchmarks is the reverse physics engine.

The cheat detection mechanism we tested is implemented on the following hardware: the CPU used in the benchmarks was an Intel Core i5 750 processor running at 2.66 GHz with 4.0 GB RAM. The GPU was an Nvidia GeForce GTX 480 with 480 processing cores, 1.5 GB memory and version 3.1 of the Nvidia CUDA framework.

The results of the first benchmark are shown in figure 5. It shows the total execution time of the cheat detection system. We can observe that with a low number of clients, the CPU is faster than the GPU. The reason for this is the added latency of moving data and code to the GPU. With more than 100 clients in the game, the execution time for the CPU exceeds that of the GPU, and the performance gap steadily increases up to 6000 clients, which is the maximum number of tested clients. This is due to the size of the memory on our test machines. When the number of clients increases, the cheat detection processing on the GPU scales much better than on the CPU.
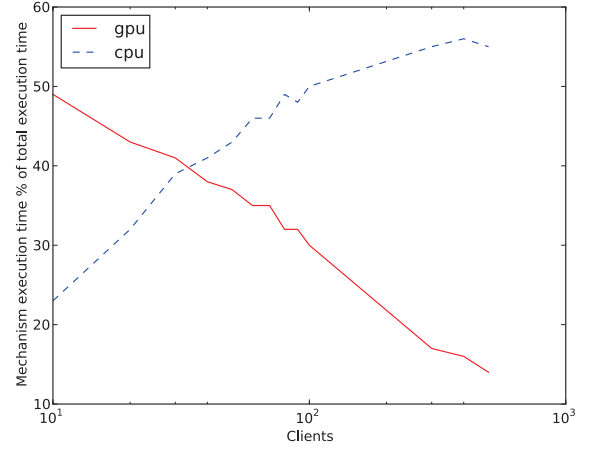
When the cheat detection mechanism is processed on the GPU, the CPU is relieved of performing these tasks and can work on other game relevant computation.

To determine the offloading effect the GPU has on the CPU, we have measured how much of the total execution time is spent on processing cheat detection mechanisms. Figure 6 shows the results of the second benchmark. The results show that for a small number of clients, the penalty for transferring data over the PCI Express bus to the GPU is significant, making the CPU more effective for a small number of clients. With more than 50 clients, the GPU implementation spends less time on cheat detection than the CPU implementation. As the number of clients increases, the time spent on cheat detection continues to drop to below 15 percent for the GPU implementation. The CPU version stabilizes around 50 percent. To improve the performance of the GPU implementation with a low number of clients, it is possible to buffer more samples before executing the mechanisms on the GPU.

VI. DISCUSSION

We have seen how the CPU and the GPU implementations of our cheat detection mechanism perform differently when we increase the numbers of clients in the game. The difference between the two is smallest when the number of checks performed on the GPU is small. However, as the number of clients increases, the increase in execution time of the CPU implementation is much steeper compared to the increase in the GPU implementation. This indicates that the GPU implementation is the more scalable of the two. It is primarily due to the highly parallel architecture of the GPU. Physics operations for a large numbers of clients are independent of each other. They constitute an embarrassingly parallel workload that maps well to the multi-threaded architecture of the GPU. As further

work, both the CPU and the GPU implementation can be further optimized. The CPU implementation can be extended with threading and SIMD operations, and the GPU version can be extended with asynchronous transfers, optimized access to global memory accesses and elimination of branching in the compute kernels.

The cheat detection mechanism we have implemented for our system is easy to parallelize because physics computations for clients are independent of each other. Similar systems with workloads that contain operations that can be performed simultaneously by a large number of threads can benefit from using a GPU to offload the processing. When offloading operations to a GPU, it is important to remember that the GPU is most efficient if it has enough data to process. It is also important that the tasks offloaded map well to the multi-threaded architecture of the GPU. Operations that require only a few calculations over a small number of threads do not run very efficiently on a GPU. This is mainly due to the delay associated with transferring data and code over the PCI Express bus to the GPU. We can also observe this effect in our benchmarks when the number of clients playing the game is reduced to below 50. With the next generation of CPUs from Intel and AMD, we see a trend with GPUs integrated as a part of the CPU die. Such solutions might reduce the overhead of offloading computations to the GPU.

A challenge in implementing parts of a program on a GPU is that developers have to think differently compared to a CPU implementation. A GPU implementation requires much more tweaking and optimization to reap the full benefits of the architecture.

Although we have experimented with cheat detection in a game simulation, the GPU can be used for several additional tasks. If the game uses a physics engine that supports GPU execution, it might be able to perform all physics calculations on the server. This will reduce the control of the game clients and remove the need for a cheat detection mechanism for consistency of movements for entirely. This can also contribute to lowering the hardware requirements on the client side of the game. The cost will however increase on the server side, since game servers traditionally have not been equipped with GPUs.

VII. Conclusion

Even though there is an increasing popularity of on-line multi-player games, cheating is prevalent. This destructive behavior degrades the gaming experience of honest game players. The game industry has always been a step behind the cheaters, struggling to keep up with new and creative cheating methods. Although the existing solutions are not sufficient to eliminate cheating, there is an increasing amount of research attempting to reduce cheating in on-line multi-player games. Because of the large diversity of the types of existing on-line games, the existing cheats and the cheating mechanism that aim to battle them, are equally diverse. In this paper, we have investigated how GPUs perform compared to CPUs with respect to processing cheat detection mechanisms.

Our results show that a system processing a cheat detection mechanism on a GPU can outperform the same mechanism running on a CPU, even with only a simple physical model, but depending on the number of clients due to the GPU data transfer costs. Although cheat detection mechanisms vary greatly from game to game, a mechanism that checks for consistency in physical calculations can be migrated to the GPU to achieve a performance boost. We have also observed that it is able to offload the CPU by moving the processing of a cheat detection mechanism to the GPU allowing the CPU to perform other tasks while the cheat detection mechanism is executing.

References

[1] Even Balance, Inc. PunkBuster Online Countermeasures. *http://www.evenbalance.com/*, Accessed July 2010.

[2] W.-c. Feng, E. Kaiser, and T. Schluessler. Stealth measurements for cheat detection in on-line games. In *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 15–20, Worcester, Massachusetts, USA, 2008.

[3] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 134–139, Cork, Ireland, 2004.

[4] P. Kabus, W. W. Terpstra, M. Cilia, and A. P. Buchmann. Addressing cheating in distributed mmogs. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–6, Hawthorne, NY, USA, 2005.

[5] NVIDIA. NVIDIA Next Generation CUDA Compute Architecture: Fermi. *http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf*, Accessed August 2010.

[6] Pritchard, M. How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It. *http://www.gamasutra.com/features/20000724/pritchard_pfv.htm*, Accessed May 2009.

[7] Valve. Steam Message. *http://storefront.steampowered.com/Steam/Marketing/message/837/?l=english*, Accessed July 2010.

[8] Valve. Valve Anti-Cheat System. *https://support.steampowered.com/kb_article.php?p_faqid=370*, Accessed July 2010.

[9] S. D. Webb and S. Soh. Cheating in networked computer games: a review. In *DIMEA '07: Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts*, pages 105–112, Perth, Australia, 2007.

[10] Wikipedia. Warden (software). *http://en.wikipedia.org/wiki/Warden_(software)*, Accessed July 2010.

[11] J. Yan and B. Randell. A systematic classification of cheating in online games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–9, Hawthorne, NY, USA, 2005.

[12] S. F. Yeung and J. C. S. Lui. Detecting cheaters for multiplayer games: Theory, design and implementation. In *NIME '05: IEEE International Workshop on Networking Issues in Multimedia Entertainment*, pages 1178–1182, Las Vegas, Nevada, USA, 2005.

Paper V: Reducing Processing Demands for Multi-Rate Video Encoding: Implementation and Evaluation

Title: Reducing Processing Demands for Multi-Rate Video Encoding: Implementation and Evaluation [34].

Authors: H. Espeland, H. K. Stensland, D. H. Finstad, P. Halvorsen.

Published: International Journal of Multimedia Data Engineering and Management (IJMDEM), Volume 3, Issue 2, IGI Global, 2012.

Reducing Processing Demands for Multi-Rate Video Encoding:
Implementation and Evaluation

Håvard Espeland, University of Oslo and Simula Research Laboratory, Norway

Håkon Kvale Stensland, University of Oslo and Simula Research Laboratory, Norway

Dag Haavi Finstad, University of Oslo and Simula Research Laboratory, Norway

Pål Halvorsen, University of Oslo and Simula Research Laboratory, Norway

ABSTRACT

Segmented adaptive HTTP streaming has become the de facto standard for video delivery over the Internet for its ability to scale video quality to the available network resources. Here, each video is encoded in multiple qualities, i.e., running the expensive encoding process for each quality layer. However, these operations consume both a lot of time and resources, and in this paper, the authors propose a system for reusing redundant steps in a video encoder to improve the multi-layer encoding pipeline. The idea is to have multiple outputs for each of the target bitrates and qualities where the intermediate processing steps share and reuse the computational heavy analysis. A prototype has been implemented using the VP8 reference encoder, and their experimental results show that for both low- and high-resolution videos the proposed method can significantly reduce the processing demands and time when encoding the different quality layers.

Keywords: Multimedia, Multi-Rate, Performance, Quality Adaption, Video Encoding, VP8

INTRODUCTION

The amount of video data available in the Internet is exploding, and the number of video streaming services both live and on-demand, is quickly increasing. For example, consider the rapid deployment of public available Internet video archives providing a wide range of content like newscasts, movies and scholarly videos. In this respect, Internet users uploaded one hour

of video to YouTube every second in January 2012 (YouTube, 2012). Furthermore, all major (sports) events like European soccer leagues, NFL Hockey, NBA basketball, NFL football, etc. are streamed live with only a few seconds delay, e.g., bringing the 2010 Winter Olympics (Zambelli, 2009), 2010 FIFA World Cup (Move Networks, 2008) and NFL Super Bowl (Move Networks, 2008) to millions of concurrent users over the Internet, supporting a wide range of devices ranging from mobile phones to HD displays. The number of videos streamed from such services are in the order of tens of billions

per month (Flosi, 2010; YouTube, 2012), and leading industry movers conjecture that traffic on the mobile-phone networks will also soon be dominated by video content (Cisco Systems, Inc., 2010).

Adaptive HTTP streaming is frequently used in the Internet and is currently the de facto video delivery solution. For example, Move Networks (2008) was one of the first providers of segmented adaptive HTTP streaming, and has later been followed by major actors like Microsoft (Zambelli, 2009), Apple (Pantos et al., 2010) and Adobe (2010). Recently, there are also standardization efforts by MPEG (Stockhammer, 2011). In these systems, the bitrate (and thus video quality) can be changed dynamically to match the varying bandwidth, giving a large advantage over non-adaptive systems that are frequently interrupted due to buffer underruns or data loss. The video is thus encoded in multiple bitrates matching different devices and different network conditions.

Today, H.264 is the most frequently used codec. However, an emerging alternative is the simpler VP8 which is very similar to H.264's baseline profile and supposed to be well suited for web-streaming with native support in major browsers, royalty free use and similar video quality as H.264 (Seeling et al., 2010; Ozer, 2010). Nevertheless, for both codecs, the challenge in the multi-rate scenario is that each version of the video requires a separate processing instance of the encoding software. This may be a challenge in live scenarios where all the rates must be delivered in real-time, and in the YouTube case, it will just take an enormous data center to keep the upload rate. Thus, the process of encoding videos in multiple qualities and data rates is both time and resource consuming.

Our initial idea was presented by Finstad et al. (2011) and in this paper, we expand our evaluation and further discuss our results. In particular, we analyze the processing overheads of multi-rate scenarios, and to reduce resource requirement, we investigate possibilities for reusing the output from different steps in the encoding pipeline as the same video elements are processed multiple times with only slightly different parameters. We use the VP8 processing pipeline as a case study, and we present a prototype that supports running multiple parallel VP8 encoder instances. The main contributions of our work are:

- Inspired by several transcoding approaches trying to reuse motion vectors (Kuo & Jayant, 2003; Zhou, Zhou, & Xia, 2008; Senda & Harasaki, 1999), we propose a way of reusing decisions from intra and inter prediction in a video encoder to avoid computational expensive steps that are redundant when encoding for multiple *target bitrates*, i.e., macroblock mode decision, intra prediction and inter prediction between the instances. The method can be used in any video codec comprising an analysis and encoding step with similar structure as H.264 and VP8.
- A prototype and demonstrator has been implemented using the VP8 reference encoder as a case study. We have performed a wide range of experiments using both low- and high-resolution videos with various data rates and content types. The results show that the computational demands are significantly reduced at the same rates and approximately the same qualities as the VP8 reference implementation.

RELATED WORK

Our proposed multi-rate encoder is based on the idea of sharing and reusing the results from the computational expensive steps. The idea is inspired by the Scalable Video Coding technique (Schwarz et al., 2007), but differs in key aspects: First, no special client player support is necessary. SVC has very little market penetration in comparison to VP8 player availability, which in contrast makes the multi-rate technique useful without requiring changes for the users. Second, the layers of SVC affect the output quality (Wien et al., 2007), which also may be a reason why it has not been widely adopted for online streaming. In our approach

we optimize the currently used VP8 codec in a way that is more similar to transcoding. In the area of video transcoding, there are several approaches where motion vectors are reused (Kuo & Jayant, 2003; Zhou et al., 2008; Senda & Harasaki, 1999; Youn, Sun, & Lin, 1999). For example, in context of spatial downscaling, methods for synthesizing a new motion vector by reuse of the original motion vectors from the higher resolution bit-stream can be performed. In this context, Kuo and Jayant (2003) discuss transcoding with the reuse of motion vectors. Their paper investigates the statistical characteristics of the macroblocks associated with the best matching motion vectors, and they then define a likelihood score, which is used for picking the motion vectors. Similarly, Zhou et al. (2008) propose a spatial downscaling algorithm reusing the motion vectors, and a more advanced method for refining the synthesized vectors is discussed. Furthermore, Senda and Harasaki (1999) describe a real-time software transcoder with motion vector reuse. They discuss a method for reusing downscaled motion vectors, which evaluate the scaled motion vectors and their neighbors. A new method for reducing the number of candidate motion vectors is proposed, and the best one is picked by finding the one with the lowest mean absolute error. Transcoding is also investigated by Youn et al. (1999) where they observed that reusing the incoming motion vectors become non-optimal, due to the reconstruction errors. They are proposing to use a fast-search adaptive motion vector refinement to improve the motion vectors. Further, Zhou and Sun (2004) propose a new algorithm to do fast inter-mode decision and motion estimation for H.264, but both these approaches are different from our proposed solution. Fonseca et al. (2007) proposed using an open loop in H.264 that is to use the original frame as input to the prediction analysis instead of the reconstructed frame. By doing this, they showed that the quality loss was small and varied with the HD sequences evaluated. The open loop approach can be used to remove the intra-frame dependencies, and Wu et al. (2009) made a parallel version of x264

for a stream processor. Our approach does not use the original frame for prediction analysis; instead we are reusing prediction parameters for other bitrates which is closer to the reconstructed frame than the original.

We have briefly outlined some examples where the video processing reuses the output of previous executions. However, none of these papers reuse the analysis step for use with several encoder instances. In our work, we target the modern multi-rate adaptive HTTP streaming solutions with the goal of decreasing the processing requirement when uploading new content to a video archive or reducing the latency in a live streaming scenario. In summary, we investigate the possibility for reusing data from parts of the encoding pipeline to be able to output multiple video streams, and we therefore next present a brief overview of the VP8 codec. We show the basic processing pipeline, and we identify the most expensive operations by presenting some VP8 profiling results.

THE VP8 CODEC

The VP8 codec (Bankoski, Wilkins, & Xu, 2011) was originally developed by On2 Technologies as a successor to VP7, and is a modern codec for storing progressive video. After acquiring On2 Technologies in 2010, Google released VP8 as the open source *webm* project, a royalty-free alternative to H.264. The webm format was later added as a supported format in the upcoming HTML5 standard, several major browsers have implemented playback support for the format since webm is expected to be one of the major streaming formats on the web in the coming years.

Many of the features in the VP8 codec are heavily influenced by H.264. It has similar functionality as the H.264 Baseline Profile. One of the differences is that VP8 have an adaptive binary arithmetic coder instead of CAVLC. VP8 is however not designed to be an all-purpose codec; it primarily targets web and mobile application. This is why VP8 has omitted features such as interlacing, scalable coding, slices and

color spaces other than 4:2:0. This reduces encoder and decoder complexity while retaining video quality for the most common use case, i.e., making VP8 a good choice for lightweight devices with limited resources.

A VP8 frame is either of type *intra-frame* or *inter-frame*, corresponding to I- and P-frames in H.264, and it has no equivalent to B-frames. In addition, VP8 introduces the concept of tagging a frame as *altref* and *golden* frames, which are stored for reference in the decoder. When predicting, blocks may use regions from the immediate previous, from the last *golden* or from the last *altref* frame. Every key frame is both a *golden* and *altref* frame; other frames can optionally be marked as *golden* or *altref*. *Altref* frames are special and never shown to the user; instead they are only used for prediction.

The encoding loop of VP8 is very similar to that of H.264. The process consists of an analysis stage, which decides if intra or inter prediction shall be used, DCT, quantization, dequantization, iDCT, followed by an in-loop deblocking filter. The result of the quantization step is entropy coded using a context adaptive boolean entropy coder and stored as the output bitstream. The output bitrate of the resulting video is dependent on the prediction parameters in the bitstream and quantization parameters.

MULTI-RATE ENCODING

The multi-rate encoder is based on the reference VP8 encoder, released as part of the webm project. Provided in Figure 1 is a simplified call graph of the VP8 reference encoder. In this call graph, we can see the flow of the program, how many times a function has been called, and how large percentage of the execution time is spent in different parts of the code. The basic flow of the entire encoder is illustrated in the upper part of Figure 2 with an analysis and encode part of the pipeline.

The *analysis* part consists of macroblock mode decision and intra/inter prediction, this corresponds to vp8_rd_pick_inter_mode in Figure 1. The *encode* part refers to transform,

quantization, dequantization and inverse transform, corresponding to the functions vp8_encode_inter* and vp8_encode_intra* for the various block modes chosen. *Output* involves entropy coding and writing the output bitstream to file, this part of the encoder is not shown in the call graph. Our profiling of the VP8 encoder give similar results as for H.264 (Huang et al., 2006; Espeland, 2008), and shows that during encoding of the *foreman* test sequence, over 80% of the execution time is spent in the analysis part of the code, i.e., if this part can be reused for encoding operations for other rates, the resource consumption can be greatly reduced. This can be done since the outputs have identical characteristics except for the bitrate, and the main difference between them is the quantization parameters, that is regardless of the *target bitrate*, the analysis step which includes searching for motion vectors and other prediction parameters can be done without considering the *target bitrate*, consequently trading complexity for prediction accuracy. The prediction accuracy is reduced since the input to the analysis step does not match exactly the reconstructed frame (which has been quantized differently). The reduced prediction accuracy will lead to some degradation in quality given the same bitrate, and we will evaluate the effects in a variety of scenarios in this paper.

To evaluate this approach, we have implemented a VP8 encoder with support for multiple outputs. In our approach, we reuse a single analysis step for several instances of the encoding part as seen in Figure 2. This mitigates the requirements for re-doing the computational heavy analysis step, and at the same time allows the encoding instances to emit different output bitrates by varying the quantization parameters in the encoder step. The encoder starts one thread for each specified bitrate where each of these threads corresponds to a separate encoding instance. The instances have identical encoding parameters such as keyframe interval, subpixel accuracy, etc., except for the *target bitrate* provided. Since the bitrate varies, each instance must maintain its own state and reconstruction buffers. The threads

Figure 1. Profile of the main parts of the reference VP8 encoder
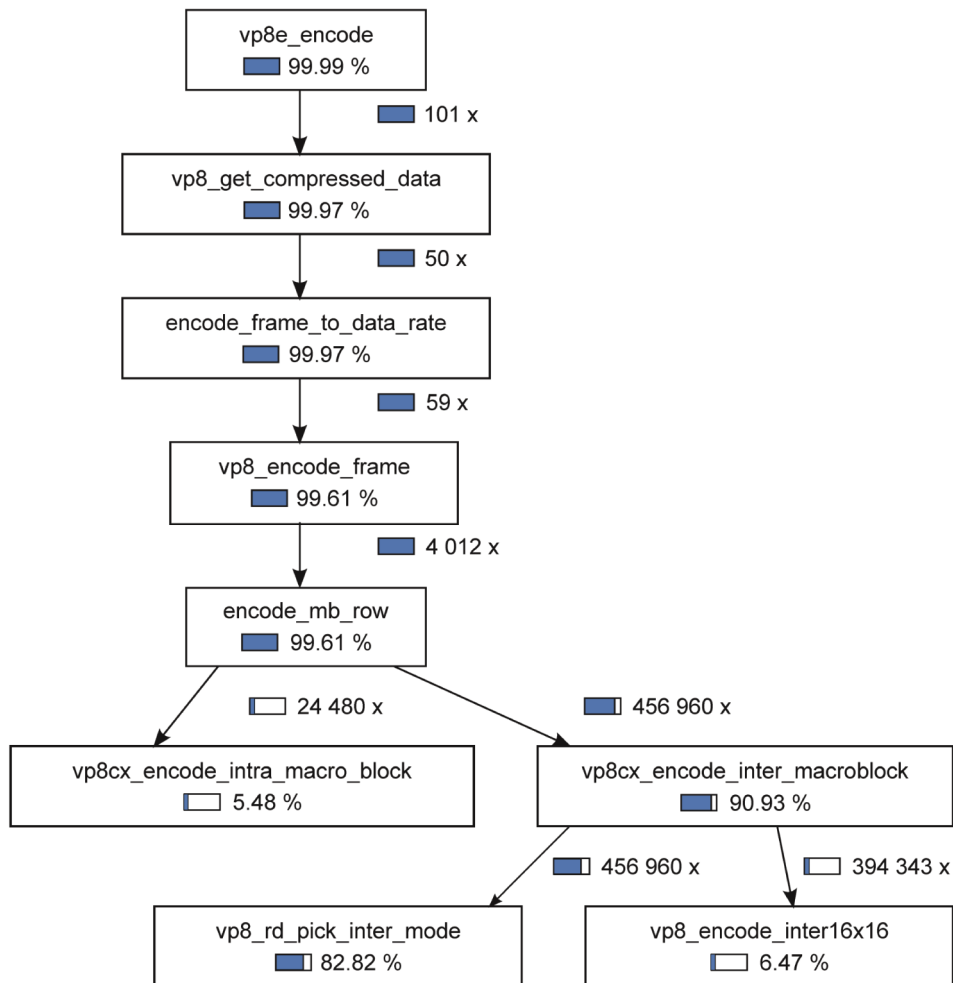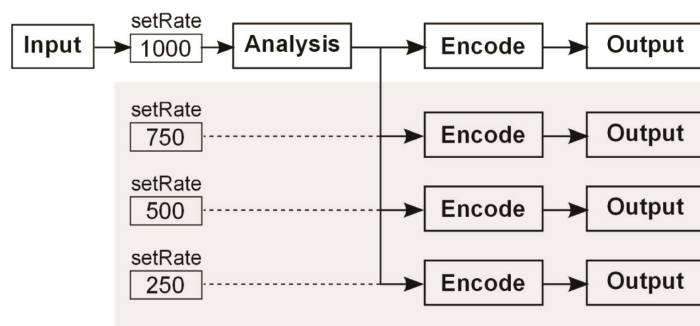

Figure 2. Encoder pipeline diagram

are synchronized on a frame by frame basis, where the main encoding instance analyses the frame before the analysis computations are made available to the other threads. This involves macroblock mode decision, intra- and inter-prediction. The non-main encoding instances reuse these computations directly without doing the computationally intensive analysis steps. Most notably vp8_rd_pick_inter_mode (Figure 1) is only performed by the main encoding instance. Since the VP8 encoder is not written with the intention of running multiple encoding instances in parallel the encoder went through significant changes in order to adapt it to run multiple instances in parallel.

EXPERIMENTS

We have performed experiments for several scenarios. One example is streaming to *mobile devices* over 3G networks. Here, Akamai (2010) recommends that video should be encoded at 250 kbps for low quality and 450 kbps for high quality. Typical 3G networks can deliver bandwidths of 384 kbps (UMTS) to 7.2 Mbps (HSDPA), and we have here measured the resource consumption for coding the standard *foreman* test sequence in *CIF* resolution using bitrates of 250, 450, 750 and 1000 kbps. Furthermore, to also test the other end of the scale, we have evaluated *HD* resolution videos. Typical ADSL lines can deliver from about 750 kbps to 8 Mbps, and for this scenario, we have performed experiments using the 1080p resolution standard test sequences *pedestrian* and *blue sky* encoded at 1500, 2000, 2500 and 3000 kbps. To measure the performance, we have used *time* to measure the consumed CPU time. Additionally, to evaluate the resulting video quality, the PSNR values are measured by the VP8 video encoder. Bitrates shown in the plots are the *resulting bitrates* achieved in output bitstreams, and not the specified *target bitrates*. *Resulting bitrates* are generally a bit lower than *target bitrates* because of a conserva-

tive rate estimator. All experiments were run on a 4-core Intel Core i5 750 processor.

Encoding Results

To evaluate our multi-rate encoder, we have first plotted the total CPU time used when encoding the *foreman* sequence in Figure 3(a) for the four different output rates. To see if there is a difference for different chosen *prediction bitrates* when using the multi-rate encoder, we have included one test for each prediction bitrate. These results are compared to the combined CPU time used when encoding the videos for the same rates using the reference encoder with both a single thread and multiple threads. The CPU time used in the multi-rate approach is more than 2.5 times faster than encoding the four sequences using the reference encoder. The multi-rate approach scales further if the number of encoded streams is increased. In addition, the time spent in kernel space is far less in the multi-rate approach compared to the reference encoder, and we believe this is a result of reading the source video from disk only once.

To see if there are differences between low and high resolution videos, we have also looked at *HD* sequences to validate our approach. Figure 4(a) shows the *pedestrian* test clip with a *prediction bitrate* of 2000 kbps. We observe a 2.06 times reduction in CPU time for the multi-rate encoder as we saw for the *foreman* sequence.

Finally, since we reuse motion vectors for the encoding, we looked at different videos with different amount and kind of motion. The *pedestrian* has a fixed camera with objects (people) moving. The *blue sky* video has more or less fixed objects, but with a moving camera. The *blue sky* results are plotted in Figure 5(a) with a performance gain of 2.47 the performance of the reference encoder. Thus, for all our experiments using different rates, resolutions and content types, our multi-rate encoder reduce the total resource consumption.

Figure 3. CIF streaming scenario foreman

(a) CPU time

(b) Rate-distortion curve
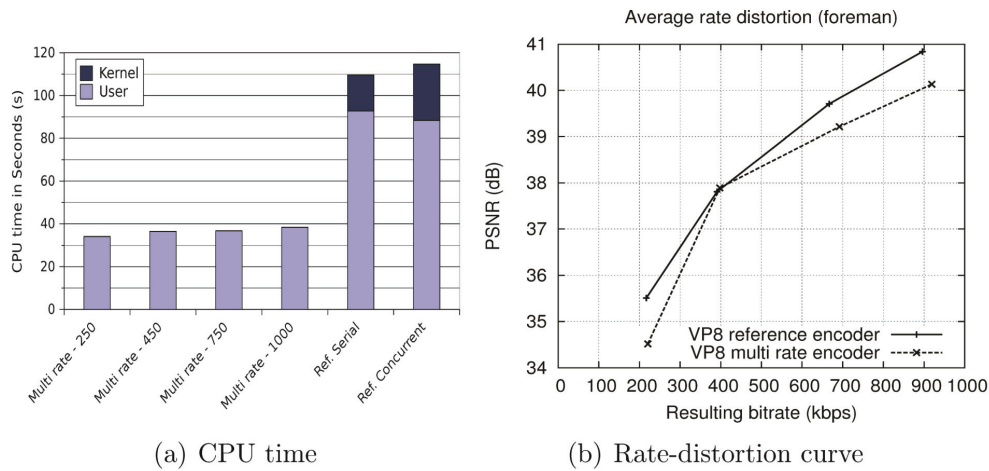
Quality Assessment

Using prediction parameters generated from a different bitrate than the *target bitrate* does have implications for the video quality. To investigate the tradeoff between reduced processing time versus degraded video quality, we have plotted a rate-distortion curve for the *foreman* sequence with a *prediction bitrate* of 450 kbps in Figure 3(b). We can see that reference encoder produces

about 1 dB higher peak signal-to-noise ratio (PSNR) at the same bitrate than the multi-rate encoder. Depending on the intended usage, the significantly reduced CPU time might outweigh the small reduction in quality.

The degradation in video quality is due to the instances' reuse of analysis computations. As described in the multi-rate encoding section, the analysis part of the encoder pipeline is only carried out by the encoder instance
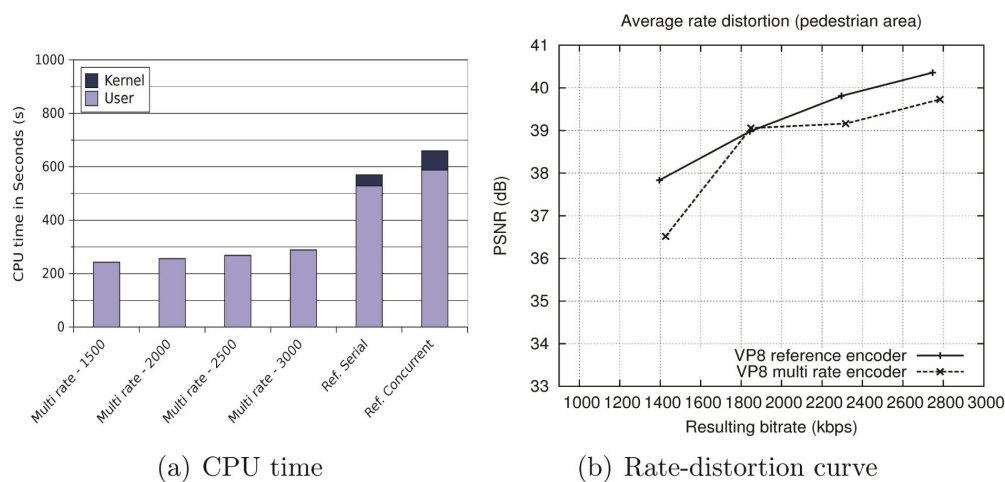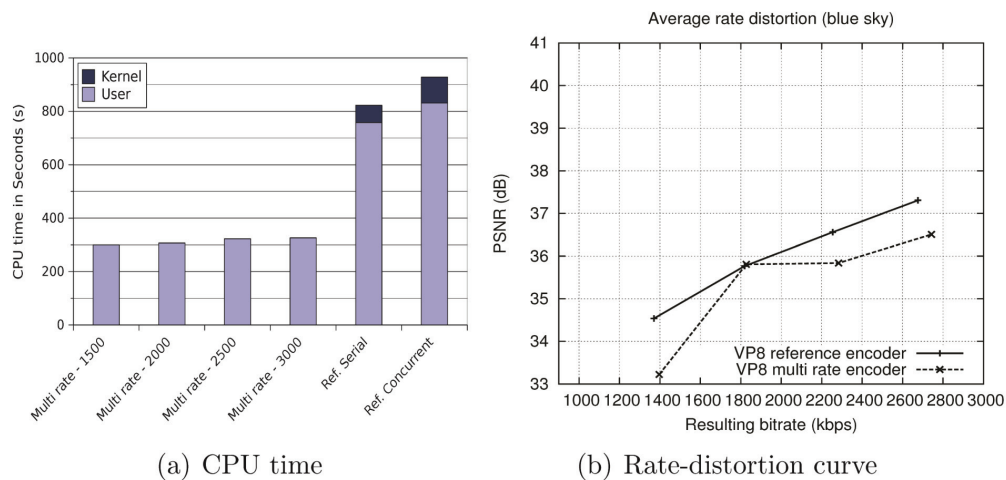
Figure 4. HD streaming scenario pedestrian

(a) CPU time

(b) Rate-distortion curve

Figure 5. HD streaming scenario blue sky

(a) CPU time

(b) Rate-distortion curve

targeting the *prediction bitrate*, and is hence only subject to the constraints of this instance. The mode decisions and motion vectors for the other instances will differ from the optimal parameters (as chosen by the normal version) and this leads to degradation in video quality.

Similarly, when considering the distortion of the *HD* sequences, we have plotted rate-distortion curves in Figure 4(b) and Figure 5(b) for *pedestrian* and *blue sky*, respectively. The reference encoder produces output that has 1.0 to 1.5 dB higher PSNR than the multi-rate encoder and distortion achieved for the two *HD* clips are very similar.

The suitability of PSNR for video quality assessment is frequently discussed, and it is often unclear what the difference means in terms of the logarithmic scale. From the plot in Figure 4(b), we can see that the PSNR of the output from the reference encoder is up to 1.32 dB better than the multi-stream encoder outputs for the *pedestrian* sequence, in the range of 1500 kbps to 3000 kbps. To see what this really means, a sample output of the "worst-case" scenario from Figure 4(b) can be seen in Figure 6. From this output, we can see that there is little visual difference between the reference encoder output and the multi-stream encoder. We also looked at the average structural similarity (SSIM) index

number for the reference encoder and the multi-stream encoder. The SSIM numbers are 0.861 and 0.837, respectively, i.e., the difference is small. Thus, the quality reduction is small (we did not see a difference viewing the resulting videos, but it might be different for other types of content). In Figure 7, the "worst-case" scenario from Figure 3(b) can be seen. In this sequence, the PSNR for the reference encoder is of 0.99 dB better than the multi-rate encoder. The SSIM index numbers for this scenario are 0.873 for the reference encoder and 0.856 for the multi-stream encoder.

Choosing the Prediction Bitrate

To evaluate which *prediction bitrate* gives the minimal distortion of the videos, we have plotted rate-distortion curves for *foreman* with various prediction rates in Figure 8. We can see that the *resulting bitrate* is lower for the multi stream encoder than the reference encoder, except for when the *prediction bitrate* exactly matches the *target bitrate*, resulting in a small spike in the plot.

The lowest *prediction bitrate* (250 kbps) incurs the largest distortion difference of 2 dB for the 1000 kbps *resulting bitrate*. When using a 450 kbps *prediction bitrate*, the distortion

Figure 6. Quality difference for the "worst-case" scenario in Figure 4(b) of 1.32 dB PSNR of 1500 kbps

(a) Reference encoder

(b) Multi-rate encoder

difference is about 1 dB for bitrates between 250 kbps and 1000 kbps. By further increasing the *prediction bitrate*, we see that the distortion difference between the multi stream and reference increases to 4 dB for the lowest output 250 kbps. Thus, the smallest distortion can be observed when using a *prediction bitrate* close to the average of the smallest and highest output bitrate, and we get a smaller penalty when the *prediction bitrate* is smaller than the output bitrate than vice versa.

Similar results can be observed when evaluating the *pedestrian* sequence, shown in Figure 9. Lower *prediction bitrates* incur less distortion difference than higher *prediction bitrates* compared to the *target bitrate*. The distortion difference is further reduced by choosing a bitrate closer to the average of the extremes.

We have shown that choosing the correct *prediction bitrate* when doing multi-rate encoding has a profound effect on the quality of the output videos. Although CPU time was also

Figure 7. Quality difference for the "worst-case" scenario in Figure 3(b) of 0.99 dB PSNR of 250 kbps

(a) Reference encoder

(b) Multi-rate encoder

Figure 8. Rate-distortion curve for CIF test sequence foreman with different prediction bitrate (in kbps)

(a) prediction bitrate: 250

(b) prediction bitrate: 450

(c) prediction bitrate: 750

(d) prediction bitrate: 1000
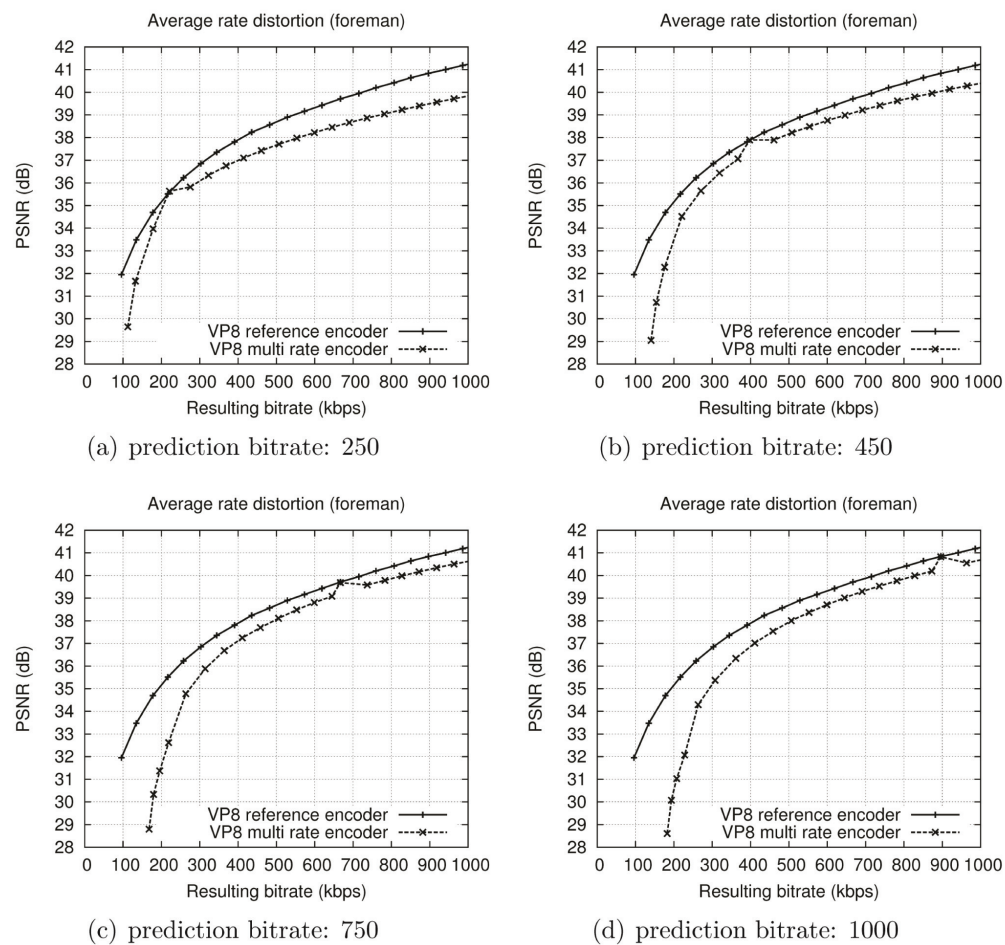
affected as shown in Figure 3(a), the difference was much less considerable. Because of the distortion, having a too wide range of *target bitrates* when doing multi-rate encoding is discouraged (see for example Figure 9(d)), but for quality ranges typically used in segmented streaming as shown in our test sequences, the results prove that multi-rate encoding is useful.

Quality Impact for Different Content Types

The previous sections have shown that our multi-output encoding approach is usable for both low- and high-resolution videos. However, encoding performance and picture quality also depend on the content type (Ni et al., 2011). In this respect, video content is classified as being high or low in both the spatial and temporal complexity (ITU-T, 1999) measured by spatial and temporal information metrics, respectively. Thus, there might be differences between videos with different amount of motion and detail as well as the resolution. To further evaluate how our multi-rate approach performs, we have performed experiments with a number of standard CIF and HD test sequences (XIPH.org, 2012) from different classifications.

Figure 9. Rate-distortion curve for HD test sequence pedestrian with different prediction bitrate (in kbps)

(a) prediction bitrate: 1500

(b) prediction bitrate: 2000

(c) prediction bitrate: 2500

(d) prediction bitrate: 3000
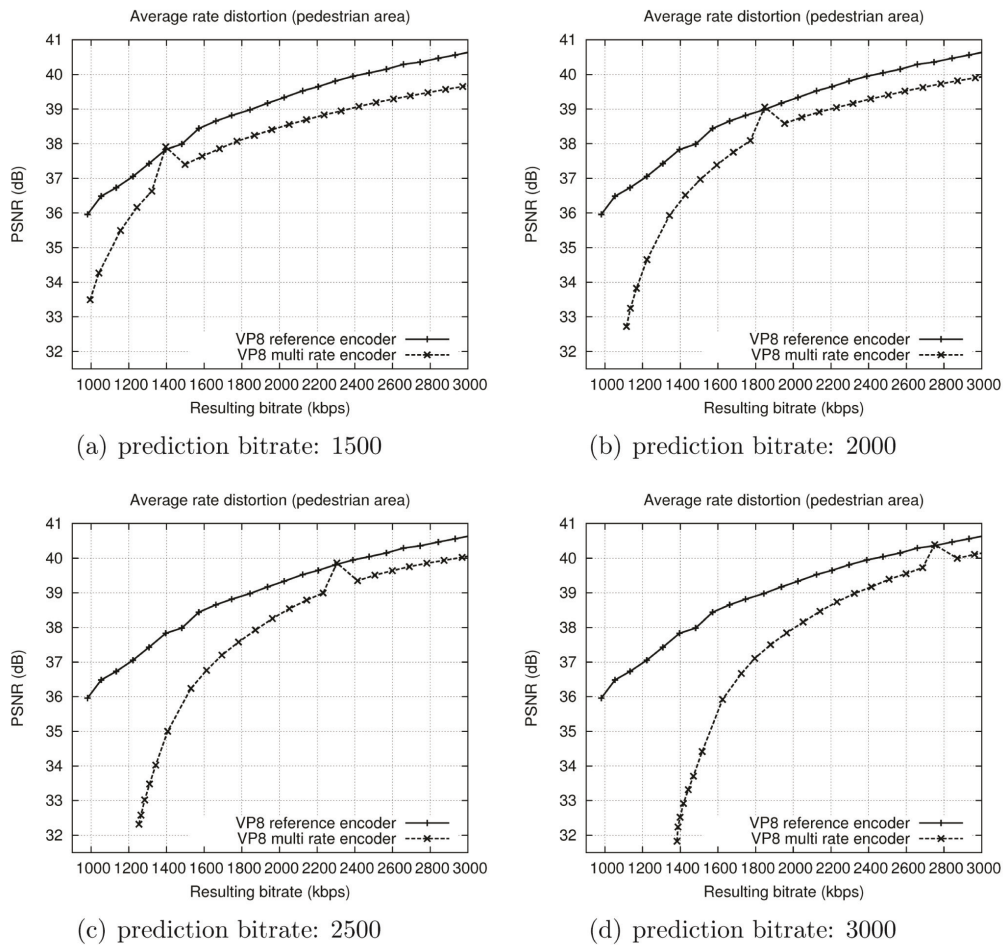
CIF Resolution

As we found in quality assessment section, the *foreman* sequence exhibited a PSNR quality loss compared to the reference encoder of approximately 1 dB in the range 250 - 900 Kbps. However, this difference was not visible in the example frames shown in Figure 7. Low rates gave a high loss if the prediction rate was high, but gave acceptable results for a large range of bitrates. In this section, we will investigate if this also holds for other CIF resolution videos. We used 16 standard test sequences (Table 1)

using prediction rates of 100, 250, 500, 750 and 1000 Kbps. Each of these clips has different amount of detail and motion. Since we reuse motion vectors, we also include motion metrics using the MPEG 7 standard (Jeannin & Divakaran, 2001) and ITU-T-Rec-P.910 (1999) in order to see if this influences the results. We have also done tests on CPU time when encoding, and in Figure 10 we can see the speedup of our multi-rate encoder varying somewhat depending on content. However, the gain for all evaluated videos are significant compared to the reference encoder.

Table 1. CIF test sequences and their temporal complexity (higher number means more motion)

Test Sequence	MPEG7	ITU P.910	Test Sequence	MPEG7	ITU P.910
bridge-close	0.01	7.20	bridge-far	0.48	7.04
coastguard	1.57	34.91	container	0.15	9.70
flower	1.43	39.19	foreman	7.90	36.25
hall	0.51	9.19	highway	1.51	50.79
mobile	0.37	33.02	mother-daughter	0.69	9.54
news	0.63	30.32	paris	0.17	15.89
silent	1.55	12.67	stefan	4.21	42.16
tempete	1.24	21.39	waterfall	0.00	8.24

Figure 11 shows the results using 500 Kbps as prediction rate as a representative example, i.e., the tests in section indicates that a prediction rate somewhere in the middle of the target range is better. The average PSNR vary as expected between different videos, hence, the different y-axis in the plots, but we can observe that for most videos the average loss of PSNR compared to the reference encoder is less than 1 dB in the range of 250 - 900 Kbps output videos. There are exceptions, including *flower*, *mobile*, *mother-daughter*, *coastguard,* and *stefan* sequences that have higher losses compared to their references of up to 2 dB in the range. In most of these situations, there is no visible difference as also described in quality assessment section, but for some frames at the very lower end of the range, some very small visible effects can be seen. On the other hand, we can also see sequences with very little quality loss (in PSNR) such as the *paris*, *bridge-close*, *highway* and *silent* videos.

HD Resolution

For the HD resolution, we found in quality assessment section that the *pedestrian* and *blue sky* test sequences exhibited a PSNR loss of about 1.0 to 1.5 dB compared to the reference encoder. As for the CIF sequence, this difference in quality was not visible in the example frame shown in Figure 6. To see if the results are similar with other contents, we have used five HD standard test sequences available in 1080p resolution.

The results for the HD experiments are shown in Figure 12. We show only the 1500 Kbps prediction rate results as this is the best prediction rate for HD resolution according to Figure 9, but the experiments give similar results using the other rates. As in the previous section, the average PSNR is varying between the different videos as expected. However, the general trend in all the five sequences is the same: The loss of PSNR compared to the

Figure 10. Speedup at CIF resolution with Multi-Rate encoder compared to reference encoder
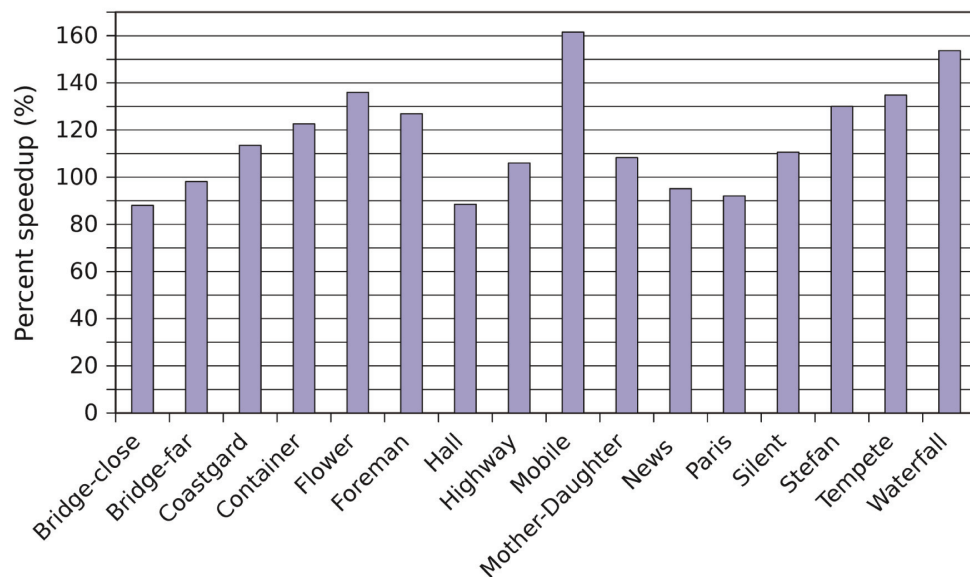
reference encoder is less than 1 dB for bitrates in the range of 1500 - 3000 Kbps. For bitrates less that 1500 Kbps, the loss of PSNR drops off faster, i.e., for all resolutions, the quality drops more and faster towards the lower end of the target interval. The worst observed difference in PSNR is slightly over 2 dB at 1000 Kbps for the *pedestrian* and *rush hour* sequence. The sequences showing the best results are the *sunflower* and *tractor* videos with worst-case PSNR less than 2 dB. As above, these differences in video quality are hardly visible to the user. We have also evaluated the CPU used for the HD sequences as can be seen in Figure 13. The results are similar to the performance of the lower resolution videos with speedup varying somewhat depending on content. Again, the gain is for all evaluated videos significant compared to the reference encoder.

DISCUSSION AND OPEN ISSUES

To prove our idea, we have implemented a prototype which reuses the most expensive operations based on a performance profile of the encoding pipeline. In particular, our multi-rate encoder reuses the analysis part consisting of macroblock mode decision and intra/inter prediction. The experimental results indicate that we can encode the different videos at the same rates with approximately the same qualities compared to the VP8 reference encoder, while reducing the encoding time significantly. However, our prototype is a small proof-of-concept implementation, and there are numerous open issues.

Though rarely visible, our results do show that there is a small quality degradation using our multi-rate encoder compared to the reference encoder. However, it is likely that the degradation can be reduced by further refining encoding parameters such as motion vectors to better suit the *target bitrate* from the *prediction bitrate*. One open issue is therefore to look into solutions for improving the quality for the other bitrates, aside from correctly choosing the *prediction bitrate*. By virtue of our method of reusing analysis computations directly, the quality will suffer when the *target bitrate* is not equal to the *prediction bitrate*. One plausible explanation of the quality degradation could be that videos

Figure 11. Average quality distortion for different videos using 500 kbps prediction bitrate note: different y-axis cutoff, scale retained

(a) Bridge-close   (b) Bridge-far   (c) Coastguard   (d) Container

(e) Flower   (f) Foreman   (g) Hall   (h) Highway

(i) Mobile   (j) Mother-Daughter   (k) News   (l) Paris

(m) Silent   (n) Stefan   (o) Tempete   (p) Waterfall
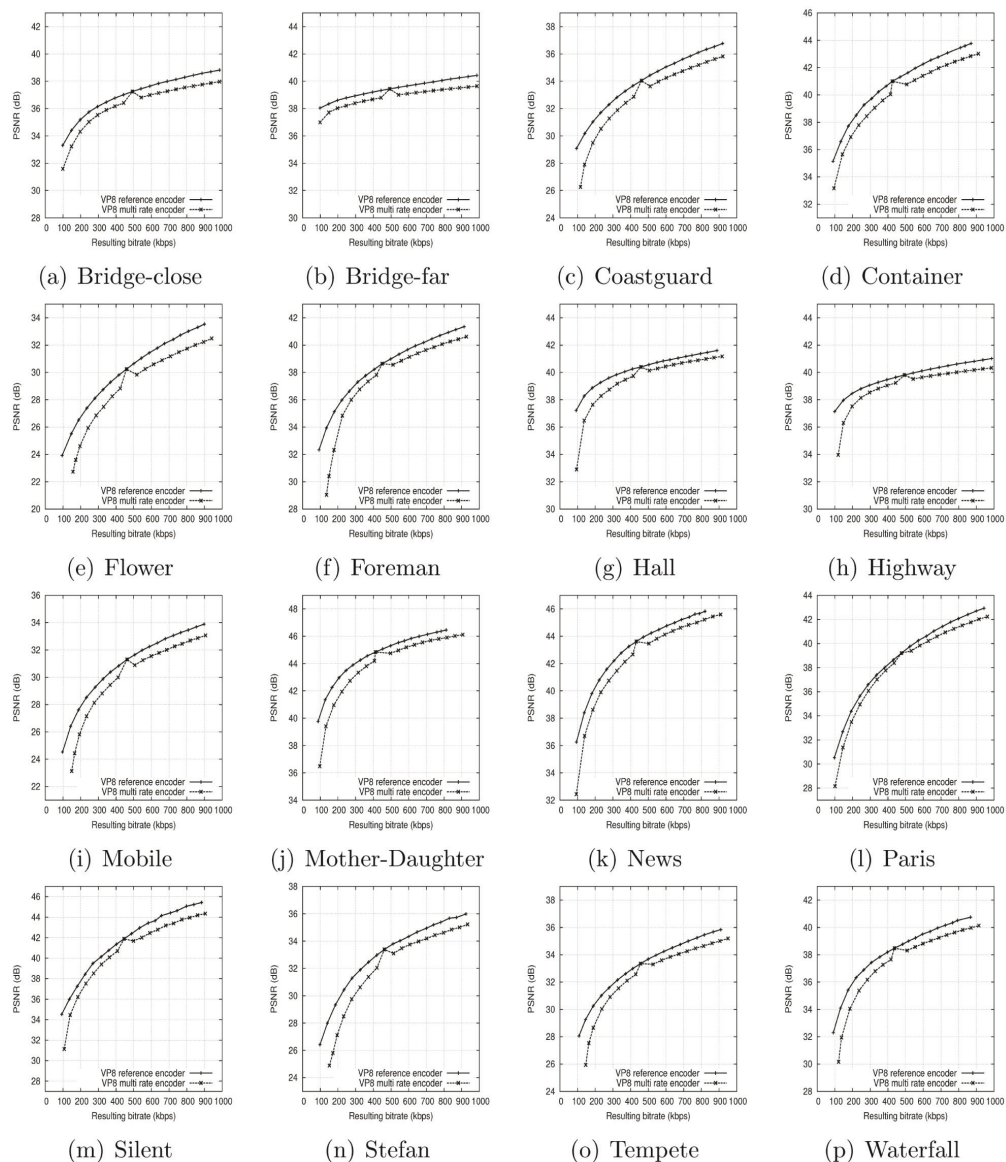
with high levels of motion produce many motion vectors, and that the motion vectors then chosen in the analysis stage are not meant to be used at another bitrate. To investigate this, we used two objective metrics for finding the temporal complexity of the test sequences (Table 1) and compared these metrics with the observed quality degradation. However, we cannot explain the quality degradation based on video motion alone. For example, both the *flower* and *highway* sequences feature camera panning and have similar motion complexity according to Table 1, but they result in very different quality degradations. Similarly, *stefan*

Figure 12. Average quality distortion for different 1080p videos using 1500 kbps prediction bitrate

(a) Pedestrian area

(b) Rush hour

(c) Station2

(d) Sunflower

(e) Tractor
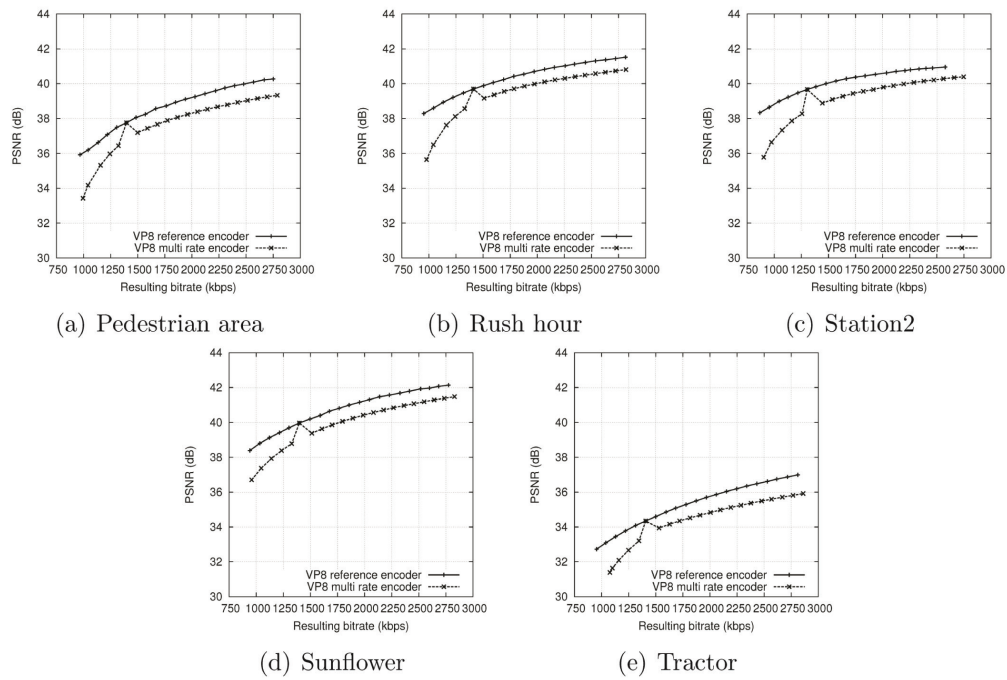
Figure 13. Speedup at HD resolution with Multi-Rate encoder compared to reference encoder
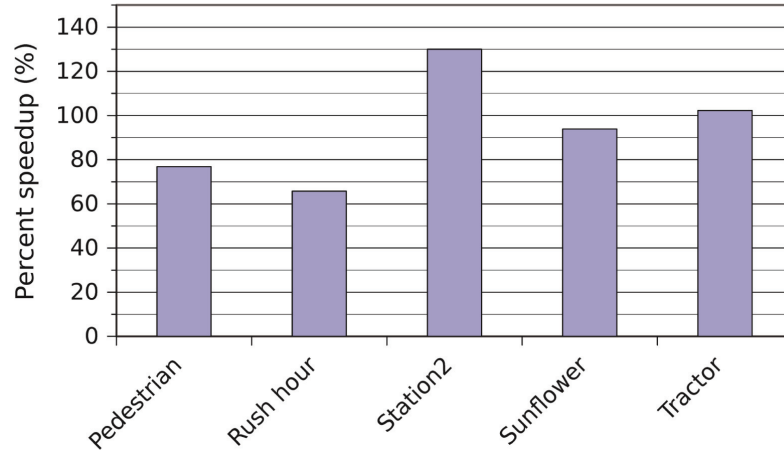
and *foreman* have high levels of motion, but only the former result in a quality loss of more than 1 dB. As such, we can rule out that this is caused by high levels of temporal complexity in the content alone.

Another contributor to the reduced quality compared to the regular encoder is that the input to the analysis step and the reconstructed frame are not identical (they differ in terms of quantization level). This matters since the prediction in the encoder (which uses the reconstructed frame as usual) does not produce identical pixels as the analysis step did in its prediction. The result being that motion vectors and other prediction modes are not optimally selected for the *target bitrate* since they were chosen using slightly different data. We believe this also explains the *spike* seen when the prediction rate matches the target rate, and the optimal prediction parameters are chosen; when the reconstructed frame and the frame used for prediction differ even slightly, we expect the quality to drop (since non-optimal modes can be chosen). We note that using the original frame instead of the reconstructed frame as original to the analysis stage (open loop) is a well-known technique to remove dependencies in video encoders (Fonseca et al., 2007) which reduces the prediction quality and hence the video quality. Our technique differs in that instead of using the original frame in analysis, we use the reconstructed frame for a near bitrate which is much more similar to the *target bitrate*'s reconstructed frame than the undistorted original frame.

Currently, prediction modes found in the analysis stage is used as-is without adaptations, even though it was found using another input frame (reconstructed with another quantization level). One potential quality improvement could be to do *predictor refinement*, inspired by the approach taken by Zhou et al. (2008). Since the prediction modes are found using a very similar image, we expect the motion vectors and intra-frame predictors to be close to the optimal result. Using refinements, we can improve these be doing a local search in the area near the shared prediction. This would,

however, lead to increased complexity in the encoder. Moreover, the quality assessment section demonstrates how reuse of the analysis computations impacts the quality/complexity tradeoff for encoding the same input at different rates. A limitation with our multi-rate encoder is that all the bit streams encoded must use the same number of reference frames, or in the case of VP8, the same golden frames for the method to be viable. It may be possible to scale the spatial resolution in the different outputs by also scaling prediction parameters as done by some transcoding approaches, but quality impact of this is left as further work. Another potential for further work is to investigate if there are other parts of the VP8 encoder where the processing can be fanned out like in the analysis step. Also, a systematic review of all the encoder modes and decisions to pinpoint if some parameters are causing more impact than others in our prototype encoder is left as further work.

Another important point is the generality of the presented idea. In the prototype, we used VP8 as a case study since it is an emerging open-source codec, and the source code is much smaller than H.264. However, VP8 is very similar to the baseline profile in H.264, and in general, most video codecs use similar ideas for compression. Thus, our ideas are not implementation specific to VP8, but also applicable for other codecs like MPEG-1/2/4, H.263/4, VC-1, Theora, etc., which compress the video data in a similar way.

CONCLUSION

A large fraction of the Internet video services use an adaptive HTTP streaming technology. By encoding the video in multiple bitrates matching different devices and different network conditions, the bitrate (and thus video quality) can be changed dynamically to match the varying bandwidth, giving a large advantage over non-adaptive systems that are frequently interrupted due to buffer underruns or data loss. However, encoding video into multiple bitrates

is a resource expensive task. We have therefore investigated the effect of running multiple encoding instances in parallel, where the different instances reuse intermediate results. This way, several encoding steps are avoided for the sub-sequent encoding operations. In particular, we have analyzed and performed experiments with Google's VP8 encoder, encoding different types of video to multiple rates for various scenarios. Our main contribution is that we propose a way of reusing decisions from intra and inter prediction in the video encoder to avoid computational expensive steps that are redundant when encoding for multiple *target bitrates* of the same video object. The method can be used in any video codec comprising an analysis and encoding step with similar structure as H.264 and VP8. Furthermore, the method has been implemented in the VP8 reference encoder as a case study, and the experimental results show that the computational demands are significantly reduced at the same rates and approximately the same qualities compared to the VP8 reference implementation, i.e., for a negligible quality loss in terms of PSNR, the processing costs can be greatly reduced. However, the quality loss is dependent on the distance from the initial bitrate, i.e., if the gap between the output bitrates is too large, the quality loss becomes larger. In such scenarios, we still need multiple instances of the whole operation.

The VP8 codec is similar H.264 and several others, and our approach should be suitable for comparable encoding pipeline as well. However, implementing a prototype and showing the same experimental results are tasks left as further work. Additionally, our aim has been to point at an operation that potentially can be optimized, and we suggested one possible solution, i.e., reusing the macro-block mode decision, intra prediction and inter prediction between the parallel encoding instances. However, as indicated in the discussion, there are several open issues where potentially other steps that can be improved with respect to the resource consumption or the resulting video quality – all promising research topics to investigate.

ACKNOWLEDGMENT

REFERENCES

Adobe Inc. (2010). *HTTP dynamic streaming on the Adobe Flash platform*. Retrieved from http://www.adobe.com/content/dam/Adobe/en/products/hds-dynamic-streaming/pdfs/hds_datasheet.pdf

Akamai Inc. (2010). *Akamai HD for iPhone encoding best practices*. Retrieved from http://www.akamai.com/dl/whitepapers/Akamai_HDNetwork_Encoding_BP_iPhone_iPad.pdf

Bankoski, J., Wilkins, P., & Xu, Y. (2011). *VP8 data format and decoding guide*. Retrieved from IETF website: http://tools.ietf.org/html/draft-bankoski-vp8-bitstream-01

Cisco Systems, Inc. (2010). *Visual networking index*. Retrieved from http://www.cisco.com/en/US/netsol/ns827/networking_solutions_sub_solution.html

Espeland, H. (2008). *Investigation of parallel programming on heterogeneous multiprocessors* (Master's thesis, University of Oslo). Retrieved from http://www.duo.uio.no/sok/work.html?WORKID=82232

Finstad, D. H., Stensland, H. K., Espeland, H., & Halvorsen, P. (2011). Improved multi-rate video encoding. In *Proceedings of the IEEE International Symposium on Multimedia* (pp. 293-300).

Flosi, S. L. (2010). *comScore releases April 2010 U.S. online video rankings* (Press Release). Reston, VA: comScore, Inc.

Fonseca, T. A. da, Liu, Y., & de Queiroz, R. L. (2007). Open-loop prediction in h.264/avc for high definition sequences. In *Proceedings of the Sociedade Brasileira de Telecomunicações*.

Huang, Y.-W., Hsieh, B.-Y., Chien, S.-Y., Ma, S.-Y., & Chen, L.-G. (2006). Analysis and complexity reduction of multiple reference frames motion estimation in h.264/avc. *IEEE Transactions on Circuits and Systems for Video Technology*, *16*(4), 507–522. doi:10.1109/TCSVT.2006.872783

ITU-T. (1999). *Subjective video quality assessment methods for multimedia applications* (p. 910). Geneva, Switzerland: ITU-T.

Jeannin, S., & Divakaran, A. (2001). MPEG-7 visual motion descriptors. *IEEE Transactions on Circuits and Systems for Video Technology*, *11*(6), 720–724. doi:10.1109/76.927428

Kuo, C.-C., & Jayant, N. (2003). An adaptive non-linear motion vector resampling algorithm for down-scaling video transcoding. In *Proceedings of the International Conference on Multimedia and Expo* (pp. 229-232).

Move Networks. (2008). *Internet television: Challenges and opportunities (Tech. Rep.)*. American Fork, UT: Author.

Ni, P., Eg, R., Eichhorn, A., Griwodz, C., & Halvorsen, P. (2011). Flicker effects in adaptive video streaming to handheld devices. In *Proceedings of the 19th ACM International Conference on Multimedia* (pp. 463-472).

Ozer, J. (2010). *First look: H.264 and VP8 compared.* Retrieved from http://www.streamingmedia.com/articles/editorial/featured-articles/first-look-h.264-and-vp8-compared-67266.aspx

Pantos, R., Batson, J., Biderman, D., May, B., & Tseng, A. (2010). *HTTP live streaming.* Retrieved from http://tools.ietf.org/html/draft-pantos-http-live-streaming-04

Schwarz, H., Marpe, D., & Wiegand, T. (2007). Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, *17*(9), 1103–1120. doi:10.1109/TCSVT.2007.905532

Seeling, P., Fitzek, F. H. P., Ertli, G., Pulipaka, A., & Reisslein, M. (2010). Video network traffic and quality comparison of VP8 and h.264 svc. In *Proceedings of the 3rd Workshop on Mobile Video Delivery* (pp. 33-38).

Senda, Y., & Harasaki, H. (1999). A realtime software mpeg transcoder using a novel motion vector reuse and a simd optimization techniques. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (pp. 2359-2362).

Stockhammer, T. (2011). Dynamic adaptive streaming over HTTP - standards and design principles. In *Proceedings of the 2nd ACM Conference on Multimedia Systems* (pp. 133-144).

Wien, M., Schwarz, H., & Oelbaum, T. (2007). Performance analysis of SVC. *IEEE Transactions on Circuits and Systems for Video Technology*, *17*(9), 1194–1203. doi:10.1109/TCSVT.2007.905530

Wu, N., Wen, M., Wu, W., Ren, J., Su, H., Xun, C., & Zhang, C. (2009). Streaming hd h.264 encoder on programmable processors. In *Proceedings of the 17th ACM International Conference on Multimedia* (pp. 371–380).

Xiph.org. (2012). *Test media.* Retrieved from http://media.xiph.org/video/derf/

Youn, J., Sun, M.-T., & Lin, C.-W. (1999). Motion vector refinement for high-performance transcoding. *IEEE Transactions on Multimedia*, *1*(1), 30–40. doi:10.1109/6046.748169

YouTube. (2012). *Holy nyans! 60 hours per minute and 4 billion views a day on youtube.* Retrieved from http://youtube-global.blogspot.com/2012/01/holy-nyans-60-hours-per-minute-and-4.html

Zambelli, A. (2009). *Smooth streaming technical overview.* Retrieved from http://learn.iis.net/page.aspx/626/smooth-streaming-technical-overview/

Zhou, H., Zhou, J., & Xia, X. (2008). The motion vector reuse algorithm to improve dual-stream video encoder. In *Proceedings of the 9th International Conference on Signal Processing* (pp. 2359-2362).

Zhou, Z., & Sun, M.-T. (2004). Fast macroblock inter mode decision and motion estimation for h.264/mpeg-4 avc. In *Proceedings of the International Conference on Image Processing* (pp. 789-792).

Håvard Espeland is currently a research fellow at Simula Research Laboratory and the University of Oslo at the Media Performance Group. He finished his MSc in 2008 with a thesis on parallel programming on heterogenous architectures. He stayed at UiO and started his PhD work focusing on systems support for multimedia processing. In this field he has worked on several aspects, including P2G, a framework and language for realtime multimedia processing, I/O scheduling optimizations and various architecture-level experiments and optimizations. At the University, Håvard also gives lectures on programming heterogeneous architectures. Håvard expects to submit his PhD thesis on multimedia processing by 2012.

Håkon Kvale Stensland is a research fellow at the Media Performance Group at Simula Research Laboratory and the University of Oslo, Norway. He finished his master degree (MSc) on fault tolerant routing in SCI networks in 2006. Håkon stayed at the University of Oslo and is now the lab manager at the iAD Center for research based innovation at the Department of Informatics. His research interests are within the field of distributed processing, and heterogeneous processing architectures. He is also interested in the state of the art in asymmetric multi-core processors like the nVIDIA Kepler-architecture. Håkon expects to submit his PhD thesis on the programing of multimedia workloads in heterogeneous architectures by 2012.

Dag Haavi Finstad was a master student in the Media Performance Group at Simula Research Laboratory and the University of Oslo, Norway. He received his master degree (MSc) in 2011 after implementing a prototype of the multi-rate encoder concept presented in this paper. Dag is now working as a developer at Varnish Software AS.

Pål Halvorsen is a researcher at the Media Performance Group at Simula Research Laboratory and a tenured professor in the Network and Distributed systems group at the Department of Informatics, University of Oslo, Norway. His research activities focus mostly on support for distributed multimedia systems (ranging from media on-demand systems to massive multiplayer games), including operating systems, storage and retrieval, communication and distribution. Pål received his master degree (Cand.Scient.) in 1997 and his doctoral degree (Dr.Scient.) in 2001, both from the Department of Informatics, University of Oslo, Norway.

Paper VI: LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition

Title: LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition [102].

Authors: K. Raaen, H. Espeland, H. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz.

Published: Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) - The 2012 International Conference on Parallel Processing Workshops, IEEE, 2012.

LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition

Kjetil Raaen, Håvard Espeland, Håkon K. Stensland, Andreas Petlund, Pål Halvorsen, Carsten Griwodz

NITH, Norway   Simula Research Laboratory, Norway   IFI, University of Oslo, Norway

Email: raakje@nith.no, {haavares, haakonks, apetlund, paalh, griff}@ifi.uio.no

Abstract—**Supporting thousands of interacting players in a virtual world poses huge challenges with respect to processing. Existing work that addresses the challenge utilizes a variety of spatial partitioning algorithms to distribute the load. If, however, a large number of players needs to interact tightly across an area of the game world, spatial partitioning cannot subdivide this area without incurring massive communication costs, latency or inconsistency. It is a major challenge of game engines to scale such areas to the largest number of players possible; in a deviation from earlier thinking, parallelism on multi-core architectures is applied to increase scalability. In this paper, we evaluate the design and implementation of our game server architecture, called LEARS, which allows for lock-free parallel processing of a single spatial partition by considering every game cycle an atomic tick. Our prototype is evaluated using traces from live game sessions where we measure the server response time for all objects that need timely updates. We also measure how the response time for the multi-threaded implementation varies with the number of threads used. Our results show that the challenge of scaling up a game-server can be an embarrassingly parallel problem.**

I. Introduction

Over the last decade, online multi-player gaming has experienced an amazing growth. Providers of the popular online games must deliver a reliable service to thousands of concurrent players meeting strict processing deadlines in order for the players to have an acceptable quality of experience (QoE).

One major goal for large game providers is to support as many concurrent players in a game-world as possible while preserving the strict latency requirements in order for the players to have an acceptable quality of experience (QoE). Load distribution in these systems is typically achieved by partitioning game-worlds into areas-of-interest to minimize message passing between players and to allow the game-world to be divided between servers. Load balancing is usually completely static, where each area has dedicated hardware. This approach is, however, limited by the distribution of players in the game-world, and the problem is that the distribution of players is heavy-tailed with about 30% of players in 1% of the game area [5]. To handle the most popular areas of the game world without reducing the maximum interaction distance for players, individual spatial partitions can not be serial. An MMO-server will experience the most CPU load while the players experience the most "action". Hence, the worst case scenario for the server is when a large proportion of the players gather in a small area for high intensity gameplay.

In such scenarios, the important metric for online multi-player games is latency. Claypool et. al. [7] classify different types of games and conclude that for first person shooter (FPS) and racing games, the threshold for an acceptable latency is 100ms. For other classes of networked games, like real-time strategy (RTS) and massively multi-player online games (MMOGs) players tolerate somewhat higher delays, but there are still strict latency requirements in order to provide a good QoE. The accumulated latency of network transmission, server processing and client processing adds up to the latencies that the user is experiencing, and reducing any of these latencies improves the users' experience.

The traditional design of massively multi-player game servers rely on *sharding* for further load distribution when too many players visit the same place simultaneously. Sharding involves making a new copy of an area of a game, where players in different copies are unable to interact. This approach eliminates most requirements for communication between the processes running individual shards. An example of such a design can be found in [6].

The industry is now experimenting with implementations that allow for a greater level of parallelization. One known example is Eve Online [8] where they avoid *sharding* and allow all players to potentially interact. Large-scale interactions in Eve Online are handled through an optimized database. On the local scale, however, the server is not parallel, and performance is extremely limited when too many players congregate in one area. With LEARS, we take this approach even further and focus on how many players that can be handled in a single segment of the game world. We present a model that allows for better resource utilization of multi-processor, game server systems which should not replace spatial partitioning techniques for work distribution, but rather complement them to improve on their limitations. Furthermore, a real prototype game is used for evaluation where captured traces are used to generate server load. We compare multi-threaded and single-threaded implementations in order to measure the overhead of parallelizing the implementation and showing the experienced benefits of parallelization. The change in responsiveness of different implementations with increased load on the server is

studied, and we discuss how generic elements of this game design impact the performance on our chosen platform of implementation.

Our results indicate that it is possible to design an "embarrassingly parallel" game server. We also observe that the implementation is able to handle a quadratic increase of in-server communication when many players interact in a game-world hotspot.

The rest of the paper is organized as follows: In section II, we describe the basic idea of LEARS, before we present the design and implementation of the prototype in section III. We evaluate our prototype in section IV and discuss our idea in section V. In section VI, we put our idea in the context of other existing work. Finally, we summarize and conclude the paper in section VII and give directions for further work in section VIII.

II. LEARS: THE BASIC IDEA

Traditionally, game servers have been implemented much like game clients. They are based around a main loop, which updates every active element in the game. These elements include for example player characters, non-player characters and projectiles. The simulated world has a list of all the active elements in the game and typically calls an "update" method on each element. The simulated time is kept constant throughout each iteration of the loop, so that all elements get updates at the same points in simulated time. This point in time is referred to as a *tick*. Using this method, the active element performs all its actions for the tick. Since only one element updates at a time, all actions can be performed directly. The character reads input from the network, performs updates on itself according to the input, and updates other elements with the results of its actions.

LEARS is a game server model with support for lockless, relaxed-atomicity state-parallel execution. The main concept is to split the game server executable into lightweight threads at the finest possible granularity. Each update of every player character, AI opponent and projectile runs as an independent work unit.

White et al. [15] describe a model they call a *state-effect pattern*. Based on the observation that changes in a large, actor-based simulation are happening *simultaneously*, they separate read and write operations. Read operations work on a consistent previous state, and all write operations are batched and executed to produce the state for the next tick. This means that the ordering of events scheduled to execute at a tick does not need to be considered or enforced. For the design in this paper, we additionally remove the requirement for batching of write operations, allowing these to happen anytime during the tick. The rationale for this relaxation is found in the way traditional game servers work. In the traditional single-threaded main-loop approach, every update is allowed to change any part of the simulation state at any time. In such a scenario the state at a given time is a combination of values from two different points in time, current and previous, exactly the same situation that occurs in the design presented here.

The second relaxation relates to the atomicity of game state updates. The fine granularity creates a need for significant communication between threads to avoid problematic lock contentions. Systems where elements can only update their own state and read any state without locking [1] do obviously not work in all cases. However, game servers are not accurate simulators, and again, depending on the game design, some (internal) errors are acceptable without violating game state consistency. Consider the following example: Character A moves while character B attacks. If only the X coordinate of character A is updated at the point in time when the attack is executed, the attack sees character A at a position with the new X coordinate and the old Y coordinate. This position is within the accuracy of the simulation which in any case is no better than the distance an object can move within one tick.

On the other hand, for actions where a margin of error is not acceptable, transactions can be used keeping the object's state internally consistent. However, locking the state is expensive. Fortunately, most common game actions do not require transactions, an observation that we take advantage of in LEARS.

These two relaxations allow actions to be performed on game objects in any order without global locking. It can be implemented using message passing between threads and retains consistency for most game actions. This includes actions such as moving, shooting, spells and so forth. Consider player A shooting at player B: A subtracts her ammunition state, and send bullets in B's general direction by spawning bullet objects. The bullet objects runs as independent work units, and if one of them hits player B, it sends a message to player B. When reading this message, player B subtracts his health and sends a message to player A if it reaches zero. Player A then updates her statistics when she receives player B's message. This series of events can be time critical at certain points. The most important point is where the decision is made if the bullet hits player B. If player B is moving, the order of updates can be critical in deciding if the bullet hits or misses. In the case where the bullet moves first, the player does not get a chance to move out of the way. This inconsistency is however not a product of the LEARS approach. Game servers in general insert active items into their loops in an arbitrary fashion, and there is no rule to state which order is "correct".

The end result of our proposed design philosophy is that there is no synchronization in the server under normal running conditions. Since there are cases where transactions are required, they can be implemented outside the LEARS event handler running as transactions requiring locking. In the rest of the paper, we consider a practical implementation of LEARS, and evaluate its performance and scalability.

III. DESIGN AND IMPLEMENTATION

In our experimental prototype implementation of the LEARS concept, the parallel approach is realized using thread pools and blocking queues.
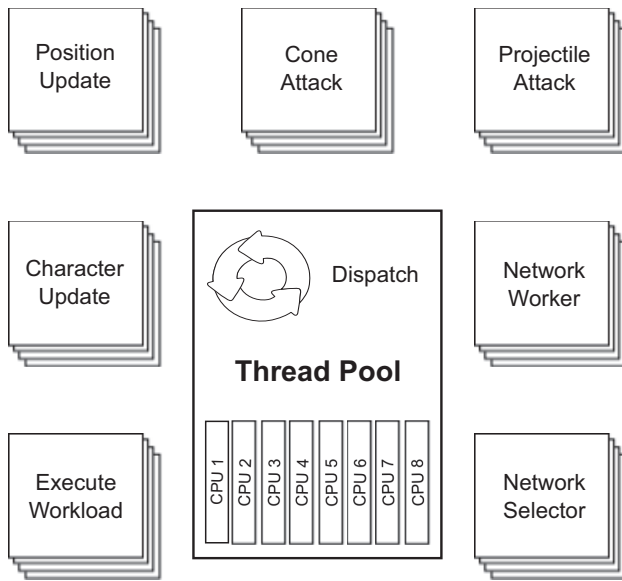
Figure 1. Design of the Game Server

A. Thread pool

Creation and deletion of threads incur large overheads, and context switching is an expensive operation. These overheads constrain how a system can be designed, i.e., threads should be kept as long as possible, and the number of threads should not grow unbounded. We use a *thread pool* pattern to work around these constraints, and a thread pool executor (the Java `ThreadPoolExecutor` class) to maintain the pool of threads and a queue of tasks. When a thread is available, the executor picks a task from the queue and executes it. The thread pool system itself is not preemptive, so the thread runs each task until it is done. This means that in contrast to normal threading, each task should be as small as possible, i.e., larger units of work should be split up into several sub-tasks.

The thread pool is a good way to balance the number of threads when the work is split into extremely small units. When an active element is created in the virtual world, it is scheduled for execution by the thread pool executor, and the active element updates its state exactly as in the single threaded case. Furthermore, our thread pool supports the concept of delayed execution. This means that tasks can be put into the work queue for execution at a time specified in the future. When the task is finished for one time slot, it can reschedule itself for the next slot, delayed by a specified time. This allows active elements to have any lifetime from one-shot executions to the duration of the program. It also allows different elements to be updated at different rates depending on the requirements of the game developer.

All work is executed by the same thread pool, including the slower I/O operations. This is a consistent and clear approach, but it does mean that game updates could be stuck waiting for I/O if there are not enough threads available.

B. Blocking queues

The thread pool executor used as described above does not constrain which tasks are executed in parallel. All systems elements must therefore allow any of the other elements to execute concurrently.

To enable a fast communication between threads with shared memory (and caches), we use *blocking queues*, using the Java `BlockingQueue` class, which implements queues that are synchronized separately at each end. This means that elements can be removed from and added to the queue simultaneously, and since each of these operations are extremely fast, the probability of blocking is low. In the scenario analysed here, all active elements can potentially communicate with all others. Thus, these queues allow information to be passed between active objects. Each active object that can be influenced by others has a blocking queue of messages. During its update, it reads and processes the pending messages from its queue. Messages are processed in the order they were put in the queue. Other active elements put messages in the queue to be processed when they need to change the state of other elements in the game.

Messages in the queues can only contain relative information, and not absolute values. This restriction ensures that the change is always based on updated data. For example, if a projectile needs to tell a player character that it took damage, it should only inform the player character about the amount of damage, not the new health total. Since all changes are put in the queue, and the entire queue is processed by the same work unit, all updates are based on up-to-date data.

C. Our implementation

To demonstrate LEARS, we have implemented a prototype game containing all the basic elements of a full MMOG with the exception of persistent state. The basic architecture of the game server is described in figure 1. The thread pool size can be configured, and will execute the different workloads on the CPU cores. The workloads include processing of network messages, moving computer controlled elements (in this prototype only projectiles) checking for collisions and hits and sending outgoing network messages.

Persistent state do introduce some complications, but as database transactions are often not time critical and can usually be scheduled outside peak load situations, we leave this to future work.

In the game, each player controls a small circle ("the character") with an indicator for which direction they are heading (see figure 2). The characters are moved around by pressing keyboard buttons. They also have two types of attack, i.e., one projectile and one instant area of effect attack. Both attacks are aimed straight ahead. If an attack hits another player character, the attacker gets a positive point, and the character that was hit gets a negative point. The game provides examples of all the elements of the design described above:

- The player character is a long lifetime active object. It processes messages from clients, updates states and potentially produces other active objects (attacks). In

384

203

Figure 2. Screen shot of a game with six players.

addition to position, which all objects have, the player also has information about how many times it has been hit and how many times it has hit others. The player character also has a message queue to receive messages from other active objects. At the end of its update, it enqueues itself for the next update unless the client it represents has disconnected.

- The frontal cone attack is a one shot task that finds player characters in its designated area and sends messages to those hit so they can update their counters, as well as back to the attacking player informing about how many were hit.
- The projectile is a short lifetime object that moves in the world, checks if it has hit anything and reschedules itself for another update, unless it has hit something or ran to the end of its range. The projectile can only hit one target.

To simulate an MMORPG workload that grow linearly with number of players, especially collision checks with the ground and other static objects, we have included a synthetic load which emulates collision detection with a high-resolution terrain mesh. The synthetic load ensures that the cache is regularly flushed to enhance the realism of our game server prototype compared to a large-scale game server.

The game used in these experiments is simple, but it contains examples of all elements typically available in the action based parts of a typical MMO-like game.

The system described in this paper is implemented in Java. This programming language has strong support for multi-threading and has well-tested implementations of all the required components. The absolute values resulting from these experiments depend strongly on the complexity of the game, as a more complex game would require more processing.

In addition, the absolute values depend on the runtime environment, especially the server hardware, and the choice of programming language also influence absolute results from the experiments. However, the focus of this paper is the relative results, as we are interested in comparing scalability of the multi-threaded solution with a single-threaded approach and whether the multi-threaded implementation can handle the quadratic increase in traffic as new players join.

IV. EVALUATION

To have a realistic behavior of the game clients, the game was run with 5 human players playing the game with a game update frequency of 10 Hz. The network input to the server from this session was recorded with a timestamp for each message. The recorded game interactions were then played back multiple times in parallel to simulate a large number of clients. To ensure that client performance is not a bottleneck, the simulated clients were distributed among multiple physical machines. Furthermore, as an average client generates 2.6 kbps network traffic, the 1 Gbps local network interface that was used for the experiments did not limit the performance. The game server was run on a server machine containing 4 Dual-Core AMD Opteron 8218 (2600 MHz) with 16 GB RAM. To ensure comparable numbers, the server was taken down between each test run.

A. Response latency

The most important performance metric for client-server games is response latency from the server. From a player perspective, latency is only visible when it exceeds a certain threshold. Individual peaks in response time are obvious to the players, and will have the most impact on the Quality of Experience, hence we focus on peak values as well as averages in the evaluation.

The experiments were run with client numbers ranging from 40 to 800 in increments of 40, where the goal is to keep the latencies close to the 100 ms QoE threshold for FPS games [7]. Figure 3 shows a box-plot of the response time statistics from these experiments. All experiments used a pool of 48 worker threads and distributed the network connections across 8 IP ports.

From these plots, we can see that the single-threaded implementation is struggling to support 280 players at an average latency close to 100 ms. The median response time is 299 ms, and it already has extreme values all the way to 860 ms, exceeding the threshold for a good QoE. The multi-threaded server, on the other hand, is handling the players well up to 640 players where we are getting samples above 1 second, and the median is at 149 ms.

These statistics are somewhat influenced by the fact that the number of samples is proportional to the update frequency. This means that long update cycles to a certain degree get artificially lower weight.

Figure 4 shows details of two interesting cases. In figure 4(a), the single-threaded server is missing all its deadlines with 400 concurrent players, while the multi-threaded version

385

204

(a) Single-threaded server   (b) Multi-threaded server
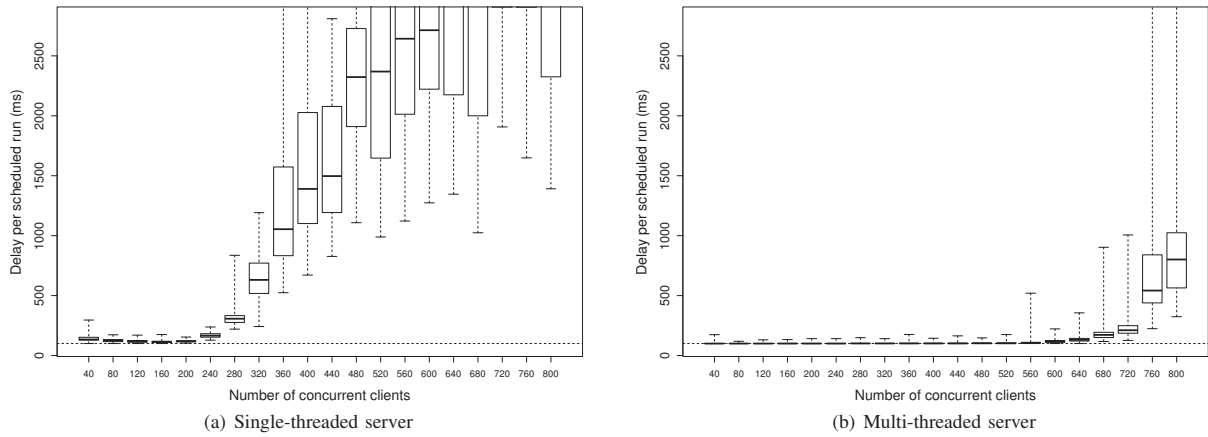
Figure 3. Response time for single- and multi-threaded servers (dotted line is the 100 ms threshold).

(a) 400 concurrent clients   (b) 800 concurrent clients
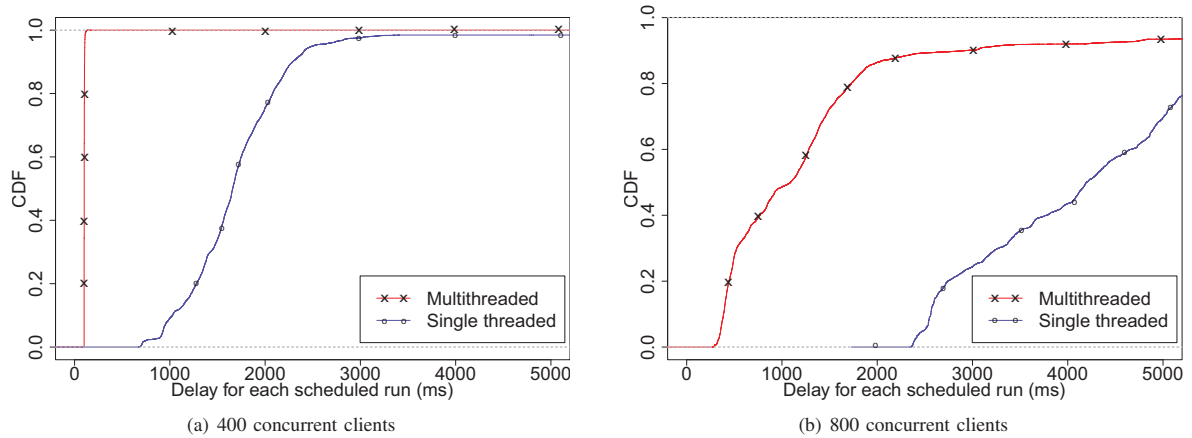
Figure 4. CDF of response time for single- and multi-threaded servers with 400 and 800 concurrent clients.
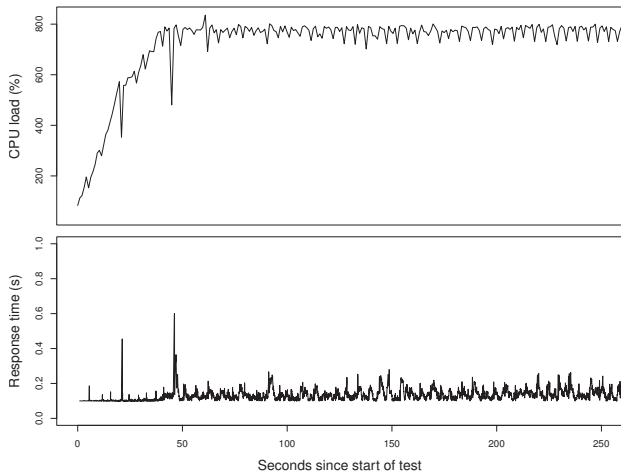
Figure 5. CPU load and response time for 620 concurrent clients on the multi-threaded server.

is processing almost everything on time. At 800 players (figure 4(b)), the outliers are going much further for both cases. Here, even the multi-threaded implementation is struggling to keep up, though it is still handling the load significantly better than the single-threaded version, which is generally completely unplayable.

B. Resource consumption

We have investigated the resource consumption when players connect to the multhreaded server as shown in figure 5. We present the results for 620 players, as this is the highest number of simultaneous players that server handles before significant degradation in performance, as shown in figure 3(b). The mean response time is 133 ms, above the ideal delay of 100 ms. Still, the server is able to keep the update rate smooth, without significant spikes. The CPU utilization grows while the clients are logging on, then stabilizes at an almost full CPU utilization for the rest of the run. The two spikes in response time happen while new players log in to the server at a very fast rate (30 clients pr. second). Receiving a new player requires a lock in

386
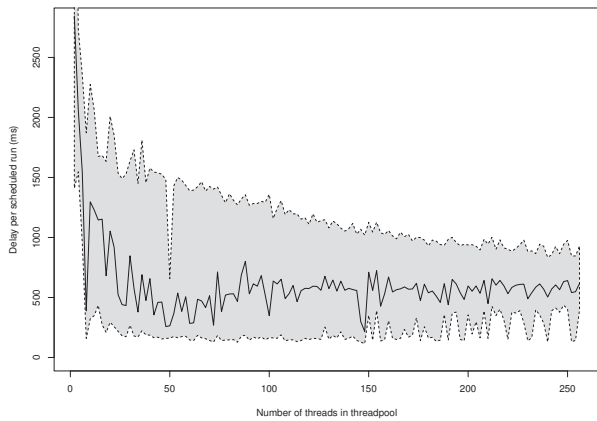
Figure 6. Response time for 700 concurrent clients on using varying number of threads. Shaded area from 5 to 95 percentiles.

the server, hence this operation is, to a certain degree, serial.

C. Effects of thread-pool size

To investigate the effects of the number of threads in the threadpool, we performed an experiment where we kept the number of clients constant while varying the number of threads in the pool. 700 clients were chosen, as this number slightly overloads the server. The number of threads in the pool was increased in increments of 2 from 2 to 256. In figure 6, we see clearly that the system utilizes more than 4 cores efficiently, as the 4 thread version shows significantly higher response times. At one thread per core or more, the numbers are relatively stable, with a tendency towards more consistent low response times with more available threads, to about 40 threads. This could mean that threads are occasionally waiting for I/O operations. Since thread pools are not pre-emptive, such situations would lead to one core going idle if there are no other available threads. Too many threads, on the other hand, could lead to excessive context switch overhead. The results show that the average is slowly increasing after about 50 threads, though the 95-percentile is still decreasing with increased number of threads, up to about 100. From then on the best case is worsening again most likely due to context switching overhead.

A game developer needs to consider this trade-off when tuning the parameters for a specific game.

V. Discussion

Most approaches to multi-threaded game server implementations in the literature (e.g., [1]) use some form of *spatial partitioning* to lock parts of the game world while allowing separate parts to run in parallel. Spatial partitioning is also used in other situations to limit workload. The number of players that game designers can allow in one area in a game server is limited by the worst-case scenario. The worst case scenario for a spatially partitioned game world is when everybody move

to the same point, where the spatial partitioning still ends up with everybody in the same partition regardless of granularity. This paper investigates an orthogonal and complementary approach which tries to increase the maximum number of users in the worst case scenario where all players can see each other at all times. Thus, spatial partitioning could be added to further scale the game server.

Experiments using multiple instances of a single-threaded server are not performed, as having clients distribueted acrosss multiple servers would mean partitioning the clients in areas where they can not interact, making numbers from such a scenario incomparable to the multithreaded solutions.

The LEARS approach does have *limitations* and is for example not suitable if the outcome of a message put restrictions on an object's state. This is mainly a game design issue, but situations such as trades can be accommodated by doing full transactions. The following example where two players trade illustrates the problem: Player A sends a message to player B where he proposes to buy her sword for X units. After this is sent, player C steals player A's money, and player A is unable to pay player B should the request go through. This is only a problem for trades *within* a single game tick where the result of a message to another object puts a constraint on the original sender, and can be solved by means such as putting the money in escrow until the trade has been resolved, or by doing a transaction outside of LEARS (such as in a database). Moreover, the design also adds some overhead in that the code is somewhat more complex, i.e., all communication between elements in the system needs to go through message queues. The same issue will also create some runtime overhead, but our results still demonstrate a significant benefit in terms of the supported number of clients.

Tasks in a thread pool can not be pre-empted, but the threads used for execution can. This distinction creates an interesting look into the performance trade-off of pre-emption. If the number of threads in the threadpool is equal to the number of CPU cores, we have a fully cooperative multitasking system. Increasing the number of threads allow for more pre-emption, but introduces context-switching overhead.

VI. Related Work

At Netgames 2011 [12], we presented a demo with a preliminary version of LEARS. Significant research has been done on how to optimize game server architectures for online games, both MMOGs and smaller-scale games. In this section, we summarize some of the most important findings from related research in this field. For example, "Red Dwarf", the community-based successor to "Project Darkstar" by Sun Microsystems [13], is a good example of a parallel approach to game server design. Here, response time is considered one of the most important metrics for game server performance, and suggests a parallel approach for scaling. The described system uses transactions for all updates to world state, including player position. This differs from LEARS, which investigates the case for common actions where atomicity of transactions is not necessary.

Work has also been done on scaling games by looking at the optimization as a data management problem. The authors in [14] have developed a highly expressive scripting language called SGL that provides game developers a data-driven AI scheme for non-player characters. By using query processing and indexing techniques, they can efficiently scale to a large number of non-player objects in games. This group also introduces the concept *state-effect pattern* in [15], which we extend in this paper. They test this and other parallel concepts using a simulated actor interaction model, in contrast to this paper which evaluates a running prototype of a working games under realistic conditions.

Moreover, Cai et al. [4] present a scalable architecture for supporting large-scale interactive Internet games. Their approach divides the game world into multiple partitions and assigns each partition to a server. The issues with this solution is that the architecture of the game server is still a limiting factor in worst case scenarios as only a limited number of players can interact in the same server partition at a given time. There have also been proposed several middleware systems for automatically distributing the game state among several participants. In [9], the authors present a middleware which allows game developers to create large, seamless virtual worlds and to migrate zones between servers. This approach does, however, not solve the challenge of many players that want to interact in a popular area. The research presented in [10] shows that proxy servers are needed to scale the number of players in the game, while the authors discuss the possibility of using grids as servers for MMOGs. Beskow et al. [3] have also been investigating partitioning and migration of game servers. Their approach uses core selection algorithms to locate the most optimal server. We have worked on how to reduce latency by modifying the TCP protocol to better support time-dependent applications [11]. However, the latency is not only determined by the network, but also the response time for the game servers. If the servers have a too large workload, the latency will suffer.

In [2], the authors are discussing the behavior and performance of multi-player game servers. They find that in the terms of benchmarking methodology, game servers are very different from other scientific workloads. Most of the sequentially implemented game servers can only support a limited numbers of players, and the bottlenecks in the servers are both game-related and network-related. The authors in [1] extend their work and use the computer game Quake to study the behavior of the game. When running on a server with up to eight processing cores the game suffers because of lock synchronization during request processing. High wait times due to workload imbalances at global synchronization points are also a challenge.

A large body of research exits on how to partition the server and scale the number of players by offloading to several servers. Modern game servers have also been parallelized to scale with more processors. However, a large amount of processing time is still wasted on lock synchronization, or the scaling is limited by partitioning requirements. In our game server design, we provide a complementary solution and try to eliminate the global synchronization points and locks, i.e., making the game server "embarrassingly parallel" which aims at increasing the number of concurrent users per machine.

VII. Conclusion

In this paper, we have shown that we can improve resource utilization by distributing load across multiple CPUs in a unified memory multi-processor system. This distribution is made possible by relaxing constraints to the ordering and atomicity of events. The system scales well, even in the case where all players must be aware of all other players and their actions. The thread pool system balances load well between the cores, and its queue-based nature means that no task is starved unless the entire system lacks resources. Message passing through the blocking queue allows objects to communicate intensively without blocking each other. Running our prototype game, we show that the 8-core server can handle twice as many clients before the response time becomes unacceptable.

VIII. Future work

From the research described in this paper, a series of further experiments present themselves. The relationship between linearly scaling load and quadratic load can be tweaked in our implementation. This could answer questions about which type of load scale better under multi-threaded implementations. Ideally, the approach presented here should be implemented in a full, complete massive multiplayer game. This should give results that are fully realistic, at least with respect to this specific game.

Another direction this work could be extended is to go beyond the single shared memory computer used and distribute the workload across clusters of computers. This could be achieved by implementing cross-server communication directly in the server code, or by using existing technology that makes cluster behave like shared memory machines.

Furthermore, all experiments described here were run with an update frequency of 10 Hz. This is good for many types of games, but different frequencies are relevant for different games. Investigating the effects of running with a higher or lower frequency of updates on server performance could yield interesting results.

If, during the implementation of a complex game, it is shown that some state changes must be atomic to keep the game state consistent, the message passing nature of this implementation means that we can use read-write-locks for any required blocking. If such cases are found, investigating how read-write-locking influence performance would be worthwhile.

388

REFERENCES

[1] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 72, april 2004.

[2] A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. *Cluster Computing*, 6:355–366, October 2003.

[3] P. B. Beskow, G. A. Erikstad, P. Halvorsen, and C. Griwodz. Evaluating ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 10:1–10:6, 2009.

[4] W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee. A scalable architecture for supporting interactive games on the internet. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation (PADS)*, pages 60–67, 2002.

[5] K.-T. Chen and C.-L. Lei. Network game design: hints and implications of player interaction. In *Proceedings of the workshop on Network and system support for games (NetGames)*, 2006.

[6] H. S. Chu. Building a simple yet powerful mmo game architecture. http://www.ibm.com/developerworks/architecture/library/ar-powerup1/, Sept. 2008.

[7] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, Nov. 2005.

[8] B. Drain. Eve evolved: Eve online's server model. http://massively.joystiq.com/2008/09/28/eve-evolved-eve-onlines-server-model/, Sept. 2008.

[9] F. Glinka, A. Ploß, J. Müller-lden, and S. Gorlatch. Rtf: a real-time framework for developing scalable multiplayer online games. In *Proceedings of the workshop on Network and system support for games (NetGames)*, pages 81–86, 2007.

[10] J. Müller and S. Gorlatch. Enhancing online computer games for grids. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 4671 of *Lecture Notes in Computer Science*, pages 80–95. Springer Berlin / Heidelberg, 2007.

[11] A. Petlund. *Improving latency for interactive, thin-stream applications over reliable transport*. Phd thesis, Simula Research Laboratory / University of Oslo, Unipub, Oslo, Norway, 2009.

[12] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz. A demonstration of a lockless, relaxed atomicity state parallel game server (LEARS). In *Proceedings of the workshop on Network and system support for games (NetGames)*, pages 1–3, 2011.

[13] J. Waldo. Scaling in games and virtual worlds. *Commun. ACM*, 51:38–44, Aug. 2008.

[14] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proceedings of the international conference on Management of data (SIGMOD)*, pages 31–42, 2007.

[15] W. White, B. Sowell, J. Gehrke, and A. Demers. Declarative processing for computer games. In *Proceedings of the ACM SIGGRAPH symposium on Video games (Sandbox)*, pages 23–30, 2008.

Paper VII: P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

P2G: A Framework for Distributed Real-Time Processing of Multimedia Data

Håvard Espeland, Paul B. Beskow, Håkon K. Stensland, Preben N. Olsen,
Ståle Kristoffersen, Carsten Griwodz, Pål Halvorsen

Department of Informatics, University of Oslo, Norway
Simula Research Laboratory, Norway
Email: {haavares, paulbb, haakonks, prebenno, staalebk, griff, paalh}@ifi.uio.no

Abstract—The computational demands of multimedia data processing are steadily increasing as consumers call for progressively more complex and intelligent multimedia services. New multi-core hardware architectures provide the required resources, but writing parallel, distributed applications remains a labor-intensive task compared to their sequential counter-part. For this reason, Google and Microsoft implemented their respective processing frameworks MapReduce [10] and Dryad [19], as they allow the developer to think sequentially, yet benefit from parallel and distributed execution. An inherent limitation in the design of these *batch* processing frameworks is their inability to express arbitrarily complex workloads. The dependency graphs of the frameworks are often limited to directed acyclic graphs, or even pre-determined stages. This is particularly problematic for video encoding and other algorithms that depend on iterative execution.

With the Nornir runtime system for parallel programs [39], which is a Kahn Process Network implementation, we addressed and solved several of these limitations. However, it is more difficult to use than other frameworks due to its complex programming model. In this paper, we build on the knowledge gained from Nornir and present a new framework, called *P2G*, designed specifically for developing and processing distributed real-time multimedia data. P2G supports arbitrarily complex dependency graphs with cycles, branches and deadlines, and provides both data- and task-parallelism. The framework is implemented to scale transparently with available (heterogeneous) resources, a concept familiar from the cloud computing paradigm. We have implemented an (interchangeable) P2G *kernel language* to ease development. In this paper, we present a proof of concept implementation of a P2G execution node and some experimental examples using complex workloads like Motion JPEG and K-means clustering. The results show that the P2G system is a feasible approach to multimedia processing.

I. INTRODUCTION

Live, interactive multimedia services are steadily growing in volume. Interactively refined video search, dynamic participation in video conferencing systems and user-controlled views in live media transmissions are a few examples of features that future consumers will expect when they consume multimedia content. New usage patterns, such as extracting features in pictures to identify objects, calculation of 3D depth information from camera arrays, or generating free-view videos from multiple camera sources in real-time, add further magnitudes of processing requirements to already computationally intensive tasks like traditional video encoding. This fact is further exacerbated by the advent of high-definition videos.

Many-core systems, such as graphic processor units (GPUs), digital signal processors (DSPs) and large scale distributed systems in general, provide the required processing power, but taking advantage of the parallel computational capacity of such hardware is much more complex than single-core solutions. In addition, heterogeneous hardware requires individual adaptation of the code, and often involve domain specific knowledge. All this places additional burdens on the application developer. As a consequence, several frameworks have emerged that aim at making distributed application development and processing easier, such as Google's MapReduce [10] and Microsoft's Dryad [19]. These frameworks are limited by their design for batch processing of large amounts of data, with few dependencies across a large cluster of machines. Modifications and enhancements that address bottlenecks [8] together with support for new types of workloads and additional hardware exist [9], [16], [31]. It is also worth mentioning that new languages for current batch frameworks have been proposed [29], [30]. However, the development and processing of distributed multimedia applications is inherently more difficult. Multimedia applications also have stricter requirements for flexibility. Support for iterations is essential, and knowledge of deadlines is often imperative. The traditional batch processing frameworks do not support this.

In our Nornir runtime system for parallel processing [39], we addressed many of the shortcomings of the batch processing frameworks. Nornir is based on the idea of Kahn Process Networks (KPN). Compared to MapReduce-like approaches, Nornir adds support for arbitrary processing graphs, deterministic execution, etc. However, KPNs are designed with some unrealistic assumptions (like unlimited queue sizes), and the Nornir programming model is much more complex than that of frameworks like MapReduce and Dryad. It demands that the application developer establishes communication channels manually to form the dependency graph.

In this paper, we expand on our visions and present our initial ideas of *P2G*. It is a completely new framework for distributed real-time multimedia processing. P2G is designed

to work on continuous flows of data, such as live video streams, while still maintaining the ability to support batch workloads. We discuss the initial ideas and present a proof-of-concept prototype[1] running on x86 multi-core machines. We present experimental results using concrete multimedia examples. Our main conclusion is that the P2G approach is a step in the right direction for development and execution of complex parallel workloads.

Figure 1. Overview of nodes in the P2G system.

II. RELATED WORK

A lot of research has been dedicated to addressing the challenges introduced by parallel and distributed programming. This has led to the development of a number of tools, programming languages and frameworks to ease the development effort.

For example, several solutions have emerged for simplifying distributed processing of large quantities of data. We have already mentioned Google's MapReduce [10] and Microsoft's Dryad [19]. In addition, you have IBM's System S and accompanying programming language SPADE [13]. Yahoo have also implemented a programming language with their PigLatin language [29], other notable mentions for increased language support is Cosmos [26], Scope [6], CIEL [25], SNAPPLE [40] and DryadLINQ [41]. The high-level languages provide easy abstractions for the developers in an environment where mistakes are hard to correct.

Dryad, Cosmos and System S have many properties in common. They all use directed graphs to model computations and execute them on a cluster. System S also supports cycles in graphs, while Dryad supports non-deterministic constructs. However, not much is known about these systems, since no open implementations are freely available. MapReduce on the other hand has become one of the most cited paradigms for expressing parallel computations. While Dryad and System S use a task parallel model, MapReduce uses a data-parallel model based on keys and values. There are several implementations of MapReduce for clusters [1], multi-core [31], the Cell BE architecture [9], and also for GPUs [16]. Map-Reduce-Merge [8] adds a merge step to process data relationships among heterogeneous data sets efficiently, operations not directly supported by the original MapReduce model. In Oivos [35], the same issues are addressed, but in addition, this system provides a more expressive, declarative programming model. Finally, reducing the layering overhead of software running on top of MapReduce is the goal of Cogset [36] where the processing architecture is changed to increase performance.

An inherent limitation in MapReduce, Dryad and Cosmos is their inability to model iterative algorithms. In addition, the rigid MapReduce semantics do not map well to all types of problems [8], which may lead to unnaturally expressed solutions and decreased performance [38]. The limited support for iterative algorithms has been mitigated in HaLoop [5], a fork of Hadoop optimized for batch processing of iterative
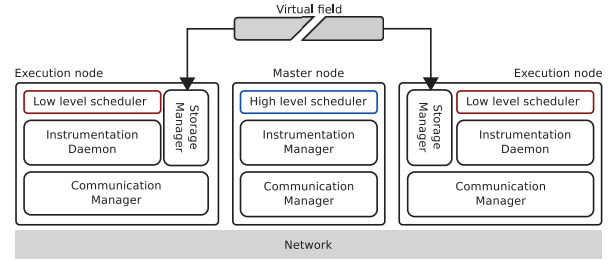
algorithms where data is kept local for future iterations of the MR steps. However, the programming model of MapReduce is designed for batch processing huge datasets, and not well suited for multimedia algorithms. Finally, Google's patent on MapReduce [11] may prompt commercial actors to look for an alternative framework.

KPN-based frameworks are one such alternative. KPNs support arbitrary communication graphs with cycles and are deterministic. However, in practice, very few general-purpose KPN runtime implementations exist. Known implementations include the Sesame project [34], the process network framework [28], YAPI [22] and our own Nornir [39]. These frameworks have several benefits, but for application developers, the KPN model has some challenges, particularly in a distributed scenario. To mention some issues, a distributed version of a KPN implementation requires a distributed deadlock detection and a developer must specify communication channels between the processes manually.

An alternative framework based on a process network paradigm is StreamIt [15], which comprises a language and a runtime system for simplifying the implementation of stream programs described by a graph that consists of computational blocks (filters) with a single input and output. Filters can be combined in fork-join patterns and loops, but must provide bounds on the number of produced and consumed messages, so a StreamIt graph is actually a synchronous data-flow process network [23]. The compiler produces code that can make use of multiple machines or CPUs, whose number is specified at compile-time, i.e., a compiled application cannot adapt to resource availability.

The processing and development of distributed multimedia applications is inherently more difficult than traditional sequential batch applications. Multimedia applications have strict requirements and knowledge of deadlines is necessary, especially in a live scenario. For multimedia applications that enable live communication, iterative processing is essential. Also, elastic scaling with the available resources becomes imperative when the workload, requirements or machine resouces change. Thus, all of the existing frameworks have some short-comings that are difficult to address, and the traditional batch processing frameworks simply come up short in our multimedia scenario. Next, inspired by the strengths of the different approach, we present our ideas for a new framework for distributed real-time multimedia processing.

---

[1]The P2G source code and workload examples are available for download from http://www.p2gproject.org/.

III. BASIC IDEA

The idea of P2G was born out of the observation that most distributed processing framework lack support for real-time multimedia workloads, and that data or task parallelism, two orthogonal dimensions for expressing parallelism, is often sacrificed in existing frameworks. With data parallelism, multiple CPUs perform the same operation over multiple disjoint data chunks. Task parallelism uses multiple CPUs to perform different operations in parallel. Several existing frameworks optimize for either task or data parallelism, not both. In doing so, they can severely limit the ability to express the parallelism of a given workload. For example, MapReduce and its related approaches provide considerable power for parallelization, but restrict runtime processing to the domain of data parallelism [12]. Functional languages such as Erlang [3] and Haskell [18] and the event-based SDL [21], map well to task parallelism. Programs are expressed as communicating processes either through message passing or event distribution, which makes it difficult to express data parallelism without specifying a fixed number of communication channels.

In our multimedia scenario, Nornir improves on many of the shortcomings of the traditional batch processing frameworks, like MapReduce and Dryad. KPNs are deterministic; each execution of a process network produces the same output given the same input. KPNs support also arbitrary communication graphs (with cycles/iterations), while frameworks like MapReduce and Dryad restrict application developers to a parallel pipeline structure and directed acyclic graphs (DAGs). However, Nornir is task-parallel, and data-parallelism must be explicitly added by the programmer. Furthermore, as a distributed, multi-machine processing framework, Nornir still has some challenges. For example, the message-passing communication channels, having exactly one sender and one receiver, are modeled as infinite FIFO queues. In real-life distributed implementations, however, queue length is limited by available memory. A distributed Nornir implementation would therefore require a distributed deadlock detection algorithm. Another issue is the complex programming model. The KPN model requires the application developer to specify the communication channels between the processes manually. This requires the developer to think differently than for other distributed frameworks.

With P2G, we build on the knowledge gained from developing Nornir and address the requirements from multimedia workloads, with inherent support for deadlines. A particularly desirable feature for processing multimedia workloads includes automatic combined task and data parallelism. Intra-frame prediction in H.264 AVC, for example, introduces many dependencies between sub-blocks of a frame, and together with other overlapping processing stages, these operations have a high potential for benefiting from both types of parallelism. We demonstrated the potential in earlier work with Nornir, whose deterministic nature showed great parallelization potential in processing arbitrary dependency graphs.

Multimedia algorithms being iterative by nature exhibit many pipeline parallel opportunities. Exploiting them are hard because intrinsic knowledge of fine-grained dependences are required, and structuring programs in such a way that pipeline parallelism can be used is difficult. Thies et al. [33] wrote an analysis tool for finding parallel pipeline opportunities by evaluating memory accesses assuming that the behaviour is stable. They evaluated their system on multimedia algorithms and gained significantly increased parallelism by utilizing the complex dependencies found. In the P2G framework, application developers model data and task dependencies explicitly, and this enable the runtime to automatically detect and take full advantage of all parallel opportunities without manual intervention.

A major source of non-determinism in other languages and frameworks lies in the arbitrary order of read and write operations from and to memory. The source of this non-deterministic behavior can be removed by adopting strict write-once semantics for writing to memory [4]. Languages that take advantage of the concept of single assignment include Erlang [3] and Haskell [18]. It enables schedulers to determine when code depending on a memory cell is runnable. This is a key concept that we adopted for P2G. While write-once-semantics are well-suited for a scheduler's dependency analysis, it is not straightforward to think about multimedia algorithms in the functional terms of Erlang and Haskell. Multimedia algorithms tend to be formulated in terms of iterations of sequential transformation steps. They act on multi-dimensional arrays of data (e.g., pixels in a picture) and provide frequently very intuitive data partitioning opportunities (e.g., 8x8-pixel macro-blocks of a picture). Prominent examples are the computation-heavy MPEG-4 AVC encoding [20] and SIFT [24] pipelines. Both are also examples of algorithms whose subsequent steps provide data decomposition opportunities at different granularities and along different dimensions of input data. Consequently, P2G should allow programmers to think in terms of fields without loosing write-once-semantics.

Flexible partitioning requires the processing of clearly distinct data units without side-effects. The idea adopted for P2G is to use *kernels* as in stream processing [15], [27]. Such a kernel is written once and describes the transformation of multi-dimensional fields of data. Where such a transformation is formulated as a loop of equal steps, the field should instead be partitioned and the kernel instantiated to achieve data-parallel execution. Each of these data partitions and tasks can then be scheduled independently by the schedulers, which can analyze dependencies and guarantee fully deterministic output independent of order due to the write-once semantics of fields.

Together, these observations determined four basic ideas for the design of P2G:

- The use of *multi-dimensional fields* as the central concept for storing data in P2G to achieve straight-forward implementations of complex multimedia algorithms.
- The use of *kernels* that process slices of fields to achieve data decomposition.
- The use of *write-once semantics* to such fields to achieve deterministic behavior.

- The use of *runtime dependency analysis* at a granularity finer than entire fields to achieve task decomposition along with data decomposition.

Within the boundaries of these basic ideas, P2G should be easily accessible for programmers who only need to write isolated, sequential pieces of code embedded in kernel definitions. The multi-dimensional fields offer a natural way to express multimedia data, and provide a direct way for kernels to fetch slices of a field in as fine a granularity as possible, supporting data parallelism.

P2G is designed to be language independent, however, we have defined a C-like language that captures many of P2G's central concepts. As such, the P2G language is inspired by many existing languages. In fact, Cray's Chapel [7] language antedates many of P2G's features in a more complete manner. P2G adds, however, write-once semantics and support for multimedia workloads. Furthermore, P2G programs consist of interchangeable language elements that formulate data dependencies between implicitly instantiated kernels, which are (currently) written in C/C++.

The biggest deviation from most other modern language designs is that the P2G kernel language makes both message passing and parallelism implicit and allows users to think in terms of sequential data transformations. Furthermore, P2G supports deadlines, which allows scheduling decisions such as termination, branching and the use of alternative code paths based on runtime observations.

In summary, we have opted for an idea that allows programmers to focus on data transformations in a sequential manner, while simultaneously providing enough information for dynamically adapting the data and task parallelization. As an end result of our considerations. P2G's fields look mostly like global multi-dimensional arrays in C, although their representation in memory may deviate, i.e., they need not be placed contiguously in the memory of a single node, and may even be distributed across multiple machines. Although this looks contrary to our message-based KPN approach used in Nornir, it maps well when slices of fields are interpreted as messages and the run-queues of worker threads as KPN channels. An obvious difference is that fields can be read as often as necessary.

IV. ARCHITECTURE

As shown in figure 1, the P2G architecture consists of a *master node* and an arbitrary number of *execution nodes*. Each execution node reports its local topology (a graph of multi-core and single-core CPUs and GPUs, connected by various kinds of buses and other networks) to the master node, which combines this information into a global topology of available resources. As such, the global topology can change during runtime as execution nodes are dynamically added and removed to accommodate for changes in the global load.

To maximize throughput, P2G uses a two-level scheduling approach. On the master node, we have a high-level scheduler (HLS), and on the execution node(s), we use a low-level scheduler (LLS). The HLS can analyze a workloads
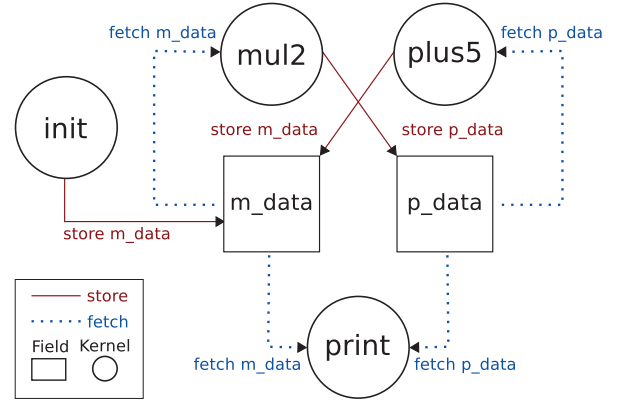
Figure 2.   Intermediate implicit static dependency graph
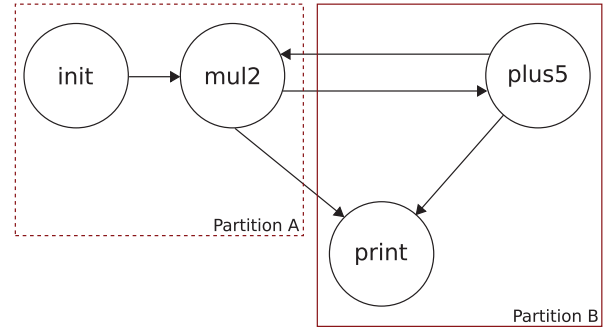
Figure 3.   Final implicit static dependency graph

store and fetch statements, from which it can generate an intermediate implicit static dependency graph (see figure 2) where edges connecting two kernels through a field can be merged, circumventing the need for a vertex representing the field (as seen in figure 3). From the intermediate graph, the HLS can then derive a final implicit static dependency graph (see figure 3). The HLS can then use a graph partitioning [17] or search based [14] algorithm to partition the workload into a suitable number of components that can be distributed to, and run, on the resources available in the topology. Using instrumentation data collected from the nodes executing the workload the final graph can be weighted with this profiling data during runtime. The weighted final graph can then be repartitioned, with the intent of improving the throughput in the system, or accommodate for changes in the global load.

Given a partial workload (such as *partition A* from figure 3), an LLS at an execution node is responsible for maximizing local scheduling decisions. We discuss this further in section V, but figure 4 shows how the LLS can combine tasks and data to minimize overhead introduced by P2G, and take advantage of specialized hardware, such as GPUs.

This idea of using a two level scheduling approach is not new. It has also been considered by Roh et al. [32], where they have performed simulations on parallel scheduling decisions for instruction sets of a functional language. Simple workloads are mapped to various simulated architectures, using a "merge-

up" algorithm, which is equivalent to our LLS, and "merge-down" algorithm, which is equivalent to our HLS. These algorithms cluster instructions in such a way that parallelism is not limited, their conclusion is that utilizing a merge-down strategy often is better.

Data distribution, reporting, and other communication patterns is achieved in P2G through an event-based, distributed publish-subscribe model. Dependencies between components in a workload are deterministically derived from the code and the high-level schedulers partitioning decisions, and direct communication occurs.

As such, P2G relies on its combination of a HLS, LLS, instrumentation data and the global topology to make best use of the performance of several heterogeneous cores in a distributed system.

V. PROGRAMMING MODEL

The programming model of P2G consists of two central concepts, the *implicit static dependency graph* (figures 2 and 3) and the *dynamically created directed acyclic dependency graph* (DC-DAG) (figure 4). We have also developed a *kernel language* (see figure 5), to make it easier to develop applications using the P2G programming model, though we consider this language to be interchangeable.

The example we use throughout this discussion consists of two primary kernels: *mul2* and *plus5*. These two kernels form a pipeline where *mul2* first multiples a value by 2 and stores this data, which *plus5* then fetches and increases by 5, *mul2* then fetches the data stored by *plus5*, and so on. The *print* kernel runs orthogonally to these two kernels and fetches and writes the data they have produced to *cout*. In combination, these three kernels form a cycle. The kernel *init* runs only once and writes some initial data for *mul2* to consume. The kernels operate on two 1-dimensional, 5 element fields. The print kernel writes *{10, 11, 12, 13, 14}, {20, 22, 24, 26, 28}* for the first *age* and *{25, 27, 29, 31, 33}, {50, 54, 58, 62, 66}* for the second, etc (as seen in figure 4). As such, the first *iteration* produces the data: *{10, 11, 12, 13, 14}, {20, 22, 24, 26, 28}* and *{25, 27, 29, 31, 33}*, and the second *iteration* produces the data: *{50, 54, 58, 62, 66}* and *{55, 59, 63, 67, 71}*, etc. Since there is no termination condition for this program it runs indefinitely.

A. Dependency graphs

The intermediate implicit static dependency graph (as seen in figure 2) is derived from the interaction between fields and kernel definitions, more precisely from the *fetch* and *store* statements of a kernel definition. This intermediate graph can be further refined by merging the edges of kernels linked through a field vertex, resulting in a final implicit static dependency graph, as depicted in figure 3. This final graph can serve as input to the HLS, which can use it to determine how best to partition the workload given a global topology. The graph can be further weighted using instrumentation data, to serve as input for repartitioning. It is important to note that these weighted graphs can serve as input to static offline

analysis. For example, it could be used as input to a simulator to best determine how to initially configure a workload, given various global topology configurations.

During runtime, the intermediate implicit static dependency graph is expanded to form a dynamically created directed acyclic dependency graph, as seen in figure 4. This expansion from a cyclic graph to a directed acyclic graph occurs as a result of our write-once semantics. As such, we can see how P2G is designed to unroll loops without introducing implicit barriers between iteration. We have chosen to call each such unrolled loop an *Age*. The LLS can then use the DC-DAG to combine tasks and data to reduce overhead introduced by P2G and to take advantage of specialized hardware, such as GPUs. It can then try different combinations of these low-level scheduling decisions to improve the throughput of the system.

We can see how this is accomplished in figure 4. When moving from *Age=1* to *Age=2*, we can see how the LLS has made a decision to reduce data parallelity. In P2G, kernels fetch slices of data, and initially *mul2* was defined to work on each single field entry in parallel, but in *Age=2*, the LLS has decreased the granularity of the fetch statement to encompass the entire field. It could also have split the field in two, leading to two kernel instances of *mul2*, working on disparate sets of the field.

Moving from *Age=2* to *Age=3*, we see how the LLS has made a decision to decrease the task parallelity. This is possible because *mul2* and *plus5* effectively form a pipeline, information that is available from the static graphs. By combining these two tasks, the individual store operations of the tasks are deferred until the data has been fully processed by each task. If the *print* kernel was not present, storing to the intermediate field *m_data* could be circumvented in its entirety.

Finally, moving from *Age=3* to *Age=4*, we can see how a decision to decrease both task and data parallelity has been taken. This renders this single kernel instance effectively instance into a classical *for-loop*, working on each data element of the field, with each task (*mul2*, *plus5*) performed sequentially on the data.

P2G makes runtime adjustments dynamically to both data and task parallelism based on the possibly oscillating resource availability and the reported performance monitoring.

B. Kernel language

From our experience with developing Nornir, we came to the realization that expressing workloads in a framework capable of supporting such complex graphs without a high-level language is a difficult task. We have therefore developed a *kernel language*. An implementation of a simple workload is outlined in figure 5, with a C++ equivalent listed in figure 6.

In the current version of our system, P2G is exposed to the developer through this *kernel language*. The language itself is not an integral part and can be replaced easily. However, it exposes several foundations of the P2G design. Most important are the kernel and field definitions, which describe the code and interaction patterns in P2G.
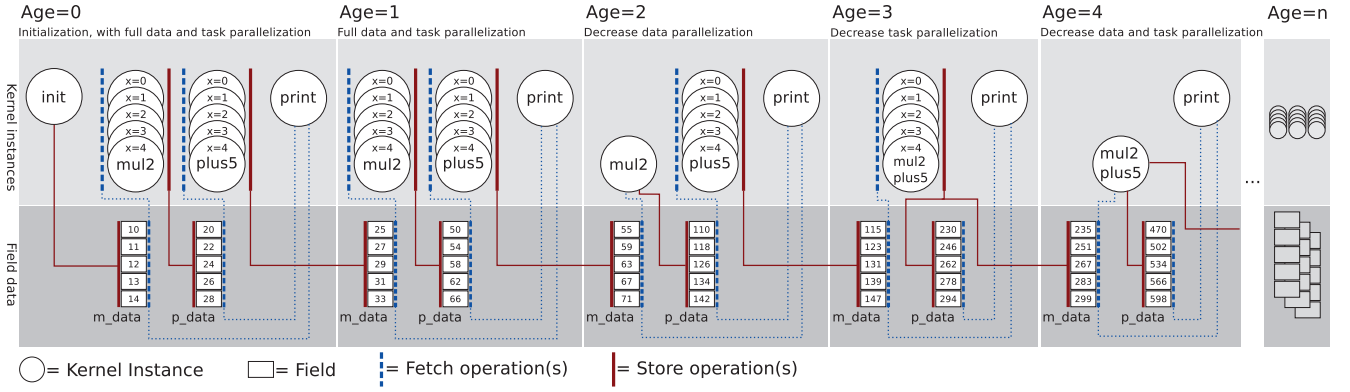
Figure 4. Dynamically created directed acyclic dependency graph (DC-DAG)

Field definitions:

```
0  int32[] m_data age;
1  int32[] p_data age;
```

Kernel definitions:

```
0  init:
1    local int32[] values;
2
3    %{
4      int i = 0;
5      for( ;i < 5; ++i )
6      {
7        put( values, i+10, i );
8      }
9    %}
10
11   store m_data(0) = values;
12
13
```

```
0  mul2:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = m_data(a)[x];
6
7    %{
8      value *= 2;
9    %}
10
11   store p_data(a)[x] = value;
12
13
```

```
0  plus5:
1    age a;
2    index x;
3    local int32 value;
4
5    fetch value = p_data(a)[x];
6
7    %{
8      value += 5;
9    %}
10
11   store m_data(a+1)[x] = value;
12
13
14
15
```

```
0  print:
1    age a;
2    local int32[] m, p;
3
4    fetch m = m_data(a);
5    fetch p = p_data(a);
6
7    %{
8    for(int i=0; i < extent(m, 0);)
9      cout << get(m, i++) << " ";
10   cout << endl;
11
12   for(int i=0; i < extent(p, 0);)
13     cout << get(p, i++) << " ";
14   cout << endl;
15 %}
```

Figure 5. Kernel and field definitions

A kernel definition's primary purpose is to describe the required interaction of a kernel instance with an arbitrary number of fields (holding the application data) through the fetch and store statements. As such, a field serves as an interaction point for kernel definitions, as can be seen in figure 2.

An important aspect of multimedia workloads is the ability to express deadlines, where it does not make sense to encode a frame if the playback has moved past that point in the

```cpp
void print( int *data, int num )
{
    for( int i = 0; i < num; ++i )
        std::cout << data[i] << " ";
    std::cout << std::endl;
}
int main()
{
    int m_data[5] = { 10, 11, 12, 13, 14 };
    int p_data[5];

    while( true )
    {
        for( int i = 0; i < 5; ++i )
            p_data[i] = m_data[i] * 2;

        print( m_data, 5 );
        print( p_data, 5 );

        for( int i = 0; i < 5; ++i )
            m_data[i] = p_data[i] + 5;
    }
    return 0;
}
```

Figure 6. C++ equivalent of mul/sum example

video-stream. Consequently, we have implemented language support for expressing deadlines. In principle, a deadline gives the application developer the option of defining a global timer: *timer t1*. This timer can then be polled, and updated, from within a kernel definition, for example *t1+100ms* or *t1 = now*. Given a condition based on a deadline such as *t1+100ms*, a timeout can occur and an alternate code-path can be executed. Such an alternate code-path is executed by storing to a different field then in the primary path, leading to new dependencies and new behavior. Currently, we have basic support for expressing deadlines in the kernel language, but the semantics of these expressions require refinement, as their implications can be considerable.

Fields in P2G have a number of properties, including a type and a dimensionality. Another property is, as mentioned above, *aging*, which allows kernels to be iterative while maintaining write-once semantics in such cyclic execution. Aging enables unique storage to the same position in a field several times,

as long as the age increases for each store operation (as seen in figure 4). In essence, this adds a dimension to the field and makes it possible to accommodate iterative algorithms. Additionally, it is important to realize that fields are not connected to any single node, and can be fully localized or distributed across multiple execution nodes (as seen in figure 1).

In defining the interaction between kernels and fields, it is encouraged that the programmer expresses the finest possible granularity of kernel definitions, and, likewise, the most precise slices possible for the kernel within the field. This is encouraged because it provides the low-level scheduler more control over the granularity of task and data decomposition. Aided by instrumentation data, it can reduce scheduling overhead by combining several instances of a kernel that process different data, or several instances of different kernels that process data in sequence (as seen in figure 4). The scheduler makes its decisions based on the implicit static dependency graph and instrumentation data.

C. Runtime

Following from the previous discussions, we can extrapolate the concept of kernel definitions to kernel instances. A kernel instance is the unit of code that is executed during runtime, and the number of kernel instances executed in parallel for a given kernel definition depends on its fetch statements.

To clarify, a kernel instance works on an arbitrary number of slices of fields, depending on the number of fetch statements of the kernel definition. For example, looking at figure 4 and 5, we can see how the *mul2* kernel, given its *fetch* statement on *m_data* with *age=a* and *index=x* fetches only a single element of the data. Thus, since the *m_data* field consists of five data elements, this means that P2G can execute a maximum possible *x* kernel instances simultaneously per age, giving *a*x* *mul2* kernel instances. Though, as we have seen, this number can be decreased by the scheduler making *mul2* work over larger slices of data from *m_data*.

With P2G we support implicit resizing of fields, this can be witnessed by looking at the kernel definition of *print* in figure 5. Initially, the extents of *m_data* and *p_data* are not defined, as such, with each iteration of the *for*-loop in *init* the local field *values* is resized locally, leading to a resize of the global field *m_data* when *values* is stored to it. These extents are then propagated to the respective fields impacted by this resize, such as *p_data*. Following the discussion from the previous paragraph, such an implicit resize can lead to additional kernel instances being dispatched.

It is worth noting that a kernel instance is only dispatched when all its dependencies are fulfilled, i.e., that the data it fetches has been stored to the respective fields and elements. Looking at figure 4 and 5 again, we can see that *mul2* stores its result to *p_data* with *age=a* and *index=x*. This means that once *mul2* has stored its results to *p_data* with *index=2* and *age=0*, this means that the kernel instance *plus5* with the fetch statement *fetch(0)[2]* can be dispatched. In our system, each kernel instance is only dispatched once, due to our write-once semantics. To summarize, the *print* kernel instance working on *age=0* becomes runnable when all the elements of *m_data* and *p_data* for *age=0* have been stored. Once it has become runnable, it is dispatched and runs only once.

VI. PROTOTYPE IMPLEMENTATION

To verify the feasibility of the P2G framework presented in this paper, we have implemented a prototype version. The prototype consists of a compiler for the kernel language and a runtime that can execute P2G programs on multi-core linux machines.

A. Compiler

Programs written for the P2G system are designed to be platform independent and feature native blocks of code written in C or C++. Heterogeneous systems are specifically targeted, but many of these require a custom compiler for the native blocks, such as nVIDIA's nvcc compiler for the CUDA system and IBM's XL compiler for the Cell Broadband Engine. We decided to compile P2G programs into C++ files, which can be further compiled and linked with native code blocks, instead of generating binaries directly. This approach gives us less control of the resulting object code, but we gain the flexibility and sophisticated optimization of the native compilers, resulting in a lightweight P2G compiler. The P2G compiler works also as a compiler driver for the native compiler and produces complete binaries for programs that run directly on the target system.

B. Runtime

The runtime prototype implements the basic features of a P2G execution node, including multi-dimensional field support, implicit resizing of fields, instrumentation and parallel execution of kernel instances on multiple processors using the implicit dependency graph formed by kernel definitions. However, at the time of writing, the prototype runtime does not yet have a full implementation of deadline expressions, this is because the semantics of the kernel language support for this feature is not fully defined yet.

The prototype targets a node with multiple processors. It is designed as a push-based system using event subscriptions on field operations. Kernel instances are executed in parallel and produce events on *store* statements, which may require resize operations. A kernel subscribes to events related to fields that it depends on, i.e., fields referenced to by the kernels *fetch* statements. When receiving such a storage event, the runtime finds all *new* valid combinations of age and index variables that can be processed as a result of the *store* statement, and puts these in a per-kernel ready queue. This means that the ready queues contain always the maximum number of parallel instances that can be executed at any time, only limited by unfulfilled data dependencies.

The low-level scheduler consists of a dependency analyzer and kernel instance dispatcher. Using the implicit dependency graph, the dependency analyzer adds new kernel instances to a ready queue, which later can be processed by the worker
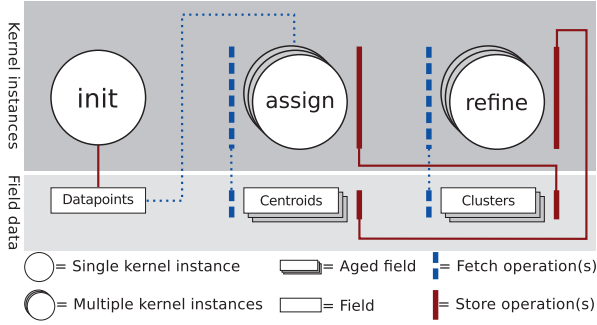
Figure 7. Overview of the *K*-means clustering algorithm

Figure 8. Overview of the MJPEG encoding process

4-way Intel Core i7	
CPU-name	Intel Core i7 860 2,8 GHz
Physical cores	4
Logical threads	8
Microarchitecture	Nehalem (Intel)
8-way AMD Opteron	
CPU-name	AMD Opteron 8218 2,6 GHz
Physical cores	8
Logical threads	8
Microarchitecture	Santa Rosa (AMD)

Table I
OVERVIEW OF TEST MACHINES

threads. Dependencies are analyzed in a dedicated thread which handles events emitted from running kernel instances that notifies on *store* and *resize* operations performed on fields. Kernel instances are executed by a worker thread dispatched from the ready queue. They are scheduled in an order that prefers the execution of kernel instances with a lower age value (older kernel instances). This ensures that no runnable kernel instance is starved by others that have no *fetch* statements or by groups of kernels that satisfy their own dependencies in aging cycles, such as the *mul2* and *plus5* kernel in figure 5.

The runtime is written in C++ and uses the blitz++ [37] library for high-performance multi-dimensional arrays. The source code for the P2G compiler and runtime can be downloaded from http://www.p2gproject.org/.

VII. WORKLOADS

We have implemented a few workloads commonly used in multimedia processing to test the prototype implementation. The P2G kernel language is able to expose both the data and task parallelism of the programs to the P2G system, so the runtime is able to adapt execution of the programs to suit the target architecture.

A. K-means clustering

K-means clustering is an iterative algorithm for cluster analysis which aims to partition n datapoints into k clusters in which each datapoint belongs to the cluster with the nearest mean. As shown in figure 7, the P2G k-means implementation consists of an *init* kernel, which generates n datapoints and *stores* them to the datapoints field. Then, it selects k of these datapoints randomly, as the initial means, and *stores* them to the centroids field. Next, the *assign* kernel *fetches* a slice of data, a single datapoint per *kernel instance*, the last calculated centroids, and *stores* this datapoint to the cluster of the closest centroids using the euclidean distance calculation. Finally, the *refine* kernel *fetches* a cluster, calculates its new mean and *stores* this information in the centroids field. The kernel definitions of *assign* and *refine* form a loop which gradually leads to a convergence in centroids, at which point the k-means algorithm has completed.

B. Motion JPEG

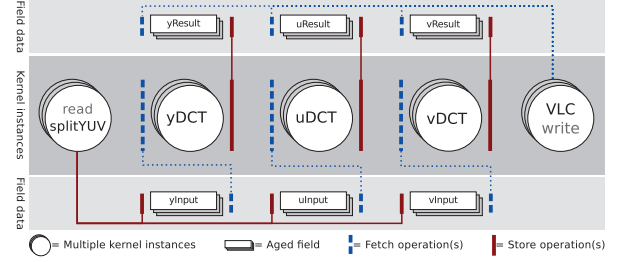Motion JPEG (MJPEG) is a video coding format using a sequence of separately compressed JPEG images. The MJPEG format provides many layers of parallelism, well suited for illustrating the potential of the framework. We focused on optimizing the discrete cosine transform (DCT) and quantization part as this is the most compute-intensive part of the codec.

The *read + splitYUV* kernel reads the input video in YUV-format and stores the data in three global fields, *yInput*, *uInput*, and *vInput*. The read loop ends when the kernel stops storing to the next age, e.g., at the end of the file. In our scenario, three YUV components can be processed independently of each other and this property is exploited by creating three kernels, *yDCT*, *uDCT* and *vDCT*, one for each component. From figure 8, we see that the respective DCT kernels are dependent on one of these fields.

The encoding process of MJPEG comprises splitting the video frames into 8x8 macro-blocks. For example, given the CIF resolution of 352x288 pixels per frame used in our tests, this generates 1584 macro-blocks of Y (luminance) data, each with 64 pixel values. This makes it possible to create 1584 instances per age of the DCT kernel transforming luminance. The 4:2:2 chroma sub-sampling yields 396 kernel instances from both the U and V (chroma) data. Each of these kernel instances stores the DCT'ed macro-block into global result fields *yResult*, *uResult* and *vResult*. Finally, the *VLC + write* kernel store the MJPEG bit-stream to disk.

VIII. EVALUATION

We have run tests with the workloads Motion JPEG and *K*-means (described in section VII). Each test was run on a *4-way Core i7* and an *8-way Opteron* (see table I for hardware specifications) ranging from *1* worker thread to *8* worker threads with *10* iterations per worker thread count. The results of these tests are reported in the figures 10 and 9, which show the mean running time in *seconds* for each machine for a given thread count with standard deviation reported as error-bars.
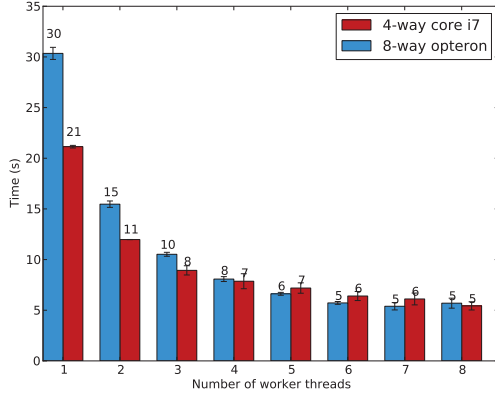
218

Figure 9.   Workload execution time for Motion JPEG

Kernel	Instances	Dispatch Time	Kernel Time
init	1	69.00 μs	18.00 μs
read/splityuv	51	35.50 μs	1641.57 μs
yDCT	80784	3.07 μs	170.30 μs
uDCT	20196	3.14 μs	170.24 μs
vDCT	20196	3.15 μs	170.58 μs
VLC/write	51	3.09 μs	2160.71 μs

Table II
MICRO-BENCHMARK OF MJPEG ENCODING IN P2G

In addition, we have performed micro-benchmarks for each workload, summarized in the tables II and III. The benchmarks summarize the number of kernel instances dispatched per kernel definition, dispatch overhead and time spent in kernel code.

A. Motion JPEG

The Motion JPEG workload is run on the standard test sequence *Foreman* encoded in *CIF* resolution. We limited the workload to process *50* frames of video.

As we can observe from figure 9, P2G is able to scale close to linearly with the resources it has available. In P2G, the dependency analyzer of the LLS runs in a dedicated thread. This affects the running time when moving from *7* to *8* worker threads. Where the eighth thread shares resources with the dependency analyzer. To compare, the standalone single threaded MJPEG encoder on which the P2G version is based upon has a running time of *30* seconds on the Opteron machine and *19* seconds on the Core i7 machine. Note that both the standalone and P2G versions of the MJPEG encoder use a naive DCT calculation, there are versions of DCT that can significantly improve performance, such as FastDCT [2].

From table II, we can see that time spent in kernel code is considerably higher compared to the dispatch overhead for the kernel definitions. The dispatch time includes allocation or reallocation of fields as part of the timing operation. As a result, *init* and *read/splitYUV* have a considerably higher dispatch time then the **DCT* operations.

We can also see that the majority of CPU-time is spent in the kernel instances of *yDCT, uDCT* and *vDCT*, which is the computationally intensive part of the workload. This
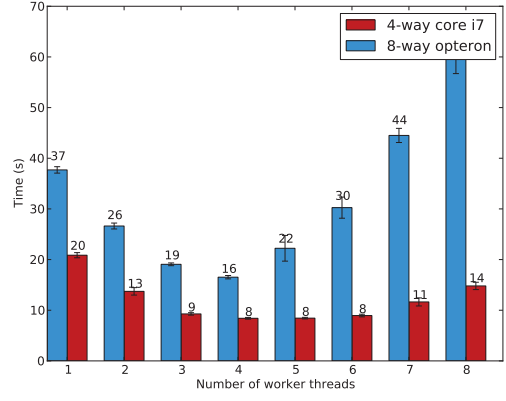

Figure 10.   Workload execution time for *K*-means

indicates that decreasing data and task granularity, as discussed in section V-A, has little impact on the throughput of the system. This is because the majority of time is already spent in kernel code.

Note that even though there are *51* instances of the *read-/write* kernel definitions, only *50* frames are encoded, because the last instance reaches the end of the video stream.

B. K-means

The *K*-means workload is run with *K=100* using a randomly generated data set containing *2000* datapoints. The *K*-means algorithm is not run until convergence, but with *10* iterations. If we do not define this break-point it is undefined when the algorithm converges, and as such, we have introduced this condition to ensure that we get a relatively stable running time for each run.

As seen in figure 10, the *K*-means workload scales to *4* worker threads. After this, the running time increases with the number of worker threads. This can be explained by the fine granularity of the *assign* kernel definition, as witnessed when comparing the dispatch time to the time spent in kernel code. This leads to the serial dependency analyzer becoming a bottle-neck in the system. As discussed in section V-A, this condition could be alleviated by decreasing the granularity of data-parallelism, in effect leading to each kernel instance of *assign* working on larger slices of data. By doing so, we would increase the ratio of time spent in kernel code compared to dispatch time and reduce the workload of the dependency analyzer. The reduction in work for the dependency analyzer is a result of the lower number of kernel instances being run.

The two different test machines behave somewhat differently in that the Opteron suffers more than the Core i7 when the dependency analyzer saturates a core. The Core i7 is able to increase the frequency of a single core to mitigate serial bottlenecks, and we think this is why the Core i7 suffers less when we meet the limitations dictated by Amdahl's law.

The considerable time *init* spends in kernel code is because it generates the data set.

Kernel	Instances	Dispatch Time	Kernel Time
init	1	58.00 μs	9829.00 μs
assign	2024251	4.07 μs	6.95 μs
refine	1000	3.21 μs	92.91 μs
print	11	1.09 μs	379.36 μs

Table III
MICRO-BENCHMARK OF K-MEANS IN P2G

C. Summary

We have shown that our prototype implementation of an execution node is able to scale with the available resources, as seen in figure 9 and 10. Our initial results indicate that the functionality of decreasing the granularity of task and data parallelity, as discussed in section V-A, is important to ensure full resource utilization.

IX. DISCUSSION

Even though support for deadlines is not yet fully implemented in the P2G runtime, the concept of deadlines formed an integral part of our design goal. The intention behind deadlines is to accommodate for live multimedia workloads, where real-time requirements are mission essential. Varying conditions over time, both in the workload and topology, may effect scheduling decisions: such as termination, branching and the use of alternative code paths based on runtime observations. This is similar to SDL, but unlike contemporary high performance languages.

In P2G, we encourage the programmer to describe the workload in as fine granularity as possible, both in the functional and data decomposition domains. The low-level scheduler has an understanding of both decomposition domains and deadlines. Given this information, the low-level scheduler can minimize overhead by combining functional components and slices of data by adapting to its available resources, be it local cores, or even GPU execution units.

Write-once semantics on fields incurs a large penalty if implemented naively, both in terms of memory usage and data cache misses. However, as the fields are virtual and do not even have to reside in continuous memory, the compiler and runtime are free to optimize field usage. This includes re-using buffers for increased cache locality when old ages are no longer referenced, and garbage collecting old ages. The explicit programming model of P2G allows the system to anticipate what data is needed in the future, which can be used for further optimizations.

Given the complexity of multimedia workloads and the (potentially) heterogeneous resources available in a modern topology, and in many cases, no knowledge of the underlying capabilities of the resources (which is common in modern cloud services), mapping these complex multimedia workloads manually to the available resources becomes an increasingly difficult task, and at some point, even impossible. This is particularly the case where resource availability fluctuates, such as in modern virtual machine parks. With batch processing, where the workloads frequently are not associated with some intrinsic deadline, this task is solved, with frameworks such as MapReduce and Dryad. However, for processing continuous streams such as iterative multimedia algorithms in an elastic manner requires new frameworks; P2G is a step in that direction.

X. CONCLUSION

With P2G, we have proposed a new flexible framework for automatic parallel, real-time processing of multimedia workloads. We encourage the programmer to specify parallelism in as fine a granularity as possible along the axes of data and task decomposition. Using our kernel language this decomposition is expressed through kernel definitions and fetch and store statements on fields. This language is independent from the P2G runtime and can easily be replaced. Given a workload defined in our kernel language it is compiled for execution in P2G. This workload can then be partitioned by the high-level scheduler of a P2G master node, which then distributes partitions to P2G execution nodes which runs the tasks locally. Execution nodes can consist of heterogeneous resources. A low-level scheduler at the execution nodes then adapted the partial (or full) workload to run optimally using resources at hand. Feedback from the instrumentation daemon at the execution node can lead to repartitioning of the workload (a task performed by the high-level scheduler). The aim is to bring the ease of batch-processing frameworks to multimedia workloads.

In this paper we have presented an execution node capable of running on a multi-way architecture. This results from our experiments running on this prototype show the potential of our ideas. However, there still remains a number of vectors for optimization. In the low-level scheduler we have identified that combining task and data to minimize overhead introduced by P2G is a first reasonable modification. Additionally, completing the implementation of a fully distributed version is in the pipeline. Also, writing workloads for heterogeneous processing cores like GPUs and non-cache coherent architectures like Intel's SCC is a further consideration. Currently, we are investigating appropriate mechanisms for both high- and low-level scheduling, garbage collection, fat binaries, resource profiling and monitoring, and efficient migration of tasks.

While a number of optimizations remain, we have determined that P2G is feasible, through the implementation of this execution node, and the successful implementation of multimedia workloads, such as Motion JPEG and k-means. With these workloads we have shown that it is possible to express multimedia workloads in the kernel language and we have implemented a prototype of an execution node in the P2G framework that is able to execute kernels and scales with the available resources.

REFERENCES

[1] Apache. Hadoop, Accessed July 2010. http://hadoop.apache.org.
[2] Y. Arai, T. Agui, and M. Nakajima. A fast dct-sq scheme for images. *Transactions of IEICE*, E71(11), 1988.
[3] J. Armstrong. A history of Erlang. In *Proc. of ACM HOTL III*, pages 6:1–6:26, 2007.
[4] R. Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *TOPLAS*, 11(4):598–632, 1989.

[5] Y. Blu, B. Howe, M. Balazinska, and M. Ernst. Haloop: Efficient iterative data processing on large clusters. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2010.

[6] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.

[7] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 23(3), 2007.

[8] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *Proc. of ACM SIGMOD*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[9] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell BE architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proc. of USENIX OSDI*, pages 10–10, 2004.

[11] J. Dean and S. Ghemawat. System and method for efficient large-scale data processing. *US Patent Application*, (US 7650331), 2010.

[12] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[13] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[14] F. Glover. Tabu search, Part I1. *ORSA journal on Computing*, 2(1):4–32, 1990.

[15] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[16] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proc. of PACT*, pages 260–269, New York, NY, USA, 2008. ACM.

[17] B. Hendrickson and T. Kolda. Graph partitioning models for parallel computing* 1. *Parallel Computing*, 26(12):1519–1534, 2000.

[18] P. H. J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. of ACM HOTL III*, pages 12:1–12:55, 2007.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of ACM EuroSys*, pages 59–72, New York, NY, USA, 2007. ACM.

[20] ISO/IEC. *ISO/IEC 14496-10:2003*, 2003. Information technology - Coding of audio-visual objects - Part 10: Advanced Video Coding.

[21] ITU. *Z.100*, 2007. Specification and Description Language (SDL).

[22] E. A. D. Kock, G. Essink, W. J. M. Smits, and P. V. D. Wolf. Yapi: Application modeling for signal processing systems. In *In Proc. 37th Design Automation Conference (DAC'2000*, pages 402–405. ACM Press, 2000.

[23] T. Lee, E.A.; Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[24] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[25] D. Murray, M. Schwarzkopf, and C. Smowton. Ciel: a universal execution engine for distributed data-flow computing. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[26] C. Nicolaou. An architecture for real-time multimedia communication systems. *Selected Areas in Communications, IEEE Journal on*, 8(3):391–400, 1990.

[27] Nvidia. Nvidia cuda programming guide 3.2, Aug. 2010.

[28] A. G. Olson and B. L. Evans. Deadlock detection for distributed process networks. In *in Proc. IEEE Int. Conf. Acoustics, Speech, Signal Processing (ICASSP*, pages 73–76, 2006.

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proc. of ACM SIGMOD*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[30] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of IEEE HPCA*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[32] L. Roh, W. A. Najjar, and A. P. W. Böhm. Generation and quantitative evaluation of dataflow clusters. In *ACM FPCA: Functional Programming Languages and Computer Architecture*, New York, NY, USA, 1993. ACM.

[33] W. Thies, V. Chandrasekhar, and S. Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society.

[34] M. Thompson and A. Pimentel. Towards multi-application workload modeling in sesame for system-level design space exploration. In S. Vassiliadis, M. Berekovic, and T. Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 222–232. Springer Berlin / Heidelberg, 2007.

[35] S. V. Valvåg and D. Johansen. Oivos: Simple and efficient distributed data processing. In *Proc. of IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 113–122, 2008.

[36] S. V. Valvåg and D. Johansen. Cogset: A unified engine for reliable storage and parallel processing. In *Proc. of IFIP International Conference on Network and Parallel Computing Workshops (NPC)*, pages 174–181, 2009.

[37] T. L. Veldhuizen. Arrays in blitz++. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE '98, pages 223–230, London, UK, 1998. Springer-Verlag.

[38] Ž. Vrba, P. Halvorsen, C. Griwodz, and P. Beskow. Kahn process networks are a flexible alternative to mapreduce. *High Performance Computing and Communications, 10th IEEE International Conference on*, 0:154–162, 2009.

[39] Ž. Vrba, P. Halvorsen, C. Griwodz, P. Beskow, H. Espeland, and D. Johansen. The Nornir run-time system for parallel programs using Kahn process networks on multi-core machines - a flexible alternative to MapReduce. *Journal of Sumpercomputing*, 27(1), 2010.

[40] D. Waddington, C. Tian, and K. Sivaramakrishnan. Scalable lightweight task management for mimd processors, 2011.

[41] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. Dryadlinq: A system for general-puropose distributed data-parallel computing using a high-level language, 2008.

Paper VIII: Bagadus: An Integrated Real-Time System for Soccer Analytics

Title: Bagadus: An Integrated Real-Time System for Soccer Analytics [114].

Authors: H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, Ø. Landsverk, M. Stenshaug, P. Halvorsen, C. Griwodz and D. Johansen.

Published: ACM Transactions on Multimedia Computing, Communications and Applications (TOMM), Volume 10, Issue 1s, ACM, 2014.

Bagadus: An Integrated Real-Time System for Soccer Analytics

HÅKON KVALE STENSLAND, VAMSIDHAR REDDY GADDAM, MARIUS TENNØE,
ESPEN HELGEDAGSRUD, MIKKEL NÆSS, HENRIK KJUS ALSTAD, ASGEIR MORTENSEN,
RAGNAR LANGSETH, SIGURD LJØDAL, ØYSTEIN LANDSVERK, CARSTEN GRIWODZ, and
PÅL HALVORSEN, University of Oslo and Simula Research Laboratory
MAGNUS STENHAUG and DAG JOHANSEN, University of Tromsø

The importance of winning has increased the role of performance analysis in the sports industry, and this underscores how statistics and technology keep changing the way sports are played. Thus, this is a growing area of interest, both from a computer system view in managing the technical challenges and from a sport performance view in aiding the development of athletes. In this respect, Bagadus is a real-time prototype of a sports analytics application using soccer as a case study. Bagadus integrates a sensor system, a soccer analytics annotations system, and a video processing system using a video camera array. A prototype is currently installed at Alfheim Stadium in Norway, and in this article, we describe how the system can be used in real-time to playback events. The system supports both stitched panorama video and camera switching modes and creates video summaries based on queries to the sensor system. Moreover, we evaluate the system from a systems point of view, benchmarking different approaches, algorithms, and trade-offs, and show how the system runs in real time.

1. INTRODUCTION

Sport analysis has become a large industry, and a large number of (elite) sports clubs study their game performance, spending a large amount of resources. This analysis is performed either manually or using one of the many existing analytics tools. In the area of soccer, several systems enable trainers

ACM Transactions on Multimedia Computing, Communications and Applications, Vol. 10, No. 1s, Article 14, Publication date: January 2014.

225

and coaches to analyze the game play in order to improve the performance. For instance, in Interplay Sports [2013], video streams are manually analyzed and annotated using a soccer ontology classification scheme. ProZone [2013] automates some of the manual annotation process by video-analysis software. In particular, it quantifies player movement patterns and characteristics like speed, velocity, and position of the athletes, and it has been successfully used at, for example, Old Trafford in Manchester and Reebook Stadium in Bolton [Salvo et al. 2006]. Similarly, STATS SportVU Tracking Technology [Stats 2013] uses video cameras to collect the positioning data of the players within the playing field in real time. This is further compiled into player statistics and performance. Camargus [2013] provides a very nice video technology infrastructure but lacks other analytics tools. As an alternative to video analysis, which often is inaccurate and resource hungry, both the Cairo's VIS.TRACK [Cairos Technologies 2013b] and ZXY Sport Tracking [ZXY 2013] systems use global positioning and radio-based systems for capturing performance measurements of athletes. Thus, these systems can present player statistics, including speed profiles, accumulated distances, fatigue, fitness graphs and coverage maps, in many different ways, such as charts, 3D graphics, and animations.

To improve game analytics, video that replays real game events becomes increasingly important. However, the integration of the player statistics systems and video systems still requires a large amount of manual labor. For example, events tagged by coaches or other human expert annotators must be manually extracted from the videos, often requiring hours of work in front of the computer. Furthermore, connecting the player statistics to the video also requires manual work. One recent example is the Muihtu system [Johansen et al. 2012], which integrates coach annotations with related video sequences, but the video must be manually transferred and mapped to the game timeline.

As these examples show, there exist several tools for soccer analysis. However, to the best of our knowledge, there does not exist a system that fully integrates all these features. In this respect, we have presented earlier [Halvorsen et al. 2013] and demonstrated [Sægrov et al. 2012] a system called Bagadus. This system integrates a camera array video capture system with the ZXY Sport Tracking system for player statistics and a system for human expert annotations. Bagadus allows the game analytics to automatically play back a tagged game event or extract a video of events extracted from the statistical player data, for example, all sprints at a given speed. Using the exact player position provided by sensors, a trainer can also follow individuals or groups of players, where the videos are presented either using a stitched panorama view or by switching cameras. Our earlier work [Halvorsen et al. 2013; Sægrov et al. 2012] demonstrated the integrated concept but did not have all operations, like generation of the panorama video, in real time. In this article, we present enhancements providing live, real-time analysis and video playback by using algorithms to enhance the image quality, parallel processing, and offloading to co-processing units like GPUs. Our prototype is deployed at Alfheim Stadium (Tromsø IL, Norway), and we use a dataset captured at a Norwegian premier league game to demonstrate our system.

The remainder of the article is structured as follows. Next, in Section 2, we give a brief overview of the basic idea of Bagadus and introduce the main subsystems. Then, we look at the video-, tracking-, and analysis-subsystems in more detail in Sections 3, 4, and 5, respectively. Then, we briefly explain the case study at Alfheim Stadium in Section 6. Section 7 provides a brief discussion of various aspect of the system before we conclude in Section 8.

2.   BAGADUS – THE BASIC IDEA

Interest in sports analysis systems has recently increased a lot, and it is predicted that sports analytics will be a real game-changer, that is, "statistics keep changing the way sports are played—and changing minds in the industry" [Dizikes 2013]. As already described, several systems exist, some for a long time, already providing game statistics, player movements, video highlights, etc. However, to a
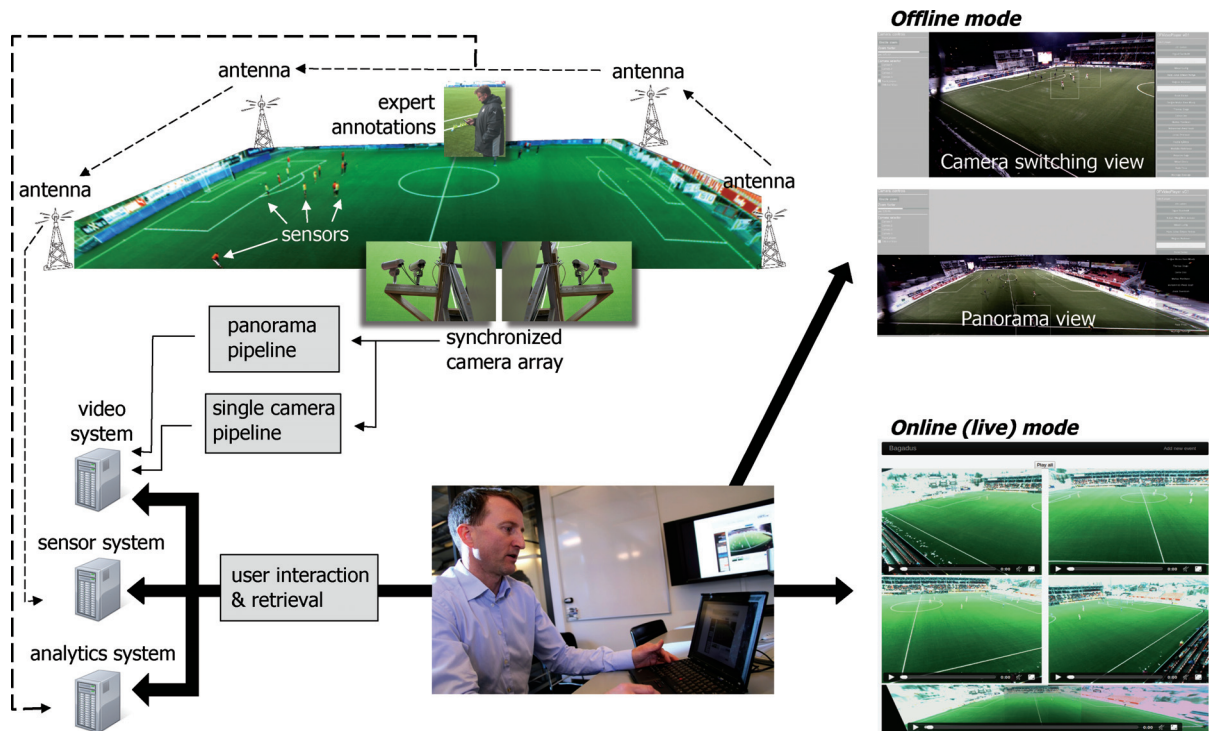
Fig. 1.   Overall Bagadus architecture.

large degree, the existing systems are offline systems, and they require a large portion of manual work to integrate information from various computer systems and expert sport analytics. In this respect, *Bagadus* is a prototype that aims to fully integrate existing systems and enable real-time presentation of sport events. Our system is built in cooperation with the Tromsø IL soccer club and the ZXY Sport Tracking company for soccer analysis. A brief overview of the architecture and interaction of the different components is given in Figure 1. The Bagadus system is divided into three different subsystems which are integrated in our soccer analysis application.

The *video* subsystem consists of multiple, small, shutter-synchronized cameras that record a high resolution video of the soccer field. They cover the full field with sufficient overlap to identify common features necessary for camera calibration and image stitching. Furthermore, the video subsystem supports two different playback options. The first allows playback of video that switches between streams delivered from the different cameras, either manually selecting a camera or automatically following players based on sensor information. The second option plays back a panorama video stitched from the different camera feeds. The cameras are calibrated in their fixed position, and the captured videos are each processed and stored using a capture–debarrel–rotate–stitch–encode–store pipeline. In an offline mode, Bagadus allows a user to zoom in on and mark player(s) in the retrieved video on the fly (see Figure 1), but this is not yet supported in the live mode used during the game.

To identify and follow players on the field, we use a *tracking* (sensor) subsystem. In this respect, tracking people through camera arrays has been an active research topic for several years. The accuracy of such systems has improved greatly, but there are still errors. Therefore, for stadium sports, an interesting approach is to use sensors on players to capture the exact position. In this area, ZXY Sport

Tracking [ZXY 2013] provides such a sensor-based solution that provides player position information. Bagadus uses this position information to track players, or groups of players, in single camera views, stitched views, or zoomed-in modes.

The third component of Bagadus is an *analytics* subsystem. Coaches have for a long time analyzed games in order to improve their own team's game play and to understand their opponents. Traditionally, this has been done by making notes using pen and paper, either during the game or by watching hours of video. Some clubs even hire one person per player to describe the player's performance. To reduce the manual labor, we have implemented a subsystem that equips members of the trainer team with a tablet (or even a mobile phone), where they can register predefined events quickly with the press of a button or provide textual annotations. In Bagadus, the registered events are stored in an analytics database and can later be extracted automatically and shown along with a video of the event.

Bagadus implements and integrates many well-known components to support our arena sports analytics application scenario. The main novelty of our approach is then the combination and integration of components enabling automatic presentation of video events based on the sensor and analytics data that are synchronized with the video system. This gives a threefold contribution: (1) a method for spatially mapping the different coordinate systems of location (sensor) data and video images to allow for seamless integration; (2) a method for recording and synchronizing the signals temporally to enable semantic extraction capabilities; and (3) the integration of the entire system into an interactive application that can be used online and offline.

Thus, in the offline mode, Bagadus will, for example, be able to automatically present a video clip of all the situations where a given player runs faster than 10 meters per second or when all the defenders were located in the opponent's 18-yard box (penalty box). Furthermore, we can follow single players and groups of players in the video and retrieve and play back the events annotated by expert users. Thus, where people earlier used a huge amount of time analyzing the game manually, Bagadus is an integrated system where the required operations and the synchronization with video is automatically managed. In the online mode, Bagadus receives expert annotated events by the team analytics team and enables immediate playback during a game or a practice session.

3. VIDEO SUBSYSTEM

To be able to record high-resolution video of the entire soccer field, we have installed a camera array using small industry cameras which, together, cover the entire field. The video subsystem then extracts, process, and delivers video events based on given time intervals, player positions, etc. There are two versions of the video subsystem. One non-real-time system and one live real-time system. Both the video subsystems support two different playback modes. The first mode allows the user to play video from the individual cameras by manually selecting a camera or by automatically following players. The second mode plays back a panorama video stitched from the four camera feeds. The non-real-time system plays back recorded video stored on disks, and because of the processing times, it will not be available before the match is finished. The live system, on the other hand, supports playing back video directly from the cameras, and events will be available in real time.

3.1 Camera Setup

To record high-resolution video of the entire soccer field, we have installed a camera array consisting of four Basler industry cameras with a 1/3-inch image sensor supporting 30fps and a resolution of 1280×960. The cameras are synchronized by an external trigger signal in order to enable a video-stitching process that produces a panorama video picture. For a minimal installation, the cameras are mounted close to the middle line under the roof covering the spectator area, that is, approximately 10 meters from the side line and 10 meters above the ground. With a 3.5mm wide-angle lens, each
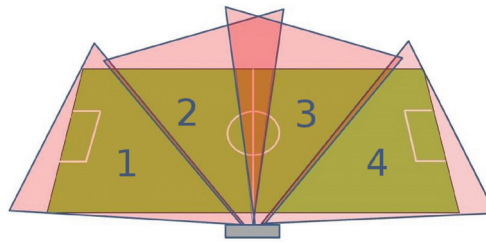
Fig. 2. Camera setup at Alfheim Stadium.

camera covers a field-of-view of about 68 degrees, that is, all four cover the full field with sufficient overlap to identify common features necessary for camera calibration and stitching (see Figure 2).

The cameras are managed using our own library, called Northlight, to manage frame synchronization, storage, encoding, etc. The system is currently running on a single computer with an Intel Core i7-3930K @ 3.2GHz and 16GB memory. Northlight integrates the SDK provided by Basler for the cameras, video encoding using x264, and color-space conversion using FFmpeg.

3.2 Digital Zoom

Bagadus supports digital zooming on tracked players, where the tracked player is kept in the center of the image while zooming in. An important operation here is interpolation, where we use known data to estimate values at unknown points when we resize or remap (i.e., distort) the image. In this respect, we have compared four different interpolation algorithms, that is, nearest neighbor, bilinear, bicubic, and Lanczos interpolation. In image processing, bicubic interpolation is often chosen over bilinear interpolation or nearest neighbor in image resampling when speed is not an issue. Lanczos interpolation has the advantages of bicubic interpolation and is known to produce sharper results than bicubic interpolation. In Bagadus, our initial tests show that the average interpolation times per frame are 4.2ms, 7.4ms, 48.3ms, and 240ms for nearest-neighbor, bilinear, bicubic, and Lanczos interpolation, respectively [Halvorsen et al. 2013]. Due to our time constraints, we use nearest-neighbor interpolation.

3.3 Stitching

Tracking game events over multiple cameras is a nice feature, but in many situations, it may be desirable to have a complete view of the field. In addition to the camera selection functionality, we therefore generate a panorama picture by combining images from multiple trigger-synchronized cameras. The cameras are calibrated in their fixed position using a classical chessboard pattern [Zhang 1999], and the stitching operation requires a more complex processing pipeline. We have alternative implementations with respect to what is stored and processed offline, but in general, we must (1) correct the images for lens distortion in the outer parts of the frame due to a fish-eye lens; (2) rotate and morph the images into the panorama perspective due to different positions covering different areas of the field; (3) correct the image brightness due to light differences; and (4) stitch the video images into a panorama image. Figure 3 shows the process of using four warped camera images into a single large panorama image. The highlighted areas in the figure are the regions where the cameras overlap.

After the initial steps, the overlapping areas between the frames are used to stitch the four videos into a panorama picture before storing it to disk. We first tried the open-source solutions given by computer vision library OpenCV, which are based on the automatic panoramic image stitcher by Brown and Lowe [2007], that is, we used the auto-stitcher functions using planar, cylindrical, and spherical projections. Our analysis shows that neither of the OpenCV implementations are perfect, having large execution times and varying image quality and resolutions [Halvorsen et al. 2013]. The fastest
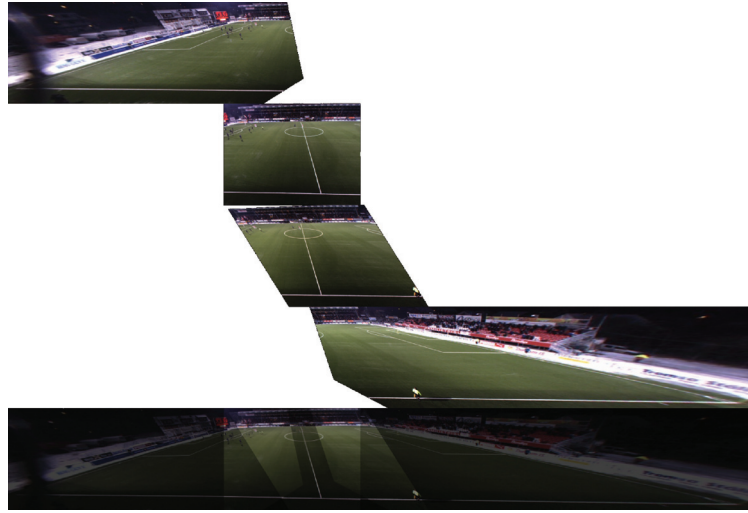
Fig. 3. The stitching process. Each image from the four different frames are warped and combined into a panorama.

(a) Stitching components in a single-threaded loop.

(b) Per-frame processing time (in milliseconds).
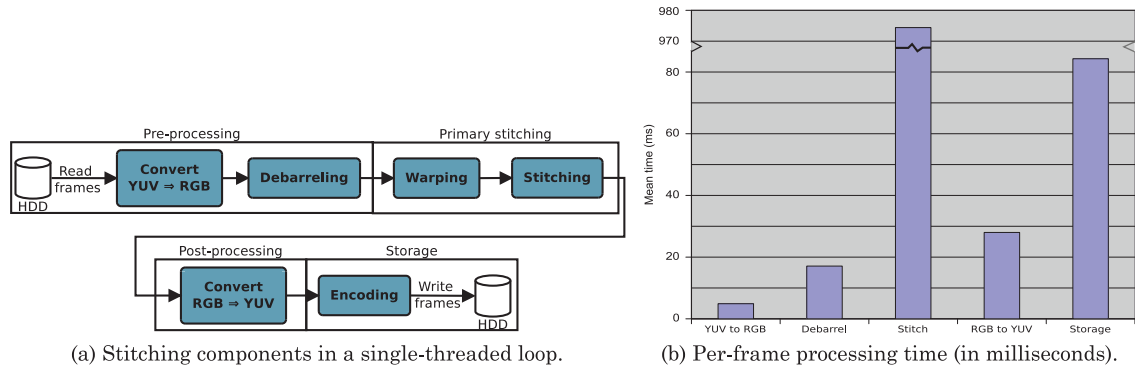
Fig. 4. The Bagadus single-threaded processing loop stitching implementation.

algorithm is the spherical projection, but it has severe barreling effects, and the execution time is 1746ms per frame—far above our real-time goal. Therefore, a different approach called homography stitching [Hartley and Zisserman 2004] has been selected, where we use a homography given by the projective geometry translating ZXY's coordinate system to pixel coordinates.

3.4 Non-Real-Time Processing Loop Implementation

As a first proof-of-concept prototype [Halvorsen et al. 2013], we implemented the stitching operation as a single-threaded sequential processing loop, as shown in Figure 4(a), that is, processing one frame per loop iteration. As seen in the figure, it consists of four main parts. One preprocessing part that reads video frames from either disk or cameras converts the video from YUV to RGB, which is used by the rest of the pipeline and debarreling to remove any barrel distortion from the cameras. For this version of the system, the debarreling functions in OpenCV is used. The next part is the primary stitching part using the homography-based stitching algorithm to stitch the four individual camera frames into a 7000×960 panorama frame. As we can observe from Figure 4(b), this is the most resource-demanding
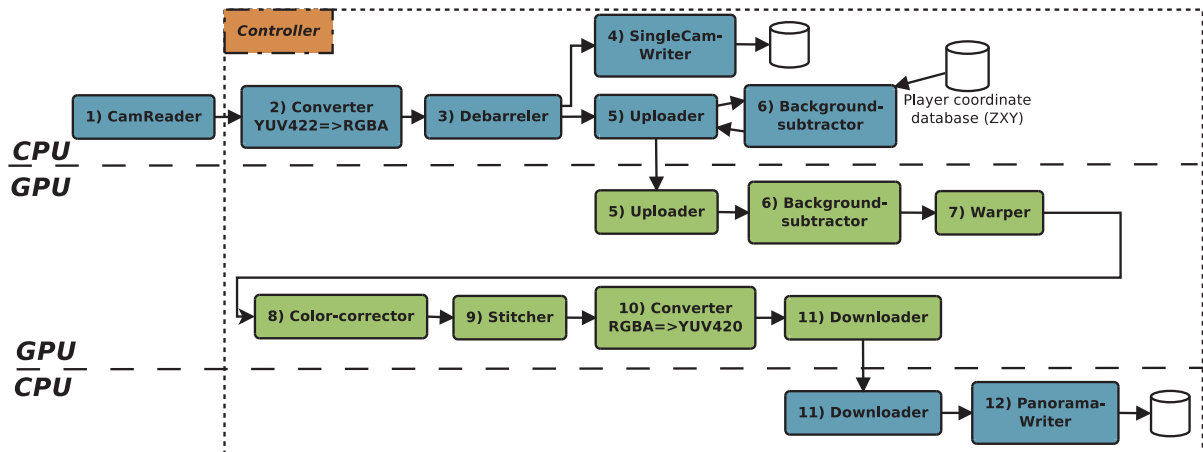
Fig. 5.   The parallel and distributed processing implementation of the stitching pipeline.

part of the system. After the stitching, the postprocessing is responsible for converting the video back from RGB to YUV due to lacking support for RGB in the x264 video encoder. The single-threaded loop means that all the steps are performed sequentially for one set of frames before the next set of frames is processed. The performance is presented in Figure 4(b), and the total execution time per panorama frame exceeds 1100ms on average. In order to meet our 30fps requirement, our next approach improves the performance by parallelizing and distributing the operations in a processing pipeline and offloading several steps onto a GPU.

3.5 Real-Time Parallel and Distributed Processing Implementation

The previous sections displayed some severe processing overheads with respect to generating a 30fps panorama video in real time. In this section, we address this by implementing the modules in a parallel pipeline in contrast to the loop previously described, and we offload compute-intensive parts of the pipeline to a modern GPU, as seen in Figure 5.

3.5.1 *Implementation.* Figure 5 shows that the parallel pipeline is separated into two main parts: one part running on the CPU, and the other part running on a GPU. Several of the CPU modules in the pipeline are the same as in the non-real-time loop. The *CamReader*, *Converter*, *Debarreler*, *SingleCamWriter*, and *PenoramaWriters* are based on the same design, but are now running in their own threads and with an updated version of the x264 encoder. The *controller module* is new and is responsible for initializing the pipeline, synchronizing the different modules, handling global errors and frame drops, and transferring data or data pointers between the different modules. The controller also checks the execution speed. If an earlier step in the pipeline runs too slow, and one or more frames have been lost from the cameras, the controller tells the modules in the pipeline to skip the delayed or dropped frame and reuse the previous frame.

A *background subtractor* module is running both on the CPU and on the GPU. This module is new in the pipeline and is responsible for determining which pixels of a video belong to the foreground and which pixel belong to the background. The background subtractor can also get input from the ZXY sensor system to improve the performance and precision. Even though we have enhanced the background subtraction with sensor data input, there are several implementation alternatives. When determining which algorithm to implement, we evaluated two different alternatives, that is,

those of Zivkovic [2004] and Zivkovic and van der Heijden [2006] and those of KaewTraKulPong and Bowden [2001]. Both algorithms uses a Gaussian mixture model (GMM), are implemented in OpenCV, and have shown promising results in other surveys [Brutzer et al. 2011]. In the end, Zivkovic provided the best accuracy, which is important for our scenario, and it was therefore selected.

There are also several modules that are running primarily on the GPU. The *Uploader* and *Downloader* are managing the dataflow to and from the GPU. The Uploader transfers RGB frames and the background subtraction player pixel maps from the CPU to the GPU for further processing. The Downloader transfers back the stitched video in YUV 4:2:0 format for encoding. Both modules use double-buffering and asynchronous transfers.

The main parts of the panorama creation is performed by the *warper*, *color-corrector*, and *stitcher* modules running on the GPU. The warper module warps (as previously described) the camera frames and the foreground masks from the background subtractor module to fit the common panorama plane. Here, we used the Nvidia Performance Primitives library (NPP) for an optimized implementation. The Color-corrector in this implementation is added to the pipeline because it is nearly impossible to calibrate the cameras to output the exact same colors because of the uncontrolled lighting conditions. This means that, to generate a best-possible panorama video, we correct the colors of all the frames to remove eventual color disparities. This operation is performed after the images are warped. The reason for this is that locating the overlapping regions is easier with aligned images, and the overlap is also needed when stitching the images together. The implementation is based on the algorithm presented in Xiong and Pulli [2009], which has been optimized to run in real-time with CUDA.

The stitcher module is similar to the homography stitcher in the loop implementation, where a seam is created between the overlapping camera frames. Our previous approach uses static cuts for seams, which means that a fixed rectangular area from each frame is copied directly to the output frame. Static cut panoramas are very fast but can introduce graphical errors in the seam area, especially when there is movement in the scene, as illustrated in Figure 6(a). Thus, to make a better visual result, a dynamic cut stitcher is introduced. This module now creates seams by first creating a rectangle of adjustable width over the static seam area. Then, it treats all pixels within the seam area as graph nodes. Each of these edges' weights are calculated using a custom function that compares the absolute color difference between the corresponding pixel in each of the two frames we are trying to stitch. The weight function also checks the foreground masks from the background subtractor to see if any player is in the pixel, and if so, it adds a large weight to the node. We then run a simplified version of the Dijkstra graph algorithm (only going up in the image) on the graph to create a minimal cost route from the bottom of the image to the end at the top. An illustration of how the final seam looks can be seen in Figure 6(b), while the seams without and with color correction are shown in Figures 6(c) and 6(d).

3.5.2 *Execution Time Evaluation.* To evaluate the processing performance of the parallel and distributed processing pipeline implementation, we used a single computer with an Intel Server Adapter i350-T4 for connecting the four cameras with gigabit ethernet, an Intel Core i7-3930K six-core processor with 32GB RAM, and a single Nvidia GeForce GTX Titan graphics processor.

The overall performance of the parallel pipeline is shown in Figure 7(a). The CPU modules are marked in blue, and the GPU modules are marked in green. The uploader and downloader module run both on the CPU and the GPU, but we have chosen to mark them as CPU modules, since they both are controlled by the CPU.

Images from all four cameras are asynchronously transfered to the GPU as soon as they are available. The number of threads and blocks on the GPU is automatically adjusted by how many cores are

(a) The original fixed-cut stitch with a straight vertical seam.

(b) The new dynamic stitch with color correction.

(c) Dynamic stitch with no color correction. In the left image, one can see the seam search area between the red lines, and the seam in yellow. In the right image, one clearly sees the seam going outside the player, but there are still color differences.

(d) Dynamic stitch with color correction. In the left image, one can see the seam search area between the red lines and the seam in yellow. In the right image, one cannot see the seam, and there are no color differences.

Fig. 6.   Stitcher comparison: improving the visual quality with dynamic seams and color correction.

available on the GPU. The modules executing on the GPU synchronize with barriers: when one module finishes, the next will be stared. Data is stored in global memory, and pointers to the data are transfered between the different modules. When processing is finished on the GPU, data is asynchronously transfered back to the CPU for encoding and writing to disk.

We can see that when executing the whole pipeline, all modules perform well below the real-time threshold. Note that the reader module is limited by the cameras which produce a new frame every 33ms. Remember that all these modules run in parallel, sharing the processing elements. Thus, since all modules perform better than the 33ms threshold, we are able to deliver panorama frames in real time. This is further demonstrated by measuring the differences between the single camera writes and the differences between the panorama writes. In Figure 7(b), we present the write differences between the frames, and we observe that a new frame is output every 33ms, that is, equal to the input rate of

233
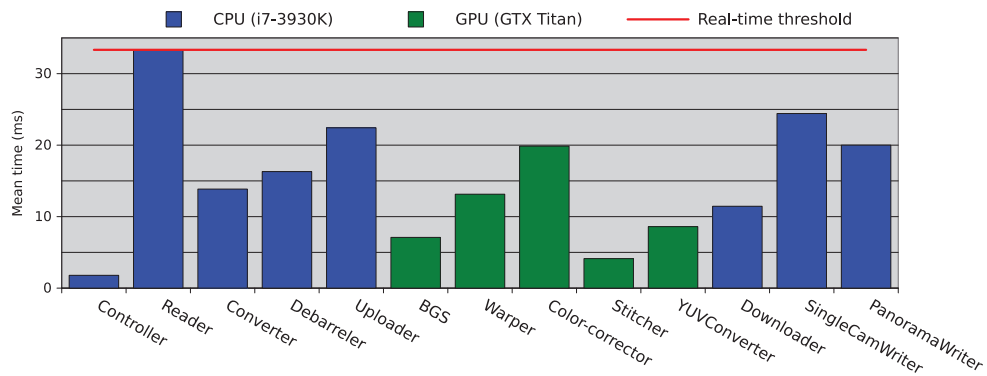
(a) The processing performance of the different pipeline modules.
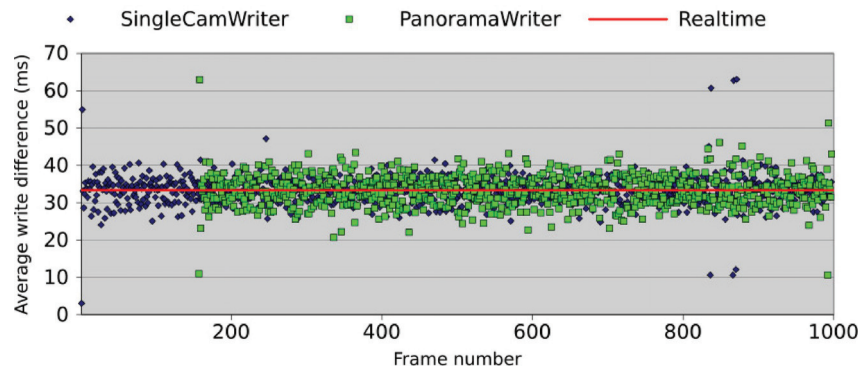
(b) Pipeline write differences (showing times for 1,000 frames). Note that the delayed start of panorama writes is caused by the frame delay buffer implemented in the uploader module.

Fig. 7. The processing performance of the parallel and distributed processing pipeline.

the cameras. These results show that our parallel and distributed processing implementation executes in real time on a single off-the-shelf computer.

4. TRACKING SUBSYSTEM

Tracking people through camera arrays has been an active research topic for several years, and many approaches have been suggested (e.g., [Ben Shitrit et al. 2011; Berclaz et al. 2011; Jiang et al. 2007; Xu et al. 2004]). The accuracy of such tracking solutions vary according to scenarios and is continuously improving, but they are still giving errors, that is, both missed detections and false positives [Ben Shitrit et al. 2011]. Often these approaches perform well in controlled lighting conditions, like indoor sport arenas, but the widely varying light conditions in an outdoor stadium provide bigger challenges.

For stadium sports, an interesting approach is to use sensors on players to capture the exact position. ZXY Sport Tracking [ZXY 2013] provides such a solution, where a sensor system submits position and orientation information at a maximum accuracy error of about one meter at a frequency of 20Hz. As indicated in Figure 1, the players wear a data chip with sensors that sends signals to antennas located around the perimeter of the pitch. The sensor data is then stored in a relational database system. Based

ACM Transactions on Multimedia Computing, Communications and Applications, Vol. 10, No. 1s, Article 14, Publication date: January 2014.

234

on these sensor data, statistics like total length run, number of sprints of a given speed, foot frequency, heart rate, etc., can be queried in addition to the exact position of all players at all times. Due to the availability of the ZXY system at our case study stadium, Bagadus uses the sensor system position information to extract videos of, for example, particular players, and the rest of the system can be used to extract time intervals of the video (e.g., all time intervals where player X sprints towards his own goal).

The ZXY sensor belt is worn by all the players on TIL (the home team); it is voluntary for the visiting team to use the sensor belts. If they choose to use the belts, they will have access to the data recorded during the match. The belts are small and compact and do not disturb the players during the match; they are also approved by FIFA for use during international matches.

Although the amount of data generated by the position sensors is small compared to video, a game of 90 minutes still produces approximately 2.4 million records. Nevertheless, as we show later in Section 6, we still have reasonable response times from when we send a complex database query until the video starts to play the corresponding query result events.

4.1 Mapping Sensor Positions to Image Pixels

The ZXY system reports the players' positions on the field using the Cartesian coordinate system. In order to locate a player in the video, we need a transformation from the sensor coordinates to the image pixels for all valid pixel coordinates in a video frame. In this respect, we calculate a 3×3 transformation matrix using fixed known points on the field, as shown in Figure 8(a). Then, using the homography between two planes, each plane can be warped to fit the other, as shown in Figures 8(c) and 8(d), using camera 2 as an example. The accuracy of the mapping is fairly good, that is, only in the outer areas of the image where debarreling have changed some pixels can we see a very small deviation between the planes. However, if we look at the mapping to the stitched image in Figure 8(b), the accuracy is reduced due to imperfections in the image processing when debarreling and, in particular, when warping and rotating. Nevertheless, at the distance between the cameras and the players, the accuracy seems to be good enough for our purposes (though inaccuracies in the mapping might also contribute to inaccurate tracking, as shown later).

In order to have a system where the players are tracked in real time, the $ZXY(x, y) \rightarrow pixel(u, v)$ mapping using the 3×3 matrix must be fast. A profile of the system when tracking all 22 soccer players indicates that about 7.2–7.7 microseconds are consumed for this operation, that is, coordinate translation is hardly noticeable compared to the other components in the system.

4.2 Automatic Camera Selection

As shown in Figure 2, the four cameras cover different parts of the field. To follow a player (or group of players) and be able to automatically generate a video selecting images across multiple cameras, we also need to map player positions to the view of the cameras. In this respect, we use the same mapping as described in Section 4.1, using our own transformation matrix for each camera. Selecting a camera is then only a matter of checking if the position of the player is within the boundaries of the image pixels. When tracking multiple players, we use the same routine and count the number of tracked players present in each camera and select the camera with the most tracked players.

5. ANALYTICS SUBSYSTEM

To improve a team's performance and understand their opponents, coaches analyze the game play in various ways. Traditionally, this has been done by making notes using pen and paper, either during the game or by watching hours of video. To reduce the manual labor, we have, in close colaboration with the coach-team, developed Muithu, a novel notational analysis system [Johansen et al. 2012] that

235
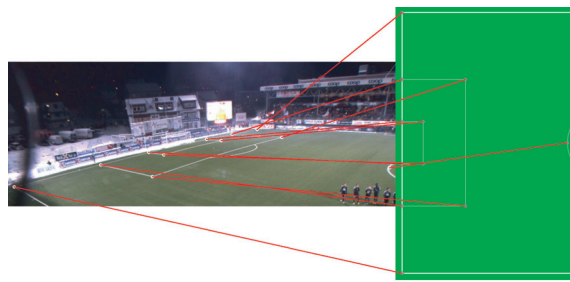
(a) Mapping between coordinates in the ZXY plane and the image plane.

(b) Warping and superimposing the ZXY plane onto the stitched image (cropped out only parts of the field for readability).

(c) Warping and superimposing the image from camera 2 to the ZXY plane.

(d) Warping and superimposing the ZXY plane onto the image from camera 2.

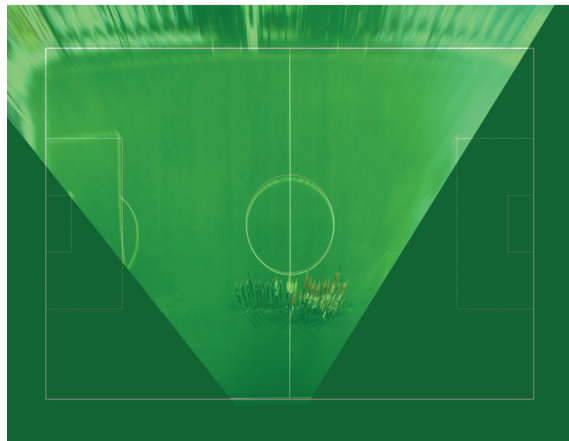Fig. 8.   Pixel mapping between the video images and the ZXY tracking system.

is non-invasive for the users, mobile, and lightweight. A cellular phone is used by head coaches during practice or games for annotating important performance events. A coach usually carries a cellular, even during practice. Thus, to avoid any extra coach devices, the cellular is used in the notational process as a notational device. Input is given using the tile-based interface shown in Figures 9(b) and 9(c), and Figure 9(a) illustrates use of the system by a coach during a recent game in the Norwegian elite division. Our experience indicates that this simple drag-and-drop user interaction requires in the order of 3 seconds per notational input. All the events in the app can be custmized by the coaches, and the number of input notations for a regular 90-minute elite soccer game varies slightly over different games, but for the 2012 season, the average is in the order of 16 events per game [Johansen et al. 2012].

In order to be usable during a game, the user interface of Muithu has to be easy to use and fast. It is therefore based on managing tiles in a drag-and-drop fashion, and it can be easily configured with input tiles and hierarchies of tiles. In the case study described in Section 6, one preferred configuration pattern for general practice is to have a two-layer hierarchy, where the root node is a number or all of the players involved. The next layer is a set of 3–4 training goals associated with each individual player. By simply touching the picture of a player on a tile, his specific training goals appear on adjacent

(a) Event tagging.   (b) Select player.   (c) Drag to event.
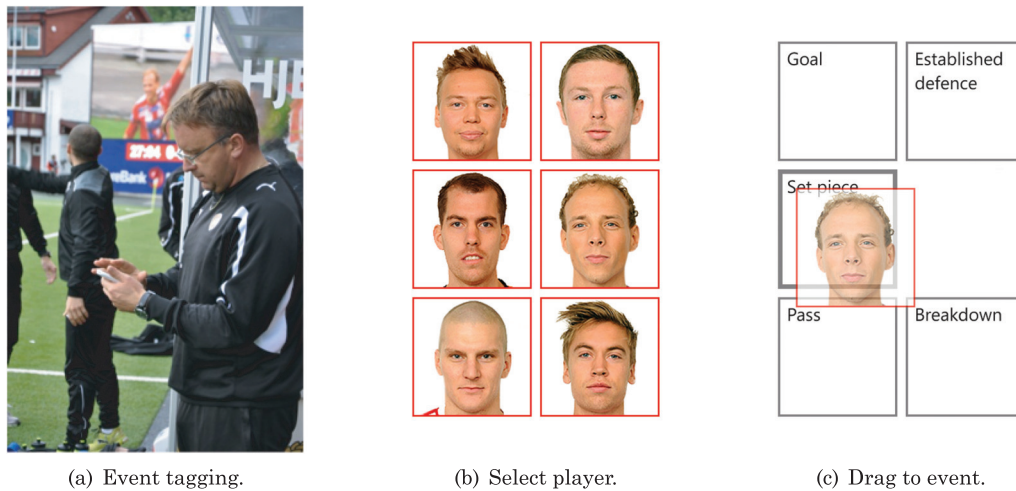
Fig. 9.   Operation of the mobile device during a game (a). Select a player (b) and drag the image tile to the appropriate event type (c) to register an event.
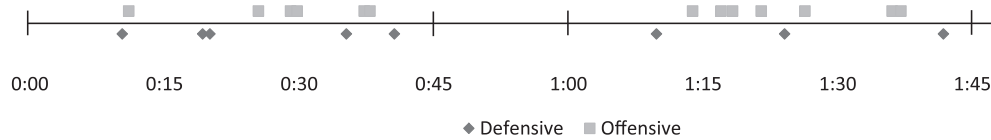
Fig. 10.   An example of notations captured during a game (time axis in HH:MM after game start). Observe that offensive notations are displayed above the timeline, defensive notations below.

tiles. Dragging the face tile over one of these goal tiles is then sufficient for capturing the intended notation.

For heated game purposes, a simpler configuration is preferred: typically one tile for offensive and one for defensive notations (see Figure 9(c)). Using this interface as an example, Figure 10 depicts the distribution of such notations during a home game in September 2012.

Recall of performance-related events without any observation aids is traditionally problematic in soccer, but the recall abilities of the head coaches using Muithu have improved rapidly approaching almost 1 (100%). A small but fundamental detail is the use of *hindsight* recording, which implies that the coach observes an entire situation and determines afterwards whether it was a notable event worth capturing. By tagging in retrospect, the coach essentially marks the end of a notable event, and the system finds the start of the sequence by a preconfigured interval length. This simple yet not so intuitive approach has reduced the number of false positives, that is, increased precision dramatically.

Only those events tagged by the head coaches are retrieved for movement patterns, strategy, and tactics evaluation. The key to this process is that the video footage is automatically retrieved from the video system when the event is selected in the video playout interface. This scales both technically and operationally, which enables expedited retrieval. The video sequence interval according to the recorded event time-stamp is a configuration option easy to change, but operational practice has shown that an interval around 15 seconds is appropriate for capturing the event on video. It is also possible to adjust this interval, both when the event is created and during playback.

Fig. 11.   The offline Linux interface (tracking three players in camera-switching mode).

```
SELECT timestamp, x_pos, y_pos
FROM zxy_oversample
WHERE (y_pos > 17.5 AND y_pos < 50.5)
   AND (x_pos >  0.0 AND x_pos < 16.5)
   AND timestamp > 45
   AND tag_id = ("the tag_id of player X")
```

Fig. 12.   Example query.

6.   ALFHEIM STADIUM CASE STUDY

We have a prototype installation at Alfheim Stadium in Tromsø (Norway). The interface of the offline prototype [Halvorsen et al. 2013][1] is shown in Figure 11, where we can follow and zoom in on particular player(s) and play back expert-annotated events from the game in panorama video- and camera-switching mode.

In the offline mode, the system has support for generating automatic summaries, that is, selecting multiple time intervals and playing it out as one video (not yet integrated into the user interface). This means that the game analytics, for example, may perform queries against the ZXY database and get the corresponding video events. An example could be to see "all the events where defender X is in the other team's 18-yard box in the second half". In this example, the position and corresponding time of player X in the former example is returned by the pseudo-query shown in Figure 12. Here, the player is located within the [0.0, 16.5] in the x-coordinate and [17.5, 50.5] on the y-axis (using the metric system) defining the 18-yard box. The returned time stamps and positions are then used to select video frames (selecting the correct camera or the panorama picture) which are automatically presented to the user. Extracting summaries like the preceding example used to be a time-consuming

---

[1]A video of the (offline) Linux-based system is available at http://www.youtube.com/watch?v=1zsgvjQkL1E. At the time of the submission, we have not been able to make a video of the online system.

Table I. Latency Profiling (in ms) of the Event Extraction Operation Using ZXY and the Video System

Operation	Mean	Minimum	Maximum	Standard deviation
Query received	2.7	1.5	5.3	0.38
Query compiled	4.9	2.9	7.8	0.61
First DB row returned	500.4	482.4	532.1	5.91
First video frame displayed	671.2	648.0	794.6	8.82
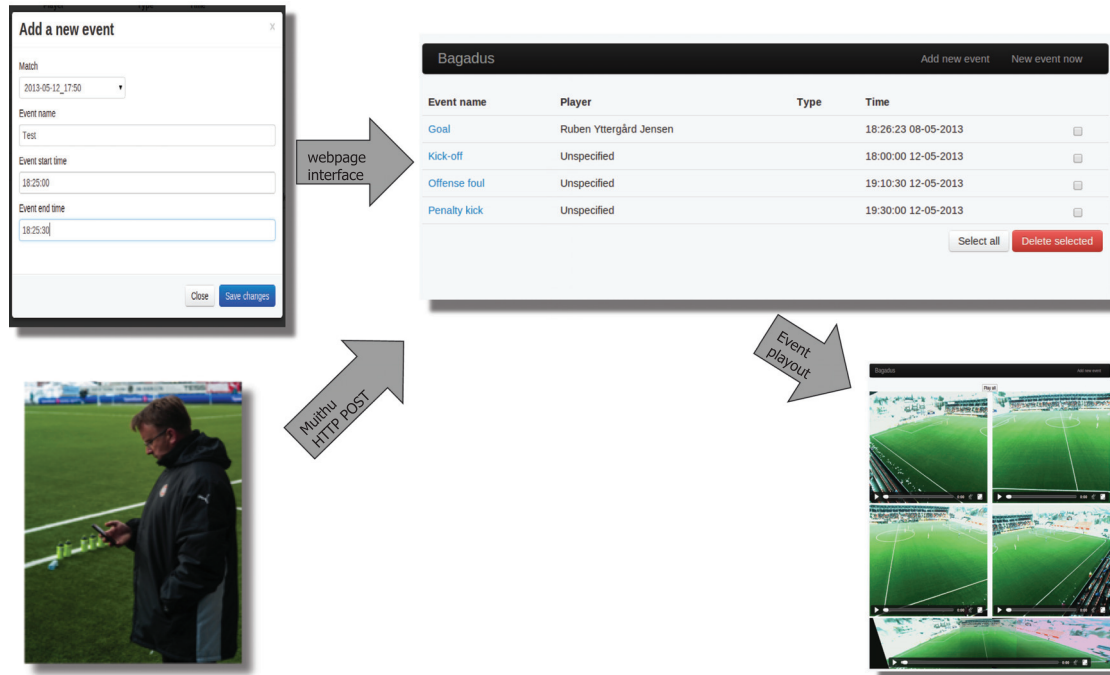
Fig. 13. The online HTML5 interface used for expert annotated events. Here the events are sorted by player and then time.

and cumbersome (manual) process. Bagadus, on the other hand, automates the video generation. For instance, the response time of returning the resulting video summary from the preceding query was measured to be around 671ms (see Table I for more detailed statistics). Note that this was measured on a local machine, that is, if the display device is remote, network latency must be added. The SQL queries are made for expert users. We have also implemented a number of predefined queries that are availible in the user interface.

As shown in the online mode HTML5 interface in Figure 13, we can in a similar way extract video events based on expert annotations. Events may be tagged through a Web interface or using the mobile phone sending an HTTP POST command, and all annotated events from the analytics subsystem then appear in the list of events. Using a standard Web browser, the corresponding videos start by clicking on the event title. Thus, the integration of subsystems enable event playout during a game or a practice session.

(a) Bagadus.   (b) Broadcaster 1.   (c) Broadcaster 2.
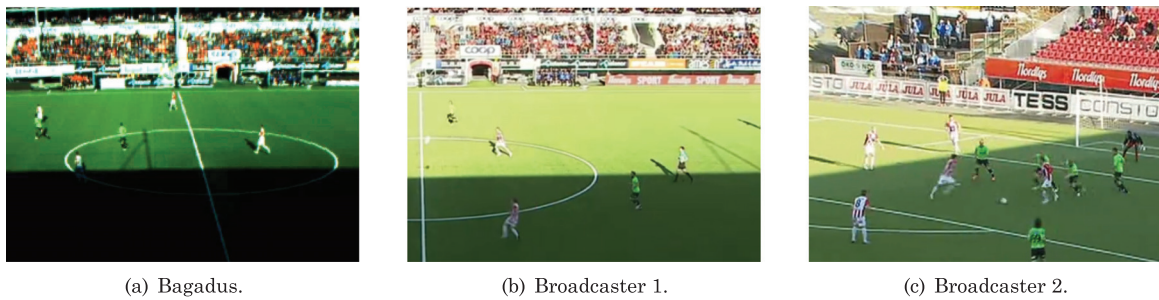
Fig. 14.   Lighting challenges at Alfheim Stadium. Comparison between Bagadus and two professional Norwegian broadcasters. (The images are from the same game but different situations during the game.)

7. DISCUSSION

Performance analysis of athletes in the sports industry is a growing field of interest. In the context of computer systems managing the technical challenges, there are numerous issues that must be addressed to provide real-time operations. In this respect, our Bagadus soccer analysis application integrates a sensor system, soccer analytics annotations, and video processing of a video camera array. There exist several components that can be used, and we have investigated several alternatives in our research. Furthermore, by providing a parallel video-processing pipeline distributing load on multiple CPUs and GPUs, Bagadus supports analysis operations at 30fps. Note, however, that our prototype aims to prove the possible integration at the system level with real-time performance, rather than being optimized for optimal resource utilization, that is, there are several areas with potential for further optimizations.

For example, most stitching processes assume the pinhole camera model where there is no image distortion because of lenses. In our work, we have observed that a camera can be calibrated to minimize lens distortion caused by imperfections in a lens but making a perfect calibration is hard. This makes finding a homography between planes difficult and error-prone, which affects the stitched result.

Another problem we have identified is parallax errors. In this respect, OpenCV's auto-stitcher has functionality for selecting seams at places where parallax errors are less obvious. However, when stitching video recorded from cameras capturing the field from the same position but with different angles (requiring rotation and warping), parallax errors will become prominent. Such problems arise because the centers of the projection of different cameras are not aligned well enough. We are looking at solutions to eliminate this problem: one of the most interesting solutions is the arrangement of cameras over cross, such as each camera capturing one side of the field, similar to Fehn et al. [2006].

Furthermore, the stitching itself can be moved from a homography-based stitching with dynamic seams to avoid moving objects to more advanced warping techniques, like the one mentioned in Lin et al. [2011]. A rather intriguing challenge would be to incorporate such a process into Bagadus and perform this approach in real time, too. Moreover, we have later found several promising alternative algorithms in the area of video processing (vision) (e.g., [Lin et al. 2011; Jin 2008; Li and Du 2010; Ozawa et al. 2012]), and there is also scope for further improvement in color correction [Xiong and Pulli 2010], since the exposure times and other parameters across the cameras may vary.

A major challenge is managing variations in lighting conditions. In most weather conditions, our current setup works fine, but our main challenge here is a low and bright sun. The visual quality is exceptional when it is partly or completely cloudy, but the striking difference between the amount of light available from highlights and shadows during a clear day leaves us with a choice of having a good dynamic range in only one region. An example from Alfheim Stadium is shown in Figure 14. When

240

there are intensely bright and dark areas in the image (Figure 14(a)), most cameras have problems creating a representative image. Particularly, in our Alfheim case study, the location of the stadium is only 2,271km from the North Pole (69.6489986°N). The sun is significant lower on the sky than most of the habitable world, resulting in challenges, as shown in the figure. In such a case, aiming for good quality in highlights leads to loss of details in shadows. Our system currently lacks the ability to make an appropriate decision which often depends on the events on the field. Professional broadcasters also experience these problems, but they have people manning cameras (and thus also the exposure settings) as well as a someone controlling the live broadcast who also can perform manual adjustments (Figures 14(c) and 14(b)).

Our system needs to handle this without human interaction and in real time. The problem is related to suboptimal auto-exposure and insufficient dynamic range on the camera sensors. Improvements can be achieved several ways. In this respect, one could solve common auto-exposure problems as proposed in Kao et al. [2011] and use real-time assembling of high-dynamic-range (HDR) video by using low-dynamic-range images [Ali and Mann 2012; Guthier et al. 2012]. Investigations of such approaches are currently ongoing.

The GPU implementation has been tested on an Nvidia GeForce Titan (GK110) GPU with compute 3.5 capabilities and has been profiled with Nvidia's Visual Profiler to investigate the possibilities of scaling the pipeline to more cameras with higher resolution. Currently, we are only using a small portion of the available PCI Express bandwidth between the CPU and the GPU. Our uploader uses 737MB/sec, and our downloader uses 291MB/sec. The theoretical bidirectional bandwidth of a 16-lane PCI Express 3.0 link is 16GB/sec. The real-time pipeline uses seven kernels running concurrently on the GPU. These seven kernels have an average compute utilization of 14.8% on this GPU. The individual CUDA kernels are also not optimized for the architecture used in our benchmarks, since the priority was to get the entire pipeline in real time. There is therefore a lot of potential on the GPU for scaling the pipeline to a larger number of cameras with higher resolution.

In our case study, we have analyzed data and retrieving video from only one game. However, we have shown earlier how one could search for events and generate video summaries on-the-fly in terms of a video playlist [Johansen et al. 2009] over large libraries of video content. In the used test scenario, there are events identified from multiple subcomponents, for example, the sensor system and the annotation system. In many cases, it would be valuable to be able to search across all the metadata and also across games. This is a feature we are currently adding, that is, the underlying video system fully supporting the video extraction, but the interface has not yet been implemented.

The design of Bagadus having three tightly integrated, but still separate subsystems, enables easy subsystem replacement. For example, we have used ZXY to track players, providing some extra nice features (heart rate, impact, etc.). However, tracking players (or, generally, objects) through video analysis is a popular research area (e.g., both in sports [Fehn et al. 2006; Yongduek et al. 1997; Iwase and Saito 2004; Kang et al. 2003] and surveillance [Fuentes and Velastin 2006; Chen et al. 2011; Siebel and Maybank 2002]). Thus, the Bagadus idea should easily be transferable to arenas where the sensor system is unavailable or to other arena sports, like ice hockey, handball, baseball, tennis, American football, rugby, etc. Similarly, video-processing components can easily be replaced to match other codec's and other filters or to suit other end devices and platforms. Equally, the annotation system can be replaced (or expanded) to retrieve metadata of events from other sources, like on-the-fly live text commentaries found in newspapers and online TV stations, like we did in our DAVVI system [Johansen et al. 2009].

One engineering challenge in systems like Bagadus is time synchronization at several levels. First, to be able to stitch several images to a panorama image, the shutters must be synchronized at the sub-millisecond level, that is, as the players are moving fast across cameras, imperfect synchronization

Fig. 15. An example of when the tracking box fails to capture the tracked player. Even though our analysis of the system indicates very infrequent errors, it may be various reasons for failed tracking, for example, both clock skew, sensor system accuracy, and coordinate mapping.

would lead to massive pixel offsets across camera perspectives resulting in severely blurred composite images of players. This is currently solved using an external trigger box (i.e., embedded trigger controller based on an ATMega16 microcontroller) which sends an input signal to the camera's electronic shutter. Another observed challenge in this respect is that the clock in the trigger box drifts slightly compared to our computer clocks depending on temperature (which changes a lot under the harsh outdoor conditions in northern Norway). While the shutters across cameras remains in sync, a drifting clock leads to slight variations in frame rate of the captured video. Similarly, Bagadus integrates several subsystems running on different systems. In this respect, the clock in the ZXY system also slightly drifts compared to the clock in our video capture machines (which will be potentially solved when we switch ZXY to the same NTP server). So far, these small errors have been identified, but since we alleviate the problem in our video player by fetching a couple of seconds more video data around a requested event time stamp, the effects have been small. Another more visible (still very infrequent) effect of time skew is that the box-marker marking the players in the video gives small misplacement errors, as shown in Figure 15. However, the bounding box is slightly larger compared to the personobject itself. This means that the player is usually contained in the box, even though not exactly in the middle. At the current stage of our prototype, we have not solved all the synchronization aspects, but it is subject to ongoing work.

The ZXY's tracking system installed at Alfheim Stadium has a maximum accuracy error of one meter (their new system reduces this error down to a maximum of 10 centimeters). This means that if a player is at a given position, the measured coordinate on the field could be ± one meter. This could give effects like those shown in Figure 15, but for the practical purposes of our case study, it has no influence on the results.

The players are tracked as described using the ZXY Sport Tracking system. Another issue which is not yet included in Bagadus is ball tracking, that is, a feature that could potentially improve the analysis further. Even though ball tracking is not officially approved by the international soccer associations due to the limited reliability and failure to provide 100% accuracy, there exist several approaches. For example, Adidas and Cairos Technologies have tried to put sensors inside the ball, that is, using a magnetic field to provide pinpoint accuracy of the ball's location inside the field [McKeegan 2007; Cairos Technologies 2013a]. Other approaches include using multiple cameras to track the ball.

Hawk-Eye [2013] is one example which tries to visually track the trajectory of the ball and display a record of its most statistically likely path as a moving image. Nevertheless, ball tracking in Bagadus is a future feature.

This article presents Bagadus in the context of sports analysis for a limited user group within a team. However, the applicability we conjecture is outside the trainer and athlete sphere, since we have a potential platform for next-generation personalized edutainment. We consider use case scenarios where users can subscribe to specific players, events, and physical proximities in real time. For instance, when the main activity is around the opponent goal, a specific target player can be zoomed into. Combine this with commonplace social networking services, and we might have a compelling next-generation social networking experience in real time.

8. CONCLUSIONS

We have presented a real-time prototype of a sports analysis system called Bagadus targeting automatic processing and retrieval of events in a sports arena. Using soccer as a case study, we described how Bagadus integrates a sensor system, a soccer analytics annotations system, and a camera array video processing system. Then, we showed how the system removes the large amount of manual labor traditionally required by such systems. We have described the different subsystems and the possible trade-offs in order to run the system in real-time mode. Compared to our initial demonstrator [Halvorsen et al. 2013], the improved processing pipeline parallelizing the operational steps and distributing workload to both CPUs and GPUs enables real-time operations, and the picture quality has been improved using dynamic seams and color correction. Furthermore, we have presented functional results using a prototype installation at Alfheim Stadium in Norway. Bagadus enable a user to follow and zoom in on particular player(s), playback events from the games using the stitched panorama video and/or the camera switching mode, and create video summaries based on queries to the sensor system.

Finally, there are still several areas for future improvements, for example, in the areas of image quality improvements handling a wide range of lighting conditions, performance enhancements as our profiling results show that we can optimize the resource utilization further and subjective user evaluations. All these areas are subjects for ongoing work, for example, we are testing algorithms discussed in Section 7 for improving the image quality, we are evaluating higher-resolution cameras like the 2K Basler aca2000-50gc, and we are further optimizing and distributing algorithms onto multiple cores and offloading calculations to GPUs for speed improvements and better utilization of both cores and buses.

REFERENCES

Mir Adnan Ali and Steve Mann. 2012. Comparametric image compositing: Computationally efficient high dynamic range imaging. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 913–916.

Horesh Ben Shitrit, Jerome Berclaz, Francois Fleuret, and Pascal Fua. 2011. Tracking multiple people under global appearance constraints. In *Proceedings of the IEEE International Conference on Computer Vision (CCV)*. 137–144.

Jerome Berclaz, Francois Fleuret, Engin Turetken, and Pascal Fua. 2011. Multiple object tracking using k-shortest paths optimization. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 9, 1806–1819.

Matthew Brown and David G. Lowe. 2007. Automatic panoramic image stitching using invariant features. *Int. J. Comput. Vision* 74, 1, 59–73.

S. Brutzer, B. Hoferlin, and G. Heidemann. 2011. Evaluation of background subtraction techniques for video surveillance. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1937–1944.

Cairos Technologies. 2013a. Goal Line Technology (GLT) system. http://www.cairos.com/unternehmen/gltsystem.php.

Cairos Technologies. 2013b. VIS.TRACK. http://www.cairos.com/unternehmen/vistrack.php.

Camargus. 2013. Premium Stadium Video Technology Inrastructure. http://www.camargus.com/.

Chao-Ho Chen, Tsong-Yi Chen, Je-Ching Lin, and Da-Jinn Wang. 2011. People tracking in the multi-camera surveillance system. In *Proceedings of the 2nd International Conference on Innovations in Bio-inspired Computing and Applications (IBICA)*. 1–4.

Peter Dizikes. 2013. Sports analytics: A real game-changer. http://web.mit.edu/newsoffice/2013/sloan-sports-analytics-conference-2013-0304.html.

Christoph Fehn, Christian Weissig, Ingo Feldmann, Markus Muller, Peter Eisert, Peter Kauff, and Hans Bloss. 2006. Creation of high-resolution video panoramas of sport events. In *Proceedings of the 8th IEEE International Symposium on Multimedia (ISM)*. 291–298.

Luis M. Fuentes and Sergio A. Velastin. 2006. People tracking in surveillance applications. *Image Vision Comput.* 24, 11, 1165–1171.

Benjamin Guthier, Stephan Kopf, and Wolfgang Effelsberg. 2012. Optimal shutter speed sequences for real-time HDR video. In *Proceedings of the IEEE International Conference on Image Systems and Techniques (IST)*. 303–308.

Pål Halvorsen, Simen Sægrov, Asgeir Mortensen, David K. C. Kristensen, Alexander Eichhorn, Magnus Stenhaug, Stian Dahl, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, and Dag Johansen. 2013. Bagadus: An integrated system for arena sports analytics a soccer case study. In *Proceedings of the 4th ACM Multimedia Systems Conference (MMSys)*. 48–59.

R. I. Hartley and A. Zisserman. 2004. *Multiple View Geometry in Computer Vision* 2nd Ed. Cambridge University Press.

Hawk-Eye. 2013. Football::Hawk-Eye. http://www.hawkeyeinnovations.co.uk/page/sports-officiating/football.

Interplay Sports. 2013. The ultimate video analysis and scouting software. http://www.interplay-sports.com/.

Sachiko Iwase and Hideo Saito. 2004. Parallel tracking of all soccer players by integrating detected positions in multiple view images. In *Proceedings of the 7th International Conference on Pattern Recognition (ICPR)*. 751–754.

Hao Jiang, Sidney Fels, and James J. Little. 2007. A linear programming approach for multiple object tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Hailin Jin. 2008. A three-point minimal solution for panoramic stitching with lens distortion. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1–8.

Dag Johansen, Håvard Johansen, Tjalve Aarflot, Joseph Hurley, Åge Kvalnes, Cathal Gurrin, Sorin Sav, Bjørn Olstad, Erik Aaberg, Tore Endestad, Haakon Riiser, Carsten Griwodz, and Pål Halvorsen. 2009. DAVVI: A prototype for the next generation multimedia entertainment platform. In *Proceedings of the 17th ACM International Conference on Multimedia (MM)*. 989–990.

Dag Johansen, Magnus Stenhaug, Roger Bruun Asp Hansen, Agnar Christensen, and Per-Mathias Høgmo. 2012. Muithu: Smaller footprint, potentially larger imprint. In *Proceedings of the 7th International Conference on Digital Information Management (ICDIM)*. 205–214.

P. Kaewtrakulpong and R. Bowden. 2001. An improved adaptive background mixture model for realtime tracking with shadow detection. In *Proceedings of the Video-Based Surveillance Systems*. 135–144.

Jinman Kang, Isaac Cohen, and Gerard Medioni. 2003. Soccer player tracking across uncalibrated camera streams. In *Proceedings of the Joint IEEE International Workshop on Visual Surveillance and Performance Evaluation of Tracking and Surveillance (VS-PETS)*. 172–179.

Wen-Chung Kao, Li-Wei Cheng, Chen-Yu Chien, and Wen-Kuo Lin. 2011. Robust brightness measurement and exposure control in real-time video recording. *IEEE Trans. Instrument. Measur.* 60, 4, 1206–1216.

Jubiao Li and Junping Du. 2010. Study on panoramic image stitching algorithm. In *Proceedings of the 2nd Pacific-Asia Conference on Circuits, Communications and Systems (PACCS)*. 417–420.

Wen-Yan Lin, Siying Liu, Y. Matsushita, Tian-Tsong Ng, and Loong-Fah Cheong. 2011. Smoothly varying affine stitching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 345–352.

Noel McKeegan. 2007. The Adidas intelligent football. http://www.gizmag.com/adidas-intelligent-football/8512/.

Tomohiro Ozawa, Kris M. Kitani, and Hideki Koike. 2012. Human-centric panoramic imaging stitching. In *Proceedings of the Augmented Human International Conferences Series (AH)*. 20:1–20:6.

ACM Transactions on Multimedia Computing, Communications and Applications, Vol. 10, No. 1s, Article 14, Publication date: January 2014.

244

Prozone. 2013. Prozone Sports – Introducing Prozone performance analysis products. http://www.prozonesports.com/products.html.

Simen Sægrov, Alexander Eichhorn, Jørgen Emerslund, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. 2012. Bagadus: An integrated system for soccer analysis (demo). In *Proceedings of the 6th International Conference on Distributed Smart Cameras (ICDSC)*.

Valter Di Salvo, Adam Collins, Barry McNeill, and Marco Cardinale. 2006. Validation of Prozone: A new video-based performance analysis system. *Int. J. Perform. Anal. Sport* 6, 1, 108–119.

Nils T. Siebel and Stephen J. Maybank. 2002. Fusion of multiple tracking algorithms for robust people tracking. In *Proceedings of the 7th European Conference on Computer Vision (ECCV)*. Lecture Notes in Computer Science, vol. 2353, Springer-Verlag, Berlin Heidelberg, 373–387.

Stats. 2013. STATS—SportVU—Football/Soccer. http://www.sportvu.com/football.asp.

Yingen Xiong and Kari Pulli. 2009. Color correction for mobile panorama imaging. In *Proceedings of the 1st International Conference on Internet Multimedia Computing and Service (ICIMCS)*. 219–226.

Yingen Xiong and Kari Pulli. 2010. Fast panorama stitching for high-quality panoramic images on mobile phones. *IEEE Trans. Consumer Electron.* 56, 2.

Ming Xu, James Orwell, and Graetne Jones. 2004. Tracking football players with multiple cameras. In *Proceedings of the International Conference on Image Processing (ICIP)*. 2909–2912.

Sunghoon Choi Yongduek, Sunghoon Choi, Yongduek Seo, Hyunwoo Kim, and Ki sang Hong. 1997. Where are the ball and players? Soccer game analysis with color-based tracking and image mosaick. In *Proceedings of the 9th International Conference on Image Analysis and Processing (ICIAP)*. Lecture Notes in Computer Science, vol. 1311, Springer-Varlag, Berlin Heidelberg, 196–203.

Zhengyou Zhang. 1999. Flexible camera calibration by viewing a plane from unknown orientations. In *Proceedings of the 7th IEEE International Conference on Computer Vision (ICCV)*. 666–673.

Z. Zivkovic. 2004. Improved adaptive gaussian mixture model for background subtraction. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR)*. 28–31. Vol. 2.

Zoran Zivkovic and Ferdinand van der Heijden. 2006. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recog. Lett.* 27, 7, 773–780.

ZXY. 2013. ZXY Sport Tracking. http://www.zxy.no/.

Paper IX: Processing Panorama Video in Real-Time

World Scientific
www.worldscientific.com

Processing Panorama Video in Real-time

Håkon Kvale Stensland*, Vamsidhar Reddy Gaddam,
Marius Tennøe, Espen Helgedagsrud, Mikkel Næss,
Henrik Kjus Alstad, Carsten Griwodz and Pål Halvorsen

University of Oslo/Simula Research Laboratory
Oslo, Norway
*haakonks@ifi.uio.no

Dag Johansen

University of Tromsø
Tromsø, Norway

There are many scenarios where high resolution, wide field of view video is useful. Such panorama video may be generated using camera arrays where the feeds from multiple cameras pointing at different parts of the captured area are stitched together. However, processing the different steps of a panorama video pipeline in real-time is challenging due to the high data rates and the stringent timeliness requirements. In our research, we use panorama video in a sport analysis system called Bagadus. This system is deployed at Alfheim stadium in Tromsø, and due to live usage, the video events must be generated in real-time. In this paper, we describe our real-time panorama system built using a low-cost CCD HD video camera array. We describe how we have implemented different components and evaluated alternatives. The performance results from experiments ran on commodity hardware with and without co-processors like graphics processing units (GPUs) show that the entire pipeline is able to run in real-time.

Keywords: Real-time panorama video; system integration; camera array.

1. Introduction

A wide field of view (panoramic) image or video is often used in applications like surveillance, navigation, scenic views, educational exhibits and sports analysis. Here, video feeds are often captured using multiple cameras capturing slightly overlapping areas, and the frames are processed and stitched into a single unbroken frame of the whole surrounding region. To prepare the individual frames for stitching and finally generating the panorama frame, each individual frame must be processed for barrel distortion, rotated to have the same angle, warped to the same plane and corrected for color differences. Then, the frames are stitched to one large panorama image, where the stitch operation also includes searching for the best possible seam in the overlapping areas to avoid seams through objects of interest in the video. Finally, the panorama frame is encoded to save storage space and transfer bandwidth and,

written to disk. As several of these steps include direct pixel manipulation and movement of large amounts of data, the described process is very resource hungry.

In [30], we described our implementation of a real-time panorama video pipeline for an arena sports application called Bagadus [11, 28], and this is an extended version providing more details. In our panorama setup, we use a static array of low-cost CCD HD video cameras, each pointing at a different direction, to capture the wide field of view of the arena. These different views are slightly overlapped in order to facilitate the stitching of these videos to form the panoramic video. Several similar non-real-time stitching systems exist (e.g. [23]), and a simple non-real-time version of this system has earlier been described and demonstrated at the functional level [11, 26]. Our initial prototype is the first sports application to successfully integrate per-athlete sensors [17], an expert annotation system [16] and a video system, but due to the non-real-time stitching, the panorama video was only available to the coaches some time after a game. The first prototype also did not use any form of color correction or dynamic seam detection. Hence, the static seam did not take into account moving objects (such as players), and the seam was therefore very visible.
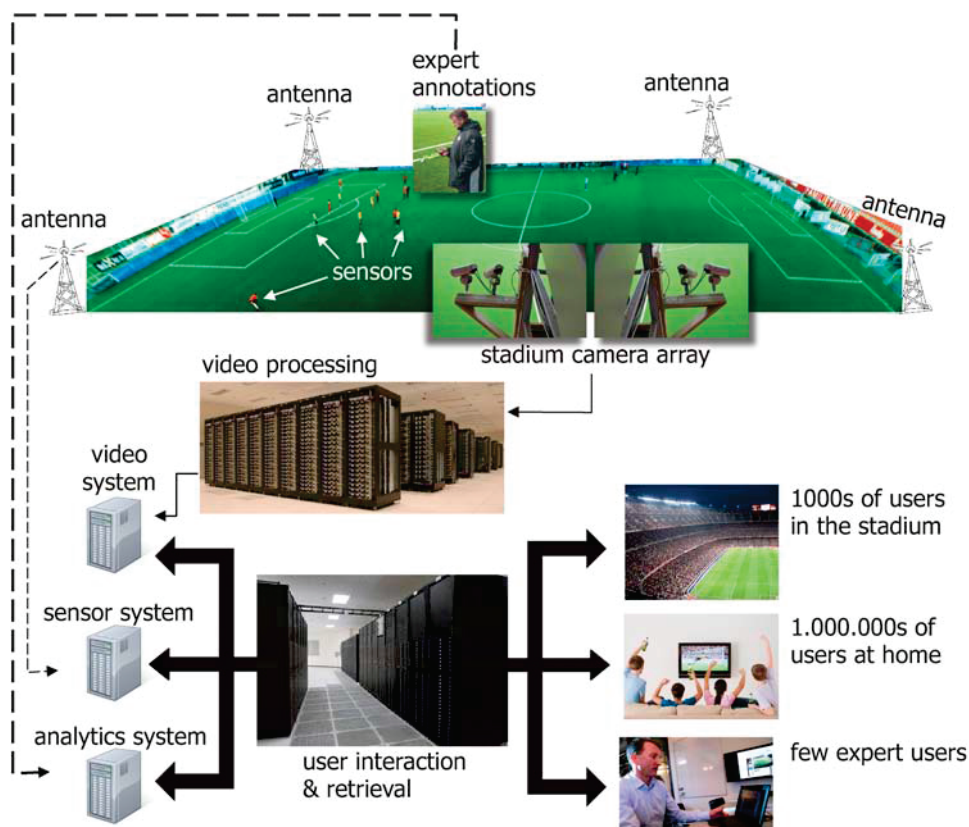

Fig. 1.   Overall system architecture.

However, new requirements like real-time performance and better visual quality have resulted in a new and improved pipeline. Using our new real-time pipeline, such systems can be used during the game. A brief overview of the architecture and interaction of the different components is given in Fig. 1. In this paper we will focus on the details of the whole pipeline from capturing images from the cameras via various corrections steps for panorama generation to encoding and storage of both the panorama video and the individual camera streams on disks. We describe how we have evaluated different implementation alternatives (both algorithms and implementation options), and we benchmark the performance with and without using graphics processing units (GPUs) as an co-processors. We evaluate each individual component, and, we show how the entire pipeline is able to run in real-time on a low-cost 6-core machine with a GPU, i.e. moving the 1 frame per second (fps) system to 30 fps enabling game analysis during the ongoing event.

The remainder of the paper is structured as follows. We give a brief overview of the basic idea of our system in Sec. 2, and then we analyze the state of the art in Sec. 3 to see if systems exist that meet our requirements. Then, we describe and evaluate our real-time panorama video pipeline in Sec. 4. Various aspect of the system are discussed in Sec. 5 before we finally conclude the paper in Sec. 6.

2. Our Sports Analysis Systems

Today, a large number of (elite) sports clubs spend a large amount of resources to analyze their game performance, either manually or using one of the many existing analytics tools. For example, in the area of soccer, several systems enable trainers and coaches to analyze the gameplay in order to improve the performance. For instance, Interplay-sports, ProZone, STATS SportVU Tracking Technology and Camargus provide very nice video technology infrastructures. These systems can present player statistics, including speed profiles, accumulated distances, fatigue, fitness graphs and coverage maps using different charts, 3D graphics and animations. Thus, there exist several tools for soccer analysis. However, to the best of our knowledge, there does not exist a system that fully integrates all desired features in real-time, and existing systems still require manual work moving data between different components. In this respect, we have presented Bagadus [11, 26], which integrates a camera array video capture system with a sensor-based sport tracking system for player statistics and a system for human expert annotations. Our system allows the game analytics to automatically playout a tagged game event or to extract a video of events extracted from the statistical player data. This means that we for example can query for all sprints faster than X or all situations where a player is in the center circle. Using the exact player position provided by sensors, a trainer can also follow individuals or groups of players, where the videos are presented either using a stitched panorama view of the entire field or by (manually or automatically) switching between the different camera views. Our prototype is currently deployed at an elite club stadium. We use a dataset captured at a premier league game to experiment and to perform

benchmarks on our system. In previous versions of the system, the panorama video had to be generated offline, and it had static seams [11]. For comparison with the new pipeline presented in Sec. 4, we next present the camera setup and the old pipeline.

2.1. *Camera setup*

To record the high resolution video of the entire soccer field, we have installed a camera array consisting of four Basler industry cameras with a 1/3-inch image sensors supporting 30 fps at a resolution of 1280×960. The cameras are synchronized by an external trigger signal in order to enable a video stitching process that produces a panorama video picture. The cameras are mounted close to the middle line (see Fig. 2), i.e. under the roof of the stadium covering the spectator area approximately 10 meters from the side line and 10 meters above the ground. With a 3.5 mm wide-angle lens, each camera covers a field-of-view of about 68 degrees, and the full field with sufficient overlap to identify common features necessary for camera calibration and stitching, is achieved using the four cameras. Calibration is done via a classic chessboard pattern [33].
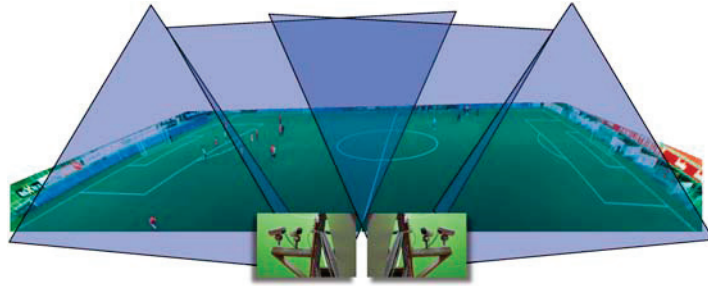
Fig. 2.   Camera setup at the stadium.

2.2. *The offline, static stitching pipeline*

Our first prototype focused on integrating the different subsystems. We therefore did not put large efforts into real-time performance resulting in an unoptimized, offline panorama video pipeline that combined images from multiple, trigger-synchronized cameras as described above. The general steps in this stitching pipeline are: (1) correct the images for lens distortion in the outer parts of the frame due to a wide-angle fish-eye lens; (2) rotate and morph the images into the panorama perspective caused by different positions covering different areas of the field; (3) rotate and stitch the video images into a panorama image; and (4) encode and store the stitched video to persistent storage. Several implementations were tested for the stitching operation such as the OpenCV planar projection, cylindrical projection and spherical projection algorithms, but due to the processing performance and quality of the output image, the used solution is a homography based algorithm.
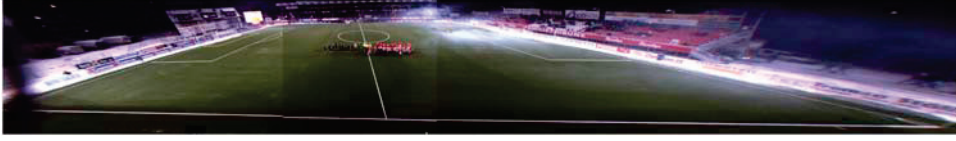
Fig. 3.   The homography-based panorama image stitched from four cameras.

The first step before executing the pipeline, is to find corresponding pixel points in order to compute the homography between the camera planes [12], i.e. the head camera plane and the remaining camera planes. When the homography is calculated, the image can be warped (step 2) in order to fit the plane of the second image. The images must be padded to have the same size, and the seams for the stitching must be found in the overlapping regions (our first pipeline used static seams). Figure 3 shows the four rotated, wrapped and stitched images. The whole process of stitching the images is described in [28]. We also calculate the homography between the sensor data plane and the camera planes to find the mapping between sensor data coordinates and pixel positions.

As can be seen in the figure, the picture is not perfect, but the main challenge is the high execution time. On an Intel Core i7-2600 @ 3.4 GHz and 8 GB memory machine, the stitching operation consumed 974 ms of CPU time to generate each 7000×960 pixel panorama image [11]. Taking into account that the target display rate is 30 fps, i.e. requiring a new panorama image every 33 ms, there are large performance issues that must be addressed in order to bring the panorama pipeline from a 1 fps system to a 30 fps system. However, the stitching operations can be parallelized and parts of it offloaded to external devices such as GPUs, which, as we will see in Sec. 4, results in a performance good enough for real-time, online processing and generation of a panorama video.

3. Related Work

Real-time panorama image stitching is becoming common. For example, many have proposed systems for panorama image stitching (e.g. [6, 14, 19–21]), and modern operating systems for smart phones like Apple iOS and Google Android support generation of panorama pictures in real-time. However, the definition of real-time is not necessarily the same for all applications, and in this case, real-time is similar to "within a second or two". For video, real-time has another meaning, and a panorama picture must be generated in the same speed as the display frame rate, e.g. every 33 ms for a 30 frame-per-second (fps) video.

One of these existing systems is Camargus [1]. The people developing this system claim to deliver high definition panorama video in real-time from a setup consisting of 16 cameras (ordered in an array), but since this is a commercial system, we have no insights to the details. Another example is Immersive Cockpit [29] which aims to generate a panorama for tele-immersive applications. They generate a stitched video

which capture a large field-of-view, but their main goal is not to give output with high visual quality. Although they are able to generate video at a frame rate of about 25 fps for 4 cameras, there are visual limitations to the system, which makes the system not well suited for our scenario.

Moreover, Baudisch *et al.* [5] present an application for creating panoramic images, but the system is highly dependent on user input. Their definition of real time is "panorama construction that offers a real-time preview of the panorama while shooting", but they are only able to produce about 4 fps (far below our 30 fps requirement). A system similar to ours is presented in [4], which computes stitch-maps on a GPU, but the presented system produces low resolution images (and is limited to two cameras). The performance is within our real-time requirement, but the timings are based on the assumption that the user accepts a lower quality image than the cameras can produce.

Haynes [3] describes a system by the Content Interface Corporation that creates ultra high resolution videos. The Omnicam system from the Fascinate [2, 27] project also produces high resolution videos. However, both these systems use expensive and specialized hardware. The system described in [3] also makes use of static stitching. A system for creating panoramic videos from already existing video clips is presented in [7], but it does not manage to create panorama videos within our definition of real-time. As far as we know, the same issue of real-time is also present in [5, 13, 23, 31].

In summary, existing systems (e.g. [7, 13, 23, 29, 31]) do not meet our demand of being able to generate the video in real-time, and commercial systems (e.g. [1, 3]) as well as the systems presented in [2, 27] do often not fit into our goal to create a system with limited resource demands. The system presented in [4] is similar to our system, but we require high quality results from processing a minimum of four cameras streams at 30 fps. Thus, due to the lack of a low-cost implementations fulfilling our demands, we have implemented our own panorama video processing pipeline which utilize processing resources on both the CPU and GPU.

4. A Real-Time Panorama Stitcher

In this paper, we do not focus on selecting the best algorithms etc., as this is mostly covered in [11]. The focus here is to describe the panorama pipeline and how the different components in the pipeline are implemented in order to run in real-time. We will also point out various performance trade-offs.

As depicted in Fig. 4, the new and improved panorama stitcher pipeline is separated into two main parts: one part running on the CPU, and the other running on a GPU using the CUDA framework. The decision of using a GPU as part of the pipeline was due to the potential high performance and the parallel nature of the workload. The decision of using the GPU for the pipeline has affected the architecture to a large degree. Unless otherwise stated (we have tested several CPUs and GPUs), our test machine for the new pipeline is an Intel Core i7-3930K, i.e. a 6-core
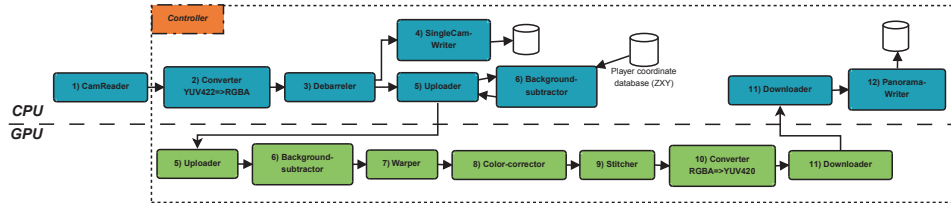
Fig. 4.   Panorama stitcher pipeline architecture. The orange and blue components run in the CPU and the green components run on the GPU.

processor based on the Sandy Bridge-E architecture, with 32 GB RAM and an Nvidia GeForce GTX 680 GPU based on the GK104 Kepler architecture.

4.1. *The Controller module*

The single-threaded Controller is responsible for initializing the pipeline, synchronizing the different modules, handling global errors and frame drops, and transferring data between the different modules. After initialization, it will wait for and get the next set of frames from the camera reader (CamReader) module (see below). Next, it will control the transfers of data from the output buffers of module N to the input buffers of module $N+1$. This is done primarily by pointer swapping, and with memory copies as an alternative. It then signals all modules to process the new input and waits for them to finish processing. Next, the controller continues looping by waiting for the next set of frames from the reader. Another important task of the Controller is to check the execution speed. If an earlier step in the pipeline runs too slow, and one or more frames has been lost from the cameras, the controller will tell the modules in the pipeline to skip the delayed or dropped frame, and reuse the previous frame.

4.2. *The CamReader module*

The CamReader module is responsible for retrieving frames from the cameras. It consists of one dedicated reader thread per camera. Each of the threads will wait for the next frame, and then write the retrieved frame to a output buffer, overwriting the previous frame. The cameras provide a single frame in YUV 4:2:2 format, and the retrieval rate of frames in the CamReader is what determines the real time threshold for the rest of the pipeline. As described above, the camera shutter synchronization is controlled by an external trigger box, and in our current configuration, the cameras deliver a frame rate of 30 fps, i.e. the real-time threshold and the CamReader processing time are thus 33 ms.

4.3. *The Converter module*

The CamReader module outputs frames in YUV 4:2:2 format. However, the stitching pipeline requires RGBA internally for processing, and the system therefore converts frames from YUV 4:2:2 to RGBA. This is handled by the Converter module using

ffmpeg and *swscale*. The processing time for these conversions on the CPU, as seen later in Fig. 11, is well below the real-time requirement, so this operation can run as a single thread.

4.4. *The Debarreler module*

Due to the wide angle lenses used with our cameras in order to capture the entire field, the images delivered are suffering from barrel distortion which needs to be corrected. We found the performance of the existing debarreling implementation in the old stitching pipeline to perform fast enough. The Debarreler module is therefore still based on OpenCVs debarreling function, using nearest neighbor interpolation, and is executing as a dedicated thread per camera.

4.5. *The SingleCamWriter module*

In addition to storing the stitched panorama video, we also want to store the video from the separate cameras. This storage operation is done by the SingleCamWriter, which is running as a dedicated thread per camera. As we can see in [11], storing the videos as raw data proved to be impractical due to the size of uncompressed raw data. The different CamWriter modules (here SingleCamWriter) therefore encode and compress frames into 3 seconds long H.264 files, which proved to be very efficient. Due to the use of H.264, every SingleCamWriter thread starts by converting from RGBA to YUV 4:2:0, which is the required input format by the x264 encoder. The threads then encode the frames and write the results to disk.

4.6. *The Uploader module*

Due to the large potential of parallelizing the panorama workload and the high computing power of modern GPUs, large parts of our pipeline run on a GPU. We therefore need to transfer data from the CPU to the GPU, i.e. a task performed by the Uploader module. In addition, the Uploader is also responsible for executing the CPU part of the BackgroundSubtractor (BGS) module (see Sec. 4.7). The Uploader consists of a single CPU thread, that first runs the player pixel lookup creation needed by the BGS. Next, it transfers the current RGBA frames and the corresponding player pixel maps from the CPU to the GPU. This is done by use of double buffering and asynchronous transfers.

4.7. *The BackgroundSubtractor module*

Background subtraction is the process of determining which pixels of a video that belong to the foreground and which pixels that belong to the background. The BackgroundSubtractor module, running on the GPU, generates a foreground mask (for moving objects like players) that is later used in the Stitcher module later to avoid seams in the players. Our BackgroundSubtractor can run like traditional systems searching the entire image for foreground objects. However, we can also

exploit information gained by the tight integration with the player sensor system. In this respect, through the sensor system, we know the player coordinates which can be used to improve both performance and precision of the module. By first retrieving player coordinates for a frame, we can then create a player pixel lookup map, where we only set the players pixels, including a safety margin, to 1. The creation of these lookup maps are executed on the CPU as part of the Uploader. The BGS on GPU then uses this lookup map to only process pixels close to a player, which reduces the GPU kernel processing times, from 811.793 microseconds to 327.576 microseconds on average on a GeForce GTX 680. When run in a pipelined fashion, the processing delay caused by the lookup map creation is also eliminated. The sensor system coordinates are retrieved by a dedicated slave thread that continuously polls the sensor system database for new samples.

Even though we enhance the background subtraction with sensor data input, there are several implementation alternatives. When determining which algorithm to implement, we evaluated two alternatives: Zivkovic [34, 35] and KaewTraKulPong [18]. Even though the CPU implementation was slower (see Fig. 5), Zivkovic provided the best visual results, and was therefore selected for further modification. Furthermore, the Zivkovic algorithm proved to be fast enough when modified with input from the sensor system data. The GPU implementation, based on [25], proved to be even faster, and the final performance numbers for a single camera stream can be seen in Fig. 5. A visual comparison of the unmodified Zivkovic implementation and the sensor system-modified version is seen in Fig. 6 where the sensor coordinate modification reduce the noise as seen in the upper parts of the pictures.
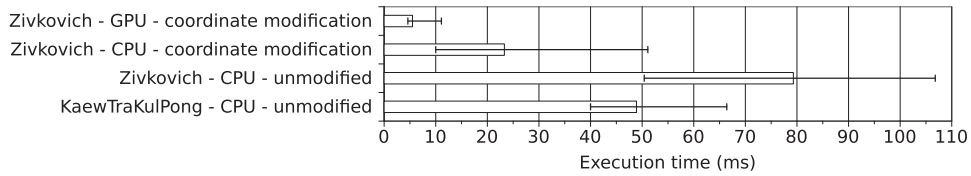

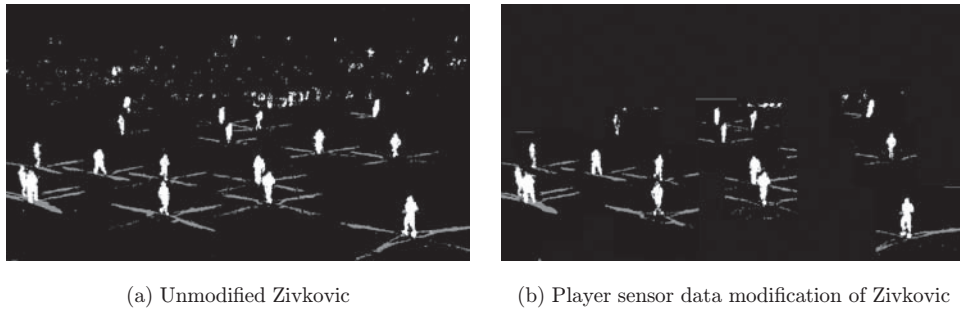Fig. 5.   Execution time of alternative algorithms for the BackgroundSubtractor module (1 camera stream).

(a) Unmodified Zivkovic   (b) Player sensor data modification of Zivkovic

Fig. 6.   Background subtraction comparison.

4.8. *The Warper module*

The Warper module is responsible for warping the camera frames to fit the stitched panorama image. By warping we mean twisting, rotating and skewing the images to fit the common panorama plane. Like we have seen from the old pipeline, this is necessary because the stitcher assumes that its input images are perfectly warped and aligned to be stitched to a large panorama. Executing on the GPU, the Warper also warps the foreground masks provided by the BGS module. This is because the Stitcher module at a later point will use the masks and therefore expects the masks to fit perfectly to the corresponding warped camera frames. Here, we use the Nvidia Performance Primitives library (NPP) for an optimized implementation.

4.9. *The Color-corrector module*

When recording frames from several different cameras pointing in different direction, it is nearly impossible to calibrate the cameras to output the exact same colors due to the different lighting conditions. This means that, to generate the best panorama videos, we need to correct the colors of all the frames to remove color disparities. In our panorama pipeline, this is done by the Color-corrector module running on the GPU.

We choose to do the color correction after warping the images. The reason for this is that locating the overlapping regions is easier with aligned images, and the overlap is also needed when stitching the images together. This algorithm is executed on the GPU, enabling fast color correction within our pipeline. The implementation is based on the algorithm presented in [32], but have some minor modifications. We calculate the color differences between the images for every single set of frames delivered from the cameras. Currently, we color-correct each image in a sequence, meaning that each image is corrected according to the overlapping frame to the left. The algorithm implemented is easy to parallelize and does not make use of pixel to pixel mapping which makes it well suited for our scenario. Figure 7 shows a comparison between running the algorithm on the CPU and on a GPU.
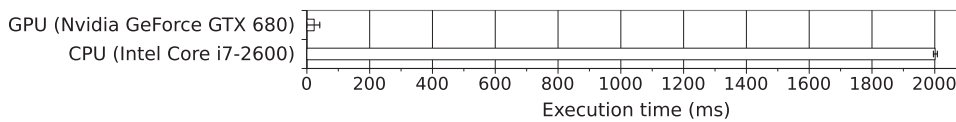

Fig. 7.   Execution time of color correction.

4.10. *The Stitcher module*

Like in the old pipeline, we use a homography based stitcher where we simply create seams between the overlapping camera frames, and then copy pixels from the images based on these seams. These frames need to follow the same homography, which is

why they have to be warped. Our old pipeline used static cuts for seams, which means that a fixed rectangular area from each frame is copied directly to the output frame. Static cut panoramas are faster, but can introduce graphical errors in the seam area, especially when there are movement in the scene (illustrated in Fig. 8).

To make a better seam with a better visual result, we therefore introduce a dynamic cut stitcher instead of the old static cut. The dynamic cut stitcher creates seams by first creating a rectangle of adjustable width over the static seam area. Then, it treats all pixels within the seam area as graph nodes. The graph is directed from the bottom to the top in such a way that each pixel points to the three adjacent ones above (left and right-most pixels only point to the two available). Each of these edge's weight are calculated by using a custom function that compares the absolute color difference between the corresponding pixel in each of the two frames we are trying to stitch. The weight function also checks the foreground masks from the BGS module to see if any player is in the pixel, and if so it adds a large weight to the node. In effect, both these steps will make edges between nodes where the colors differs and players are present have much larger weights. We then run the Dijkstra graph

Fig. 8. Stitcher comparison — improving the visual quality with dynamic seams and color correction. The first image shows the original stitch [11] with a fixed cut stitch with a straight vertical seam. The middle image shows a dynamic stitch with no color correction. The embedded thumbnail shows the seam. The bottom image shows a dynamic stitch with color correction, i.e. resulting in that the seam is no longer visible.

algorithm on the graph to create a minimal cost route from the start of the offset at the bottom of the image to the end at the top. Since our path is directed upwards, we can only move up or diagonally from each node, and we will only get one node per horizontal position. By looping through the path, we therefore get our new cut offsets by adding the node's horizontal position to the base offset.

An illustration of how the final seam looks can be seen in bottom image in Fig. 8, where the seams without and with color correction are shown in the embedded thumbnails. Timings for the dynamic stitching module can be seen in Fig. 9. The CPU version is currently slightly faster than our GPU version (as searches and branches often are more efficient on traditional CPUs), but further optimization of the CUDA code will likely improve this GPU performance. Note that the min and max numbers for the GPU are skewed by frames dropping (no processing), and the initial run being slower.
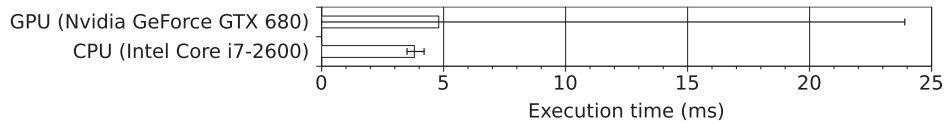
Fig. 9.   Execution time for dynamic stitching.

4.11. *The YuvConverter module*

Before storing the stitched panorama frames, we need to convert back from RGBA to YUV 4:2:0 for the H.264 encoder, just as in the SingleCamWriter module. However, due to the size of the output panorama, this conversion is not fast enough on the CPU, even with the highly optimize *swscale*. This module is therefore implemented on the GPU. In Fig. 10, we can see the performance of the CPU based implementation versus the optimized GPU based version.

Nvidia NPP contains several conversion primitives, but not a direct conversion from RGBA to YUV 4:2:0. The GPU based version is therefore first using NPP to convert from RGBA to YUV 4:4:4, and then a self written CUDA code to convert from YUV 4:4:4 to YUV 4:2:0.
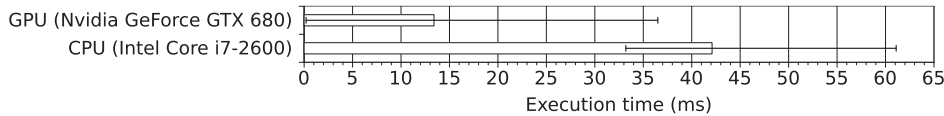
Fig. 10.   Execution time for RGBA to YUV 4:2:0 conversion.

4.12. *The Downloader module*

Before we can write the stitched panorama frames to disk, we need to transfer it back to the CPU, which is done by the Downloader module. It runs as a single CPU thread

that copies a frame synchronously to the CPU. We could have implemented the Downloader as an asynchronous transfer with double buffering like the Uploader, but since the performance as seen in Fig. 11 is very good, this is left as future work.
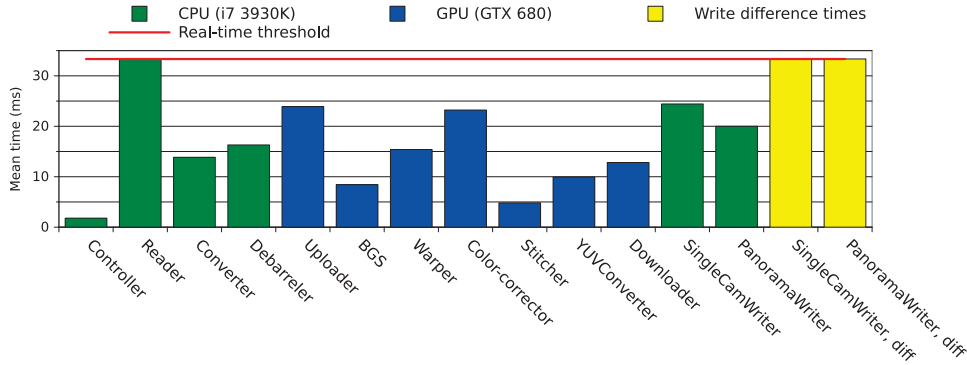
Fig. 11.   Improved stitching pipeline performance, module overview (Nvidia GeForce GTX 680 and Intel Core i7-3930K).

4.13. *The PanoramaWriter module*

The last module, executing on the CPU, is the Writer that writes the panorama frames to disk. The conversion from RGBA to YUV has already been done on the GPU, so the only steps the PanoramaWriter needs to follow, is to first encode the input frame to H.264, and then write the result to disk as three second H.264 video files.

4.14. *Pipeline performance*

In order to evaluate the performance of our pipeline, we used an off-the-shelf PC with an Intel Core i7-3930K processor and an nVidia GeForce GTX 680 GPU. We have benchmarked each individual component and the pipeline as a whole capturing, processing and storing 1000 frames from the cameras.

In the initial pipeline [11], the main bottleneck was the panorama creation (warping and stitching). This operation alone used *974 ms per frame.* As shown by the breakdown into individual components' performance in Fig. 11, the new pipeline has been greatly improved. Note that all individual components run in real-time running concurrently on the same set of hardware. Adding all these, however, gives times far larger than 33 ms. The reason why the pipeline is still running in real-time is because several frames are processed in parallel. Note here that all CUDA kernels are executing at the same time on a single GPU, so the performance of all GPU modules are affected by the performance of the other GPU modules. On earlier GPUs like the GTX 280, this was not allowed, but concurrent CUDA kernel execution was introduced in the Fermi architecture [24] (GTX 480 and above). Thus, since the

(a) SingleCamWriter inter-departure time

(b) Inter-departure time of PanoramaWriter.
The inter-departure frames delayed by five
seconds due to the two second safety
buffer in CPU/GPU transfer and the
three second delay of the sensor data

(c) Core count scalability

(d) Core frequency scalability
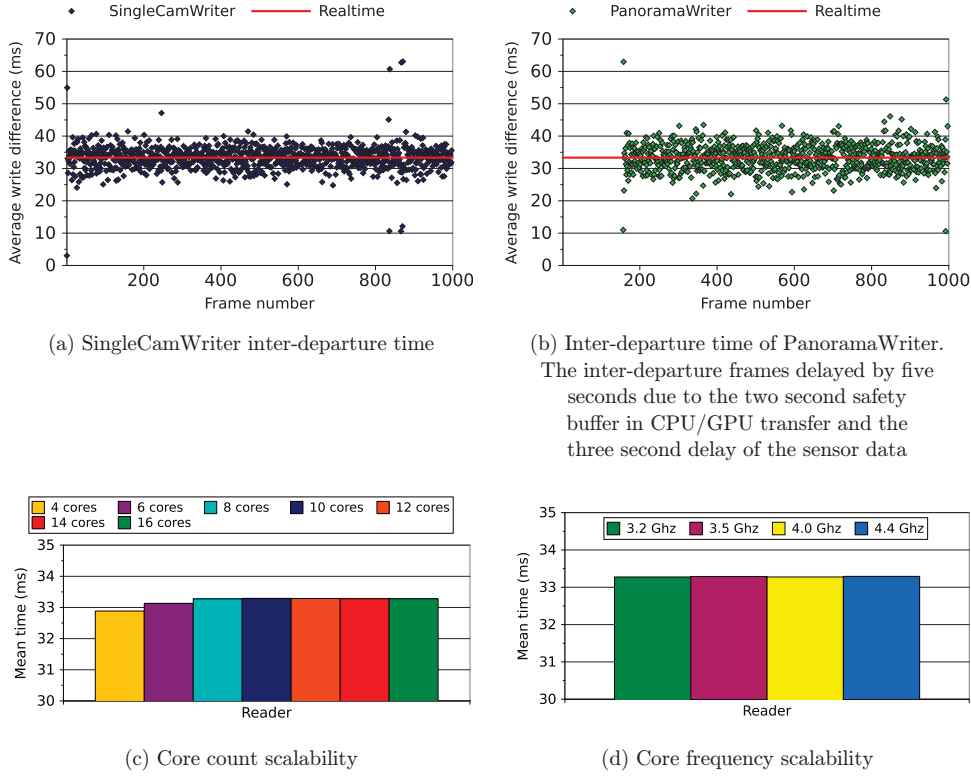
Fig. 12.   Inter-departure time of frames when running the entire pipeline. In a real-time scenario, the output rate should follow the input rate (given here by the trigger box) at 30 fps (33 ms).

Controller module schedules the other modules according to the input rate of 30 fps, the amount of resources are sufficient for real-time execution.

For the pipeline to be real-time, the output rate should follow the input rate, i.e. deliver all output frames (both 4 single camera and 1 panorama) at 30 fps. Thus, to give an idea of how often a frame is written to file, Fig. 12 shows individual and average frame inter-departure rates. The figures show the time difference between consecutive writes for the generated panorama as well as for the individual camera streams. Operating system calls, interrupts and disk accesses will most likely cause small spikes in the write times (as seen in the scatter plot in Figs. 12(a) and 12(b)), but as long as the average times are equal to the real-time threshold, the pipeline can be considered real-time. As we can see in Figs. 11, 12(c) and 12(d), the average frame inter-arrival time (Reader) is equal to the average frame inter-departure time (both SingleCamWriter and PanoramaWriter). This is also the case testing other CPU frequencies and number of available cores. Thus, the pipeline runs in real-time.

As said above and seen in Figs. 12(a) and 12(b), there is a small latency in the panorama pipeline compared to writing the single cameras immediately. The total panorama pipeline latency, i.e. the end to end frame delay from read from the camera

until written to disk, is equal to 33 ms per sequential module (as long as the modules perform fast enough) plus a 5 second input buffer (the input buffer is because the sensor system has at least 3 second latency before the data is ready for use, and we have added a 2 second buffer for GPU processing). The 33 ms are caused by the camera frame rate of 30 fps, meaning that even though a module may finish before the threshold, the Controller will make it wait until the next set of frames arrive before it is signaled to re-execute. This means that the pipeline latency is 5.33 seconds per frame on average.

5. Discussion

Our soccer analysis application integrates a sensor system, soccer analytics annotations and video processing of a video camera array. There already exist several components that can be used, and we have investigated several alternatives in our research. Our first prototype aimed at full integration at the system level, rather than being optimized for performance. In this paper, however, our challenge has been of an order of magnitude harder by making the system run in real-time on low-cost, off-the-shelf hardware.

The new real-time capability also enables future enhancements with respect to functionality. For example, several systems have already shown the ability to serve available panorama video to the masses [13, 23], and by also generating the panorama video live, the audience can mark and follow particular players and events. Furthermore, ongoing work also include machine learning of sensor and video data to extract player and team statistics for evaluation of physical and tactical performance. We can also use this information to make video playlists [15] automatically giving a video summary of extracted events. Due to limited availability of resources, we have not been able to test our system with more cameras or higher resolution cameras. However, to still get an impression of the scalability capabilities of our pipeline, we have performed several benchmarks changing the number of available cores, the processor clock frequency and GPUs with different architecture and compute resources. Figure 13[a] shows the results changing the number of available cores that can process the many concurrent threads in the CPU-part of pipeline (Fig. 12(c) shows that the pipeline is still in real-time). As we can observe from the figure, every component runs in real-time using more than 4 cores, and the pipeline as a whole using 8 or more cores. Furthermore, the CPU pipeline contains a large, but configurable number of threads (86 in the current setup), and due to the many threads of the embarrassingly parallel workload, the pipeline seems to scale well with the number of available cores.

Similar conclusions can be drawn from Fig. 14 where the processing time is reduced with a higher processor clock frequency, i.e. the pipeline runs in real-time already at 3.2 GHz, and there is almost a linear scaling with CPU frequency (Fig. 12(d)

---

[a]Note that this experiment was run on a machine with more available cores (16), each at a lower clock frequency (2.0 GHz) compared to the machine installed at the stadium which was used for all other tests.
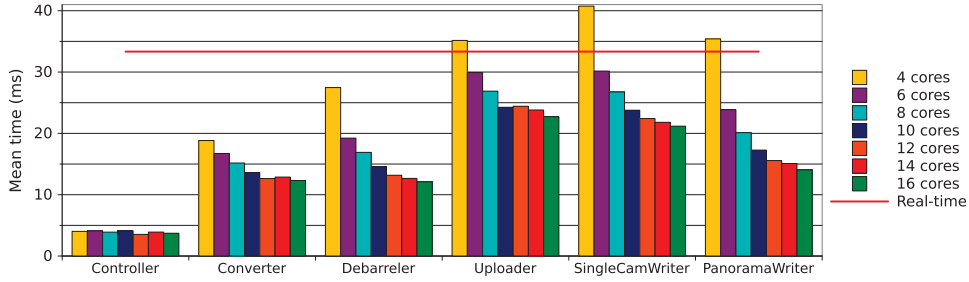
Fig. 13.   Core count scalability.
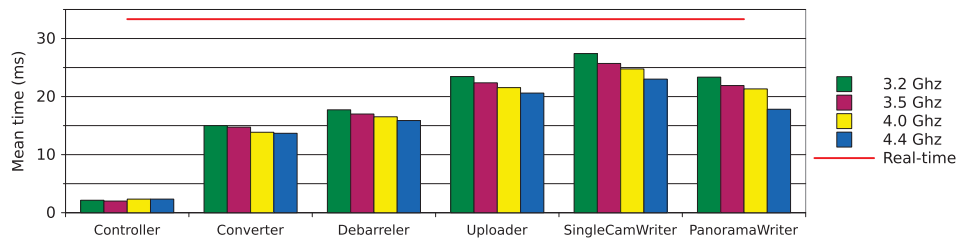
Fig. 14.   CPU frequency scalability.

shows that the pipeline is still in real-time). Especially the H.264 encoder scales very good when scaling the CPU frequency. With respect to the GPU-part of the pipeline, Fig. 15 plots the processing times using different GPUs.

The high-end GPUs GTX 480 and above (Compute 2.x and higher) all achieve real-time performance on the current setup. The GTX 280 is only compute 1.3 which does not support the concurrent CUDA kernel execution in the Fermi architecture [24], and the performance is therefore slower than real-time. As expected, more powerful GPUs reduce the processing time. For now, one GPU fulfills our real-time requirement, we did therefore not experiment with multiple GPUs, but the GPU processing power can easily be increased by adding multiple cards. Thus, based on these results, we believe that our pipeline easily can be scaled up to both higher numbers of cameras and higher resolution cameras.
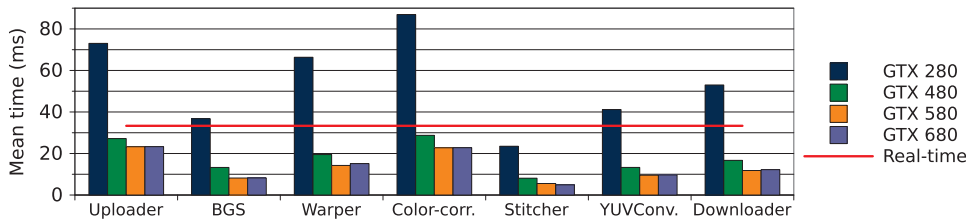
Fig. 15.   GPU comparison.

6. Conclusions

In this paper, we have presented a prototype of a real-time panorama video processing system. The panorama prototype is used as a sub-component in a real sport analysis system where the target is automatic processing and retrieval of events at a sports arena. We have described the pipeline in detail, where we use both the CPU and a GPU for offloading. Furthermore, we have provided experimental results which prove the real-time properties of the pipeline on a low-cost 6-core machine with a commodity GPU, both for each component and the combination of the different components forming the entire pipeline.

The entire system is under constant development, and new functionality is added all the time, e.g. camera-array-wide synchronized automatic exposure [8], interactive zoom and panning [9, 10], extended search functionality [22] and scaling the panorama system up to a higher number of cameras and to higher resolution cameras [9]. So far, the pipeline scales nicely with the CPU frequencies, number of cores and GPU resources. We plan to use PCI Express-based interconnect technology from Dolphin Interconnect Solutions for low latency and fast data transfers between machines. Experimental results in this respect is though ongoing work and out of scope in this paper.

Acknowledgments

References

[1] Camargus — Premium Stadium Video Technology Infrastructure, http://www.camargus.com/. [Online; accessed 01-March-2013.]

[2] Live ultra-high resolution panoramic video, http://www.fascinate-project.eu/index.php/tech-section/hi-res-video/. [Online; accessed 04-March-2012.]

[3] Software stitches 5k videos into huge panoramic video walls, in real time, http://www.sixteen-nine.net/2012/10/22/software-stitches-5k-videos-huge-panoramic-video-walls-real-time/, 2012. [Online; accessed 05-March-2012.]

[4] M. Adam, C. Jung, S. Roth and G. Brunnett, Real-time stereo-image stitching using GPU-based belief propagation, 2009, pp. 215–224.

[5] P. Baudisch, D. Tan, D. Steedly, E. Rudolph, M. Uyttendaele, C. Pal and R. Szeliski, An exploration of user interface designs for real-time panoramic photography, *Australasian Journal of Information Systems* **13**(2) (2006) 151.

[6] M. Brown and D. G. Lowe, Automatic panoramic image stitching using invariant features, *International Journal of Computer Vision* **74**(1) (2007) 59–73.

[7] D.-Y. Chen, M.-C. Ho and M. Ouhyoung, Videovr: A real-time system for automatically constructing panoramic images from video clips, in *Proc. CAPTECH*, 1998, pp. 140–143.

[8] V. R. Gaddam, C. Griwodz and P. Halvorsen, Automatic exposure for panoramic systems in uncontrolled lighting conditions: A football stadium case study, in *Proc. SPIE/ IS&T Electronic Imaging — the Engineering Reality of Virtual Reality*, 2014, pp. 90120C–90120C-9.

[9] V. R. Gaddam, R. Langseth, S. Ljødal, P. Gurdjos, V. Charvillat, C. Griwodz and P. Halvorsen, Interactive zoom and panning from live panoramic video, in *Proc. NOSSDAV*, 2014, pp. 19:19–19:24.

[10] V. R. Gaddam, R. Langseth, H. K. Stensland, P. Gurdjos, V. Charvillat, C. Griwodz, D. Johansen and P. Halvorsen, Be your own cameraman: Real-time support for zooming and panning into stored and live panoramic video, in *Proc. MMSys*, 2014, pp. 168–171.

[11] P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz and D. Johansen, Bagadus: An integrated system for arena sports analytics — a soccer case study, in *Proc. MMSys*, 2013, pp. 48–59.

[12] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision* (Cambridge University Press, 2003).

[13] K. Huguenin, A.-M. Kermarrec, K. Kloudas and F. Taiani, Content and geographical locality in user-generated content sharing systems, in *Proc. NOSSDAV*, 2012, pp. 77–82.

[14] J. Jia and C.-K. Tang, Image stitching using structure deformation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **30**(4) (2008) 617–631.

[15] D. Johansen, H. Johansen, T. Aarflot, J. Hurley, Å. Kvalnes, C. Gurrin, S. Sav, B. Olstad, E. Aaberg, T. Endestad, H. Riiser, C. Griwodz and P. Halvorsen, DAVVI: A prototype for the next generation multimedia entertainment platform, in *Proc. ACM MM*, 2009, pp. 989–990.

[16] D. Johansen, M. Stenhaug, R. B. A. Hansen, A. Christensen and P.-M. Høgmo, Muithu: Smaller footprint, potentially larger imprint, in *Proceedings of the IEEE International Conference on Digital Information Management*, 2012, pp. 205–214.

[17] H. D. Johansen, S. A. Pettersen, P. Halvorsen and D. Johansen, Combining video and player telemetry for evidence-based decisions in soccer, in *Proceedings of the International Congress on Sports Science Research and Technology Support*, 2013, pp. 197–205.

[18] P. KaewTraKulPong and R. Bowden, An improved adaptive background mixture model for real-time tracking with shadow detection, in *Video-Based Surveillance Systems* (Springer, 2002), pp. 135–144.

[19] A. Levin, A. Zomet, S. Peleg and Y. Weiss, Seamless image stitching in the gradient domain, *Computer Vision — ECCV 2004*, 2004, pp. 377–389.

[20] Y. Li and L. Ma, A fast and robust image stitching algorithm, in *Proc. WCICA* **2** (2006) 9604–9608.

[21] A. Mills and G. Dudek, Image stitching with dynamic elements, *Image and Vision Computing* **27**(10) (2009) 1593–1602.

[22] A. Mortensen, V. R. Gaddam, H. K. Stensland, C. Griwodz, D. Johansen and P. Halvorsen, Automatic event extraction and video summaries from soccer games, in *Proc. MMSys*, 2014, pp. 176–179.

[23] O. A. Niamut, R. Kaiser, G. Kienast, A. Kochale, J. Spille, O. Schreer, J. R. Hidalgo, J.-F. Macq and B. Shirley, Towards a format-agnostic approach for production, delivery and rendering of immersive media, in *Proc. MMSys*, 2013, pp. 249–260.

[24] nVIDIA. Nvidia's next generation CUDA compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010. [Online; accessed 08-March-2013].

[25] V. Pham, P. Vo, V. T. Hung *et al.*, GPU implementation of extended gaussian mixture model for background subtraction, in *IEEE International Conference on Computing*

and Communication Technologies, Research, Innovation and Vision for the Future,* 2010, pp. 1–4.

[26] S. Sægrov, A. Eichhorn, J. Emerslund, H. K. Stensland, C. Griwodz, D. Johansen and P. Halvorsen, Bagadus: An integrated system for soccer analysis (demo), in *Proc. ICDSC*, 2012, pp. 1–2.

[27] O. Schreer, I. Feldmann, C. Weissig, P. Kauff and R. Schafer, Ultrahigh-resolution panoramic imaging for format-agnostic video production, in *Proceedings of the IEEE* **101**(1) (2013) 99–114.

[28] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, Ø. Landsverk, C. Griwodz, P. Halvorsen, M. Stenhaug and D. Johansen, Bagadus: An integrated real-time system for soccer analytics, *Transactions on Multimedia Computing, Communications and Applications* **10**(1s) (2014) 14:1–14:21.

[29] W.-K. Tang, T.-T. Wong and P.-A. Heng, A system for real-time panorama generation and display in tele-immersive applications, *IEEE Transactions on Multimedia* **7**(2) (2005) 280–292.

[30] M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, V. R. Gaddam, H. K. Stensland, C. Griwodz, D. Johansen and P. Halvorsen, Efficient implementation and processing of a real-time panorama video pipeline, in *Proc. ISM*, 2013, pp. 76–83.

[31] C. Weissig, O. Schreer, P. Eisert and P. Kauff, The ultimate immersive experience: Panoramic 3d video acquisition, *Advances in Multimedia Modeling*, LNCS Vol. 7131 (Springer, 2012), pp. 671–681.

[32] Y. Xiong and K. Pulli, Color correction for mobile panorama imaging, in *Proc. ICIMCS*, 2009, pp. 219–226.

[33] Z. Zhang, Flexible camera calibration by viewing a plane from unknown orientations, in *Proceedings of the IEEE International Conference on Computer Vision*, 1999, pp. 666–673.

[34] Z. Zivkovic, Improved adaptive gaussian mixture model for background subtraction, in *Proc. ICPR* **2** (2004) 28–31.

[35] Z. Zivkovic and F. van der Heijden, Efficient adaptive density estimation per image pixel for the task of background subtraction, *Pattern Recognition Letters* **27**(7) (2006) 773–780.

Posters and live demonstrations

Transparent protocol translation and load balancing on a network processor in a media streaming scenario

Title: Transparent protocol translation and load balancing on a network processor in a media streaming scenario [33].

Authors: H. Espeland, C. Lunde, H. K. Stensland, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08.

Summary: A live demonstration of the proxy was presented at NOSSDAV.

Processing of Multimedia Data using the P2G Framework

Title: Processing of Multimedia Data using the P2G Framework [11].

Authors: P. B. Beskow, H. K. Stensland, H. Espeland, E. A. Kristiansen, P. N. Olsen, S. B. Kristoffersen, C. Griwodz, and P. Halvorsen.

Published: Proceedings of the 19th ACM international conference on Multimedia (MM), ACM, 2011.

Summary: A live demonstration of the P2G framework encoding video and dynamically adapting to the available resources was presented at ACM Multimedia.

Distributed Real-Time Processing of Multimedia Data with the P2G Framework

Title: Distributed Real-Time Processing of Multimedia Data with the P2G Framework [10].

Authors: P. B. Beskow, H. Espeland, H. K. Stensland, P. N. Olsen, S. B. Kristoffersen, E. A. Kristiansen, C. Griwodz, and P. Halvorsen.

Published: EuroSys 2011 (Poster Session), ACM, 2011.

Summary: Ideas about the P2G prototype, with early experimental results were presented at a poster session on the EuroSys conference.

A Demonstration Of a Lockless, Relaxed Atomicity State Parallel Game Server (LEARS)

Title: A Demonstration Of a Lockless, Relaxed Atomicity State Parallel Game Server (LEARS) [101].

Authors: K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz.

Published: Workshop on Network and Systems Support for Games (NetGames), IEEE / ACM, 2011.

Summary: A live demonstration of the LEARS lockless game server was presented at NetGames.

BAGADUS: An Integrated System for Soccer Analysis

Title: BAGADUS: An Integrated System for Soccer Analysis [105].

Authors: S. Sægrov, A. Eichhorn, J. Emerslund, H. K. Stensland, C. Griwodz, and P. Halvorsen

Published: Proceedings of the International Conference on Distributed Smart Cameras (ICDSC), ACM /IEEE, 2012

Summary: The offline version of the Bagadus Soccer Analysis system demonstrated at ICDSC.

Realtime Panorama Video Processing Using NVIDIA GPUs

Title: Realtime Panorama Video Processing Using NVIDIA GPUs [120].

Authors: M. Tennøe, E. O. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, P. Halvorsen, and C. Griwodz.

Published: Nvidia GPU Technology Conference, 2013.

Summary: Poster about the real-time Bagadus panorama pipeline presented at GTC.

Performance and Application of the NVIDIA NVENC H.264 Encoder

Title: Performance and Application of the NVIDIA NVENC H.264 Encoder [132].

Authors: M. A. Wilhelmsen, H. K. Stensland, V. R. Gaddam, P. Halvorsen, and C. Griwodz

Published: Nvidia GPU Technology Conference, 2014.

Summary: Poster about using hardware encoder for delivery of interactive video streams presented at GTC.

Other research papers

Efficient Implementation and Processing of a Real-time Panorama Video Pipeline

Title: Efficient Implementation and Processing of a Real-time Panorama Video Pipeline [119].

Authors: M. Tennøe, E. O. Helgedagsrud, M. Næss, H. K. Alstad, H. K. Stensland, V. R. Gaddam, D. Johansen, P. Halvorsen, and C. Griwodz.

Published: Proceedings of the International Symposium on Multimedia (ISM), IEEE, 2013.

Abstract: High resolution, wide field of view video generated from multiple camera feeds has many use cases. However, processing the different steps of a panorama video pipeline in real-time is challenging due to the high data rates and the stringent requirements of timeliness. We use panorama video in a sport analysis system where video events must be generated in real-time. In this respect, we present a system for real-time panorama video generation from an array of low-cost CCD HD video cameras. We describe how we have implemented different components and evaluated alternatives. We also present performance results with and without co- processors like graphics processing units (GPUs), and we evaluate each individual component and show how the entire pipeline is able to run in real-time on commodity hardware.

Bagadus: An Integrated System for Arena Sports Analytics   A Soccer Case Study

Title: Bagadus: An Integrated System for Arena Sports Analytics   A Soccer Case Study [46].

Authors: P. Halvorsen, S. Sægrov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen

Published: Proceedings of the 4th annual ACM conference on Multimedia Systems (MMSYS), ACM, 2013.

Abstract: Sports analytics is a growing area of interest, both from a computer system view to manage the technical challenges and from a sport performance view to aid the development of athletes. In this paper, we present Bagadus, a prototype of a sports analytics application using soccer as a case study. Bagadus integrates a sensor system, a soccer analytics annotations system and a video processing system using a video camera array. A prototype is currently installed at Alfheim Stadium in Norway, and in this paper, we describe how the system can follow and zoom in on particular player(s). Next, the system will playout events from the games using stitched panorama video or camera switching mode and create video summaries based on queries to the sensor system. Further- more, we evaluate the system from a systems point of view, benchmarking different approaches, algorithms and trade-offs.

Improved Multi-Rate Video Encoding

Title: Improved Multi-Rate Video Encoding [35].

Authors: D. H. Finstad, H. K. Stensland, H. Espeland, and P. Halvorsen

Abstract: Adaptive HTTP streaming is frequently used for both live and on-Demand video delivery over the Internet. Adaptiveness is often achieved by encoding the video stream in multiple qualities (and thus bitrates), and then transparently switching between the qualities according to the bandwidth fluctuations and the amount of resources available for decoding the video content on the end device. For this kind of video delivery over the Internet, H.264 is currently the most used codec, but VP8 is an emerging open-source codec expected to compete with H.264 in the streaming scenario. The challenge is that, when encoding video for adaptive video streaming, both VP8 and H.264 run once for each quality layer, i.e., consuming both time and resources, especially important in a live video delivery scenario. In this paper, we address the resource consumption issues by proposing a method for reusing redundant steps in a video encoder, emitting multiple outputs with varying bitrates and qualities. It shares and reuses the computational heavy analysis step, notably macro-block mode decision, intra prediction and inter prediction between the instances, and outputs video in several rates. The method has been implemented in the VP8 reference encoder, and experimental results show that we can encode the different quality layers at the same rates and qualities compared to the VP8 reference encoder, while reducing the encoding time significantly.

Improving File Tree Traversal Performance by Scheduling I/O Operations in User space

Title: Improving File Tree Traversal Performance by Scheduling I/O Operations in User space [76].

Authors: C. H. Lunde, H. Espeland, H. K. Stensland, and P. Halvorsen.

Published: Proceedings of the 28th IEEE International Performance Computing and Communications Conference (IPCCC), IEEE, 2009.

Abstract: Current in-kernel disk schedulers provide efficient means to optimize the order (and minimize disk seeks) of issued, in-queue I/O requests. However, they fail to optimize sequential multi-file operations, like traversing a large file tree, because only requests from one file are available in the scheduling queue at a time. We have therefore investigated a user-level, I/O request sorting approach to reduce inter-file disk arm movements. This is achieved by allowing applications to utilize the placement of inodes and disk blocks to make a one sweep schedule for all file I/Os requested by a process, i.e., data placement information is read first before issuing the low-level I/O requests to the storage system. Our experiments with a modified version of tar show reduced disk arm movements and large performance improvements.

Improving Disk I/O Performance on Linux

Title: Improving Disk I/O Performance on Linux [77].

Authors: C. H. Lunde, H. Espeland, H. K. Stensland, A. Petlund, and P. Halvorsen.

Published: UpTimes - Proceedings of Linux-Kongress and OpenSolaris Developer Conference, GUUG, 2009.

Abstract: The existing Linux disk schedulers are in general efficient, but we have identified two scenarios where we have observed a non-optimal behavior. The first is when an application requires a fixed bandwidth, and the second is when an operation performs a file tree traversal. In this paper, we address both these scenarios and propose solutions which both increase performance.

Making an SCI Fabric Dynamically Fault Tolerant

Title: Making an SCI Fabric Dynamically Fault Tolerant [116].

Authors: H. K. Stensland, O. Lysne, R. Nordstrœm, and H. Kohmann.

Published: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 2008.

Abstract: In this paper we present a method for dynamic fault tolerant routing for SCI networks implemented on Dolphin Interconnect Solutions hardware. By dynamic fault tolerance, we mean that the interconnection network reroutes affected packets around a fault, while the rest of the network is fully functional. To the best of our knowledge this is the first reported case of dynamic fault tolerant routing available on commercial off the shelf interconnection network technology without duplicating hardware resources. The development is focused around a 2-D torus topology, and is compatible with the existing hardware, and software stack. We look into the existing mechanisms for routing in SCI. We describe how to make the nodes that detect the faulty component do routing decisions, and what changes are needed in the existing routing to enable support for local rerouting. The new routing algorithm is tested on clusters with real hardware. Our tests show that distributed databases like MySQL can run uninterruptedly while the network reacts to faults. The solution is now part of Dolphin Interconnect Solutions SCI driver, and hardware development to further decrease the reaction time is underway.

Appendix A

BNF Grammar of the P2G Kernel Language

```
%token <n>        IDENTIFIER
%token <n>        INTEGER
%token <n>        VECTOR
%token <n>        INTRINSIC
%token <s>        NATIVE
%token TYPE
%token INDEX
%token LOCAL
%token AGE
%token LAST
%token SIZE
%token ORDERED
%token INT
%token INCR
%token WRAP
%token FETCH
%token STORE
%token DEF
%token NEXT
%token TOP_HEADER
%token TOP_CODE
%token HEADER
%token CODE
%token BIND
%token IF
%token THEN
%token ELSE
%token END
%token DOTDOT
%token EQ
%token LE
%token GE
%token HOUR
%token MIN
%token SEC
%token US
%token MS
```

```
%token FINAL
%token FINALIZE
%token FINALIZE_ON_ALL
%token FINALIZE_ON_ONE
%token TIMER
%token SET
%token NOW
%token DEADLINE

%right  '='
%left   '<' '>' GE LE EQ
%left   DOTDOT ':'
%left   '+' '-'
%left   '*' '/' '%'
%left   UNARY_MINUS_PRECEDENCE

%%

start: statements
      ;

statements: statements statement
          |
          ;

statement: field_definition
         | timer_definition
         | kernel_definition
         | TOP_HEADER ':' NATIVE /* top of generated header file */
         | TOP_CODE   ':' NATIVE /* top of generated code file */
         | HEADER     ':' NATIVE /* after field definitions in generated
             header file */
         | CODE       ':' NATIVE /* after field declarations in generated
             code file */
         ;

/**
 * % Each field is a virtual multi-dimensional array of data.
 * % It is virtual because the kernel program forbids to write to
 * % the same field in more than one unique kernel, which does
 * % naturally not make sense in reality.
 * % The data cannot be stored contiguously in memory, either,
 * % because parallel architecture will encourage pipelining of
 * % data through several kernels before writing it to a
 * % contiguous location in memory (if ever).
 * % In the process of mapping, many of these fields will be
 * % identified with each other or be optimized out.
 * % A field should not be considered a bunch of memory cells,
 * % but a bunch of memory cells at the time of a kernel's run.
 * % Example field definitions:
 * %   float128  f;
 * %   int32[4] array;
 * %   vector3<float64>[3] three_coordinates, three_more, and_three_more;
 * %   vector8 complex128[][8][][][256] weird_stuff;
 */
```

```
field_definition: datatype define_global_field_names field_attributes ';'
                | intrinsic_datatype error ';'
                ;

datatype: intrinsic_datatype field_dimensions
        | VECTOR intrinsic_datatype field_dimensions
        ;

intrinsic_datatype: INTRINSIC
                  | TYPE type_name
                  ;
type_name: IDENTIFIER
         ;

field_dimensions: field_dimensions '[' INTEGER ']'
                | field_dimensions '[' ']'
                | /* empty */
                ;
field_attributes: field_attributes AGE '(' FINALIZE_ON_ALL ')'
                | field_attributes AGE '(' FINALIZE_ON_ONE ')'
                | field_attributes AGE
                | field_attributes ORDERED
                | /* empty */
                ;
define_global_field_names: define_global_field_names ','
   define_global_field_name
                         | define_global_field_name
                         ;

timer_definition: TIMER def_timer_names timer_attributes ';'
                ;
def_timer_names: def_timer_names ',' define_timer_name
               | define_timer_name
               ;
timer_attributes: timer_attributes AGE
                | /* empty */
                ;

define_local_field_names: define_local_field_names ','
   define_local_field_name
                        | define_local_field_name
                        ;

kernel_definition: kernel_name kernel_native_attributes ':' kernel_head
   NATIVE kernel_tail kernel_deadline
                 | kernel_name kernel_native_attributes ':' kernel_head
                   kernel_tail kernel_deadline
                 | kernel_name kernel_native_attributes ':' kernel_head IF
                   NATIVE THEN kernel_tail END opt_semicolon
                   kernel_deadline
                 | kernel_name kernel_native_attributes ':' kernel_head IF
                   NATIVE THEN kernel_tail ELSE kernel_tail END
                   opt_semicolon kernel_deadline
                 ;
```

```
opt_semicolon:  ';'
                | /* empty */
                ;


/**
 * Native attributes will be used to give the dependency
 * analyser hints about special kernels.
 * In principle, a kernel can be placed anywhere, but if
 * the native code requires user input, disk access and so on,
 * it should be expressed by a list of identifiers
 * (commas strictly optional and without semantic meaning).
 */
kernel_native_attributes: BIND kernel_native_attrlist
                          | '%' error
                          | /* empty */
                          ;
kernel_native_attrlist:  kernel_native_attrlist IDENTIFIER
                         | IDENTIFIER
                         ;
kernel_head:  kernel_head fetch_statement
            | kernel_head index_var_declaration
            | kernel_head field_size_expression
            | kernel_head DEF error ';'
            | /* empty */
            ;
kernel_tail:  kernel_tail store_statement
            | kernel_tail next_statement
            | kernel_tail timer_set_statement
            | /* empty */
            ;

kernel_deadline:  kernel_deadline deadline_statement
            | /* empty */
            ;

field_size_expression: SIZE field_size '=' field_size_expr ';'
                     ;
field_size_expr:  field_size_expr '*' field_size_expr
                | field_size_expr '/' field_size_expr
                | field_size_expr '%' field_size_expr
                | field_size_expr '+' field_size_expr
                | field_size_expr '-' field_size_expr
                | '(' field_size_expr ')'
                | INTEGER
                | '-' INTEGER %prec UNARY_MINUS_PRECEDENCE
                | '+' INTEGER %prec UNARY_MINUS_PRECEDENCE
                | field_size
                  ;

field_size:  use_field_name field_ages '.' INTEGER
          ;

field_ages:  field_ages '.' AGE '(' age_expr ')'
          |
          ;
```

```
index_var_declaration: INDEX index_var_int_or_not index_var_wrap_or_not
    index_var_increment index_vars ';'
                        | LOCAL  index_var_int_or_not index_var_wrap_or_not
                          index_var_increment index_vars ';'
                        | INT  index_var_wrap_or_not index_var_increment
                          index_vars ';'
                        | AGE     index_vars ';'
                        | LOCAL datatype define_local_field_names ';'
                        | INDEX intrinsic_datatype error ';'
                        | LOCAL intrinsic_datatype error ';'
                        | AGE intrinsic_datatype error ';'
                        ;
index_var_int_or_not: INT
                        | /* empty */
                        ;
index_var_wrap_or_not: WRAP
                        | /* empty */
                        ;
index_var_increment: INCR '(' INTEGER ')'
                        | /* empty */
                        ;
index_vars: index_vars ',' index_name
          | index_name
                ;
fetch_statement: FETCH use_field_name index_use '=' use_field_name
    index_use ';'
                ;
store_statement: store_final_prefix STORE use_field_name index_use '='
    use_field_name index_use ';'
                ;
store_final_prefix: FINAL
                        |
                        ;
finalize_statement: FINALIZE use_field_name index_use ';'
                        ;
next_statement: NEXT index_name ';'
                ;
timer_set_statement: SET use_timer_name age_list '=' timer_expr ';'
                        ;

deadline_statement: DEADLINE '(' deadline_condition ')' deadline_actions
    END opt_semicolon
                        ;
deadline_condition: timer_expr expr_cond_cmp timer_expr
                        ;
deadline_actions: deadline_actions store_statement
                    | deadline_actions finalize_statement
                    | /* empty */
                        ;

/**
 * % Expression lists allow fetch and store operations with indices
 * % that are computed statically when dependencies are evaluated.
 * % It would be nice if they could be evaluated statically, but that
 * % would require constant field sizes and we don't want to limit
 * % ourselves to that.
```

```
* % As a very simple example consider the following kernel head:
* %    int64 [10] a,b;
* %    def x;
* %    fetch i=a(x);
* %    fetch j=b(9−x);
* % This initiates 10 kernels, because the fields are C−indexed and
* % both have 10 fields.
* % The fetch statements
* %    fetch i=a(x);
* %    fetch j=b(10−x);
* % on the other hand, would initiate only 9 kernels, because the
* % case x==0 is outside the range of field b.
* %
* % However, it is also possible to use larger blocks of memory
* % in a single kernel. Consider:
* %    float64 [32][32] field;
* %    fetch v = field [x:8][y:8];
* % This avoids that 32x32=1024 kernels are started. Instead,
* % it starts 16 kernels, where v is interpreted as double v[8][8].
* % However, if you want to start 1024 kernel and make all possible
* % 8x8 sub−blocks available to the native code, you have to
* % do the following:
* %    float64 [32][32] field;
* %    fetch v00 = field [x][y];
* %    fetch v01 = field [x][y+1];
* %    ...
* %    fetch v77 = field [x+7][y+7];
* % If people want this, we can create a better syntax for this later.
* % It might make sense, for example to parallelize motion vector
* % search at a super−fine granularity and let the dependency
* % analysis take care of efficiency.
*/
index_use: expr_list
        ;
expr_list: expr_list '[' exprs ']'
        | expr_list '[' exprs '|' expr_cond_a ']'
        | expr_list '[' ']'
        | age_list
        ;

age_list: age_list '(' age_expr ')'
        | age_list '.' AGE '(' age_expr ')'
        | /* empty */
        ;

age_expr: age_expr '*' age_expr
        | age_expr '/' age_expr
        | age_expr '%' age_expr
        | age_expr '+' age_expr
        | age_expr '−' age_expr
        | INTEGER
        | '−' INTEGER %prec UNARY_MINUS_PRECEDENCE
        | '+' INTEGER %prec UNARY_MINUS_PRECEDENCE
        | LAST
        | index_name
        ;
```

```
exprs: exprs ',' expr
     | expr
     ;

/** Note: Originally I used BNF for explicit precedence.
 *  Changed to bison extensions %right and %left to
 *  declare precedence to make the code more compact.
 */
expr: expr DOTDOT expr
    | expr ':' INTEGER
    | expr '*' expr
    | expr '/' expr
    | expr '%' expr
    | expr '+' expr
    | expr '-' expr
    | '(' expr ')'
    | INTEGER
    | '-' INTEGER %prec UNARY_MINUS_PRECEDENCE /* %prec overrides
       precedence of %left */
    | '+' INTEGER %prec UNARY_MINUS_PRECEDENCE
    | index_name
    ;

expr_cond_a: expr_cond_a ',' expr_cond_b
           | expr_cond_b
           ;
expr_cond_b: expr expr_cond_cmp expr
           ;
expr_cond_cmp: '<'
             | '>'
             | EQ
             | LE
             | GE
             ;

timer_expr: timer_expr '*' timer_expr
          | timer_expr '/' timer_expr
          | timer_expr '%' timer_expr
          | timer_expr '+' timer_expr
          | timer_expr '-' timer_expr
          | INTEGER timer_unit
          | '-' INTEGER timer_unit %prec UNARY_MINUS_PRECEDENCE
          | '+' INTEGER timer_unit %prec UNARY_MINUS_PRECEDENCE
          | NOW
          | use_timer_name age_list
          ;
timer_unit: HOUR
          | MIN
          | SEC
          | MS
          | US
          ;
```

```
/**
 * % Kernel names have their own namespace.
 */
kernel_name: IDENTIFIER
           ;

define_global_field_name: field_name
                        ;

define_local_field_name: field_name
                        ;

use_field_name: field_name
              ;

define_timer_name: IDENTIFIER
                 ;
use_timer_name: IDENTIFIER
              ;

/**
 * % Field names have their own namespace.
 */
field_name: IDENTIFIER
          ;

/**
 * % The only meaning of an index name is as a placeholder
 * % for one dimension needed for accessing a cell in a field.
 * % It is similar to a variable in a C for-loop, but the
 * % body can't change it. An index name can be used for
 * % computations in several dimensions (e.g. required for
 * % transposing matrices). Using a name index in a fetch
 * % operation is similar to the creation of an implicit
 * % foreach loop. One kernel should be started for every
 * % possible combination of indices. It is impossible to
 * % prevent this from happening without programming a special
 * % kernel that cuts a sub-field out of a larger field.
 * % It is expected that the evaluation of the dependency
 * % graph makes it possible to transform such mapping
 * % operations into no-ops.
 * %
 * % Index names have their own namespace.
 */
index_name: IDENTIFIER
          ;
```