International Journal of Software Engineering and Knowledge Engineering

Assessment of Linux' Data Path Implementations for Download and Streaming

PÅL HALVORSEN
Department of Informatics, University of Oslo, Oslo, Norway
Simula Research Laboratory, Oslo, Norway
paalh@{ifi.uio, simula}.no

TOM ANDERS DALSENG
Department of Informatics, University of Oslo, Oslo, Norway
tdalseng@ifi.uio.no

CARSTEN GRIWODZ
Department of Informatics, University of Oslo, Oslo, Norway
Simula Research Laboratory, Oslo, Norway
griff@{ifi.uio, simula}.no

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. Unnecessary copy operations in the data path can therefore contribute significantly to the resource consumption of streaming operations. Despite previous research, off-the-shelf operating systems have only limited support for data paths that have been optimized for streaming. Additionally, system call overhead has grown with newer operating systems editions, adding to the cost of data movement. Frequently, it is argued that these issues can be ignored because of the continuing growth of CPU speeds. However, such an argument fails to take problems of modern streaming systems into account. The dissipation of heat generated by disks and high-end CPUs is a major problem of data centers, which would be alleviated if less power-hungry CPUs could be used. The power budget of mobile devices, which are increasingly used for streaming as well, is tight, and reduced power consumption an important issue. In this paper, we prove that these operations consume a large amount of resources, and we therefore revisit the data movement problem and provide a comprehensive evaluation of possible streaming data I/O paths in the Linux 2.6 kernel. We have implemented and evaluated several enhanced mechanisms and show how to provide support for more efficient memory usage and reduction of user/kernel space switches for content download and streaming applications. In particular, we are able to reduce the CPU usage by approximately 27% compared to the best approach without kernel modifications, by removing copy operations and system calls for a streaming scenario in which RTP headers must be added to stored data for sequence numbers and timing.

Keywords: data path enhancements, streaming, content download

1. Introduction

Improvements in access network connectivity, such as digital subscriber lines, cable modems and recently passive optical networks, WLAN, as well as UMTS, and large improvements in machine hardware make distributed multimedia streaming applications increasingly popular. Numerous streaming services are available today, e.g., movie-on-demand (Broadpark, SF-anytime), news-on-demand (CNN), media content download (iTunes), online radio (BBC), Internet telephony (Skype).

The receivers of such media streams are usually well-equipped to handle a small number of streams. Senders, on the other hand, face scaling challenges. A streaming media server that is part of a large-scale on-demand streaming application will have to scale to thousands of concurrent users that request timely delivery of high-rate media streams. In addition to this, a server will also perform additional tasks such as stream management, encryption, media transcoding, adaptation and compression. And in a server cluster setup where distribution of this task to several machines appear easier, the service provider face increasing problems of power consumption and heat dissipation. Such a setup will therefore scale better if the CPU can perform the same operations using less CPU power. In alternative, distributed approaches such as peer-to-peer systems, resource consumption should be kept low to have less impact on other applications. Peer-to-peer participants may not have the latest hardware, and their users may run other applications in the foreground that should not be disturbed. Finally, it is becoming increasingly common to stream stored content from mobile devices. These have a very limited lifetime of the batteries. Reducing the resource consumption in terms of CPU cycles and enabling the use of a smaller CPU increase the time such a device can operate.

In these and many other media streaming scenarios that do not require data touching operations, the most expensive operation performed by the sender is moving data from disk to network including the encapsulation of 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, senders 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 [21,10,11].

In the last 15 years, the area of data transfer overhead has been a major thread in operating system research, and several zero-copy [24] data paths have been proposed to optimize resource-hungry data movement operation. Reducing the number of consumed CPU cycles per operation may have a large impact on system performance in several contexts, making this relevant again today. Reducing this cost is highly desirable to make resources available for other tasks or for enabling the use of smaller, less power consuming processors for the same tasks.

Therefore, 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 media streaming where we often interleave user space information like timing constraints (e.g., using the real-time protocol (RTP) header), and consequently, a lot of streaming service providers make their own implementations. We therefore investigate to which extent the generic functions are sufficient and whether dedicated support for streaming applications can still considerably improve performance. Thus, we revisit the data movement problem and provide a comprehensive evaluation of different mechanisms using different data paths in the Linux 2.6 kernel.

Our investigation and analysis of the different components in the Linux kernel show that data movement operation is still the main consumer of CPU cycles. We have therefore performed several experiments to see the real performance of retrieving data from disk and sending the data to a remote client, both in a content download scenario and in a streaming scenario using application level RTP packets. Additionally, we have implemented and evaluated several enhanced mechanisms, and we 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 the best existing case by removing copy operations and system calls for a given stream.

The rest of this paper is organized as follows: In section 2, we briefly look at the disk-network data path and show that data movement operations are still the main CPU cycle consumer. Section 3 gives a small overview of examples of existing mechanisms with respect to the main bottleneck, i.e., data copying and context switches. In section 4, we present the evaluation of existing mechanisms in the Linux 2.6 kernel, and section 5 describes and evaluates some new enhanced system calls improving the disk-network data path for streaming applications. Section 6 gives a discussion, and finally, in section 7, we conclude the paper.

2. Data Path

A typical layered architecture of the current operating systems' I/O-pipeline for retrieving data from disk and sending it out on a network is depicted in figure 1. The *file system* is responsible for storage and retrieval of digital data, e.g., video and audio, and provides services for executing requests from the application that retrieves data from the storage I/O-system. A streaming or download *application* handles requests from clients, fetches data from the file system and passes data to the communication system. The *communication system* provides services like error

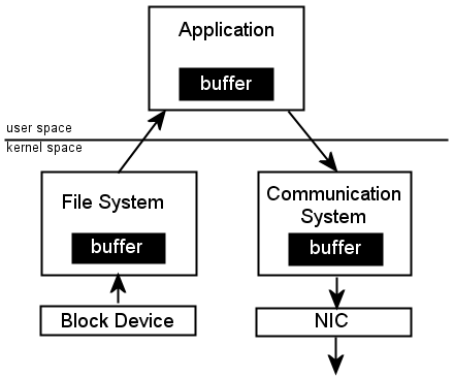


Fig. 1. Critical data path in a server.

control, flow control, etc. and manages data transfers between the end-systems over the network. Furthermore, as we can see from figure 1, each of these subsystems manage their own buffers, i.e., data is moved between the subsystems usually using data copy operations where data is physically moved through the CPU from one memory location to another.

2.1. *Profiling*

To see which system calls used which kernel functions, and to see the respective resource consumption, we made a test using `OProfile` [1], a system-wide performance profiler for Linux systems, on a machine with an Intel 845 chipset, 1.70 GHz Intel Pentium4 CPU, 400 MHz front side bus and 1 GB PC133 SDRAM. We read stored data and transmitted the data using UDP, and the results are shown in table 1.

For the *read* and *send* combination and the *mmap* and *send* combination, we see that copy operations consume almost 40 % of the CPU resources, and we also see that switching between kernel and user space is expensive. In the case of *mmap*, we also see a large overhead managing page faults. *sendfile*, on the other hand, works without data copying (and it is therefore the best alternative for content download as shown in section 4.3), but we still see some overhead in moving data for the packet headers. Thus, the layering of components in the operating system structure (see figure 1) is the reason for two large overheads (copying and switching), and we therefore look closer at these operations in the next section.

2.2. *Copying and Switching Performance*

As copying and switching performance are the main bottlenecks in download and streaming servers, we performed several measurements to see how much each of these operations costs on our test machine (see above). In figure 2, an overview of the chipset on our test machine is shown (similar to many other Intel chipsets). Transfers

Table 1. Profile of the five most expensive operations for different system calls.

system calls	samples	CPU usage in %	image name	symbol name
`read` and `send`	1281	19.4150	vmlinux	__copy_from_user_ll
	1217	18.4450	vmlinux	__copy_to_user_ll
	537	8.1388	reiserfs.ko	search_by_key
	230	3.4859	vmlinux	sysenter_past_esp
	210	3.1828	vmlinux	ip_push_pending_frames
`mmap` and `send`	2595	38.7082	vmlinux	__copy_from_user_ll
	423	6.3097	reiserfs.ko	search_by_key
	377	5.6235	vmlinux	do_page_fault
	175	2.6104	vmlinux	ip_append_data
	172	2.5656	vmlinux	ide_outb
	...			
	136	2.0286	vmlinux	sysenter_past_esp
`sendfile`	412	13.5482	vmlinux	skb_copy_bits
	178	5.8533	reiserfs.ko	search_by_key
	170	5.5903	vmlinux	ip_append_data
	165	5.4258	vmlinux	ip_push_pending_frames
	124	4.0776	vmlinux	__kmalloc

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.

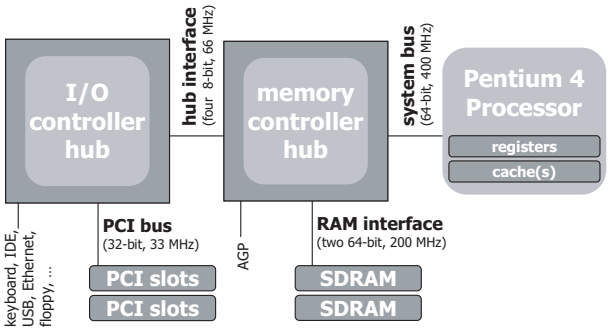


Fig. 2. Pentium4 processor and 845 chipset

Data movement performance has been measured several times, for example on Linux (2.2 and 2.4) and Windows (2000) [4]. The conclusion was that *memcpy* per-

forms well (compared to other copy functions/instructions), and Linux is in most cases faster than Windows depending on data size and used copy instruction. Therefore, to see the performance on our test machine, we tested *memcpy* using different data sizes [11]. The results confirm previous experiments [15,21,4,10] and show that the overhead is growing with the size of the data element.

With respect to switching contexts, system call overhead (and process context switches) is high on Pentium 4 [18]. 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 is approximately 920 nanoseconds for each call.

Copy and system call performance has also been an issue for hardware producers like Intel, who have 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% [15]. Thus, both copy and kernel access performance are still resource consuming and remain possible bottlenecks.

3. 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 the DASH IPC mechanism [25], Container Shipping [2], Genie [5], IO-Lite [21] and UVM virtual memory system [8] 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 *splice* system call [9], the multimedia mbuf (MMBUF) mechanism [6], the *stream* system call [19], the Hi-Tactix system [20], KStreams [16] and the *sendfile* system call (for more references, see [10]).

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 [7], event batching [22] and making application level code run in the kernel (e.g., Stream Engine [17] and KStreams [16]).

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. Only some limited support is included in the most used operating systems today, including the *sendfile* system call in Linux, AIX and *BSD, the *sendfilev* system call in Solaris and the *TransmitFile* and the *TransmitPacket* system calls in Windows. In the next section, we therefore evaluate the I/O pipeline performance of the new Linux 2.6 kernel.

4. Evaluation of 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. After a comparison of these function calls' performance in the 2.4 and 2.6 kernels in section 4.2, we present the results of our performance tests using combinations of these for **content download** operations (adding no application level information) in section 4.3 and **streaming** operations (adding and interleaving application level RTP headers for timing and sequence numbering) in section 4.4.

The experiments were performed using two machines connected by a point-to-point Ethernet connection. We used the same test machine as for the tests described in section 2. 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. We have added the user and kernel time values to get the total resource consumption, and each test was performed 10 times to get more reliable results. However, the differences between the tests are small.

4.1. *Read and Send Functions*

The functions used for retrieving data from disk into memory are usually *read* or *mmap*. Data is transferred 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. The payload is copied from the user buffer or the page cache, depending on whether *read* or *mmap* is used, to the socket buffer (*sk_buf*). Then, the

data is transferred in a DMA operation to the network device. Another approach is to use *sendfile* (see for example [24]) sending the whole file (or the specified part) 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, which are for example needed to read the payload and the generated headers that are located in different (*sk_buf*) buffers in a single operation, data can be sent from disk to network without any in-memory copy operations.

4.2. *Linux 2.4 versus Linux 2.6*

Many important changes have been made between the 2.4 and 2.6 Linux kernels, e.g., the big kernel lock (BKL), a new O(1) scheduler, the new API (NAPI), improved disk read-ahead, optimized block copy, and new versions of TCP. All these may influence the performance in our scenario. To test the real performance difference, we performed several measurements of downloading and streaming a 1 GB data file (all read from disk) using the 2.4.21 and 2.6.5 kernels, and several different combinations of system calls. As we can see from table 2, the experiments using UDP and TCP show large improvements for the new 2.6 kernel (except some cases using UDP with packets larger than the maximum transfer unit (MTU) size). For UDP, we see improvements between 5 % and 37 % for packets smaller than the MTU size, and for TCP, the respective improvements are between 6 % and 49 %. This is probably due to the improved read-ahead mechanism and optimized block copy. The tables also show that we have similar results for both content download and streaming operations.

4.3. *Content Download Experiments*

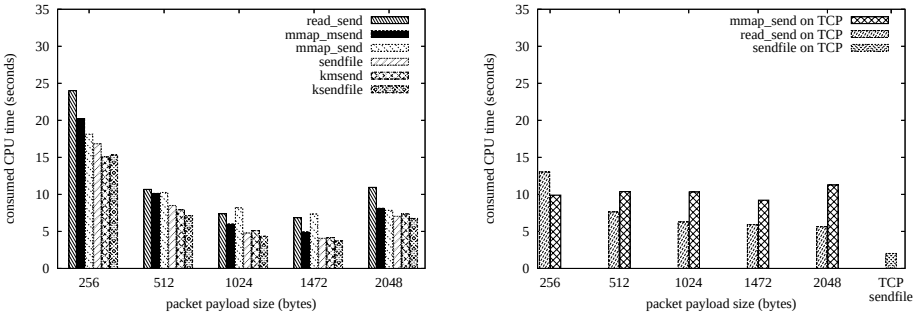
In a content download scenario, data needs only to be read from disk and sent as soon as possible without application level control. Thus, there is no need to add application level information. To evaluate the performance of the existing mechanisms in Linux in this context, we performed several tests using the different data paths and system calls described in section 4.1 and table 3 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:

- *mmap_msend* uses *mmap* to share data between file system buffer cache and the application. Then, it uses the enhanced *msend* system call to send data from the mapped file data using a virtual memory pointer instead of a physical copy. This gives no in-memory data transfers and $1 + n$ system calls.
- *kmsend* is similar to the *mmap_msend* combination using *mmap* and the enhanced *msend* system call. However, instead of making system calls for each send operation, *kmsend* manages everything in the kernel. This gives no in-memory data transfers and only 1 system call.

Table 2. Comparison of different kernels using UDP and TCP.

protocol	**operation**	**system calls**	**send size (bytes)**	**kernel 2.6.5 (seconds)**	**kernel 2.4.21 (seconds)**	**improvement in 2.6 (%)**
UDP	*download*	`read/send`	512	8.714675	11.537200	24.46
			1024	5.844012	7.357300	20.57
			2048	8.144362	5.499400	-48.10
		`mmap/send`	512	9.399971	11.038000	14.84
			1024	7.139615	7.579700	5.81
			2048	6.536707	7.431000	12.03
		`sendfile`	512	6.052180	9.030200	32.98
			1024	3.951599	6.358000	37.85
			2048	5.736328	5.331400	-7.60
	streaming	`read/send`	512	8.742471	10.918800	19.93
			1024	6.271547	7.439200	15.70
			2048	7.825210	5.336900	-46.62
		`read/writev`	512	9.351078	11.135600	16.03
			1024	6.622893	7.361700	10.04
			2048	7.602744	5.552400	-36.93
		`sendfile`	512	9.064322	13.342500	32.06
			1024	8.005983	7.638400	-4.81
			2048	7.312389	6.900000	-5.98
TCP	*download*	`read/send`	512	7.105820	10.310800	31.08
			1024	5.978091	7.733400	22.70
			2048	5.504263	6.509300	15.44
		`mmap/send`	512	7.986786	10.225300	21.89
			1024	7.827110	8.615400	9.15
			2048	7.640939	8.175900	6.54
		`sendfile`	512	2.776278	4.144200	33.01
			1024	1.996097	2.966100	32.70
			2048	1.526668	2.132500	28.41
			whole file	1.044841	1.308400	20.14
	streaming	`read/send`	512	7.381678	10.337000	28.59
			1024	5.956495	7.949900	25.07
			2048	5.355886	6.733600	20.46
		`read/writev`	512	7.790716	10.422900	25.25
			1024	6.038782	7.777100	22.35
			2048	5.441773	6.536500	16.75
		`mmap/send`	512	11.453159	19.188100	40.31
			1024	9.551148	12.779300	25.26
			2048	8.546500	10.701500	20.14
		`mmap/writev`	512	8.184156	10.611900	22.88
			1024	7.897200	9.019000	12.44
			2048	7.708428	8.373700	7.94
		`sendfile`	512	7.337885	14.399500	49.04
			1024	4.230557	7.683300	44.94
			2048	2.606704	4.421200	41.04

- *ksendfile* is a modification of *sendfile* for UDP where the kernel manages sending of all the packets instead of having one call per packet. Thus, this gives no in-memory data transfers and only 1 system call.



(a) UDP (b) TCP

Fig. 3. Content download operations

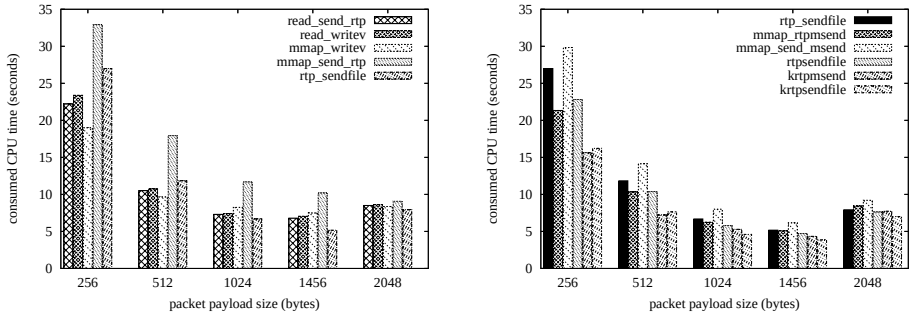
The results for UDP are shown in figure 3(a). We see, as expected, that removal of copy operations and system calls both give performance improvements. Furthermore, in figure 3(b), the results using TCP are shown. Again, we see that a quite a lot of resources can be freed using *sendfile* compared to the two other approaches that make several system calls and copy operations per data element.

Table 3. Descriptions of performed *content download* tests.

	copy calls	**user-kernel switches**	**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	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 similar to *sendfile* for TCP)

n is the number of packets
*new enhanced system call

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



(a) Existing mechanisms (b) Enhancements

Fig. 4. Streaming operations

one system call is needed to transfer the whole file.

4.4. *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 4. As shown above, the application payload can be transferred 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 4(a). 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.

5. Enhancements for RTP Streaming

Looking at the existing mechanisms described and analyzed in section 4, 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 4:

- *mmap_rtpmsend* uses *mmap* to share data between file system buffer cache

Table 4. Descriptions of performed *streaming* tests.

	copy calls	**user-kernel switches**	**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***$^{\beta}$, n ***send***, n ***send*** and n ***uncork***$^{\beta}$ 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***$^{\beta}$, n ***send***, n ***sendfile*** and n ***uncork***$^{\beta}$ calls
mmap_rtpmsend$^{\alpha}$	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$^{\alpha}$	n	$2+8n$	1 ***mmap***, n ***cork***$^{\beta}$, n ***send***, n ***msend*** and n ***uncork***$^{\beta}$ calls (no data copying using *msend*, but the RTP header must be copied from user space)
rtpsendfile$^{\alpha}$	n	$2n$	n ***rtpsendfile*** calls (*rtpsendfile* adds the RTP header copy operation to the *sendfile* system call)
krtpsendfile$^{\alpha}$	0	2	1 ***krtpsendfile*** call (*krtpsendfile* adds RTP headers to *sendfile* in the kernel)
krtpmsend$^{\alpha}$	0	2	1 ***krtpmsend*** call (*krtpmsend* adds RTP headers to the *mmap*/*msend* combination (see above) in the kernel)

n is the number of packets
$^{\alpha}$new enhanced system call
$^{\beta}$*(un)cork* is a *setsockopt* call using the *CORK* option to prevent sending incomplete segments, e.g., allowing RTP headers to be merged with the payload in a single packet

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 enhanced *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.

Here, *mmap_rtpmsend* and *krtpmsend* are targeted at applications requiring the possibility to touch data in user-space, e.g., parsing or sporadic modifications[a], whereas *rtpsendfile* and *krtpsendfile* aim at data transfers without any application-level data touching. All these enhanced system calls reduce the overhead compared to existing approaches, and with respect to overhead, *mmap_rtpmsend*, *rtpsendfile*, *krtpmsend* and *krtpsendfile* look promising. To see the real performance gain, we performed the same tests as above. Our results, shown in figure 4(b), 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% when using *krtpsendfile* where a kernel engine generates RTP headers instead of *rtp_sendfile* in the scenario with MTU-sized packets. If we to make 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 exhibit only 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*. Higher user level control that requires 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.

6. Discussion

The enhancements described in this paper to reduce the number of copy operations and system calls and their respective evaluations address mainly application scenarios where data is streamed to the client without any data manipulation at the server side. However, several of the enhanced system calls allow applications to share a buffer with the kernel and 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 question is whether data copying is still a bottleneck in systems today. The hardware has improved, and one can easily find other possible bottle-

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

neck components. For example, I/O devices such as disks and network cards are potential bottlenecks. In our context, most focus has been on storage systems where a large number of mechanisms targeted at specific applications has been proposed including scheduling, data placement, replication, prefetching (see for example [12,13] for an overview). Nevertheless, as described in section 2.2, 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. Thus, reducing the number of consumed CPU cycles per operation can a have large impact on system performance in several contexts (see section 1). Making resources available for other tasks or enabling the use of smaller, less power consuming processors for the same tasks.

Our results show that large improvements are possible making only small changes in the kernel. Figure 5 (note that the y-axis starts at 0.5) shows the performance of the different RTP streaming mechanisms relative to *read_writev*, i.e., a straight forward approach for 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 to *rtp_sendfile* which requires several of both (see table 3). With respect to consumed processor time, we achieve an average re-

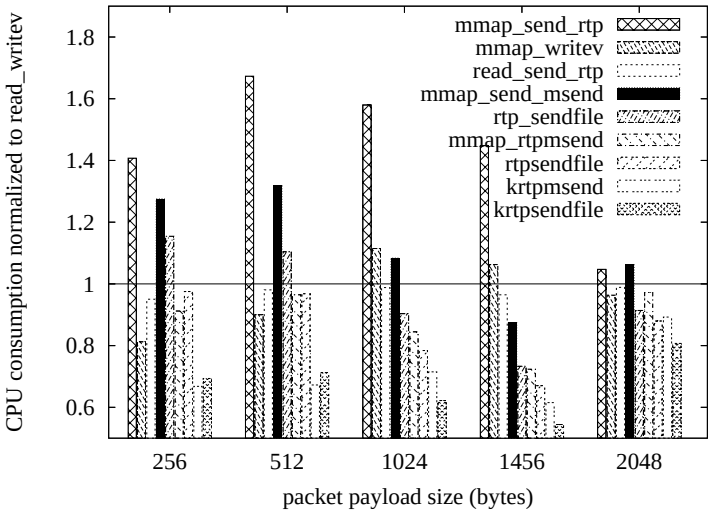


Fig. 5. Relative performance to *read_writev*

duction 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 are still potential bottlenecks and having looked at possible enhancements, let us look at what a general purpose operating system like Linux misses. Usually, the commodity operating systems aim at generality, and new system calls are not frequently added. Thus, specialized mechanisms like *krtpsendfile* and *krtpmsend* that require 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, similar to the Red Hat Content Accelerator (*tux*) [23] 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 take off. Simple enhancements like *mmap_rtpmsend* and *rtpsendfile* are generally useful, performance improving mechanisms that could be of interest in scenarios where the application does or does not touch the data, respectively.

7. 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 for applications that require packetization and the addition of headers. We have therefore implemented several enhancements to the Linux kernel, and evaluated both existing and 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 that generates 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 either allows a reduced power consumption of the sending system, better scalability, or a utilization of freed resources for other tasks.

Currently, we have other on-going kernel activities and try to offload a node using programmable sub-systems [14], and we hope to be able to integrate all of our subcomponents. We will modify the KOMSSYS video server [26] 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 only implemented as proof-of-concepts for 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. OProfile. http://oprofile.souceforge.net, September 2006.
2. 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, July 1995.
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. 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.
5. Jose Carlos Brustoloni. Interoperation of copy avoidance in network and file i/o. In *Proceedings of the Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 534–542, March 1999.
6. Milind M. Buddhiko, Xin Jane Chen, Dakang Wu, and Guru M. Parulkar. Enhancements to 4.4bsd unix for efficient networked multimedia in project mars. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, pages 326–337, June 1998.
7. Charles Coffing. An x86 protected mode virtual machine monitor for the mit exokernel. Master's thesis, Paralell & Distributed Operating System Group, MIT, May 1999.
8. Charles D. Cranor and Gurudatta M. Parulkar. The UVM virtual memory system. In *Proceedings of the USENIX Annual Technical Conference*, pages 117–130, June 1999.
9. Kevin Roland Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve i/o throughput and cpu availability. In *Proceedings of the USENIX Winter Technical Conference*, pages 327–333, January 1993.
10. Pål Halvorsen. *Improving I/O Performance of Multimedia Servers*. PhD thesis, Department of Informatics, University of Oslo, Norway, August 2001. Published by Unipub forlag, ISSN 1501-7710 (No. 161).
11. Pål Halvorsen, Tom Anders Dalseng, and Carsten Griwodz. Assessment of data path implementations for download and streaming. In *Proceedings of the International Conference on Distributed Multimedia Systems (DMS)*, pages 228–233, September 2005.
12. Pål Halvorsen, Carsten Griwodz, Vera Goebel, Ketil Lund, Thomas Plagemann, and Jonathan Walpole. Storage system support for continuous-media applications, part 1: Requirements and single-disk issues. *IEEE Distributed Systems Online*, 5(1), 2004.
13. Pål Halvorsen, Carsten Griwodz, Vera Goebel, Ketil Lund, Thomas Plagemann, and Jonathan Walpole. Storage system support for continuous-media applications, part 2: Multiple disks, memory, and integration. *IEEE Distributed Systems Online*, 5(2), 2004.
14. Øyvind Hvamstad, Carsten Griwodz, and Pål Halvorsen. Offloading multimedia prox-

ies using network processors. In *Proceedings of the International Network Conference (INC)*, pages 113–120, July 2005.
15. Intel Corporation. Block copy using PentiumIII streaming SIMD extensions (revision 1.9). ftp://download.intel.com/design/servers/softdev/copy.pdf, 1999.
16. 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.
17. Jonathan Lemon, Zhe Wang, Zheng Yang, and Pei Cao. Stream engine: A new kernel interface for high-performance internet streaming servers. In *Proceedings of International Workshop on Web Content and Caching (IWCW)*, pages 159–170, September 2003.
18. Gregory McGarry. Benchmark comparison of netbsd 2.0 and freebsd 5.3. http://www.feyrer.de/NetBSD/gmcgarry/, January 2005.
19. 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, January 1998.
20. 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, December 2002.
21. 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.
22. 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*, November 2000.
23. Red Hat. Red Hat content accelerator. http://www.redhat.com/docs/manuals/tux/, 2002.
24. Dragan Stancevic. Zero Copy I: User-mode perspective. *Linux Journal*, 2003(105), January 2003.
25. Shin-Yuan Tzou and David P. Anderson. The performance of message-passing using restricted virtual memory remapping. *Software - Practice and Experience*, 21(3):251–267, March 1991.
26. Michael Zink, Carsten Griwodz, and Ralf Steinmetz. KOM player - a platform for experimental VoD research. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 370–375, July 2001.