Scout: A Communications-Oriented Operating System

Allen B. Montz   David Mosberger   Sean W. O'Malley   Larry L. Peterson
Todd A. Proebsting

Abstract

Scout is new communication-centric operating system. The principle scout abstraction (the path) is an attempt to capture all of the operating system infrastructure necessary to insure that a given network connection can achieve high and predictable performance in the face of other connections and other system loads.

1 Introduction

As the National Information Infrastructure (NII) evolves, and digital computer networks become ubiquitous, communication will play an increasingly important role in computer systems. In fact, a recent report on the NII rejects the term "computer" because of its emphasis on computation, and instead chooses to call these systems "information appliances" that support communication, information storage, and user interactions [10]. We expect these information appliances to include video displays, cameras, Personal Digital Assists (PDAs), thermostats, and data servers, as well as more conventional compute servers and desktop workstations. In many of these cases, computation will simply be viewed as something one does to I/O data as it passes through the system.

This position paper describes Scout, a new operating system (OS) being designed for systems connected to the NII. Scout is a configurable, communication-oriented operating system, whose performance is both predictable and scales with processor performance. We envision Scout being an appropriate OS for the following types of information appliances:

- Network devices, such as cameras, that are configured into system-area (desk-area) networks.
- Individual nodes that make up a scalable storage server connected to a high-speed network.
- Hand-held and portable devices that must adapt to varying network bandwidths.
- Multimedia workstations that receive, process/filter, and display high-bandwidth video data.
- Individual nodes of a distributed-memory multicomputer that is applied to large scientific problems.
- Application gateways that forward, and sometimes filter, data between multiple network domains.

2 Tenets

Claiming an operating system is suitable for the NII is easy. What makes Scout unique is the set of insights, experiences, and enabling technologies it leverages. The following identifies these forces by stating the four tenets that underly Scout. They both respond to the limitations and problems we see in current operating systems, and anticipate the role of the operating system five to ten years out.

2.1 Software Specialization

We envision the systems connected to the NII being constructed from inexpensive, commodity components, and then specialized in software to perform a particular task. For example, a scalable storage server can be built by connecting a collection of network disks (perhaps diskful PCs), each running the software modules needed to make it function as either a conventional UNIX file server, a special-purpose video server, or an application-specific database. As another example, a network camera might run modules that encode and compress the video, fragment the video frames into network packets, and transmit the packets according to some flow control algorithm.

The alternatives to software specialization is to either build specialized hardware, which is generally not a cost-effective solution, or to run a general-purpose operating system like Unix. Running a general-purpose OS has an obvious down-side: the mechanisms are often a poor match for the task at hand, or stated another way, their generality usually stands in the way of optimizing for the cases that

0-8186-7081-9/95 $04.00 © 1995 IEEE

are important to the particular application. Likewise, there is no need for the OS to be dynamically extensible when the task to be performed by the system is known beforehand. In short, there is no reason to require the OS running on a network camera to be POSIX compliant, nor is there is any value in being able to dynamically extend such an OS to support parallel computing.

Software specialization does not imply that each system must be built from scratch. Instead, the key is to provide a framework (toolkit) for configuring in the modules required by the application. This framework must be sufficiently general to support the expected diversity, but still narrow enough to allow effective optimization. In the case of Scout, the focus is on communication, which requires support for varying degrees of reliability, security, mobility, and real-time. To support this diversity, Scout will draw heavily from a predecessor system, the x-kernel [8], in which network protocols define the fundamental building blocks from which the system is configured. Scout will go beyond the x-kernel in two ways. First, it will broaden the scope from network-specific to all communication-related functions, most notably, file access. Second, the framework itself—e.g., the scheduler and resource allocators—will be configurable as well. In summary, our first tenet is:

> *The systems connected to the NII will need to be specialized in software, and hence, Scout is designed to be configurable—a given instance contains exactly the functionality required by the system for which it is built.*

2.2 Communication-Centric Design

A principal task of the OS running on the systems connected to the NII is to shuffle data between I/O sources and sinks, and possibly to compute on it, as efficiently as possible. Whereas support for software specialization often leads to horizontally layered software, taking an end-to-end perspective suggests the importance of a vertical cut—recognizing the *path* through the system's layers that control and data follow. We observe that there has recently been a trend in OS design to improve support for the implicit concept of a path, mostly for the sake of improving the performance of layered systems. For example, *fbufs* are a path-centric buffer management mechanism [5], *packet filters* are a mechanism for determining which path an incoming packet belongs on [14], and two recent systems provide support for the thread operating on behalf of a path-like computation to migrate across protection boundaries [7].

Scout takes this trend to its logical conclusion by making the path a first class abstraction.[1] Paths are perhaps best motivated in terms of the conventional compute-centric model of an operating system, which starts with a physical machine, defines one or more "virtual machines" on top of it, and typically culminates with a task abstraction that provides all the mechanisms and resources necessary to compute. In contrast, Scout starts with a physical network link, views physical processors as the means to extend this physical network link by "virtual links", and culminates with the path abstraction that provides all the mechanisms and resources necessary to communicate. In other words, a path represents the flow of data from an I/O source, through the system, to an I/O sink. It is defined by (1) the sequence of communication modules, and (2) the set of system resources, needed to move the data from the source to the sink.

[1] The name Scout is significant because of its meaning as a "path finder"; it is not an acronym.

Making paths a first class abstraction is important for two reasons. First, paths provide a place to locate the machinery needed to address the hardest problem facing any operating system—resource management. That is, all the resources required to send, receive, and process data are allocated and scheduled on a per-path basis. Second, explicitly identifying the control and data path is a necessary first step in being able to optimize that path. For example, the scheduling decision made at task boundaries in a conventional OS interferes with the optimal execution of an I/O path. It is our experience that neither resource allocation/accounting, nor optimizing the flow of control and data through the system, can be done effectively without an explicit path abstraction. The abstractions one finds in a conventional compute-centric OS are often at odds with the needs of I/O data paths. In summary, the second tenet of Scout is:

> *The concept of a path is fundamental to a communication-oriented system, and as a consequence, it is made an explicit abstraction in Scout. The path abstraction provides a focal point for addressing resource allocation and enabling effective optimizations.*

2.3 Managing System Entropy

A major force that shapes Scout is what we refer to as computer system entropy—the growing complexity of both operating systems and machine architectures, and the unpredictable (often inexplicable) artifact that results when the two are mixed. On the one hand, operating systems have become significantly more complex. We've gone from monolithic kernels (UNIX) , to microkernels with OS servers in user tasks (Mach), to microkernels with some of the servers put back in the kernel (OSF/1 MK 6.0, FLEX), to extensible systems that support a late binding of exactly what services are provided and where they are implemented (Spring, Lipto). At the same time, machine architectures are becoming more sophisticated. Instruction pipelines are becoming deeper (making it increasingly difficult to avoid processor stalls), and the growing disparity between memory and processor speeds is making it increasingly important to make effective use of the cache.

Given this situation, it is little wonder that operating systems are not becoming faster as fast as processors are becoming faster [11]. Our experiences integrating the x-kernel into Mach illustrates the problem [4]. For example, we have protocol stacks that have no better latency on DECstation 5000 workstations than they did on Sun3 workstations, and changing the load order of the object modules in the system sometimes led to a 50% change in network software latency. Both of these results are due to collisions in the instruction cache (I-cache). We have shown that paying careful attention to I-cache effects can improve protocol latency by 25%[9]. In the case of the data cache (D-cache), we found that only 5% of the message fragments brought into the cache when they are received are still there by the time the application is scheduled to run [12]; other studies have uncovered similar effects [3]. We also found that making the wrong branch prediction on a critical path can mean the difference between whether or not a real-time embedded system is able to live within its instruction budget [6]. Finally, we have observed numerous cases where the performance of micro-benchmarks bear little relationship to the performance of the system as a whole, and in fact, optimizing a particular module is just as likely to hurt overall system performance as help it. Others have reported similar phenomena [2].

The bottom line is that OS performance will not scale until the OS is able to consistently take advantage of those very architectural features that are responsible for improved performance—caches and instruction pipelines. Doing this is particularly important for real-time applications, and puts all applications on the microprocessor performance curve. This leads to our third tenet:

Scout will provide performance that both scales with processor performance, and is predictable—improvements to individual components of the system will lead to the expected improvement to the overall system.

2.4 Exploiting Compiler Technology

A key design principle in RISC architectures is that much of the responsibility for achieving good performance falls to the compiler. While this principle as been aggressively applied to application code, it is often given less attention in the case of operating system code. For example, the C programming language was originally designed in the early 70's to support writing system code. Since then, operating systems have changed, hardware has changed, and compiler technology has improved, but C has remained basically the same. While numerous programming languages, language extensions, and compiler optimizations have been proposed to support application-level programming in this new environment, little or nothing has been done to support low-level system programming. After 20 years it is time to revisit the issue of how a programming language and/or compiler could be enhanced to better support the construction of OS and other low-level system code.

Improving OS performance is the main reason to revisit the compiler issue. For example, our experience with protocol software shows that there is an opportunity for the compiler to automatically organize OS code so that it is more compatible with the memory architectures on modern RISC processors [1]. We have also found code patterns unique to operating systems, but not necessarily common in application-level code, that can be more heavily optimized [1]. Such optimizations are necessary if the operating system has any hope of staying on the processor performance curve.

Not only is the compiler a key to performance, but it is also an important tool for easing the task of the operating system implementor. This is especially critical for an OS like Scout that is designed to support specialization—the compiler is the right place to generate specialized code in a systematic and portable fashion. Similarly, the compiler can generate tedious, low-level code. For example, we have found in the x-kernel that some of the most error-prone and difficult to port components of the system—the code that deals with byte-order and byte-alignment issues—can readily be generated automatically. Not utilizing compiler support in cases like this is a waste of valuable programmer time and effort. Thus, Scout's final tenet is as follows:

Scout will leverage compiler technology for two purposes—to help put the OS on the processor performance curve, and to simplify the process of constructing and porting the OS.

3 Summary

The single most important idea in Scout is the introduction of the path abstraction. A path encapsulates the flow of data from an I/O source to an I/O sink. It consists of two things: the sequence of communication modules that define the path's semantics (e.g., its reliability, security, and real-time behavior), and the collection of system resources needed to process and forward the data along the path (e.g., CPU time, memory buffers, cache space). What one traditionally thinks of as an application program can either be a source or sink module, or perhaps a "filter" in the middle of a path.

The abstractions one finds in conventional, compute-centric operating systems are defined in terms of paths. For example, paths, rather than threads, are the schedulable entity in Scout. A path may be annotated with preemption information that defines points where a context switch from that path to another is permitted. It is even possible for a path (or some fragment of a path) to execute at interrupt time. Similarly, rather than beginning with protection domains (tasks) as a primary abstraction, and trying to manage the flow of I/O data across multiple domains, Scout starts with the path abstraction, and annotates it with zero or more fault-isolation boundaries [13] and zero or more privacy boundaries.

The path abstraction provides the focal point for realizing the goals outlined above. First, paths define an infrastructure for composing together various collections of protocols to provide different communication services. Second, paths are the right place to locate the machinery needed to address the hardest problem facing any operating system—resource management. That is, all the resources required to send and receive data, including the CPU, memory buffers, the I/O bus, the cache, and the TLB, are allocated and scheduled on a per-path basis. Third, the path abstraction defines a structure for both writing and generating code that leads to scalable and predictable performance. In particular, it suggests a programming paradigm that explicitly identifies the critical path and makes this critical path available for optimization. The optimizations come from both providing the right underlying OS mechanisms and from applying compiler techniques.

In summary, Scout provides a communication-oriented software architecture for building OS code that is specialized for the different information appliances available on the NII. By introducing an explicit path abstraction, we expect to be able to effectively optimize the critical paths through the layers of communication modules. These path-enabled optimizations, plus enhanced compiler support, will result in a specializable operating system that has both predictable and scalable performance.

References

[1] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, Oct. 1993.

[2] A. F. Brian N. Berhad, Richard P. Draves. Using microbenchmarks to evaluate system performance. In *Proc. Third Workshop on Workstation Operating Systems*, pages 148–153, Key Biscayne, FL (USA), Apr. 1992. IEEE.

[3] J. B. Chen and B. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Dec. 1993.

[4] P. Druschel, M. B. Abbott, M. Pagels, and L. L. Peterson. Network subsystem design. *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, 7(4):8–17, July 1993.

[5] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 189–202, Dec. 1993.

[6] P. Druschel, L. L. Peterson, and B. S. Davie. Experience with a high-speed network adaptor: A software perspective. In *Proceedings of the SIGCOMM '94 Symposium*, Aug. 1994.

[7] B. Ford and J. Lepreau. Evolving Mach 3.0 to use migrating threads. In *Winter 1994 Usenix Conference*, Jan. 1994.

[8] 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, Jan. 1991.

[9] D. Mosberger, L. L. Peterson, and S. W. O'Malley. Protocol latency: Mips and reality. Technical Report 95-02, Department of Computer Science, University of Arizona, Feb. 1995.

[10] NIST. *R&D for the NII: Technical Challenges.* Gaithersburg, MD, Mar. 1994.

[11] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. Summer 1990 USENIX Conf.*, pages 247–256, Anaheim, CA (USA), June 1990. USENIX.

[12] M. A. Pagels, P. Druschel, and L. L. Peterson. Analysis of cache and TLB effectiveness in processing network I/O. Technical Report 94-08, Department of Computer Science, University of Arizona, Mar. 1994.

[13] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th Symposium on Operating System Principles*, pages 203–216, Dec. 1993.

[14] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Winter 1994 Usenix Conference*, Jan. 1994.