

Stream Engine: A New Kernel Interface for High-Performance Internet Streaming Servers

Jonathan Lemon, Zhe Wang, Zheng Yang and Pei Cao
Cisco Systems, Inc.
{jlemon, zwang, zyang, cao}@cisco.com

Abstract

As high-speed Internet connections and Internet streaming media become widespread, the demand for high-performance, cheap Internet streaming servers increases. In this paper, we look into the performance limitations of streaming server applications running on PC servers with Linux, and propose a new kernel optimization called “stream engine” that combines both copy elimination and context switch avoidance to double the streaming server throughput. Our experiments with stream engine show that for Internet streaming, eliminating context switches is just as important as eliminating data copying. Using profile data, we also project the benefits of TCP offloading hardware implementing part or all of the stream engine optimization.

1 Introduction

As the adoption of broadband connections accelerates over the years, the demand and consumption of streaming media on the Internet also increase rapidly. The result is a constant increase in the throughput requirement of streaming servers.

Today’s Internet streaming servers are mostly built with general-purpose servers. Unlike traditional VoD applications, Internet streaming protocols run over standard IP infrastructure and use TCP/UDP as the underlying data transport. The streaming protocols themselves undergo constant changes as vendors seek to perfect end-user experiences. Because of frequent changes in the

protocols, streaming server vendors so far have avoided building special-purpose hardware for Internet streaming. Rather, PC-based servers, because of their low price/performance and good development environment, have been the platform of choice.

Unfortunately, PC-servers running generic operating systems are not well-suited for streaming workloads, which are characterized by high data rates and high protocol processing overhead. This paper demonstrates the limitations of traditional OS kernels for streaming servers through performance profiling of a server implementation, and proposes a new kernel interface, “Stream Engine”, to address these limitations.

The novelty of the “stream engine” interface lies in its combination of data-copy avoidance and context-switch avoidance for applications like Internet streaming. While reducing data copying is a well-known technique to improve server throughput, it was only through implementation experiments that we learnt the significant overhead of user-kernel context switches in Internet streaming applications. The user-level context switches cannot be avoided by any application architecture change (for example, changing from a one-stream-per-thread server implementation to an event-driven server implementation). Rather, a new kernel mechanism and interface are needed.

We have implemented “stream engine” in the Windows Media streaming server running on a PC. The result is a two-fold increase in streaming server throughput, up to 1.2Gbps on current-generation dual-processor 1.7Ghz Pentium servers. In con-

trast, eliminating data copies by itself would only improve throughput by 30-40% (based on our experiments and back-of-envelope calculations). As a result, the “stream engine” optimization is incorporated in the Content Engine appliances, a shipping product in Cisco’s Content Networking line of products.

Our experience with streaming engine also provides inputs on the requirements and interfaces for TCP-offloading hardware for Internet streaming. Most of the TCP offloading hardware devices developed today are for iSCSI acceleration [Inc02a, Inc01, Inc02b, Int02], and the interfaces are oriented toward SCSI-based storage. We show that for Internet streaming, the benefits from conventional TCP offloading engines are limited. Rather, TCP offloading hardware needs to interface with stream engine to offload packet assemblies and data transmissions from file buffer cache, and TCP offloading hardware that implements stream engine entirely on board can potentially quadruple the server throughput.

2 Characteristics of Internet Streaming Protocols

There are three main streaming protocols used on the Internet today: Windows Media Technology (WMT) from Microsoft, RealNetwork Streaming from RealNetworks, and QuickTime streaming from Apple Computers. Certain major content providers such as AOL and Yahoo also have their own streaming protocols. Despite their differences, the streaming protocols share some common characteristics.

Virtually all of the protocols utilize both a control connection and a data connection. Over the control connection, the media player sets up a streaming session with the media server, and passes along streaming control messages such as pause, fast-forward and rewind. In protocols that support variable bit rate streaming (i.e. adjusting streaming quality based on available bandwidth), the control connection is also used by the client to inform the server of changes in available bandwidth in the

network. The control connection is typically a TCP connection established by the client. In the case of RealNetwork streaming and QuickTime streaming, the control connection uses the IETF standard RTSP (Real-Time Streaming Protocol) protocol [SRL98] with vendor-specific extensions.

The data connection is used to transmit streaming media data from the server to the client. It can be based on either UDP or TCP. In virtually all streaming protocols, the data transmissions are “paced”, that is, the data are not sent to the client all at once, but rather sent in blocks with delays between the blocks. The delays are determined by either the bit rate of the stream or the properties of the encoding scheme. The “pacing” of data transmissions is necessary to avoid overflowing the client-side buffer, particularly in the case of high-speed Internet connections. Yet, the “pacing” causes many user-kernel context switches.

Almost all protocols share the following usage patterns:

- The control connection is mostly idle during the streaming session while the data connection has packets going from the server to the client in a periodic, repetitive fashion;
- Most of the processing logic is devoted to the messages sent over the control connection, while most of the traffic is caused by data blocks traveling on the data connection;
- A message over the control connection almost always changes the behavior of the streaming server on the data connection;
- The processing logic for the control messages typically does not scan the streaming data.

For example, a typical streaming protocol behaves as the following. When a client wants to view a streaming video, it initiates a TCP connection to the streaming server. The TCP connection is the control connection; the client sends the player version info, the URL of the streaming media, and client credentials to the server over this connection. There maybe challenges/responses between the client and the server until the server authorizes

the client. Afterward, the server extracts the properties of the streaming media such as length and bit rates from the media file and sends the properties to the client. The client and the server then negotiate a suitable bit rate to serve the media to the client.

The server then initiates a UDP data stream (the data “connection”) to the client, and starts sending streaming data in blocks to the client. A fixed-duration delay exists between transmissions of data blocks to “pace” the transmission. If network conditions change and the client wants to reduce the bit rate of the stream, the client sends a message over the control connection, and the server changes the composition of the data blocks as a result, sending frames encoded in a lower bit-rate.

3 Implementation Experience

Our project is to implement a high-performance streaming server on PC-server architectures. We started with a quick initial prototype and then tried to identify and eliminate the performance bottlenecks.

3.1 Initial Prototype

We implemented the above-described streaming protocol on PC servers running Linux 2.4.16. In the initial prototype, for “time-to-market” reasons, we adopted a one-stream-per-process programming model. (In Linux 2.4.16, threads have the same kernel overhead as processes, so a one-stream-per-thread model would have similar performance.) This programming model is much easier to code and debug than the event-driven model in which one process serves many clients and processes messages from all clients.

In the prototype, a parent process listens on the designated port and waits for incoming control connections. For each new client it spawns a child process to handle the client. The child process first performs the normal session exchanges with the client, then initiates the data connection to the client. Afterwards, the child process sends the first

block of streaming data to the client, sleeps for a fixed duration, then sends the second block, and the process repeats. The sleep time is calculated using the block size and the bit rate of the streaming media.

The child process also monitors incoming messages over the control connection from the client. If a “seek”, “fast-forward”, or “rewind” message arrives, the process adjusts the starting location of the data block in the streaming file. If a “bit rate upgrade” or “bit rate downgrade” message arrives, the process changes the frame-selection procedure and selects frames encoded in different bit rates from the streaming media file.

While the implementation was straightforward and robust, its performance is less than desirable. On our initial experimental platform, which is a dual-processor 866MHz Pentium III PC server with 2GB of RAM and 20 disks, the throughput is only around 150Mb/s.

3.2 Identifying Performance Bottlenecks

We then embarked on the investigation to improve server performance. The first observation was that the system was limited by disk reads. It was suggested to us that perhaps a specially designed storage system is needed, instead of the regular UNIX file system that we have been using. Upon closer inspection, however, we realized that streaming files are accessed sequentially most of the time, which means that prefetching and large “disk allocation block size” can be very effective. Furthermore, servers today have a large amount of RAM and can easily afford a sizable file buffer cache for the video files. Since streaming workload tends to be heavily skewed [dMSK02], a large main memory cache can be quite effective at reducing the number of disk reads. Hence, we hypothesized that, combining the sequential access pattern and the large memory cache, a properly tuned UNIX file system implementation can work reasonably well for video files.

We went ahead and tuned the file system pa-

rameters, increasing the continuous-allocation size in the file system (i.e. the “extent-size”) to 128KB and the pre-fetch size to 128KB. The tuning improved the server throughput to around 400Mb/s, under the workload where 100 video files are streamed concurrently, 10 on each disk. Each disk spindle can in fact support 120Mb/s of read throughput on those video files, rivaling the per-spindle throughput of some of the specially-designed streaming storage systems.

With the improvement in disk throughput, the system became CPU bound. We considered changing the programming model to event-driven, as doing so would reduce the number of processes and the associated kernel process-scheduling overhead. However, changing the programming model is a significant effort, and before we plunge into that, we need to determine the potential payoff of the effort.

After noticing that most of the CPU is consumed by the kernel, we focused on performance profiling of the kernel using “kernprof” [Sou02], the Linux kernel profiling tool. Kernprof uses the PC-sampling technique; for each kernel routine, it lists the number of times that a PC sample falls in that routine.

Table 1 shows the kernprof results of the top 10 routines from a typical streaming load test. During the test the system is serving 720 streams of a 500Kbps media file. Eight of the routines listed in the table are kernel code, and they account for 52% of the CPU usage.

From Table 1, we can see that data copying, which involves reading the file data from kernel space to user space (i.e. `file_read_actor`) and sending the data from the user space to kernel space (i.e. `csum_partial_copy_generic`), accounts for 33% of the CPU. Thus, if the data copies are eliminated completely, the server throughput can be improved by 33%.

This “back-of-envelope” calculation correlates with another experiment that we run. We changed the streaming server to use Linux’s “sendfile” interface to send the streaming data blocks. Since Linux’s “sendfile” implementation still incurs a copy from the file buffer cache to the network

socket buffer (in other words, it eliminates just one instead of two copies), we expect to see half of the benefit of zero data copying. Indeed, the throughput improvement that we observed is between 15% and 20%.

The kernel’s process scheduling routine, “schedule”, which is the one most affected by the large number of processes, only accounted for 9% of the CPU. This significantly dampened our enthusiasm for changing the programming model to event-driven. Even if the application is changed to an event-driven programming model and uses only one process, the kernel’s CPU overhead would only be reduced by 9%. Of course, this is not a rigorous analysis; depending on how the event-driven application is constructed, it might have other benefits such as improved process cache hit ratios, etc. Nevertheless, given the significant coding effort involved in changing the application to event-driven model, it’s clear that we should focus on eliminating data copying and other system bottlenecks instead of changing the programming model.

One aspect that the profiling data do not fully capture is the impact of user-kernel context switches on the system throughput. Even though researchers have long pointed out that context switches can be made cheap [Lie93], expensive context switches are a fact of life for PC architectures running Linux. During the system’s run-time we noticed that the number of context switches are high, over 10000 per second. This is due to the fact that the streaming server’s main operation mode is to send a data block, sleep for some duration, wake up and send another one. Each data block transmission, sleep and wakeup are context switches between user-level and kernel.

Are there ways to eliminate the context switches? We notice that in the streaming server implementation, the actions of each child process are quite monotonic and repetitive, with few sophisticated processing logic. Hence, one possibility is to construct a state-machine for each child process to capture the monotonic operations, and “drop” it into the kernel. Combining this with data copy elimination could provide a significant performance boost.

Routine	Sampling Count	Percentage	Cumulative Percentage
file_read_actor	16508	18.6%	18.6%
csum_partial_copy_generic	12435	14.0%	32.6%
default_idle	11271	-	-
schedule	8141	9.1%	41.7%
USER	6592	7.4%	-
tcp_sendmsg	1868	2.1%	43.8%
_kfree_skb	1758	1.9%	45.7%
ace_interrupt	1721	1.9%	47.6%
skb_release_data	1386	1.5%	49.1%
ip_fw_check	1347	1.5%	51.6%

Table 1: Results from kernprof on the top 10 routines. The total sample count is 100000. The first column lists the routine name; USER means user-level code, and default_idle means CPU idle time. In this particular test, since we did not have enough client machines to drive the server to maximum load, the server’s CPU is about 11% idle, as reflected in the sampling count for the routine “default_idle”. We have found from other full-load tests that the system is indeed CPU bound. The second column is the total sample count of the routine as reported by kernprof. The third column is the percentage of the CPU usage that the routine incurs. The last column counts kernel routines only and is the cumulative sum of the CPU usage incurred by the kernel routines.

Furthermore, such optimization is not limited to a particular streaming protocol. As discussed before, all streaming protocols share the common characteristics of repetitive data transmissions. Hence, the optimization applies to all streaming protocols. We call this optimization the “stream engine”.

4 The Stream Engine Interface

We propose an optimization technique called “Stream Engine” for Internet streaming applications. This technique has been implemented as a kernel-level mechanism in our streaming server, and is suitable as an interface for TCP-offloading hardware for Internet streaming.

4.1 Definitions of a Stream Engine

A stream engine encapsulates the operations that the kernel (or potentially a TCP-acceleration hardware device) does for a streaming media stream. It is defined by a “context”, which contains all the necessary information for the engine to perform its

tasks. The user-level application passes the following pieces of information to the kernel, which stores them in the context of the stream engine:

- Data connection description: the transport mode (UDP or TCP) of the connection, and the file descriptor of the connection;
- Streaming source description: there are two types of streaming source:
 - type FILE, which contains the file descriptor of the streaming media file, the starting and stopping offsets (in bytes) of the data to be sent to the client;
 - type BUFFER, which contains the memory address of the buffer, length of the buffer, a flag indicating if the buffer is a circular buffer, and in the case of circular buffer, a user-level variable that is the end position of useful data in the buffer.

Type FILE sources are used when streaming from an encoded streaming media file. Type BUFFER sources are used when streaming from a live source, in which case the encoded

stream data are fed to the streaming server and stored in a user-level buffer by the receiving process.

- Transmission interval in milliseconds. The stream engine will always transmit one data block per interval.
- Data block assembly instructions, which tell the engine how to assemble data packets from the file data or buffer data. Clearly, the instructions depend on the streaming protocol and the encoding format. There can be two kinds of instructions:
 - “simple stepping”, which means that the stream engine should send a fixed-size block of data at every interval with a certain stride between each block. The description includes the block size and the stride size.
 - a data transformation routine: the stream engine will read a block of data from the stream source into a buffer, then apply the data transformation routine to obtain a new buffer to be sent to the client.

Simple stepping is used with encoding schemes that encode the streaming media file into equal-sized blocks. Some popular streaming media formats follow this scheme.

The data transformation routine is versatile and can support any encoding scheme. In our implementation, we use it primarily for streaming quality adaptation, where the routine extracts the correct frames from the data blocks and assembles them to be sent to the client. Due to limited resources, we did not look into mechanisms to support arbitrary data transformation routines in a safe manner (such as those proposed in SPIN [BSP⁺95]), but rather coded a few common ones for the streaming application to choose from.

- File descriptor of the control connection. Any incoming message on the connection terminates the stream engine’s operation immedi-

ately, and the kernel returns to the application for further instructions.

We implemented stream engine as a blocking system call. The system call takes as an argument a data structure “stream_eng_ctx” which encloses the above pieces of information. The call returns when the streaming operation completes or when a message arrives at the control connection. The return parameters include the reason that the system call returned, and the total number of data blocks sent to the client.

In other words, the stream engine runs in the kernel until some action needs to be taken on the stream (e.g. a control message arrives from the client to change the stream rate or position), at which point it terminates and the call returns to the user application. The user application can then initiate a new stream engine call. Hence, changes in a video stream session, for example, changing the transmission rate, is accomplished by having a message sent to the control file descriptor to terminate the current stream engine, and then initiating a new stream engine.

In our streaming server implementation, the child process initiates a stream engine by calling the system call, when it is ready to start streaming data to the client. The child process then waits for the call to return. After the call returns, depending on the reason, the child process either processes the control messages and then starts a new stream engine, or terminates the stream and cleans it up.

In our current design, the admission control of the streams is handled by the user level application, and the kernel does not perform further resource checking. The reason is that in the streaming server, the admission control is not purely resource based, but rather depends on licensing terms and network configuration. However, the design does have the drawback that if the user application is not careful and over-commits the resource, the stream engine would not be able to pump data out at the specified data rate. In future versions we plan to investigate adding resource checks in the kernel for stream engines.

4.2 Performance Improvements from Stream Engine

We applied the stream engine optimization to the streaming server described in Section 3. The end result is that the throughput is almost doubled. While the old implementation can service 720 streams of 500Kbps media at 90% of the system capacity, the new implementation can service 1360 streams of 500Kbps media at 90% of the system capacity. The improvement showed that eliminating context switches leads to about 60% increase in system throughput.

Table 2 lists the top routines from the kernel profile result for the new implementation. The top nine kernel routines account for about 50% of total CPU used. Comparing Table 1 and Table 2, one can see that stream engine eliminates the data copies and the context switch and process scheduling overhead. Instead, the overheads of checksum calculation (it has to be done somehow), timer handling (due to sleep/wakeup of the stream engines) and the NIC processing routines become more prominent. Some of these overheads can be alleviated by the use of TCP offloading hardware.

On a new state-of-art PC server with dual-processor 1.7Ghz Pentium system with 2GB of RAM and 20 disks, the streaming server with stream engine optimization can achieve 1.2Gbps throughput on streaming 300Kbps media files.

5 Implications for TCP Offloading Hardware

Recently there have been a number of commercial TCP Offloading Engine (TOE) products [YCM⁺02, Inc02a, Inc01, Inc02b, Int02, Mog03]. Their primary target is the IP storage market (e.g. iSCSI), and the goal is to offload TCP/IP processing from the host CPU. Some of the TOE products are implemented using network processors, while others are implemented using ASICs. The TOE is usually embedded in a Network Interface Card (NIC) or a Host Bus Adapter (HBA).

We are investigating the use of TOE cards to improve the throughput of Internet streaming servers. Internet streaming, however, is different from IP storage and has quite different requirements.

Based on our experience, we believe that stream engine is an appropriate interface for TCP offloading engines. For Internet streaming applications, just offloading the processing of TCP/IP from the host computer brings limited benefits; it will not eliminate the data copying and the context switches. A full-fledged implementation of stream engine will bring the most performance benefits. Short of that, a TOE card must be able to work with the host system kernel's stream engine implementation. By this, we mean that the TOE's interface with the host system must support sending data from a linked-list of data buffers in the file buffer cache, with a starting offset and size of the data, and implementing the packet assembly routine on the file buffer data.

We illustrate our observations using back-of-envelope calculations based on full kernel profile results from our experiments:

- Using a TOE that does not work with stream engine. We calculate the benefit of a conventional TOE NIC that simply offloads the TCP/IP protocol processing from the host CPU. Assuming that the TOE offloads all checksum calculations, TCP stack operations, IP stack operations, NIC transmission and interrupt processing, the throughput can be improved by about 25%.
- Using a TOE that works with stream engine. If stream engine is implemented in the host system's kernel, and the TOE card can work with the implementation to offload the TCP/IP send/receive processing, checksum calculation, and NIC interrupts, the throughput can be improved by up to 60%. That is, in addition to the improvement from the stream engine implementation, the system's throughput can be improved further by 60%.
- Using a TOE that implements stream engine. If the TCP offloading hardware is powerful

Routine	Sampling Count	Percentage	Cumulative Percentage
stream_engine_op	16896	18.6%	18.6%
csum_partial	12075	13.3%	31.9%
default_idle	9343	-	-
timer_bh	2659	2.9%	34.8%
ace_interrupt	2657	2.9%	37.7%
__kfree_skb	2509	2.7%	40.4%
skb_clone	2297	2.5%	42.9%
kfree	2277	2.5%	45.4%
ace_start_xmit	2212	2.4%	47.8%
add_timer	1909	2.1%	49.9%

Table 2: Kernprof results from streaming server using stream engine. The total CPU sample size is 100000. The columns are similar to those in Table 1. Note that again the server is not driven to full load in this test.

enough to implement the entire stream engine, we find that the load on the host CPU is reduced by 75%. In other words, if, in addition to TCP offloading, the TOE card also handles the stream engine logic including the timer processing, socket buffer management and disk read processing, then the host CPU only needs to handle a quarter of load. Assuming that the TOE card has enough parallel processing power to keep up with the host CPU, this implies that the system throughput can be potentially quadrupled.

In summary, any TOE for Internet streaming application needs to work with stream engine either as part or all of implementation. The above projections assume that the TOE card’s CPU is powerful enough to keep up with the host CPU while offloading the work of the stream engine. In reality, the CPUs on the TOE cards are typically slower than the host CPU, and multiple TOE cards, each handling a subset of the streams, would be needed for maximum throughput.

6 Related Work

Research has long pointed out the need to eliminate data copying when building high-performance network servers [Dru94]. Many kernel mechanisms and interfaces have been proposed to address

data copy elimination, including fbufs [DP93], I/O Lite [PDZ00] and sendfile [mp00]. Our work confirmed the importance of copy elimination in high-performance streaming servers. The existing schemes however do not alleviate the extensive amount of context switches incurred by streaming applications. We showed that for streaming protocols, eliminating context switches is equally important.

Recently there have also been a number of projects looking at building high-performance streaming servers using either specialized operating systems or specialized hardware. For example, “Hi-Tactix” is a kernel built by Hitachi system development laboratory that is a streaming oriented real-time OS providing high I/O performance and transmission rate guarantees [Moa02]. We are also aware of at least one small company building blade servers with specialized I/O backplane for Internet streaming purposes. Different from these efforts, we focus on improving streaming throughput on Linux by introducing a generic kernel optimization, and we are mainly interested in generic PC servers.

The debate between event-driven programming model and one-stream-per-process programming model is not limited to streaming servers. For example, it’s well known that an event-driven Web server such as Flash [PDZ99] performs much better than a one-client-per-process Web server such as Apache [Com03]. However, event-driven pro-

gramming models are typically harder to develop and harder to maintain. Our experience with the streaming server prototype and stream engine seems to suggest that, at least in the case of Internet streaming applications, changing the programming model to event-driven would provide limited benefits, while eliminating data copies and context switches provides much higher payoff and entails much shorter development time. We note that recently other researchers have arrived at similar conclusions [vBCB03].

Extensible operating systems [BSP⁺95, KEG⁺97, MMO⁺94, LMB⁺96] would make it much easier to implement optimizations such as stream engine. For example, the SPIN operating system provides a ready mechanism for user applications to drop arbitrary data transformation routines into the kernel. Unfortunately, these operating systems are not yet suitable for commercial products. Hence, we designed stream engine for Linux to accommodate streaming server specific optimizations.

Finally, the design of better streaming protocols over the Internet is an active research area which this paper is not focused on. Rather, our study looks into what it takes for a server to perform well for existing Internet streaming protocols. We believe that the stream engine interface will benefit future variations of streaming protocols, as eliminating data copies and context switches is key to high throughput for many networking applications.

7 Conclusion and Future Work

In this paper, we introduce the “stream engine” kernel interface, and show that it is a suitable kernel optimization for Internet streaming applications. We have implemented the stream engine mechanism in the kernel and have converted one streaming server implementation to use it. We are currently in the process of converting other Internet streaming protocol implementation to use stream engine.

Our experience shows that in addition to data copying, user-kernel context switches can also re-

duce system throughput significantly, particular for applications that incur frequent context switches due to “paced” data transmissions. We are currently looking into whether stream engine can benefit other non-streaming protocols that incur “paced” data transmissions, and whether the interface needs to be further extended to support those protocols.

We are also working with TOE vendors to investigate having the TOE interface with a host system’s stream engine implementation. In particular, we are looking into the partition of responsibilities between the host system and the TOE hardware and the kernel/hardware interfaces in this case.

References

- [BSP⁺95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.
- [Com03] Open-Source Community. The apache software foundation, 2003.
- [dMSK02] Jacobus Van der Merwe, Subhabrata Sen, and Charles Kalmanek. Streaming video traffic: Characterization and network impact. In *Proceedings of 7th International Workshop on Web Content Caching and Distribution*, August 2002.
- [DP93] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.
- [Dru94] P. Druschel. Operating systems support for highspeed networking, 1994.
- [Inc01] WindRiver Systems Inc. Complete tcp/ip offload for high-

- speed ethernet networks. In *http://www.windriver.com/products/tina/tcpip_offload.pdf*, September 2001.
- [Inc02a] Alacritech Inc. The 1000x1 internet protocol processor: Patented, third-generation tcp/ip offload engine (toe) asic. In *http://www.alacritech.com/html/081902a.html*, August 2002.
- [Inc02b] Trebia Networks Inc. Snp-1000i dual port tcp offload engine. In *http://www.trebia.com/products/snp_1000i.shtml*, October 2002.
- [Int02] Intel. Intel pro/1000t ip storage adaptor. In *http://www.intel.com/network/connectivity/resources/doc_library/data_sheets/pro1000_T_IP_SA.pdf*, February 2002.
- [KEG⁺97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.
- [Lie93] Jochen Liedtke. Improving ipc by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, December 1993.
- [LMB⁺96] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.
- [MMO⁺94] Allen Brady Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system (abstract). In *Operating Systems Design and Implementation*, page 200, 1994.
- [Moa02] Damien Le Moai. Cost-effective streaming server implementation using hi-tactix. In *ACM Multimedia 2002*, 2002.
- [Mog03] Jeff Mogul. Tcp offload is a dumb idea whose time has come. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.
- [mp00] Linux man page. Linux system call sendfile(2). In *http://www.die.net/doc/linux/man/man2/sendfile.2.html*, 2000.
- [PDZ99] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server, 1999.
- [PDZ00] 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, 2000.
- [Sou02] SGI Developer Central Open Source. Linux kernprof (kernel profiling). In *http://oss.sgi.com/projects/kernprof/*, 2002.
- [SRL98] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (rtsp). Technical Report RFC-2326, IETF Network Working Group, 1998.
- [vBCB03] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on*

Hot Topics in Operating Systems (HotOS IX), 2003.

- [YCM⁺02] Eric Yeh, Herman Chao, Venu Manem, Joe Gervais, and Bradley Booth. Introduction to tcp/ip offload engine (toe). In http://www.10gea.org/SP0502IntroToTOE_F.pdf, April 2002.