Assessment of Data Path Implementations for Download and Streaming

Pål Halvorsen[1,2], Tom Anders Dalseng[1], Carsten Griwodz[1,2]
[1]IFI, University of Oslo, Norway [2]Simula Research Laboratory, Norway
Email: {paalh, tdalseng, griff}@ifi.uio.no

Abstract

Distributed multimedia streaming systems are increasingly popular due to technological advances, and numerous streaming services are available today. On servers or proxy caches, there is a huge scaling challenge in supporting thousands of concurrent users that request delivery of high-rate, time-dependent data like audio and video, because this requires transfers of large amounts of data through several sub-systems within a streaming node. Since the speed increase for memory accesses does not follow suite with the CPU speed, copy operations can be a severe limiting factor on the streaming performance of off-the-shelf operating systems, which still have only limited support for data paths that have been optimized for streaming despite previous research proposals. We observe furthermore that while CPU speed continues to increase, system call overhead has grown as well, adding to the cost of data movement. In this paper, we therefore revisit the data movement problem and provide a comprehensive evaluation of possible streaming data I/O paths in Linux 2.6 kernel. We have implemented and evaluated enhanced mechanisms and show how to provide support for more efficient memory usage and reduction of user/kernel space switches for streaming applications.

1 Introduction

Improvements in access network connectivity with flat-rate Internet connections, such as DSL, cable modems and recently E-PON, and large improvements in machine hardware make distributed multimedia streaming applications increasingly popular and numerous streaming services are available today, e.g., movie-on-demand (Broadpark), news-on-demand (CNN), media content download (iTunes), online radio (BBC), Internet telephony (Skype), etc. At the client side, there is usually no problem presenting the streamed content to the user. On the server side or on intermediate nodes like proxy caches, however, the increased popularity and the increasing data rates of access networks make the scaling challenge even worse when thousands of concurrent users request delivery of high-rate, time-dependent data like audio and video.

In such media streaming scenarios (and many others) that do not require data touching operations, the most expensive server-side operation is moving data from disk to network including encapsulating the data in application- and network packet headers. A proxy cache may additionally forward data from the origin server, make a cached copy of a data element, perform transcoding, etc. Thus, both servers and intermediate nodes that move large amounts of data through several sub-systems within the node may experience high loads as most of the performed operations are both resource and time consuming. Especially, memory copying and address space switches consume a lot of resources [1, 2], and since improvements in memory access speed do not keep up with the increase in CPU speed, these operations will be a severe limiting factor on streaming performance of off-the-shelf operating systems having only limited support for optimized data paths.

In the last 15 years, the area of data transfer overhead has been a major thread in operating system research. In this paper, we have made a Linux 2.6 case study to determine whether more recent hardware and commodity operating systems like Linux have been able to overcome the problems and how close to more optimized data paths the existing solutions are. The reason for this is that a lot of work has been performed in the area of reducing data movement overhead, and many mechanisms have been proposed using virtual memory remapping and shared memory as basic techniques. Off-the-shelf operating systems today frequently include data path optimizations for common applications, such as web server functions. They do not, however, add explicit support for streaming media, and consequently, a lot of streaming service providers make their own implementations. We investigate therefore to which extent the generic functions are sufficient and whether dedicated support for streaming applications can still considerably improve performance. We revisit this data movement problem and provide a comprehensive evaluation of different mechanisms using different data paths in the Linux 2.6 kernel. We have performed several experiments to see the real performance retrieving data from disk and sending data as RTP packets to remote clients. Additionally, we have also implemented and evaluated enhanced mechanisms and we can show that they still improve the performance of streaming operations by providing means for more efficient memory usage and reduction of user/kernel space switches. In particular, we are able to reduce the CPU usage by approximately 27% compared to best existing case removing copy operations and system calls for a given stream.

The rest of this paper is organized as follows: Section 2 gives a small overview of examples of existing mechanisms. In section 3, we present the evaluation of existing mechanisms in the Linux 2.6 kernel, and section 4 describes and evaluates some new enhanced system calls improving the disk-network data path for streaming applications. Section 5 gives a discussion, and finally, in section 6, we conclude the paper.

2 Related Work

The concept of using buffer management to reduce the overhead of cross-domain data transfers to improve I/O performance is rather old. It has been a major issue in operating systems research where variants of this work have been implemented in various operating systems mainly using virtual memory remapping and shared memory as basic techniques. Already in 1972, *Tenex* [3] used virtual copying, i.e., several pointers in virtual memory to one physical page. Later, several systems have been designed which use virtual memory remapping techniques to transfer data between protection domains without requiring several physical data copies. An interprocess data transfer occurs simply by changing the ownership of a memory region from one process to another. Several general purpose mechanisms supporting a zero-copy data path between disk and network adapter have been proposed, including Container Shipping [4], IO-Lite [1], and UVM virtual memory system [5] which use some kind of page remapping, data sharing, or a combination. In addition to mechanisms removing copy operations in all kinds of I/O, some mechanisms have been designed to create a fast in-kernel data path from one device to another, e.g., the disk-to-network data path. These mechanisms do not transfer data between user and kernel space, but keep the data within the kernel and only map it between different kernel sub-systems. This means that target applications comprise data storage servers for applications that do not manipulate data in any way, i.e., no data touching operations are performed by the application. Examples of such mechanisms are the *stream* system call [6], the Hi-Tactix system [7], KStreams [8] and the *sendfile* system call (for more references, see [2]).

Besides memory movement, system calls are expensive operations, because each call to the kernel requires two switches. Even though in-kernel data paths remove some of this overhead, many applications still require application level code that makes kernel calls. Relevant approaches to increase performance include batched system calls [9] and event batching [10].

Although these examples show that an extensive amount of work has been performed on copy and system call avoidance techniques, the proposed approaches have usually remained research prototypes for various reasons, e.g., they are implemented in own operating systems (having an impossible task of competing with Unix and Windows), small implementations for testing only, not integrated with the main source tree, etc. Therefore, only some limited support is included in the most used operating systems today like the *sendfile* system call in UNIX, Linux, AIX and *BSD. In the next section, we therefore evaluate the I/O pipeline performance of the new Linux 2.6 kernel.

3 Existing Mechanisms in Linux

Despite all the proposed mechanisms, only a limited support for various streaming applications is provided in commodity operating systems like Linux. The existing solutions for moving data from storage device to network device usually comprise combinations of the *read/write*, *mmap* and *sendfile* system calls. Below, we present the results of our performance tests using combinations of these for **content download** operations (adding no application level information) and **streaming** operations (adding application level RTP header for timing and sequence numbering).

3.1 Test Setup

The experiments were performed using two machines connected by a point-to-point Ethernet connection. The test machine has an Intel 845 chipset, 1.70 GHz Intel Pentium4 CPU, 400 MHz front side bus and 1 GB PC133 SDRAM. The resource usage is measured using the *getrusage* function measuring consumed user and kernel time to transfer 1 GB of data stored using the Reiser file system in Linux 2.6. Below, we have added the user and kernel time values to get the total resource consumption, and each test is performed 10 times to get more reliable results. However, the differences between the tests are small.

3.2 Copy and Switching Performance

Before looking at the disk-network data transfer performance, we first look at the memory copy and system call performance themselves. In figure 1, an overview of the chipset on our test machine is shown (similar to many other Intel chipsets). Transfers between device and memory are typically performed using DMA transfers that move data over the PCI bus, the I/O controller hub, the hub interface, the memory controller hub and the RAM interfaces. A memory copy operation is performed moving data from RAM over the RAM interfaces, the memory controller hub, the system (front side) bus through the CPU and back to another RAM location with the possible side effect of flushing the cache(s). Data is (possibly unnecessarily) transferred several times through shared components reducing the overall system performance.

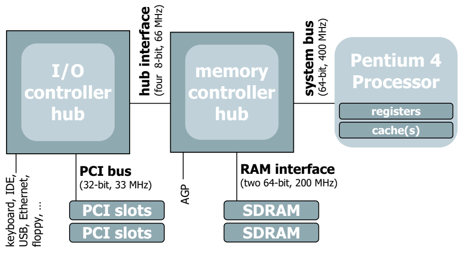
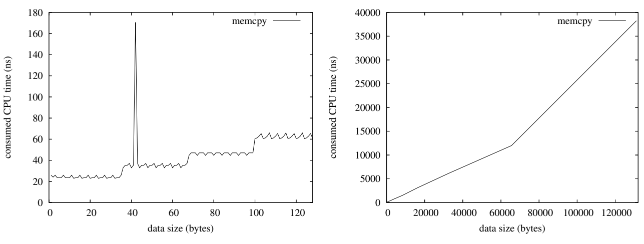

I/O controller hub
hub interface (four 8-bit, 66 MHz)
memory controller hub
system bus (64-bit, 400 MHz)
Pentium 4 Processor
registers
cache(s)
keyboard, IDE, USB, Ethernet, floppy, ...
PCI bus (32-bit, 33 MHz)
PCI slots
PCI slots
AGP
RAM interface (two 64-bit, 200 MHz)
SDRAM
SDRAM

Figure 1. Pentium4 processor and 845 chipset

In [11], memory copy performance was measured on Linux (2.2 and 2.4) and Windows (2000) where the conclusion was that *memcpy* performs well (compared to other copy functions/instructions), and Linux is in most cases faster than Windows depending on data size and used copy instruction. Furthermore, to see the performance on our test machine, we tested *memcpy* using different data sizes. Figure 2a shows that the overhead is slowly growing with the size of the data element, but after reaching a certain size (having cache size effects) the overhead increases more or less linearly with the size (figure 2b)[1].



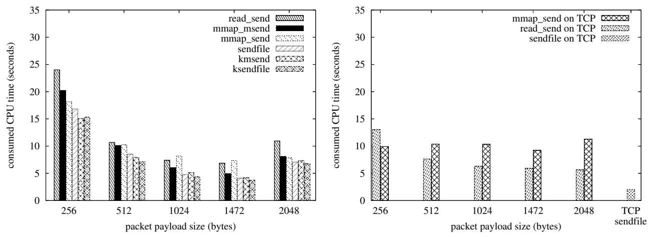
a) b)

Figure 2. User space memory copy speed

With respect to switching contexts, system call overhead (and process context switches) are time consuming on Pentium4 [12]. To get an indication of the system call overhead on our machine, we measured the *getpid* system call, accessing the kernel and only returning the process id. Our experiments show that the average time to access the kernel and return back is approximately 920 nanoseconds for each call.

Copy and system call performance has also been an issue for hardware producers like Intel, who has added new instructions, in particular MMX and SIMD extensions useful for copy operations and *sysenter* and *sysexit* instructions particularly for system calls. For example, using SIMD instructions, the block copy operation speed was improved by up to 149% in the Linux 2.0 kernel, but the reduction in CPU usage was only 2% [13]. Thus, both copy and kernel access performance still are resource consuming and remain possible bottlenecks.

[1]The single high peek in figure 2a at 42 bytes is due to one single test being interrupted by an external event. Additionally, some strange artifacts can be seen when the size of the memory address are not 4-byte aligned with even higher peeks in user space. This is also apparent for larger data sizes and is probably due to different instructions.



a) UDP b) TCP

Figure 3. Content download operations

3.3 Disk-Network Path

The functions used for retrieving data from disk into memory are usually *read* or *mmap*. Data is transfered using DMA from device to memory, and in case of *read*, we require an in-memory copy operation to give the application access to data whereas *mmap* shares data between kernel and user space. To send data, *send* (or similar) can be used where the payload is copied from the user buffer or the page cache depending on whether *read* or *mmap* is used, respectively, to the socket buffer (*sk_buf*). Then, the data is transfered in a DMA operation to the network device. Another approach is to use *sendfile* sending the whole file in one operation, i.e., data is sent directly from a kernel buffer to the communication system using an in-kernel data path. Thus, if gather DMA operations are supported, i.e., needed because the payload and the generated headers are located in different (*sk_buf*) buffers, data can be sent from disk to network without any in-memory copy operations.

3.4 Content Download Experiments

The first test we performed looking at the whole disk-network data path was in a content download scenario. Here, data needs only to be read from disk and sent as soon as possible without application level control. Thus, there is no need to add application level information. To evaluate the performance, we performed several tests using the different data paths and system calls described in section 3.3 and table 1-A using both TCP and UDP. For UDP we also added three enhanced system calls to be able to test a download scenario similar to *sendfile* with TCP.

The results for UDP are shown in figure 3a. We see, as expected, that removal of copy operations and system calls both give performance improvements. Furthermore, in figure 3b, the results using TCP are shown. Again, we see that a quite a lot of resources can be freed using *sendfile* compared to the two other approaches making several system calls and copy operations per data element. Note, however, that it seems that *getrusage* is still not fully implemented for TCP in the 2.6 kernel. Thus, the TCP and UDP experiments are not directly comparable.

From the results, we can see that the existing *sendfile* over TCP performs very well compared to the other tests

A – content download

	co	s	calls to the kernel
read_write	$2n$	$4n$	n ***read*** and n ***write*** calls (TCP will probably gather several smaller elements into one larger MTU-sized packet)
mmap_send	n	2+2n	1 ***mmap*** and n ***send*** calls (TCP will gather several smaller elements into one larger MTU-sized packet)
sendfile (UDP)	0	$2n$	n ***sendfile*** calls
sendfile (TCP)	0	2	1 ***sendfile*** call
mmap_msend†	0	2+2n	1 ***mmap*** and n ***msend*** calls (*msend*† sends data over UDP using the virtual address of a *mmap*'ed file instead of copying the data)
kmsend†	0	2+2n	1 ***kmsend*** call (*kmsend*† combines *mmap* and *msend* (see above) in the kernel until the whole file is sent)
ksendfile†	0	2	1 ***ksendfile*** call (*ksendfile*† performs *sendfile* over UDP in the kernel a la *sendfile* for TCP)

B – RTP streaming

	co	s	calls to the kernel
read_send_rtp	$2n$	$4n$	n ***read*** and n ***send*** calls (RTP header is placed in user buffer in front of payload, i.e., no extra copy operation)
read_writev	$3n$	$4n$	n ***read*** and n ***writev*** calls (RTP header is generated in own buffer, *writev* write data from two buffers)
mmap_send_rtp	$2n$	2+8n	1 ***mmap***, n ***cork***, n ***send***, n ***send*** and n ***uncork*** calls (need one send call for both data and RTP header)
mmap_writev	$2n$	2+2n	1 ***mmap*** and n ***writev*** calls
rtp_sendfile	n	$8n$	n ***cork***, n ***send***, n ***sendfile*** and n ***uncork*** calls

C – enhanced RTP streaming

	co	s	calls to the kernel
mmap_rtpmsend†	n	2+2n	1 ***mmap*** and n ***rtpmsend*** calls (*rtpmsend*† copies RTP headers from user space and adds payload from *mmap*'ed files as payload in the kernel)
mmap_send_msend†	n	2+8n	1 ***mmap***, n ***cork***, n ***send***, n ***msend*** and n ***uncork*** calls (no data copying using *msend*, but the RTP header must be copied from user space)
rtpsendfile†	n	$2n$	n ***rtpsendfile*** calls (*rtpsendfile*† adds the RTP header copy operation to the *sendfile* system call)
krtpsendfile†	0	2	1 ***krtpsendfile*** call (*krtpsendfile*† adds RTP headers to *sendfile* in the kernel)
krtpmsend†	0	2	1 ***krtpmsend*** call (*krtpmsend*† adds RTP headers to the *mmap*/*msend* combination (see above) in the kernel)

co = number of *copy operations*, s = number of *switches* between user and kernel space, n = number of packets, † = new enhanced system call

Table 1. Descriptions of the performed tests

as applications only have to make one single system call to transfer a whole file. Consequently, if no data touching operations, no application level headers or timing support are necessary, *sendfile* seems to be efficiently implemented and achieves a large performance improvement compared to the traditional *read* and *write* system calls, especially when using TCP where only one system call is needed to transfer the whole file.

3.5 Streaming Experiments

Streaming time-dependent data like video to remote clients typically requires adding per-packet headers, such as RTP headers for sequence numbers and timing information. Thus, plain file transfer optimizations are insufficient, because file data must be interleaved with application generated headers, i.e., additional operations must be performed. To evaluate the performance of the existing mechanisms, we performed several tests using the set of data paths and system calls listed in table 1-B. As shown above, the application payload can be transfered both with and without user space buffers, but the RTP header must be copied and interleaved within the kernel. Since TCP may gather several packets into one segment, i.e., the RTP headers will be useless, we have only tested UDP. The results of our tests are shown in figure 4a. Compared to the ftp-like operations in the previous section, we need many system calls and copy operations. For example, compared to the *sendfile* (UDP) and the enhanced *ksendfile* tests in figure 3, there are a 21% and a 29% increase in the measured overhead for the *rtp_sendfile* using Ethernet MTU-sized packets, respectively. This is because we now also need an additional *send* call for the RTP header. Thus, the results indicate that there is a potential for improvements. In the next section, we therefore describe some possible improvements and show that already minor enhancements can achieve large gains in performance.

4 Enhancements for RTP streaming

Looking at the existing mechanisms described and analyzed in the previous section, we are more or less able to remove copy operations (except the small RTP header), but the number of user/kernel boundary crossings is high. We have therefore implemented a couple of other approaches listed in table 1-C. With respect to overhead, *mmap_rtpmsend*, *rtpsendfile*, *krtpmsend* and *krtpsendfile* look promising:

- *mmap_rtpmsend* uses *mmap* to share data between file system buffer cache and the application. Then, it uses the enhanced *rtpmsend* system call to send data copying a user-level generated RTP header and adding the mapped file data using a virtual memory pointer instead of a physical copy. This gives n in-memory data transfers and $1 + n$ system calls. (A further improvement would be to use a virtual memory pointer for the RTP header as well)
- *krtpmsend* uses *mmap* to share data between file system buffer cache and the application and uses the en-

hanced *msend* system call to send data using a virtual memory pointer instead of a physical copy. Then, the RTP header is added in the kernel by a kernel-level RTP engine. This gives no in-memory data transfers and only 1 system call.

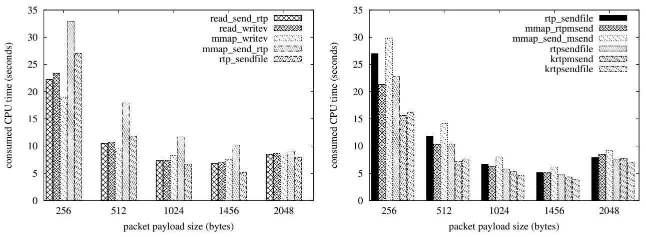
- *rtpsendfile* is a modification of the *sendfile* system call. Instead of having an own call for the RTP header transfer, an additional parameter (a pointer to the buffer holding the header) is added, i.e., the data is copied in the same call and sent as one packet. This gives only n in-memory data transfers and n system calls.
- *krtpsendfile* uses *ksendfile* to transmit a UDP stream in the kernel, in contrast to the standard *sendfile* requiring one system call per packet for UDP. Additionally, the RTP header is added in the kernel having an in-kernel RTP engine. This gives no in-memory data transfers and only 1 system call.

The two first mechanisms are targeted at applications requiring the possibility to touch data in user-space, e.g., parsing or sporadic modifications[2], whereas the last two mechanisms aim at data transfers without application-level data touching. All these enhanced system calls reduce the overhead compared to existing approaches, and to see the real performance gain, we performed the same tests as above. Our results, shown in figure 4b, indicate that simple mechanisms can remove both copy and system call overhead. For example, in the case of streaming using RTP, we see an improvement of about 27% using *krtpsendfile* where a kernel engine generates RTP headers compared to *rtp_sendfile* in the scenario with MTU-sized packets. If we need the same user level control making one call per packet, the *rtpsendfile* enhancement gives at least a 10% improvement compared to existing mechanisms. In another scenario where the application requires data touching operations, the existing mechanism only have small differences. If comparing the results for MTU-sized packets, *read_send_rtp* (already optimized to read data into the same buffer as the generated RTP header) performs best in our tests. However, using a mechanism like *krtpmsend* gives a performance gain of 36% compared to *read_send_rtp*. Similar user level control by making one call per packet is achieved by *mmap_rtpmsend* which gives a 24% gain. Additionally, similar results can in general also be seen for smaller packet sizes (of course with higher overhead due to a larger number of packets), and when the transport level packet exceeds the MTU size, additional fragmentation of the packet introduces additional overhead.

5 Discussion

The enhancements described in this paper to reduce the number of copy operations and system calls mainly address application scenarios where data is streamed to the client

[2]Non-persistant modifications to large parts of the files require a data copy in user space, voiding the use of the proposed mechanisms.



a) Existing mechanisms b) Enhancements

Figure 4. Streaming performance

without any data manipulation at the server side. However, several of the enhanced system calls also show that the application can share a buffer with the kernel and can interleave other information into the stream. Thus, adding support for data touching operations, like checksumming, filtering, etc. without copying, and data modification operations, like encryption, transcoding, etc. with one copy operation, should be trivial. The in-kernel RTP engine also shows that such operations can be performed in the kernel (as kernel stream handlers), reducing copy and system call overhead.

An important issue is whether data copying is still a bottleneck in systems today. The hardware has improved, and one can easily find other possible bottleneck components. However, as shown in section 3.2, data transfers through the CPU are time and resource consuming and have side effects like cache flushes. The overhead increases approximately linearly with the amount of data, and as the gap between memory and CPU speeds increases, so does the problem. Additionally, in figure 5 (note that the y-axis starts at 0.5), we show the performance of the different RTP streaming mechanisms relative to *read_writev*, i.e., a straight forward approach reading data into an application buffer, generating the RTP header and writing the two buffers to the kernel using the vector write operation. Looking for example at MTU-sized packets, we see that a lot of resources can be freed for other tasks. We can also see that less intuitive but more efficient solutions than *read_writev* that do not require kernel changes exist, for example using *sendfile* combined with a *send* for the RTP header (*rtp_sendfile*). However, the best enhanced mechanism, *krtpsendfile*, removes all copy operations and makes only one access to the kernel compared

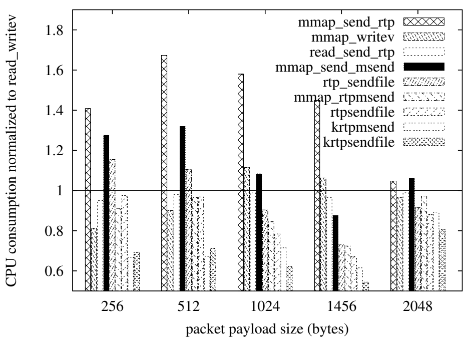


Figure 5. Relative performance to *read_writev*

to *rtp_sendfile* which requires several of both (see table 1). With respect to consumed processor time, we achieve an average reduction of 27% using *krtpsendfile*. Recalculating this into (theoretical) throughput, *rtp_sendfile* and *krtpsendfile* can achieve 1.55 Gbps and 2.12 Gbps, respectively. Assuming a high-end 3.60 GHz CPU like Pentium4 660 and an 800 MHz front side bus, the respective numbers should be approximately doubled. These and higher rates are also achievable for network cards (e.g., Force10 Network's E-Series), PCI express busses and storage systems (e.g., using several Seagate Cheetah X15.4 in a RAID). Thus, the transfer and processing overheads are still potential bottlenecks, and the existing mechanisms should be improved.

Now, having concluded that data transfers and system calls still are possible bottlenecks and having looked at possible enhancements, let us look at what a general purpose operating system like Linux miss. Usually, the commodity operating systems aim at a generality, and new system calls are not frequently added. Thus, specialized mechanisms like *krtpsendfile* and *krtpmsend* having application specific, kernel-level RTP-engines, will hardly ever be integrated into the main source tree and will have to live as patches for interested parties like streaming providers, e.g., like the Red Hat Content Accelerator (*tux*) for web services. However, support for adding application level information (like RTP headers) to stored data will be of increasing importance in the future as streaming services really explode. Simple enhancements like *mmap_rtpmsend* and *rtpsendfile* might be general, performance improving mechanisms that could be of interest in scenarios where the application does or does not touch the data, respectively.

6 Conclusions

In this paper, we have shown that (streaming) applications still pay a high (unnecessary) performance penalty in terms of data copy operations and system calls if those applications require packetization such as addition of RTP headers. We have therefore implemented several enhancements to the Linux kernel, and evaluated both existing and the new mechanisms. Our results indicate that data transfers still are potential bottlenecks, and simple mechanisms can remove both copy and system call overhead if a gather DMA operation is supported. In the case of a simple content download scenario, the existing *sendfile* is by far the most efficient mechanism, but in the case of streaming using RTP, we see an improvement of at least 27% over the existing methods using MTU-sized packets and the *krtpsendfile* system call with a kernel engine generating RTP headers. Thus, using mechanisms for more efficient resource usage, like removing copy operations and avoiding unnecessary system calls, can greatly improve a node's performance. Such enhancements free resources like memory, CPU cycles, bus cycles, etc. which now can be utilized by other applications or providing support for more concurrent streams.

Currently, we also have other kernel activities on-going, and we hope to be able to integrate our subcomponents. We will also modify the KOMSSYS video server to use the proposed mechanisms and perform more extensive tests including a workload experiment looking at the maximum number of concurrent clients able to achieve a timely video playout. Finally, we will optimize our implementation, because most of the enhancements are implemented as proof-of-concept removing copy operations and system calls. We have made no effort in optimizing the code, so the implementations have large potential for improvement, e.g., moving the send-loop from the system call layer to the page cache for the *krtpsendfile* which will remove several file lookups and function calls (as for the existing *sendfile*).

References

[1] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, February 2000.

[2] Pål Halvorsen. *Improving I/O Performance of Multimedia Servers*. PhD thesis, Department of Informatics, University of Oslo, Oslo, Norway, August 2001.

[3] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S.Tomlinson Bolt Beranek. Tenex, a paged time sharing system for the pdp-10. *Communications of the ACM*, 15(3):135–143, March 1972.

[4] Eric W. Anderson. *Container Shipping: A Uniform Interface for Fast, Efficient, High-Bandwidth I/O*. PhD thesis, Computer Science and Engineering Department, University of California, San Diego, CA, USA, July 1995.

[5] Charles D. Cranor and Gurudatta M. Parulkar. The UVM virtual memory system. In *Proceedings of the USENIX Annual Technical Conference*, pages 117–130, Monterey, CA, USA, June 1999.

[6] Frank W. Miller and Satish K. Tripathi. An integrated input/output system for kernel data streaming. In *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, pages 57–68, San Jose, CA, USA, January 1998.

[7] Damien Le Moal, Tadashi Takeuchi, and Tadaaki Bandoh. Cost-effective streaming server implementation using hi-tactix. In *Proceedings of the ACM International Multimedia Conference (ACM MM)*, pages 382–391, Juan-les-Pins, France, December 2002.

[8] Jiantao Kong and Karsten Schwan. Kstreams: Kernel support for efficient end-to-end data streaming. Technical Report GIT-CERCS-04-04, College of Computing, Georgia Institute of Technology, Atlanta, GA, USA, 2004.

[9] Charles Coffing. An x86 protected mode virtual machine monitor for the mit exokernel. Master's thesis, Paralell & Distributed Operating System Group, MIT, Cambridge, MA, USA, May 1999.

[10] Christian Poellabauer, Karsten Schwan, Richard West, Ivan Ganev, Neil Bright, and Gregory Losik. Flexible user/kernel communication for real-time applications in elinux. In *Proceedings of the Workshop on Real Time Operating Systems and Applications and Second Real Time Linux Workshop*, Orlando, FL, USA, November 2000.

[11] Edward Bradford. Runtime: Block memory copy (part 2) – high-performance programming techniques on linux and windows. http://www-106.ibm.com/developerworks/library/l-rt3/, July 1999.

[12] Gregory McGarry. Benchmark comparison of netbsd 2.0 and freebsd 5.3. http://www.feyrer.de/NetBSD/gmcgarry/, January 2005.

[13] Intel Corporation. Block copy using PentiumIII streaming SIMD extensions (revision 1.9). ftp://download.intel.com/design/servers/-softdev/copy.pdf, 1999.