A case for Virtual Channel Processors

Position paper

Derek McAuley
Intel Research Cambridge
15 JJ Thomson Av
Cambridge, UK
derek.mcauley@intel.com

Rolf Neugebauer*
Intel Research Cambridge
15 JJ Thomson Av
Cambridge, UK
rolf.neugebauer@intel.com

ABSTRACT

Modern desktop and server computer systems use multiple processors: general purpose CPU(s), graphic processor (GPU), network processors (NP) on Network Interface Cards (NICs), RAID controllers, and signal processors on sound cards and modems. Some of these processors traditionally have been special purpose processors but there is a trend towards replacing some of these with embedded general purpose processors. At the same time main CPUs become more powerful; desktop CPUs start featuring Simultaneous Multi-Threading (SMT); and Symmetric Multi-Processing (SMP) systems are widely used in server systems. However, the structure of operating systems has not really changed to reflect these trends — different types of processors evolve at different timescales (largely driven by market forces) requiring significant changes to operating systems kernels to reflect the appropriate tradeoffs.

In this position paper we propose to re-vitalise the old idea of channel processors by encapsulating operating system I/O subsystems in *Virtual Channel Processors (VCPs)*. VCPs perform I/O operations on behalf of an OS. They provide similar development, performance, and fault isolation as dedicated (embedded) I/O processors do while offering the flexibility to split functionality between the main processor(s) and dedicated processors without affecting the rest of the OS. If part of a VCP is executed on the main processor, we propose to make use of virtual machine technology and SMT/SMP features to isolate its performance from that of the rest of the system and to protect the system from faults within the VCP.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design; D.4.4 [**Operating Systems**]: Communications Management

General Terms

Design

*Contact Author

ACM SIGCOMM 2003 Workshops, August 25–29, 2003, Karlsruhe, Germany.

Keywords

Virtual Channel Processors, I/O virtualisation, protocol offloading

1. INTRODUCTION

A modern computer system contains several different processors apart from the main CPU. These include graphics engines and signal or audio processors which provide specialised functionality, such as vector and real-time "analog" signal processing. However, recently this trend has been accelerated by proposals and products placing additional processing on I/O cards and within the I/O subsystem. These processors partially replace functionality previously implemented in hardware but are mainly targeted at moving computation away from the main CPU onto the I/O cards. A prime example is the emergence of network cards containing TCP Offloading Engines (TOE) where the processor on the network card is executing the TCP protocol stack on behalf of the operating system. More common approaches off-load only part of the network protocol stack processing to the hardware, e.g., checksum calculation. This trend is partly motivated by the emergence of storage networks using IP storage solutions [14] such as iSCSI [32]. The main argument for the use of TOE or other off-loading techniques is to reduce the load imposed on the main processor and to isolate the performance impact this additional load would have on the rest of the system. Similar reasons can be found for the use of specialised RAID controller cards[1].

At the same time the speed of main CPUs, their support chipsets and memory interfaces increase significantly faster and more continuously than the speed of embedded processors typically used for separate I/O co-processors. We believe that this trend can be mainly attributed to market economies.

With these observations in mind, the question becomes on whether it makes sense to invest into developing specialised offloading technology, especially considering the cost and complexity of embedded software development and required modifications to an OS, if in a few month time the main processor could handle the additional load? To give an example, a recent paper presented some experimental results of a software iSCSI implementation using a 800 MHz Pentium® III client and a dual 733 MHz Pentium® III server and suggested that TOE capable network cards or specialised iSCSI capable NICs should be used to relieve the main CPU [31]. In a subsequent paper [30] the same authors evaluate the performance of currently available hardware-based approaches on a more

[1]"Soft modems" are a notable if marginal exemption: I/O processing is actually moved from a specialised signal processor back onto the main processor, despite the quite stringent timeliness requirements [17].

up-to-date system (1.6 GHz AMD Athlon MP). They concluded that the currently available hardware-based approaches, such as TOE, do not provide a significant performance benefit over software based approaches using the main processor and attribute this result mainly to the disparity in processing powers. While only few improvements have been made to the hardware based solutions since, main processor speeds have almost doubled with 3 GHz processors being widely available. While this widening disparity may be a temporary phenomenon it is clear that the correct tradeoff will remain a moving target. Furthermore, even with faster offloading technology available it has been argued that "TCP offload is a dumb idea" for most applications as it is complex to implement and offers little performance benefits [23].

Another common justification for offloading processing onto additional processors is to provide performance isolation. The argument is that the performance of applications executing on the main CPU is not *directly* impacted by the processing performed by the additional processors as they use dedicated hardware resources. More importantly, a significant performance impact in the end systems can also be attributed to general processing overheads, e.g., interrupt processing [8]. This issue is not likely to be resolved by faster main processors alone. On the contrary, with ever more complex processors it is likely to become more pronounced. However, to some extent, both performance isolation and interrupt overheads could be addressed by simultaneous multi-threading (SMT) technology, such as Intel®'s Hyper-Threading Technology (HT) [22]. There is experimental evidence that even for non-specialised OSes, HT can provide significant network performance improvements [15]. Isolation could be achieved by using separate hardware threads, while processing overheads could be addressed by dedicating a (fixed portion) of a hardware thread to interrupt processing, akin to related approaches dedicating a separate physical CPU for protocol processing [25, 24, 28], albeit at significantly reduced cost.

An important observation is that the structure of traditional operating systems is not necessarily conducive to supporting these and other emerging technologies; they are difficult to adapt to the ever changing tradeoffs between different implementation options, as exemplified by the slow availability of the different off-loading options discussed above. As anecdotal evidence consider that the Linux network stack (2.4.18) checks in 25 different source files (excluding device drivers) if the hardware supports checksum offloading. While these modifications are fairly local it appears a non-trivial change. Consider the changes which would be required for supporting more substantial hardware offloading or even more radical hardware designs such as the Twin Cities prototype[2]. Furthermore, changing the operating system to track these tradeoffs may jeopardise overall system stability, as previously "monolithic" kernel subsystems, like the IP network stack are broken up. Even if the changes are local to a kernel subsystem they have the potential to disrupt the rest of the kernel as they execute in the same protection and resource domain.

To address these issues, we propose to virtualise the I/O subsystems of a traditional OS kernel (and potentially other kernel subsystems) by encapsulating them in Virtual Channel Processors (VCPs) akin to the use of channel processors in mainframe computers [11]. VCPs perform I/O operations on behalf of the rest of the operating system and constitute both separate protection *and* scheduling domains. Within a VCP, developers can make the tradeoff of where to place I/O functionality: on dedicated hardware on I/O interface cards or on the main processor. If placed on the main processor isolation can be provided by virtual machine technology offering fault and performance isolation as well as a "clean slate" to developers of I/O subsystems.

The rest of this paper is structured as follows. In the next section we highlight the problems arising from a traditional OS kernel structure using the aforementioned iSCSI example. In section 3 we describe our virtual channel processor approach in more detail and, by re-visiting the iSCSI example, illustrate how it addresses these issues. In section 4 we outline our implementation plan, which is based on the Xen virtual machine monitor [2]. Section 5 discusses some related work and in section 6 we summarise our proposal.

2. EXAMPLE: iSCSI

iSCSI [32] is an emerging standard which aims at using commodity Ethernet network components to build Storage Area Networks allowing clients to issue SCSI commands over TCP/IP to access block devices on "remote" network attached storage. For an operating system, iSCSI is an interesting example as it exercises two previously independent subsystems, namely the file-system stack and the network stack, and thus poses some interesting challenges on operating systems [31]. Figure 1 illustrates how a pure software iSCSI implementation can be provided. Applications access files through the file system layer, which in turn uses a block device layer to issue SCSI commands. However, instead of using the SCSI commands to control a locally attached disk, they are entered at the top of the operating system's network stack, and traverse it as normal network packets would, i.e., through the TCP/IP layer and the Ethernet layer before being handed to the network card for transmission.

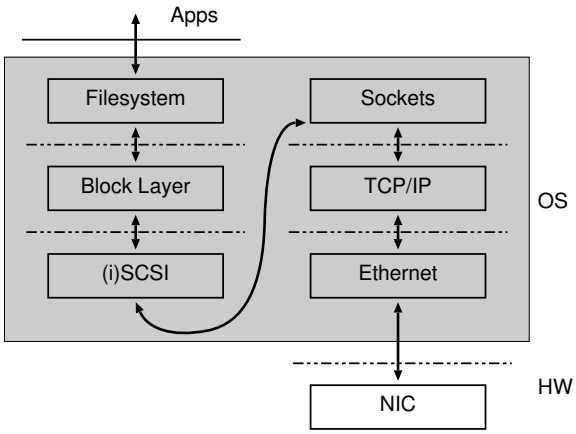

Figure 1: iSCSI software implementation

This oversimplified diagram highlights a number of issues. The implementation adds complexity: previously independent kernel subsystems now depend on each other. Operating systems typically use non-public, sometimes ill-defined, and changing APIs between their components[3]. Adding a feature which crosses subsystems may trigger unwanted side-effects, cause previously unencountered interactions, and complicate the maintainability of the kernel. Note, that some of these issues could be resolved with standard software engineering practice, such as modularisation and

[2]Twin Cities is a experimental prototype connecting a Intel® IXP 1240 network processor and its attached network interface directly to the Front Side Bus (FSB) of a Pentium® III processor [12].

[3]Microsoft's Windows network implementation [21] with the definition of the NDIS and TDI interfaces for network devices and transport protocols respectively appear to be a step in the right direction.

strong encapsulation. However, these techniques are not typically utilised throughout OS kernel code.

Furthermore, such an implementation may have a significant performance impact, not because it executes on the same processor but because of its implementation. For example the same network stack is now shared between the traditionally separate file I/O and network I/O: normal network processing may interfere with iSCSI network processing in the network stack, essentially introducing more opportunities for resource contention, e.g., memory contention for buffer caches and network receive and transmit buffers. There may also be potential problems with locking, as locking in the filesystem stack is typically independent of locking in the network stack. Latency may also be increased since file I/O requests have to go through two stacks in the kernel rather than just one.

A common proposal to solve these problems is to use specialised network cards implementing the TCP/IP network stack for iSCSI devices, or place an iSCSI interface on the network I/O card. Instead, we propose to virtualise the I/O interface in operating systems and encapsulate much of the I/O functionality in a Virtual Channel Processor.

3. VIRTUAL CHANNEL PROCESSORS

Figure 2 gives an overview of our proposed virtualised I/O architecture. We assume a Virtual Machine Monitor (VMM) being used to create and manage virtual machines. This VMM provides similar functionality as the Control Program (CP) in VM/370 [33], Disco [6], the VMWare Server ESX kernel, the Denali isolation kernel [37], or the Xen Hypervisor [2]. These VMMs allow multiple guest operating systems to execute on the same machine by providing them with a virtualised machine interface. In addition to guest OSes, we propose to also place parts of the I/O subsystem in virtual machines, constituting the part of a VCP executing on the main CPU(s). These virtual I/O machines export a virtual I/O interface, defining a virtual channel processor. The virtual I/O interface, indicated by solid lines in the diagram (in contrast to the non-public APIs in figure 1 between kernel components) are based on message queues, allowing the decoupling of computations in different virtual machines. Messages are passed asynchronously and contain references to the data residing in shared memory regions, allowing zero-copy communication on the data path.

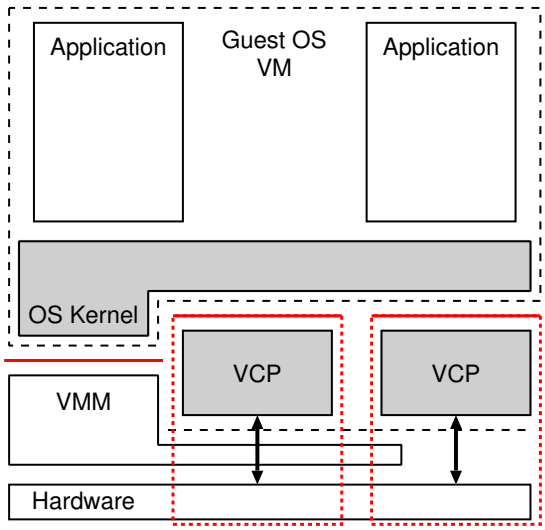

Figure 2: Virtualisation of kernel subsystems

A VCP may contain device drivers talking directly to the hardware device if the VMM supports this feature. Otherwise, it would have to access the hardware via an interface provided by the VMM. Furthermore, it is conceivable (and desirable) to allow user-level applications to instantiate and communicate directly with a VCP. However, this does require changes to the guest operating system's API to its user-level processes.

Placing parts of the I/O subsystem into separate virtual machines has several advantages. The obvious advantage is that the rest of the system is isolated from certain software faults within the I/O subsystem, potentially increasing the overall system stability (drivers contribute to a large proportion of operating system bugs [9]). Furthermore, faults in experimental software and software under development can be contained more easily when executed within a virtual machine, as only the virtual machine executing the experimental code is affected. Fault containment is a motivation shared with Hive [7][4]. Another software engineering benefit is that a virtual machine provides a "clean slate" programming environment. Programmers are freed from the constraints of the, say, Linux or Windows driver or I/O model and are not tied to a particular existing implementation and implementation environment. This significantly simplifies the implementation of a VCP. For example, programmers do not need to worry about locking with respect to concurrency with other kernel subsystems or worry about SMP related locking issues if a VCP only ever executes on a single CPU. We do acknowledge, that an initial implementation of VCPs may introduce instabilities and potential bugs in the main operating system itself as non-trivial changes are required to incorporate the concepts of VCPs into the kernel. However, we believe that after this initial modification the containment of the I/O subsystems in a VCP offers greater benefits.

A second benefit is that the VCP abstraction provides more flexibility to respond to changing system environments. Rather than hard-coding a software/hardware divide, virtualising I/O allows to evolve the technology used within the virtual channel processor independent of the rest of the system. In other words, the virtual machine approach does not preclude the use of hardware to accelerate I/O, rather it allows the choice of where to place the functionality without affecting the rest of the system — only the implementation within the virtual machine is affected. This obviously assumes that the virtual I/O interface, presented to the rest of the operating system, does not need to be changed. Designing suitable virtual interfaces is therefore a requirement for virtualising I/O. This interface is likely to be based on the message queue paradigm using shared memory, as it allows for efficient bulk data transfer, avoiding unnecessary data copies, while decoupling the individual components. It's worth pointing out that the design principles of such interfaces is well understood, partially motivated by the limitation of existing interface [4, 5, 26].

VCPs represent separate scheduling and resource allocation entities. Thus, placing I/O processing in a virtual machine makes it easier to control how much resources are consumed for I/O processing. We plan to leverage SMT features found in modern processors, i.e., dedicate a hardware thread or a guaranteed share of a hardware thread for the virtual channel processing performed by the main CPU(s), comparable to the dedicated CPU used in the TCP server system [28] or in the ETA prototype [29].

We note that the VMM does not need to "understand" the se-

[4]It is worth pointing out that fault containment alone is not sufficient to achieve better overall system stability and dependability — an appropriate recovery mechanism is also required. Furthermore, recovery from bad hardware state may require additional support from the hardware itself.

mantics of the I/O operations performed by the VCPs in detail. For example, we expect to be able to support the managing of buffer space in message queues using the same virtual memory interface the VMM exports to Guest OSes. Similar, the VMM has to export an asynchronous event notification mechanisms to Guest OSes, i.e., virtualised interrupts. The same mechanism could be used for the communication between VCP and other virtual machines. The only special requirement VCPs have on the VMM is specialised treatment with respect to scheduling. However, this is easily accomplished if an entire hardware thread is dedicated to a VCP (slack CPU resources of this hardware thread may still be shared on a best-effort basis).

3.1 iSCSI virtualised

With the general idea of virtual channel processors introduced, we now revisit the iSCSI example. Figure 3 shows one possible implementation of an iSCSI subsystem as a virtual channel processor and is in stark contrast to the more traditional software implementation depicted in Figure 1.

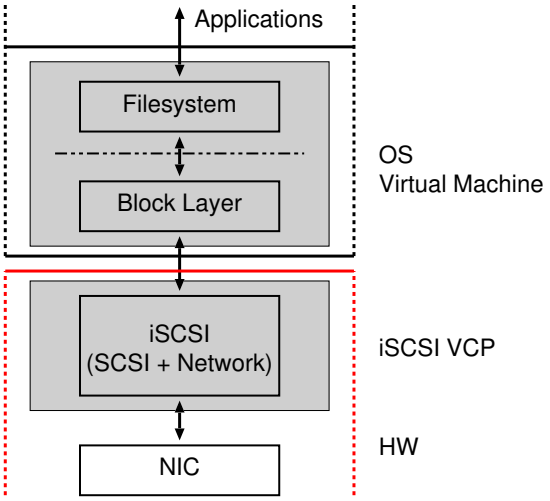

Figure 3: iSCSI virtualisation

The filesystem and block device layer remain virtually unchanged, but are now executed in a guest OS virtual machine. The block device layer may have to be changed to interface with the virtual I/O interface exported by the newly introduced iSCSI virtual machine. This new component now encapsulates both the iSCSI layer and a network stack connected to a network interface.

The network stack in the iSCSI virtual machine could be simplified and streamlined for iSCSI traffic, e.g., with respect to locking and supported TCP options, thus also making it less susceptible to complex bugs. Since the network stack is encapsulated within the virtual machine, we are also free to move some of the network processing onto the NIC should this provide better performance. Again, this decision could be made independent of the other components and would not effect the rest of the operating system. Moreover, if the network interface itself would be suitably virtualised both the standard network subsystem and the iSCSI virtual machine could used the same physical network device.

4. IMPLEMENTATION PLAN

We are planning to implement a prototype of the described virtual channel architecture using one of the recent virtual machine monitors, namely the Xen hypervisor [2]. Xen virtualises the IA-32[16] architecture and currently is capable of running multiple instances of the Linux operating system in separate virtual machines[5].

In contrast to full virtualisation of the underlying hardware Xen uses an approach termed para-virtualisation [37]. That is, it only virtualises a (large) subset of the physical hardware while providing idealised abstractions for the parts of the architecture which are difficult or extremely inefficient to virtualise. This results in changes being required to guest operating systems to be hosted by the Xen VMM.

In case of the Linux guest OS, known as XenoLinux, these changes are limited to the machine dependent part of the kernel source and practically introduce a new architecture which strongly resembles, but is not identical to the physical hardware. For example, the Linux kernel is executed in ring 1 instead of ring 0 of the IA-32 (user-level processes execute in ring 3 as in standard Linux and run unmodified Linux binaries). Thus, operations which require the privileges of ring 0, e.g., page table updates, are provided through a different interface rather than virtualised one. The architecture specific part of the Linux virtual memory system has been modified to reflect this change. In general, this form of para-virtualisation promises performance benefits over full virtualisation while keeping the changes required to guest operating systems to a minimum.

Unfortunately, the I/O namespace can currently not easily been completely virtualised, i.e., it is currently not possible to give individual virtual machines access to only portions of the physical I/O namespace (VMs can be given I/O privileges, allowing them to access the entire I/O namespace with the obvious reduction of isolation between VMs). Rather than attempting to virtualise and then emulate a specific physical device, an approach taken, e.g., by VMWare workstation [35], we plan to extend the idea of para-virtualisation to physical devices as well. Our current plan is to provide a virtualised I/O interface, again based on message queues, directly from the hardware. This can be provided by an approach similar to the Arsenic GigaBit Ethernet prototype [27]. An interesting alternative is offered by the Twin Cities prototype [12] which would allow direct interaction between the VCP and the hardware without VMM interaction. In this context we are also evaluating the VI architecture [10] as a potential option. The para-virtualisation of I/O interfaces is in line with the current Xen implementation which provides I/O virtualisation to Guest OSes through a similar interface. However, Xen currently executes device drivers for the physical device within the virtual machine monitor. We plan to evaluate the benefits and drawbacks of executing device drivers in separate VMs or as part of a VCP.

Our initial prototype will focus on a virtual channel processor implementation providing iSCSI functionality, as used as an example in this paper. We plan to compare its performance both against other software implementations as well as hardware assisted implementations. While previous work suggests potentially substantial performance benefits for performing network I/O processing on a separate processor [25, 28, 29] we are particularly interested in investigating the benefits of SMT technologies, such as Hyper-Threading, in this context. Using SMT for VCPs is interesting, because it offers potential performance improvements at significantly lower cost than SMP systems, and challenging, as hardware threads share some functional units of the processor which can become contented. Furthermore, we are interested in evaluating the overheads introduced through the use of virtualisation.

[5]Our choice for Xen is motivated in part due to its relative maturity and completeness compared to other available VMM prototypes and in part due to our involvement in its development.

For example, for network transmit and receive throughput the Xen and XenoLinux prototype only introduces a very small overhead of 0.2% for an MTU size of 1500 bytes when compared to standard Linux, but incurs a significantly higher overhead of 25% on transmission and 45% on reception for a smaller MTU size of 500 bytes (measure using `ttcp`). This significant per-packet overhead can be attributed to the current implementation where the VMM performs fire-walling and (de)-multiplexing. We believe that some of these performance bottlenecks can be removed when the network interface is used in more dedicated environments, e.g., within a VCP used for a particular purpose, or with more intelligent hardware devices performing some of this functionality.

5. RELATED WORK

The idea of placing I/O subsystems into separate virtual machines is not new. For example, as early as 1969 software for a telecommunication subsystem was placed in a special virtual machine on a CP-67 (a precursor to VM/370) controlled micro-computer to allow bulk remote access to the machine [13]. The idea of channel processors (albeit real ones) for I/O processing also originates in the early mainframe world [11]. With our proposal, we revitalise these ideas and apply them in a modern context.

A number of systems, e.g., Piglet [25], AsyMOS [24], the ETA prototype [29], and TCP servers [28], introduce virtual network device interfaces and use dedicated processors on an SMP system for protocol processing. Virtual channel processors are similar to this approach, however, rather than dedicating an entire processor for this purpose they provide a virtual machine, thus allowing a more dynamic allocation of resources. We hope that through higher processor speeds and use of SMT technology with appropriate scheduling algorithms similar performance benefits can be achieved as those reported for the above systems.

Some of our reasons for placing the I/O subsystems into separate virtual machines are shared with some of the motivations behind micro-kernels architectures, e.g., MACH [1] or L4 [19, 20]. For these it was proposed to move device drivers into user-level servers. However, we believe that the IPC interface typically used in micro-kernels is inferior to a message queue interfaces, especially with respect to bulk-data transfers. Furthermore, with appropriate VMM support our proposed system does not need to suffer from the extra cost of kernel/user mode crossings typically incurred on micro-kernel systems.

Staged computation [36, 18] decouples computation and has been demonstrated to achieve better scalability and performance. By virtualising I/O interface and using a message queue based interface we hope to gain similar benefits.

Finally, some of our goals are comparable to those of extensible operating systems projects like Vino [34] and Spin [3]. These systems use software mechanisms or safe languages to protect parts of the kernel against others. Instead, we propose to use virtual machine technology, such as provided by Xen, for isolation, thus providing more freedom to developers to use a programming language of their choice.

6. SUMMARY

In this position paper we have presented the concept of virtual channel processors. VCPs encapsulate the I/O subsystem of traditional operating system kernels, exporting a virtual I/O interface to other parts of the system.

We have argued that this approach has the benefit over traditional kernel structures by offering a safer execution environment for new or immature kernel code while providing resource isolation between kernel subsystems.

While we do not yet know if the tradeoffs we presented — placing more computation on the main CPU, leveraging more rapidly increasing CPU speeds and emerging technologies, such as SMT, instead of deploying dedicated off-loading engines — are justifiable today, we have argued that the virtual channel processor concept provides more flexibility compared to the current trend of hardware accelerating specific function of the I/O subsystem. VCPs allow the hardware/software tradeoff being made without having to changes the core of an operating system.

We have outlined an implementation plan which is based on the virtual machine monitor technology provided by the Xen hypervisor. Our initial prototype will target an iSCSI implementation such as we have used as an illustrative example for this paper.

7. REFERENCES

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Proc. of Summer Usenix*, pages 93–113, July 1986.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, A. Madhavapeddy, R. Neugebauer, I. Pratt, and A. Warfield. Xen 2002: The Xenoserver Hypervisor. Technical Report UCAM-CL-TR-533, Cambridge Computer Laboratory, Jan. 2003.

[3] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th Symposium on Operating Systems Principles (SOSP'95)*, pages 267–284, Copper Mountain, CO, USA, Dec. 1995.

[4] R. J. Black. *Explicit Network Scheduling*. PhD thesis, University of Cambridge Computer Laboratory, Apr. 1995. Available as Technical Report no. 361.

[5] J. C. Brustoloni and P. Steenkiste. Effects of Buffering Semantics on I/O Performance. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 277–291, Seattle, WA, USA, Oct. 1996.

[6] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proc. of the 15th Symposium on Operating Systems Principles (SOSP'95)*, pages 143–156, Copper Mountain, CO, USA, Dec. 1995.

[7] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proc. of the 15th ACM SIGOPS Symposium on Operating Systems Principles (SOSP'95)*, pages 12–25, Copper Mountain, Colorado, USA, Dec. 1995.

[8] J. Chase, A. Gallatin, and K. Yocum. End-System Optimisation for High-Speed TCP. *IEEE Communications*, 39(4):68–74, Apr. 2001.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical Study of Operating Systems Errors. In *Proc. of the 18th Symposium on Operating Systems Principles (SOSP'01)*, pages 73–88, Lake Louise, Canada, Oct. 2001.

[10] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, Dec. 1997.

[11] R. Cormier, R. Dugan, and R. Guyette. System/370 Extended Architecture: The Channel Subsystem. *IBM J. of Research and Development*, 27(3):206–218, May 1983.

[12] F. Hady, T. Bock, M. Cabot, J. Chu, J. Meinecke, K. Oliver,

and W. Talarek. Speeding Complex Networking Applications with Twin Cities, a Heterogeneous Multiprocessing Prototype. *Accepted for IEEE Network*, 2003.

[13] E. C. Hendricks and T. C. Hartman. Evolution of a Virtual Machine Subsystem. *IBM Systems J.*, 18(1):111–142, 1997.

[14] IETF. IP Storage Working Group. http://www.ietf.org/html.charters/ips-charter.html.

[15] R. Illikkal and R. Huggahalli. Benefits of Threads in TCP/IP Processing. Submitted for publication, Jan. 2003.

[16] Intel Corporation. *IA-32 Intel® Architecture Software Developer's Manual; Volume 1: Basic Architecture*, 2003.

[17] M. B. Jones and S. Saroiu. Predictability Requirements of a Soft Modem. In *Proc. of the Int. Conf. on Measurements and Modeling of Computer Systems*, pages 37–49, Cambridge, MA, USA, June 2001.

[18] J. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *Proc. of the Usenix Annual Technical Conference*, pages 103–114, Monterey, CA, June 2002.

[19] J. Liedtke. On μ-Kernel Construction. In *Proc. of the 15th Symposium on Operating Systems Principles (SOSP'95)*, pages 237–250, Copper Mountain, CO, USA, Dec. 1995.

[20] J. Liedtke. Toward Real Microkernels. *Comm. of the ACM*, 39(9):70–77, Sept. 1996.

[21] D. MacDonald and W. Barkley. Microsoft Windows 2000 TCP/IP Implementation Details. White paper, Microsoft, 2000.

[22] D. Marr, F. Binns, D. Hill, G. Hinten, D. Koufaty, J. Miler, and M. Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technical Journal*, 6(1), Feb. 2002.

[23] J. Mogul. TCP offload is a dumb idea whose time has come. In *Proc. of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, May 2003.

[24] S. Muir and J. Smith. AsyMOS - An Asymmetric Multiprocessor Operating System. In *Proc. of 1st IEEE Conf. on Open Architectures and Network Programming (OPENARCH'98)*, San Francisco, CA, USA, Apr. 1998.

[25] S. Muir and J. Smith. Functional Divisions in the Piglet Multiprocessor Operating System. In *Proc. of the 8th European SIGOPS Workshop*, pages 255–260, Sintra, Portugal, Sept. 1998.

[26] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, LA, USA, Feb. 1999.

[27] I. Pratt and K. Fraser. Arsenic: A User-Accessible Gigabit Ethernet Interface. In *Proc. of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 67–76, Los Alamitos, CA, Apr. 2001.

[28] M. Rangarajan, A. Bohra, K. Banerjee, E. Carrera, R. Bianchini, L. Iftode, and W. Zwaenepoel. TCP Servers: Offloading TCP/IP Processing in Internet Servers. Design, Implementation, and Performance. Technical Report DCS-TR-481, Rutgers University, Department of Computer Science, Mar. 2002.

[29] G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Fong. ETA: Experience with a Intel®Xeon™Processor as a Packet Processing Engine. In *Proc. of the Symposium on High Performance Interconnects*, Palo Alto, CA, Aug. 2003. accepted for publication.

[30] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: When Does Hardware Support help? In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, USA, Mar. 2003.

[31] P. Sarkar and K. Voruganti. IP Storage: The Challenge Ahead. In *Proc. of the 19th IEEE Symposium on Mass Storage Systems*, College Park, MD, USA, Apr. 2002.

[32] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. iSCSI. Internet Draft, draft-ietf-ips-iscsi-19.txt, Nov. 2002.

[33] L. Seawright and R. MacKinnon. VM/370 - A Study of Multiplicity and Usefulness. *IBM Systems J.*, pages 4–17, 1979.

[34] C. Small and M. Seltzer. VINO: An Integrated Platform for Operating Systems and Database Research. Technical Report TR-30-94, Harvard University, Cambridge, MA, USA, 1994.

[35] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMWare Workstation's Hosted Virtual Machine Monitor. In *Proc. of the Usenix Annual Technical Conference*, pages 1–14, Boston, MA, June 2001.

[36] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of the 18th Symposium on Operating Systems Principles (SOSP'01)*, pages 230–243, Lake Louise, Canada, Oct. 2001.

[37] A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proc of the 5th Symposium on Operating Systems Design and Implementation*, pages 195–209, Boston, MA, Dec. 2002.