Adding a Flow-Oriented Paradigm to Commodity Operating Systems

Cristian Soviani Stephen A. Edwards Angelos Keromytis
Department of Computer Science, Columbia University in the City of New York

Abstract

The speed of CPUs and memories has historically outstripped I/O, but emerging network and storage technologies promise to invert this relationship. As a result, fundamental assumptions about the role of the operating system in computing systems will have to change.

We propose an operating and application architecture that removes the CPU and memory from the path of high-speed I/O. In our model, the operating system becomes a data-flow manager and applications merely direct this flow instead of directly participating in it.

Our proof-of-concept prototype, which we implemented on an FPGA board, nearly doubled the throughput of a simple cryptographic networking application, suggesting our model can provide a substantial improvement.

1 Introduction

Traditional computing assumes I/O speeds are substantially slower than CPU-memory speed. Emerging network and storage technologies have been stretching this assumption to its limits. Optical wire-speeds already range from 10–40 Gbps, on par with CPU memory bandwidth, while hard-disk drives are delivering 2.5 Gbps. With both wire-speed and storage density technologies scaling at faster Moore's exponents than silicon, it is not unreasonable to assume that raw I/O speeds can continue to outgrow CPU-memory speeds this decade. This speed inversion will have a profound impact on computing.

We propose an operating system and application architecture that removes the memory and CPU from the data path for applications that handle high-bandwidth data flows. The role of the operating system becomes that of data-flow management, while applications are concerned purely with signalling. 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 toward programmable peripheral (I/O) devices. An alternative view is that hardware devices are composed into virtual processing pipelines, completely removing the CPU and main memory from the data-intensive tasks for which they are increasingly unsuitable due to performance considerations. Our hypothesis is that application-specific data-flow handling policies can be executed inside the operating system kernel, which composes and configures peripherals, manages flows, and handles exceptions. We further hypothesize that such functionality can be retrofitted to existing operating systems and applications, instead of requiring a complete redesign.

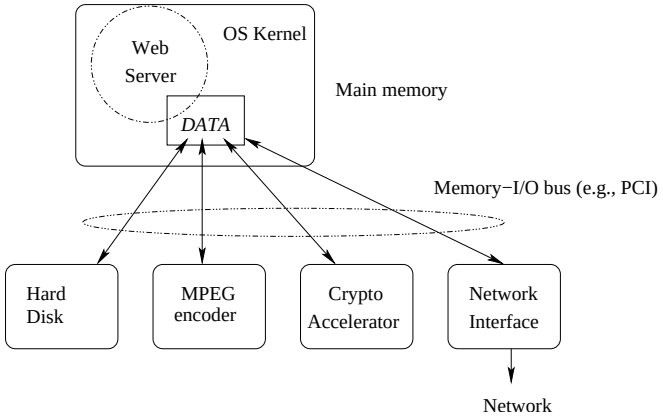

Figure 1: The status quo: main memory as data cache.

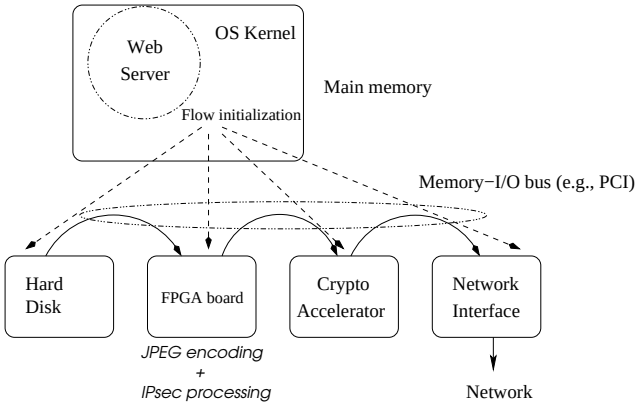

Figure 2: Our approach: operating system as a data switch. Here, we use an FPGA to accelerate some processing.

Our proposed architecture is a first step toward abandoning the concept of memory-centric computing, a paradigm ingrained in computer scientists for several decades. This shift in architectural thinking is both necessary for performance reasons and natural, given the increasing use of task-specific and programmable hardware. Our proof-of-concept prototype indicates that our system can at least double I/O throughput for large flows, and is orthogonal to other performance optimization approaches (e.g., faster interconnection buses).

After reviewing related work, we present our model in Section 3 and some experimental evidence that it can provide substantial throughput gains (Section 4).

2 Related Work

Despite considerable work in reducing the amount and cost of memory copying, the focus of all such efforts has been only to remove the operating system from the data path; the application remains responsible for handling data. Such approaches only partially address the memory-as-bottleneck problem; our architecture completely removes memory from the data path.

The Scout OS [17] represents I/O transactions using the concepts of paths [18]. System code is divided into software 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. Our proposed architecture essentially extends the concept of the Scout path to operate across multiple peripherals (with the OS managing flow scheduling), completely eliminating memory as a data-staging area and allowing for non-network-oriented data processing.

Aron and Druschel [10] describe 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. However, memory remains a key component in the data processing cycle, thus remaining a processing bottleneck. Pai et al.'s IO-Lite [20] 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 zero-copy approaches include Pasquale et al.'s "container shipping" [21], and the work of O'Malley and Peterson [19]. 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. Brustoloni [4] 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. This approach requires synchronization between the client and server processes for proper alignment of data buffers on reception. Because of the requirement that data must reside in the file cache, at least one copy to memory (two PCI transactions) is needed.

The Exokernel operating system [9] separates 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. Engler et al. [12] propose 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 [16] provided a vertically-structured single-address-space architecture that did away with all data copying between kernel and applications.

Krishnamurthy [14] examines new hardware architectures for meeting the QoS requirements of different streams at wire speeds, without need for special-purpose ASICs. He developed an architecture, SharedStreams, that uses programmable network interfaces to implement various packet scheduling disciplines that enforce different QoS parameters for different data streams. To achieve 10 Gbps Ethernet performance, the architecture uses FPGA cards and programmable network interfaces that implement the packet disciplines, as well as direct network-to-network interface data transfers to avoid CPU and main memory overheads (they also use disk-to-card transfers, although the disks are directly attached to the network cards through a SCSI interface — the cards are fully-programmable computers on a PCI board). The focus of the work is on packet schedulers, whereas our work is looking at the intersection of flow scheduling and data computation, managing data flows across a series of peripherals that transform the data.

NodeOS [22] introduced the concept of *short-cuts* in domains (software modules implementing certain types of protocol processing), which is used to quickly forward packets to another domain or another node. This notion is conceptually similar to, albeit much simpler than our direct card-to-card data transfer approach, which operates directly on hardware.

3 Our Approach

We propose a new operating system architecture that removes or minimizes the role of the memory and CPU from the data path for applications that handle large flows of data. The role of the operating system becomes that of data-flow management, while applications operate purely at the signalling 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.

Our architecture, Figure 2, eliminates all data copies to main memory, including those between peripherals and RAM, and reduces the number of data transfers over the shared bus (e.g., PCI) by half. In this architecture, the application simply specifies the data path in terms of requirements (e.g., "transfer file /html/index.html to remote host 128.59.19.56 using HTTP 1.0 over SSL; the remote TCP port is 32156"). The operating system translates these requirements to specific configuration directives to the various peripheral devices, which may include configuring the devices, downloading code to perform some data processing, scheduling the data transfers, and dealing with failures and other exceptional situations. Data then flows directly between the various peripherals, which may be connected via a shared bus (e.g., PCI) or via direct interconnects (e.g., FireWire, USB, Rapid I/O). In some sense, our

proposal generalizes the concept of a path, first introduced in ScoutOS [18], to span multiple peripheral devices, completely eliminate memory as a data-staging area, and allow for non-network-oriented data processing. By design, our approach is not applicable to certain classes of applications, e.g., interactive applications such as a text editor.

In our architecture, applications specify their data processing policies using a simple API that sends flow-control requests to the operating system kernel. Such applications control all aspects of data flow management, such as scheduling, exception handling, and resource composition. The runtime environment in which these programs will be executed will ensure the safety of the kernel and enforce isolation between different flows. We define a *flow* to be any stream of data, possibly as small as a single packet. However, since there are costs associated with creating and tearing down a flow, we will only consider those flows where the number of packets is high enough to make the flow creation cost negligible. We plan to quantify this cut-off point eventually.

Operating systems and applications often perform operations on the data they move around, which might appear to make our architecture unrealistic. However, many modern peripheral devices support a considerable amount of programmability, e.g., the Intel IXP family of network processors. Similar functionality has been proposed for hard drives and controllers [2, 3, 24]. We utilize such functionality to inject code snippets that implement simple operations such as header processing or database-record matching. Preliminary work in this direction has been promising [13].

We envision the operating system offloading certain parts of data-flow processing to programmable hardware ranging from network processors (such as the LANI processors on Myrinet systems, or Intel IXP cards) to Field-Programmable Gate Arrays (FPGAs), as shown in Figure 2, which represents a simple web-based Video-on-Demand service. Here, the FPGA board is configured by the operating system to subsume the role of a specialized JPEG encoder and IPsec header-processing engine. Data flows from the hard disk to this card, from there to a hardware accelerator, and on to a network interface for transmission. Of course, such programmability may be available on the network interface or the hard disk controller—the exact location is not important from an architectural viewpoint. We do not envision re-programming the FPGAs and other programmable components very frequently, since the serial interface though which reprogramming is usually done is slow. As stated before, our architecture is oriented toward large data-flow applications where configurations change infrequently.

Our proposed scheme can easily take advantage of programmable peripherals, as opposed to the awkward way current operating systems access them. 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. Our architecture can also take advantage of point-to-point interconnection buses such as FireWire.

Our approach works even better when entire systems can be integrated on a chip, where multiple busses are the rule. Our approach will use such flexibility for an additional speed-up.

3.1 Server Model

Figure 2 captures our model of the underlying hardware: the system is composed of a series of peripheral devices classified into three categories: data sources, data sinks, and data transformers. Hard drives and network interfaces are both data sources and sinks, although they will assume a single role in the context of an individual data flow. Video cameras are an example of a data source, while a sound card is an example of a data sink. Data transformers include crypto accelerators, DSPs, MPEG encoder/decoders, and general-purpose FPGAs. Some devices may fall under all three categories (e.g., a programmable network card).

These devices are interconnected using topologies such as shared-bus (e.g., PCI/PCI-X) and daisy-chain (e.g., FireWire devices). When FPGAs are needed, they are either attached to the shared bus or are part of a chain. Of course, in some interconnection strategies (e.g., daisy-chains) device locations can affect both performance and possible functionality. Although our ideas would be best implemented on hardware specifically designed to support our configurable flows, FPGAs are an excellent alternative to completely custom hardware, especially when mixed with existing programmable peripherals such as network interfaces with on-board processors. Applications and the OS kernel reside in main memory and otherwise operate much as they do in current systems, with the exception of the changes in functionality imposed by our architecture.

3.2 New Operating System Components

Our architecture needs several new components: a signalling API, a resource scheduler, programmable peripheral support, legacy device handling, and exception handling. Here, we describe the required functionality of each component and the challenges in designing and implementing them.

Signalling API The first component is a signalling 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. This API must be able to accommodate a wide variety of devices accessible through a small number of different bus architectures (PCI/PCI-X, USB, etc.). Such an API and runtime environment can be used to both effectively manage all aspects of a data flow and express interesting application policies with respect to data transmission requirements.

To begin with, applications that use our architecture will be associated with kernel modules that capture their data flow needs as interactions between the hardware components and download these into the operating system kernel. The modules will control the data flow in terms of flow control, requirements, initialization, and teardown. The modules will interact with devices through the runtime environment and a standardized driver-side API. We believe an API through which applications can specify their needs in terms of data flow management and scheduling is the best approach.

The driver-side API must support at minimum device status and capability sensing, programming the direct memory access (DMA) controllers on the various peripherals, and exception handling.

Resource Scheduler Since we expect multiple applications to run on the same system, we must provide a resource scheduler to coordinate the various virtual pipelines that process different data flows. The scheduler must take into consideration not only the performance requirements of each individual flow and application and the relative priorities of the various applications, 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 bus contention, interrupt latency, and other external factors as other potential performance-limiting factors).

However, if there are considerable discrepancies among the maximum throughputs of various devices (e.g., a 10 Gbps network interface supported by a 1 Gbps cryptographic accelerator), multiple copies of the the slow component can be run in parallel to increase performance. Even if a particular flow must run on a specific device, e.g., because processing requires state dependent on previous operations, we can improve aggregate throughput through replication and load balancing. Thus, our scheduler must be able to consider the potential for parallel scheduling as well as global system requirements [15]. Whether this can be done efficiently is an open question.

Programmable Peripherals Our approach uses and sometimes requires 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 [13]. Dynamic code generation (DCG) techniques can be used to program these peripherals on the fly, and adapt them to specific tasks. Maintaining the original semantics of the operating system network stack is an obvious challenge.

While few of today's peripherals support the programmability we demand, the number is increasing; we hope our architecture is compelling enough to encourage many more.

We envision using FPGAs as co-processors for data transformation tasks such as video transcoding, cryptographic operations, network protocol header processing. These will be configured at flow set-up time by the scheduler and multiplexed across different flows with the same functionality requirements (i.e., one such board may be able to do IP processing for the majority of data flows in the system). FPGAs can also be used as stepping stones for legacy peripherals, which often require data transfers to involve memory. While this would imply that direct card-to-card data transfers are impossible with such peripherals, an FPGA can masquerade as a high-speed memory buffer directly attached to the I/O bus.

The magnitude of benefits conferred by programmability remains an open question. An obvious approach is to use precompiled customizable modules that implement the various pieces of functionality we may want to download to the FPGA, making our environment mostly static and avoiding the overheads involved in frequent re-programming of FPGAs (and other programmable peripherals). But we suspect dynamic code generation techniques can be applied in our approach.

Legacy Device Handling Even when programmable peripherals are not available per se, features such as TCP data checksumming and NIC/crypto integrated processing can be found on modern network interfaces. Such features allow the operating system to avoid touching the data, which otherwise requires its 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. However, some devices, such as certain ones for image processing, 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.

Exception Handling Finally, the operating system must be able to handle exceptions. Such exceptions generally 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). The former category is the easier one to handle. The module managing a flow can address the exception in 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 re-programmed, can provide the lost functionality.

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 module itself, the application, or the operating system stack. We believe it will be possible to move state between the operating system network stack and the module to migrate data flows between different peripherals and the network stack. This should be transparent to the application, provided that the application's requirements are met.

Note that 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, and as a template for dynamic code generation [8, 23] and specialization [1, 6, 25].

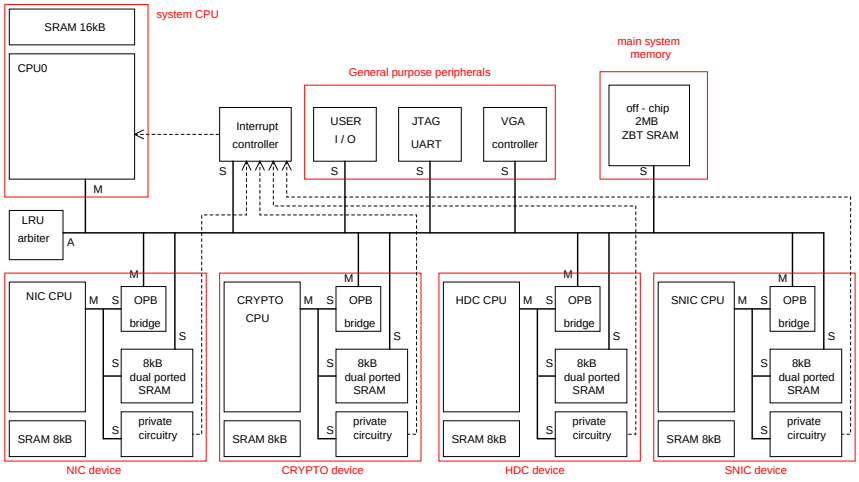

Figure 3: Block diagram of the proof-of-concept prototype, which uses a Xilinx Virtex-2 XC2V2000-4 FPGA board and an 81 MHz system clock. The whole architecture, which includes three peripherals (network interface, crypto card, hard drive), main CPU and memory, and a trivial OS supervising the simulated data flows, was implemented on an FPGA board. The fourth peripheral was not used.

3.3 Designing and Implementing Custom Peripherals

Especially when using FPGAs, our approach needs the ability to quickly design and implement bus-resident peripherals and their corresponding device drivers. Our starting point is a domain-specific language (developed at Columbia) for describing Unix device drivers [7] and a hardware/software co-design language that allows a peripheral and its device driver to be specified and synthesized simultaneously [11].

Given our goal of using FPGAs as highly programmable flow processors, it will be necessary to greatly simplify the task of creating data processors that communicate via existing communication protocols such as buses and serial links. We believe that a combination of domain-specific languages able to describe the hardware/software boundary with a particular eye toward communication protocols coupled with high-level synthesis tools is the solution for this problem.

4 Preliminary Experiments

To estimate the performance improvement we can expect from our approach, we built a proof-of-concept prototype of the architecture. Rather than implement the necessary drivers and OS extensions, on a large FPGA board we assembled a system with a CPU, main memory, and several peripherals communicating over a PCI-like bus. Figure 3 depicts the architecture we implemented. The system ran a mock OS kernel that managed data transfers to/from RAM and between peripherals, based on the scenario.

We simulated a system where the network interface (leftmost NIC) receives packets that are decrypted by a cryptographic accelerator board (CRYPTO), with the results stored on the hard drive (HDC). The fourth device (SNIC) is idle. At a high level of abstraction, the data flows correspond to those found in a secure web or file server. Each device has a command queue; when a command is finished, the device interrupts the CPU and immediately starts the next command. The CPU tries to keep the command queues full (if possible), to avoid stalls in data processing. The transfers between devices are done via ring buffers, a common control-register configuration in high-performance peripherals, and are also managed by the CPU.

The devices implement some abstract commands specific to their functionality. The NIC simply needs to be told where in memory (whether main memory or another board's PCI-addressable buffers) the next received packet should be placed. The CRYPTO device is told where to receive data, which key to process them with, and where to place the output (again, in terms of memory locations). Finally, the HDC writes data found at a specific memory location to the drive, and fetches data from the drive to a specific memory address.

We ran two sets of experiments, one simulating memory-centric data processing (Figure 1) and another using our proposed architecture (Figure 2). In each, we streamed 1 million packets through all peripherals. When the stream consisted of 64-byte packets, the times were 15.5 and 13.6 seconds for the memory-centric and our approach respectively, a modest 13% speedup. However, when we used 1024-byte packets, the times were 114.6 and 59.2 seconds (9,150 *vs.* 17,700 packets/sec) respectively, corresponding to nearly twice the throughput. We expect even better performance with larger data packets—a reasonable upper bound would be 1460 bytes (a maximum-size Ethernet frame minus IP and TCP headers). Although this is only a rough estimate, we believe that it illustrates the potential performance improvements inherent in our proposed architecture.

We also experimented with pushing application-layer logic inside the kernel, partially implementing the scenario from Figure 2. Our preliminary results [5] show that simply removing the application from the data path (but without otherwise minimizing bus and memory utilization), throughput increases by up to 25%.

5 Conclusions

He have proposed a new architecture for operating systems, one that departs from the memory-centric approach of the past 40 years. Our architecture structurally resembles that of a network switch, enabling for fast data flows between peripherals, removing the CPU and (more importantly) the main memory from the data path, as these have proven to be the limiting factors in modern architectures.

Our architecture will enable high-performance applications and allow more efficient use of server resources, beyond the limits of modern operating systems. It will allow experimentation and deployment of new types of system architectures, allowing use of novel hardware designs, device interconnection strategies, and load-balancing and scheduling algorithms. Our approach will also eliminate one commonly perceived problem with the wide-spread use of secure protocols and cryptography, that of performance, and it will enable a new class of systems that are able to support high-performance applications across a wide range of areas of computer science that are currently economically infeasible.

While to fully realize our vision represents a substantial development project, preliminary results are promising. Our proof-of-concept prototype (Figure 3) ran at nearly double the throughput with our approach, which we think of as a lower bound because of the simplicity of the prototype.

We are proposing nothing less than a new way of thinking about operating systems, one that enables high-performance applications and allows efficient use of server resources, beyond the limits of current systems. Our architecture will not only improve application performance, but will allow the use of security protocols with very low performance impact, as well as enable experimentation and deployment of new types of system architectures, novel hardware designs, device interconnect strategies, scheduling algorithms, and so on.

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

[5] M. Burnside and A. D. Keromytis. The Case For Crypto Protocol Awareness Inside The OS Kernel. *ACM SIGARCH Computer Architecture News*, 33(1):57–64, March 2005.

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

[7] Christopher L. Conway and Stephen A. Edwards. NDL: a domain-specifi c language for device drivers. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Washington, DC, June 2004.

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

[9] 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.

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

[11] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of Emsoft*, Jersey City, New Jersey, September 2005.

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

[13] 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.

[14] R. B. Krishnamurthy. *Scalable Real-time Architectures and Hardware Support for High-Speed QoS Packet Schedulers*. PhD thesis, Georgia Institute of Technology, April 2003.

[15] 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.

[16] 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.

[17] 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 Operating Systems Design and Implementation (OSDI)*, 1994.

[18] 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.

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

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

[21] 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.

[22] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullman, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications (JSAC)*, 19(3):473–487, March 2001.

[23] 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.

[24] 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.

[25] 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.