High-Speed I/O: The Operating System As A Signalling Mechanism

Position paper

Matthew Burnside
Computer Science Department
Columbia University
mb@cs.columbia.edu

Angelos D. Keromytis
Computer Science Department
Columbia University
angelos@cs.columbia.edu

ABSTRACT

The design of modern operating systems is based around the concept of memory as a cache for data that flows between applications, storage, and I/O devices. With the increasing disparity between I/O bandwidth and CPU performance, this architecture exposes the processor and memory subsystems as the bottlenecks to system performance. Furthermore, this design does not easily lend itself to exploitation of new capabilities in peripheral devices, such as programmable network cards or special-purpose hardware accelerators, capable of card-to-card data transfers.

We propose a new operating system architecture that removes the memory and CPU from the data path. The role of the operating system becomes that of data-flow management, while applications operate purely at the signaling level. This design parallels the evolution of modern network routers, and has the potential to enable high-performance I/O for end-systems, as well as fully exploit recent trends in programmability of peripheral (I/O) devices.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Input/Output

General Terms

Design, Security, Performance, Languages.

Keywords

Operating Systems, Data Streaming, Architecture

1. INTRODUCTION

Modern computing has become network-centric, with the utility of certain services measured in terms of aggregate throughput to the clients. Such applications include web-based services, database query processing, video and audio on demand, *etc.* The common characteristic of all these applications is their demand on all aspects of the server system, *i.e.,* hard drives, memory, processor, internal bus, network interface, *etc.* Furthermore, because of the computational complexity in some of these services' components, hardware accelerator cards are often employed to improve performance. Examples of such cards include SSL/TLS/IPsec accelerators, general-purpose cryptographic engines, MPEG encoders, digital signal processors (DSPs), and so on.

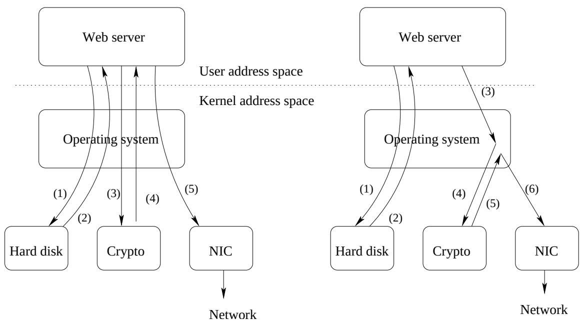

Figure 1: Current operating system I/O design. Typical operating systems follow the design on the left, which gives applications complete control over data-flow at the expense of performance. Occasionally, operating systems take advantage of specific features (such as a crypto accelerator in the network stack) in a semi-transparent way, as shown on the right.

From the operating system point of view, these devices are viewed as data transformation units. The role of the operating system then is to create a virtual pipeline of such devices to achieve the application's goals. This can be achieved in one of two ways, shown in Figure 1. In a typical operating system design, shown on the left, the operating system exports device functionality directly to the application, which is responsible for managing the flow of data among the various peripherals. While this is perhaps the most flexible approach, it suffers from two obvious performance bottlenecks: data must be copied several times between the peripherals and main memory, often over a shared bus (*e.g.,* PCI), and the same data must be copied between the application and kernel address space, also requiring two context switches per data unit. The *effective* bandwidth of the I/O bus is $1/n$ where n is the number of peripherals the flow of data must pass through. In slightly more advanced designs, shown on the right sub-fi gure, the operating system directly takes advantage of hardware accelerators transparently. In this case, the application is responsible for providing hints as to the desired processing. We describe such a design in Section 3. The advantage of this approach is that it reduces the amount of memory-to-memory

ACM SIGCOMM 2003 Workshops, August 25–29, 2003, Karlsruhe, Germany.
Copyright 2003 ACM 1-58113-748-6/03/0008 ...$5.00.

data copying and minimizes context switches. Note also that this approach can make efficient use of programmable peripherals, but does not *depend* on them.

However, in both designs, the use of main memory and CPU cycles are the limiting factor [61]. This is a fundamental limitation in the current scheme of composing different devices and it calls for a new I/O-centric operating system architecture that moves away from the concept of "main memory as data repository", shown in Figure 2. Such a rethinking is made all the more necessary by the increasing divergence between memory and CPU/network speeds, as shown in Figure 3.

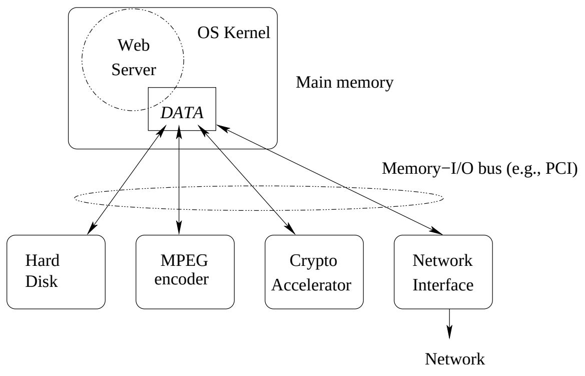

Figure 2: Main memory as data repository.

We propose a new operating system architecture, shown in Figure 5. In our scheme, the operating system becomes a manager of data flows among the various peripheral devices. Thus, it subsumes a role equivalent to that of a network switch — a shift from the early-router design that was also driven by the need for higher performance. Applications are limited exclusively to the signaling plane. They are responsible for initializing and tearing down flows, specifying the performance policies that govern these flows, and exception handling. The operating system must implement these policies by acting as a resource scheduler. The actual data transfers occur directly between cards.

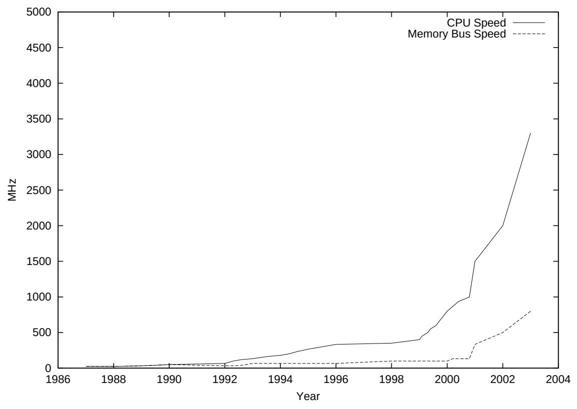

Figure 3: CPU speed *vs.* memory speed (x86 processor series) Note that the high-end memory buses (333-800MHz) have a throughput which is 1/3 of their rate speed. *Source: http://www.intel.com/*

We define a *flow* to be any stream of data, as small as a single packet. However, since there are costs associated with creating and tearing down a flow, realistically, we will only discuss those flows where the number of packets is high enough to make the flow creation cost negligible.

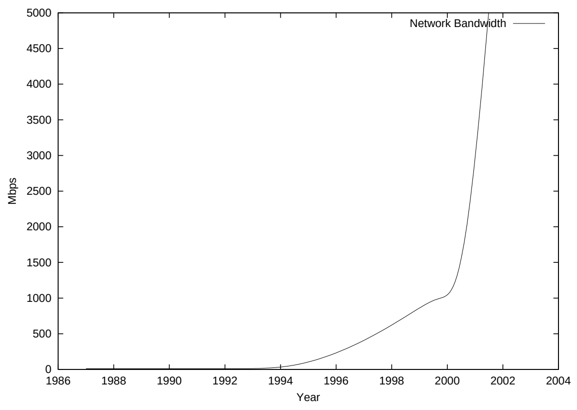

Figure 4: Network bandwidth from 1987-2003.

It is often the case that operating systems and applications do not simply shift data around but perform some simple operations, such as header processing. It would appear that this would make impossible a realization of our architecture. Observe, however, that many modern peripheral devices support a considerable amount of programmability, *e.g.,* the Intel IXP family of network cards. Similar functionality has been proposed for hard drives and controllers [2, 3, 56]. We can utilize such functionality to inject code snippets that implement simple operations such as header processing. Preliminary work in this direction has been promising [28]. Furthermore, our architecture easily takes advantage of such programmable peripherals, as opposed to the awkward manner in which they are used by current operating systems. Even when programmability is not a feature, scatter-gather I/O can be used to compose the data with network headers without having to copy the data into main memory. Finally, the proposed architecture allows us to efficiently take advantage of interconnection buses that allow for point-to-point data transfer between devices, such as daisy-chain FireWire or USB.

In the remainder of this paper, we expand on our vision of this new operating system architecture, identifying the main new components and their functionality. We describe some early work aimed at optimizing the performance of cryptographic operations as used by common network-security protocols such as SSL/TLS [19] and SSH, which motivated our re-thinking of operating system design. We close the paper by discussing related work.

2. ARCHITECTURE

To implement our proposed architecture, we need several new components in the operating system:

- A signaling API that applications such as web servers use to initialize flows across a series (pipeline) of peripheral devices, and specify the performance requirements of such flows. Since the API must be able to accommodate a wide variety of devices accessible through a small number of different bus architectures (PCI/PCI-X, FireWire, USB, *etc.*), we believe that a procedural approach is inflexible. Instead, we envision a simple language and associated runtime environment, along the lines of BPF [43], which will control the data flow operations. We call this the *Flow Management Language (FML).* Another possibility is to leverage on prior work in

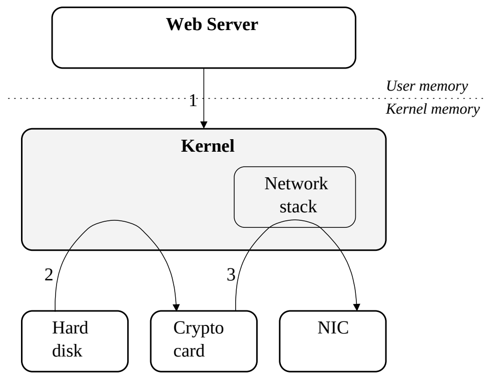

Figure 5: Operating system as a data-switch.

service composition for programmable infrastructures [68, 30]. FML programs will interact with devices through the runtime environment and a standardized driver-side API. The minimal operations that must be supported are device status and capability sensing, programming the direct memory access (DMA) controllers on the various peripherals, and exception handling.

Obviously, FML must be safe to execute inside the operating system. We believe that an approach similar to [34], which extends the processing capabilities of BPF while retaining the safety guarantees, can be applied to FML. Other techniques described in that work can be used to validate the rights of applications to reserve and use system resources.

Finally, if the runtime environment (interpreter) proves to be a performance bottleneck, we can use just-in-time (JIT) compilation techniques to convert the FML programs to machine code. We do not expect this to be necessary, however, since FML does not need to "touch" data (as, for example, BPF does) and thus should not be in the data-path.

- Since we expect multiple applications, or instances of the same application, to run on the same system, we must provide a resource scheduler to coordinate the various virtual pipelines that process the different data flows. The scheduler must take into consideration not only the performance requirements of each individual flow and application, but also the relative speeds of the various devices that process a flow. Generally speaking, the peak performance of a flow will be ultimately limited by the speed of the slowest device that must process the data (discounting for the moment bus contention, interrupt latency, and other external factors as other potential performance-limiting factors).

  Note, however, that if there are considerable discrepancies between the maximum throughput of various devices (*e.g.,* a 10 Gbps network interface supported by a 1 Gbps cryptographic accelerator), it is likely that the slow component will be replicated to exploit increased performance through parallelization. Even if a particular flow must remain tied to a specific device, *e.g.,* because processing requires state dependent on previous operations — as is the case for some compression algorithms, we can improve aggregate throughput through replication and load balancing.

  Thus, our scheduler (and FML) must be able to take into consideration the potential for parallel scheduling, as well as the "global" system requirements [39]. Recent work in global flow scheduling [31] seems directly applicable here, and we propose to investigate further.

- As we already mentioned, our approach makes use, and in some cases requires, the use of programmable peripherals such as smart network interface cards (*e.g.,* the IXP family of NICs) and hard drive controllers. Such capabilities can be used to avoid using the main processor altogether, or to implement needed functionality that is not otherwise available in the system, *e.g.,* cryptographic processing on a programmable network interface [28]. Dynamic code generation (DCG) techniques can be used to program these peripherals on the fly, and adapt them to specific tasks[1]. One challenge in using such capabilities is how to maintain the original semantics of the operating system network stack. One potential approach is to generate the code that runs on the programmable devices from the same code that the operating system itself uses.

- Even when programmable peripherals are not available *per se,* features such as TCP data-checksumming are increasingly being found on modern network interfaces. Such features allow the operating system to avoid having to "touch" the data, which would require their transfer to main memory. Instead, the operating system can use scatter-gather DMA to compose (and decompose) packets as they travel across devices. Depending on the specifics of the network protocols used, this composition can be done in two ways. First, protocol headers can be attached to the data as they are DMA'ed between devices, if the receiving device (and all subsequent devices of that flow) supports "pass-through" processing[2]. However, some devices, *e.g.,* certain image-processing devices, do not support pass-through. For these, the operating system must build the appropriate headers separately from the data, and "join" the two at the last step before it is transmitted to the network.

- Finally, the operating system and the FML runtime must be able to handle exceptions. Generally, such exceptions fall in one of two categories: data-processing errors (*e.g.,* device failure or data corruption) or external exceptions (*e.g.,* a TCP Reset or a Path MTU Discovery ICMP response from a router [45]).

  The former category is in some sense the easiest to handle: the FML program managing a flow can address the exception in one of several ways.

  1. It can try to re-start the device, or re-issue the data, if the fault was transient. In some cases (*e.g.,* corrupt data from a video camera), the error may simply be ignored, if it is not persistent.
  2. The flow can be redirected to use a different device. This can be another instance of the failed device, or it may be a different type of device which, when reprogrammed, can offer the lost functionality.

[1]For example, for a given TCP connection, the entire TCP/IP stack could be dynamically compiled and compressed, with all lookups and variables set to constants, and then the stack offloaded to a programmable NIC

[2]That is, process only part of the data while passing through the remaining data untouched. This is common in cryptographic accelerators used for IPsec and SSL/TLS processing.

3. The application can be notified about the failure and left to determine the appropriate course of action.
4. The flow may be switched to using the operating system code for data processing, trading functionality for performance.
5. If everything else fails, the flow will be terminated and a notification will be sent to the application.

The second type of exception can be handled by the FML program itself, the application, or the operating system stack.

It is interesting to note that, based on the above discussion, the "traditional" network stack implemented by current operating systems will remain useful for a number of reasons: as a fail-over when a device fails, for processing requests involving small amounts of data (which does not justify the overhead of setting up a flow and using up the relevant resources), for exception handling (driven by the FML program), and as a template for the dynamic code generation (*e.g.,* [15, 55]) and specialization mechanism [1, 12, 62].

We believe that additional functionality will be required by the operating system and the components we described above, but such will be identified as we progress our implementation of the architecture. We next describe some preliminary work we have done in the OpenBSD kernel, that lays the foundations for our architecture.

3. ACCELERATING CRYPTOGRAPHIC PROTOCOLS

We briefly describe some preliminary work in realizing our new operating system architecture. This work was motivated by a very practical problem: how to accelerate cryptographic protocols that transfer bulk data (*e.g.,* SSL/TLS or SSH) by using hardware cryptographic accelerators present in the system. We started by taking the most straightforward approach, which is exporting the appropriate functionality through a unix device driver. We built the OpenBSD cryptographic framework (OCF) [37], which is an asynchronous service virtualization layer inside the kernel that provides uniform access to cryptographic hardware accelerator cards. To allow user-level processes to take advantage of hardware acceleration facilities, a `/dev/crypto` device driver abstracts all the OCF functionality and provides an *ioctl()*-based command set that allows interaction with the OCF.

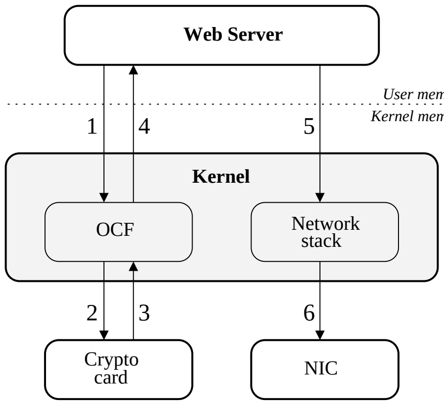

Figure 6: Use of cryptographic accelerators in a traditional operating system.

In network-security protocols that use cryptographic accelerators, the user-level process that implements the protocol, *e.g.,* a web server serving HTTPS requests, issues one or more crypto requests via `/dev/crypto,` followed by a *write()* or *send()* call to transmit the data, as can be seen in Figure 6. The web server uses the `/dev/crypto` device driver to pass data to the OCF (step 1), which delegates the SSL encryption and MAC'ing to the cryptographic accelerator card (step 2). The card performs the requested operations and the data are returned to the web server (steps 3 and 4). The web server uses the *write()* system call to pass the data to the network stack (step 5), which transmits the data on the wire using a network interface card (step 6). Similarly, a *read()* or *recv()* call is followed by a number of requests to `/dev/crypto.`

This implies considerable data copying to and from the kernel, and unnecessary process context switching. One alternative approach in the spirit of our new operating system architecture is to "link" some crypto context to a socket or file descriptor, such that data sent or received on that file descriptor are processed appropriately by the kernel. The impact of such data copying has been recognized in the past, as we discuss in Section 4, and has impacted TLS performance, as shown in [44]. As much as 25% of the overhead can be attributed to data copying. As cryptographic accelerators become faster, the relative importance of this source of overhead will increase.

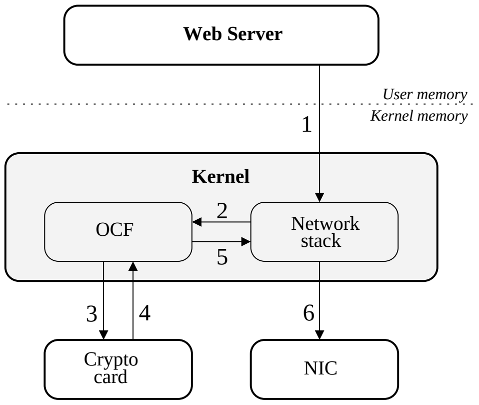

Figure 7: Encrypting and transferring a buffer, with socket layer extensions.

Our approach reduces the per-buffer cost down to a single *write()* and two context switches. (This is the same penalty as sending a buffer over the network with no crypto at all.) The fundamental change is that the network stack is crypto-aware. Figure 7 demonstrates how a buffer is encrypted and transferred using our extensions to the socket layer; the web server passes the buffer to the network stack using *write()* (step 1), which passes the data directly to the OCF (step 2). The OCF delegates the crypto to the accelerator card (step 3), as before, but the data are returned to the network stack (step 5) rather than user memory. The network stack then finishes processing the data and transmits it (step 6). The implementation of this scheme was extremely straightforward, consisting of adding about 80 lines of code in the kernel and changing less than 10 lines of code in our sample application that implemented a simple cryptographic protocol along the lines of SSL/TLS. The benefits were overall fairly modest, giving us a 10% improvement in throughput. However, this represents only a minor (and trivial) step towards our new architecture — most of the gains will be had from eliminating the DMA transfer to memory (consolidating steps 4 and 6). We are currently working toward such an implementation.

Another potential approach to reducing data copying overhead is to do "page sharing" of data buffers; when a request is given to */dev/crypto,* the kernel removes the page from the process's address space and maps it in its own. When the request is completed, it re-maps the page back to the process's address space, avoiding all data copying. This works well as long as */dev/crypto* remains a synchronous interface. However, to take full advantage of page sharing, applications will have to be extensively modified to ensure that data buffers are kept in separate pages, and pages that are being shared with the kernel are not accessed while an encryption/decryption request is pending. Finally, page-sharing does not, by itself, take advantage of NICs with integrated cryptographic support, although it can improve the performance in that scenario.

Another I/O performance bottleneck has been that of small requests. Although we did not examine its effects in this paper, previous work [42, 37] has demonstrated that they can have a significant negative impact on the performance. Since many cryptographic protocols, *e.g.,* SSH login, use small requests, the gains from cryptographic accelerators are smaller than one might hope for. There are several possible approaches: request-batching, kernel crossing and/or PCI transaction minimization, or simply use of a faster processor. These are more cost-effective solutions to deploying a hardware accelerator, as has already been pointed in the context of the TLS handshake phase [14].

4. RELATED WORK

[21] describes a mechanism for optimizing high-bandwidth I/O called *fbufs*. *Fbufs* combine a page remapping technique with dynamically mapped, group-wise shared virtual memory to increase performance during multiple domain crossings (*e.g.,* from user to kernel space). Essentially, data to be migrated across multiple domains are placed in an *fbuf,* and that *fbuf* is a memory buffer that is made accessible to each domain. Their experiments show that *fbufs* can offer an order of magnitude better throughput than page remapping. IO-Lite [52] is closely related to *fbufs,* but contains additional support for file system access and a file cache. One problem with such systems is their poor performance when dealing with small data buffers, since the fixed overhead for handling a message becomes the dominant cost, eclipsing memory copying. Other similar zero-copy approaches include "container shipping" [53], and the work described in [50]. These mechanisms were designed to minimize data copying between peripheral devices, kernel memory and user-process address space, thus do not directly address the problems of bus contention and memory bandwidth limitations, nor do they allow applications to efficiently take advantage of special-purpose hardware or direct interconnects between peripheral devices (*e.g.,* FireWire or USB daisy-chained devices). [10] showed that general-purpose operating systems (they used NetBSD, a unix variant) can easily emulate zero-copy I/O semantics at the system-call level without requiring any modification of applications, while offering performance comparable to the specially-designed systems described above.

[9] addresses the problem of data transfers between the network interface and the filesystem, which requires data buffers to be cached for later use and is thus incompatible with most zero-copy mechanisms. The author proposes the combination of mapped file I/O, using the standard unix *mmap()* system call, and copy avoidance techniques to minimize data copying at the server. However, the proposes approach requires synchronization between the client and server processes to ensure proper alignment of data buffers on reception. Furthermore, because of the requirement that data must reside in the file cache, at least one data copy operation to memory (*i.e.,* two PCI bus transactions) is needed.

The Scout operating system [46] represents I/O transactions using the concepts of paths [48]. System code is divided into modules, like IP or ETH, (for the IP module and the Ethernet device driver, respectively), which are arranged into *paths* along which packets travel. Paths are defined at build time, so when a module receives an incoming packet it does not have to expend cycles determining where to route the packet; that information is encoded in the path itself.

The Exokernel operating system [16, 26, 35, 27, 24, 29] takes a different approach, separating protection of hardware resources from management of hardware resources. This allows applications to perform their own management; hence, they can implement their own disk schedulers, memory managers, *etc.* Under an Exokernel architecture, a web server can, for example, serve files to a client while avoiding all in-memory data touching by the CPU by transmitting files directly from the file cache. Similar work includes [7, 59]. [25] proposes the elimination of all abstractions in the operating system, whose role should be to securely expose the hardware resources directly to the application. The Nemesis operating system [41, 8], driven by the need to support multimedia applications, provided a vertically-structured single-address-space architecture that did away with all data copying between traditional kernel and applications.

The Click Router [47] demonstrated the value of a polling-like scheme, versus an interrupt-driven one, for heavily-multiplexed systems. Because the various protocol modules are dynamically composed, various inefficiencies and redundant copying can also be eliminated. The value of clocked interrupts was also demonstrated by work on Soft Timers [4]. Similar prior work includes the *x-kernel* [33] and the router plugins architecture [18, 17].

A different technique for increasing I/O performance is the use of Application-Specific Handlers (ASH) [66, 65]. ASHs are user-written code fragments that execute in the kernel, in response to message arrivals. An ASH directs message transfers and sends the messages it handles, eliminating copies and reducing send-response latency. The authors demonstrate a 20% performance increase on an AN2 ATM network. A precursor to ASHs that was less flexible in terms of the operations allowed by the message handlers is described in [23]. A scheme similar to ASHs, Active Messages [64, 11], allows messages to invoke application-specified handlers instead of using the traditional thread-processing approach, thus avoiding the overhead of thread management and some unnecessary data copying. A similar approach is taken in [54]. Fast Sockets [58] provide an implementation of the Berkeley Sockets API on top of Active Messages. Optimistic Active Messages [67] removes some of the restrictions inherent in Active Messages, achieving the same performance while relaxing some of the restrictions inherent in the latter. U-Net [63] avoids the passage of data through kernel memory by performing a DMA transfer directly into the user buffer, letting applications perform their own flow control.

The Fast Messages library [40] introduces a user-level library that tries to eliminate unnecessary copying that is otherwise needed to perform routine protocol processing operations like header addition/removal or payload delivery. Another user-level approach for fast I/O is the Alloc Stream Facility [38]. Fundamentally, all these schemes place the application in the center of data transfers, making main memory and the application itself (*i.e.,* the main CPU) potential performance bottlenecks. Applicatin Device Channels (ADCs) [22] link network device drivers with applications, allowing direct processing of most protocol messages without requiring operating system intervention. LRP [20] applies early demultiplexing and lazy protocol processing at the receiver's priority in the Aegis operating system, which is otherwise similar to the *eager* approach taken by Application-Specific Handlers. A more conventional approach, described in [6, 5], improves the implementation of the *select()* system call, used by most event-driven network service implementations.

In [13], Clayton and Calvert propose a data-stream language that may be viewed as a simple version of the FML described in Section 2. By describing protocols with a data-stream-based language, they are able to arrange and implement protocol functions conveniently. Their compiler is also able to remove *at compile time* many of the ineffi encies introduced by traditional structured programming techniques, such as boundary crossings and lost optimization opportunities due to information hiding. Related work includes the Chromium system [32], which is a stream processing language designed specifi cally for interactively rendering graphics on clusters of machines. By abstracting away the underlying graphics architecture and network topology, it can support an array of applications in differing environments.

[36, 51, 57] present more proof of the validity and usefulness of the I/O streams paradigm, examined in the context of multimedia application processing. They use their Imagine architecture and *StreamC* language to compile C-like code into VLSI and demonstrate that, on average, stream programs take 98% of the execution time of hand-optimized assembly versions of the same code under an automated scheduler. [60] discusses their experience and evaluation in using a custom-built board, the Imagine Stream Processor, for high-performance 3-D teleconferencing. Our concept of stream processing transcends special-purpose hardware, aiming to enable traditional computer architectures. Other related work includes [49].

5. CONCLUSIONS

We proposed a new operating system architecture that removes the memory and CPU from the data path. The role of the operating system becomes that of data-flow management, while applications operate purely at the signaling level. This design parallels the evolution of modern network routers, and has the potential to enable high-performance I/O for end-systems, as well as fully exploit recent trends in programmability of peripheral (I/O) devices.

6. REFERENCES

[1] M. B. Abbott and L. L. Peterson. Increasing Network Throughput by Integrating Network Layers. *IEEE/ACM Transactions on Networking (ToN)*, 1(5), October 1993.

[2] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[3] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proceedings of the USENIX Annual Technical Conference*, pages 307–322, June 2000.

[4] M. Aron and P. Druschel. Soft Timers: Effi cient Microsecond Software Timer Support for Network Processing. *ACM Transactions of Computer Science*, 18(3):197–228, 2000.

[5] G. Banga, P. Druschel, and J. Mogul. Better Operating System Features for Faster Network Servers. In *Proceedings of the Workshop in Internet Server Performance*, June 1998.

[6] G. Banga and J. Mogul. Scalable Kernel Performance for Internet Servers Under Realistic Loads. In *Proceedings of the USENIX Technical Conference*, June 1998.

[7] B. 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 *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, December 1995.

[8] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol Implementation in a Vertically Structured Operating System. In *Proceedings of the 22nd Annual Conference on Local Computer Networks*, 1997.

[9] J. C. Brustoloni. Interoperation of Copy Avoidance in Network and File I/O. In *Proceedings of IEEE INFOCOM*, pages 534–542, 1999.

[10] J. C. Brustoloni and P. Steenkiste. Effects of Buffering Semantics on I/O Performance. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 277–291, December 1996.

[11] G. Chiola and G. Ciaccio. A Performance-oriented Operating System Approach to Fast Communications in a Cluster of Personal Computers. In *In Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA), volume I*, pages 259–266, July 1998.

[12] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of ACM SIGCOMM*, September 1990.

[13] R. V. Clayton and K. L. Calvert. Structuring Protocols as Data Streams. In *Proceedings of the 2nd Workshop on High-Performance Protocol Architectures*, December 1995.

[14] C. Coarfa, P. Druschel, and D. Wallach. Performance Analysis of TLS Web Servers. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, February 2002.

[15] D. R. Engler and M. F. Kaashoek. DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation. In *Proceedings of ACM SIGCOMM*, August 1996.

[16] D. R. Engler, *et al*. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[17] D. S. Decasper. *A software architecture for next generation routers*. PhD thesis, Swiss Federal Institute of Technology, 1999.

[18] D. S. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *Proceedings of ACM SIGCOMM*, pages 229–240, October 1998.

[19] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, IETF, January 1999.

[20] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 216–275, October 1996.

[21] P. Druschel and L. L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 189–202, 1993.

[22] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of ACM SIGCOMM*, pages 2–13, August 1994.

[23] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost Gb/s LAN. In

Proceedings of ACM SIGCOMM, pages 14–24, August 1994.

[24] D. R. Engler. *The Exokernel Operating System Architecture*. PhD thesis, MIT, October 1998.

[25] D. R. Engler and M. F. Kaashoek. Exterminate All Operating System Abstractions. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, pages 78–85, 1995.

[26] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. The Operating System Kernel as a Secure Programmable Machine. In *Proceedings of the Sixth SIGOPS European Workshop*, pages 62–67, September 1994.

[27] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and Flexible Application-Level Networking on Exokernel Systems. *ACM Transactions on Computer Systems*, pages 49–83, February 2002.

[28] L. George and M. Blume. Taming the IXP Network Processor. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[29] R. Grimm. Exodisk: maximizing application control over storage management. Masters Thesis, Massachusetts Institute of Technology, May 1996.

[30] M. W. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. Technical Report MS-CIS-98-25, Department of Computer and Information Science, University of Pennsylvania, February 1998.

[31] M. W. Hicks, A. Nagarajan, and R. van Renesse. User-specifi ed Adaptive Scheduling in a Streaming Media Network. In *Proceedings of IEEE OPENARCH*, April 2003.

[32] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, pages 693–702, 2002.

[33] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[34] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: packet fi ltering for low-cost network monitoring. In *Proceedings of the IEEE Workshop on High-Performance Switching and Routing (HPSR)*, pages 121–126, May 2002.

[35] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, October 1997.

[36] U. J. Kapasi, P. Mattson, W. J. Dally, J. D. Owens, and B. Towles. Stream Scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, December 2001.

[37] A. D. Keromytis, J. L. Wright, and T. de Raadt. The Design of the OpenBSD Cryptographic Framework. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[38] O. Krieger, M. Stumm, and R. Unrau. The Alloc Stream Facility: A Redesign of Application-Level Stream I/O. *IEEE Computer*, 27(3), March 1994.

[39] K. Lakshman, R. Yavatkar, and R. Finkel. Integrate CPU and Network-I/O QoS Management in the Endsystem. *Computer Communications*, pages 325–333, April 1998.

[40] M. Lauria, S. Pakin, and A. A. Chien. Effi cient Layering for High Speed Communication: Fast Messages 2.x. In *Proceedings of the 7th IEEE Symposium on High Performance Distributed Computing (HPDC)*, July 1998.

[41] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[42] M. Lindemann and S. W. Smith. Improving DES Coprocessor Throughput for Short Operations. In *Proceedings of the 10th USENIX Security Symposium*, pages 67–81, August 2001.

[43] S. McCanne and V. Jacobson. A BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter Technical Conference*, pages 259–269, January 1993.

[44] S. Miltchev, S. Ioannidis, and A. D. Keromytis. A Study of the Relative Costs of Network Security Protocols. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 41–48, June 2002.

[45] J. Mogul and S. Deering. Path MTU Discovery. RFC 1191, November 1990.

[46] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A Communications-Oriented Operating System (Abstract). In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[47] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, December 1999.

[48] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–167, 1996.

[49] J. Nieh and M. Lam. SMART: A Processor Scheduler for Multimedia Applications. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, December 1995.

[50] S. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[51] J. D. Owens, S. Rixner, U. J. Kapasi, P. Mattson, B. Towles, B. Serebrin, and W. J. Dally. Media Processing Applications on the Imagine Stream Processor. In *Proceedings of the IEEE International Conference on Computer Design*, pages 295–302, September 2002.

[52] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unifi ed I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[53] J. Pasquale, E. Anderson, and P. K. Muller. Container Shipping: Operating System Support for I/O-intensive applications. *IEEE Computer*, 27(3):84–93, March 1994.

[54] C. Poellabauer and K. Schwan. Lightweight Kernel/User Communication for Real-Time and Multimedia Applications, June 2001.

[55] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek.

’C and tcc: A Language and Compiler for Dynamic Code Generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):324–369, 1999.

[56] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proceedings of the Conference on Very Large DataBases*, August 1998.

[57] S. Rixner. *A Bandwidth-efficient Architecture for a Streaming Media Processor*. PhD thesis, MIT, February 2001.

[58] S. Rodrigues, T. Anderson, and D. Culler. High-Performance Local-Area Communication Using Fast Socket. In *Proceedings of the USENIX Technical Conference*, January 1997.

[59] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, October 1996.

[60] B. Serebrin, J. D. Owens, C. H. Chen, S. P. Crago, U. J. Kapasi, B. Khailany, P. Mattson, J. Namkoong, S. Rixner, and W. D. Dally. A Stream Processor Development Platform. In *Proceedings of the International Conference on Computer Design (ICCD)*, September 2002.

[61] A. S. Tanenbaum. *Computer Networks, 3rd Edition*. Prentice Hall, 1996.

[62] E. N. Volanschi, G. Muller, C. Consel, L. Hornof, J. Noye, and C. Pu. A Uniform and Automatic Approach to Copy Elimination in System Extensions via Program Specialization. Technical Report RR-2903, Institut de Recherche en Informatique et Systemes Aleatoires, France, 1996.

[63] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP)*, 1995.

[64] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, 1992.

[65] D. A. Wallach. *High-Performance Application-Specific Networking*. PhD thesis, MIT, January 1997.

[66] D. A. Wallach, D. Engler, and M. F. Kaashoek. ASHs: Application-specifi c handlers for high-performance messaging. In *Proceedings of ACM SIGCOMM*, August 1996.

[67] D. A. Wallach, W. C. Hsieh, K. L. Johnson, M. F. Kaashoek, and W. E. Weihl. Optimistic Active Messages: A Mechanism for Scheduling Communication with Computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[68] Y. Yemini and S. daSilva. Towards Programmable Networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, October 1996.