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

978-1-4244-8355-6/10/$26.00 ©2010 IEEE

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

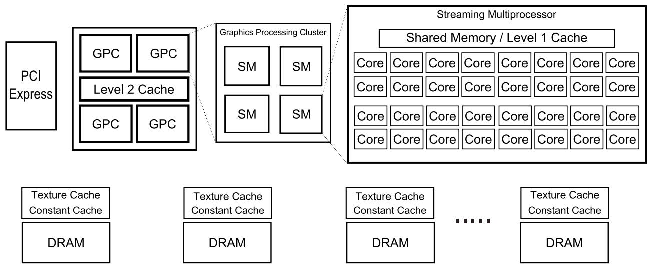

Figure 1. Nvidia GF100 Compute Architecture

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 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.

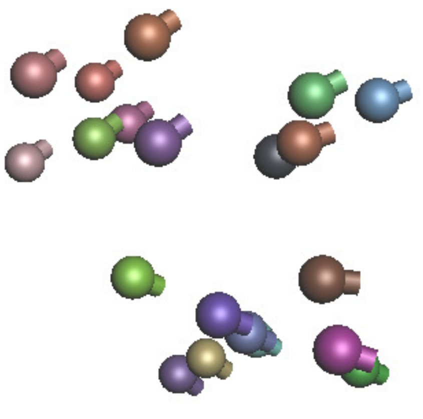

Figure 2. Screenshot of the graphical representation of player objects in the virtual environment.

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,

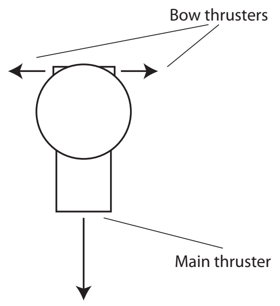

Figure 3. Illustration of a game object with bow thrusters in the front and the main thruster at the back.

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:

$$\sum \mathbf{F} = m\mathbf{a} \tag{1}$$

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

$$\Omega = \frac{\mathrm{d}\omega}{\mathrm{d}t} \tag{2}$$

$$\omega = \frac{\mathrm{d}\alpha}{\mathrm{d}t} \tag{3}$$

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.

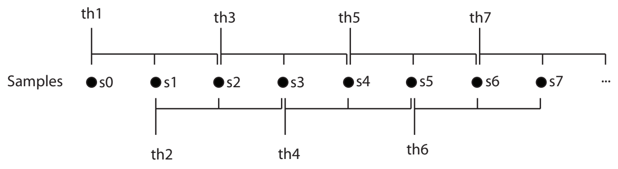

Figure 4. Sample reading and execution pattern of the threads.

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

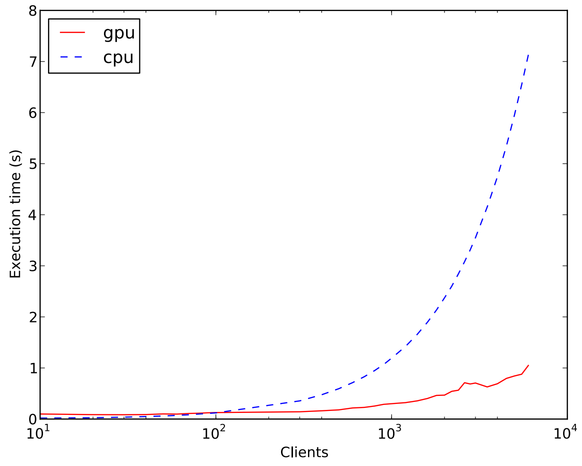

Figure 5. Execution time (in seconds) of the cheat detection mechanism on the GPU and the CPU.

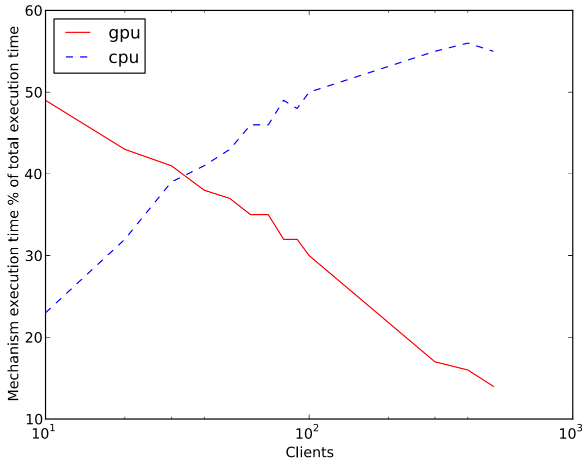

Figure 6. Percent of time spent on cheat detection processing on host using the GPU and the CPU.

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.