Scout: A Communications-Oriented Operating System

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

TR 94-20

Abstract

This white paper describes Scout, a new operating system being designed for systems connected to the National Information Infrastructure (NII). Scout provides a communication-oriented software architecture for building operating system code that is specialized for the different systems that we expect to be available on the NII. It includes an explicit path abstraction that both facilitates effective resource management and permits optimizations of the critical path that I/O data follows. These path-enabled optimizations, along with the application of advanced compiler techniques, result in a system that has both predictable and scalable performance.

June 17, 1994

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

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 choses to call these systems "information appliances" that support communication, information storage, and user interactions [33]. 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 white 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.

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 next section identifies these forces by stating the four tenets that underly Scout. Following sections then describe the main research problems that must be addressed to realize Scout, compare Scout to other systems, and outline our research plan.

2 Tenets

This section presents the four tenets that shape 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 are important to the particular application. Likewise, there is no need for the OS to be dynamically extensibile 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 [27, 34], 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 Path Abstraction

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 [16], *packet filters* are a mechanism for determining which path an incoming packet belongs on [48], and two recent systems provide support for the thread operating on behalf of a path-like computation to migrate across protection boundaries [21, 24].

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.

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

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

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 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 [28]), to microkernels with OS servers in user tasks (Mach [3]), to microkernels with some of the servers put back in the kernel (OSF/1 MK 6.0, FLEX [10]), to extensible systems that support a late binding of exactly what services are provided and where they are implemented (Spring [24], Lipto [18]). 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 [36]. Our experiences integrating the x-kernel into Mach illustrates the problem [15]. 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). 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 [37]; other studies have uncovered similar effects [11]. 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 [17]. 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 [9].

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 Compiler Support

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 the OS.*

2.5 Summary

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.

3 Research

This section identifies the significant research topics that we must address to realize Scout. Although many of the topics are synergistic, we group them into three broad categories to help simpilify the discussion: path-related issues, software tools, and low-level optimizations.

3.1 Path Abstraction

A path is an OS abstraction that 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 [47] and zero or more privacy boundaries.

The path abstraction provides the focal point for realizing the goals outlined in Section 2. 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.

3.1.1 Integrated Resource Allocation

How to effectively allocate and schedule resources is perhaps the most fundamental problem faced by any operating system. In the case of Scout, the problem is one of allocating resources to paths. From a network perspective, each path needs the resources to extend the *quality of service* (QoS) provided by the network through to the application. Scout is not concerned with any specific QoS policy, but instead provides the mechanism needed to support a spectrum of policies.

Currently, the only resource that is explicitly allocated/scheduled is the processor. This is true of conventional OS's, as well as most designed specifically for real-time applications. All other resources, including the cache, the I/O bus, and often even memory buffers, are implicitly acquired once a thread is scheduled on the CPU. Furthermore, since existing systems do not treat paths as first class objects, these allocation and scheduling decisions have to be revisited at domain boundaries. There simply is no end-to-end, per-path resource management.

In contrast, Scout does resource allocation and scheduling on a per-path basis. What is more, resources are explicitly allocated in a coordinated and integrated manner. That is, Scout will provide mechanisms for reserving enough I/O bus bandwidth to receive data at network speeds, enough memory buffers to absorb bursty data, and enough data and instruction cache space to process the data without thrashing. It is important to coordinate the scheduling of all these resources, not just the processor, because these other resources both limit scalability and create unpredictable performance behavior.

Consider the example of memory allocation in more detail. On emerging high-speed networks, the static memory allocations found in most systems (e.g., a TCP connection is allocated 16kbytes of buffering when it is opened) is not sufficient. The potential bursts that will need to be absorbed on the receiver, and the amount of data that will need to be buffered for possible retransmission on the sender could easily be on the order of several megabytes. It will be necessary to dynamically, and perhaps even speculatively, allocate memory to paths. Moreover, this allocation must be to the entire path, independent of how many protection domains it traverses. In contrast, resources are typically accounted for on a per-domain basis in existing systems. Finally, it is not only important how much memory is allocated to each path, but also where that memory is (i.e., what address it is at). This is because the address influences where the buffer lies in the data cache. One could imagine allocating memory to concurrently active paths so as to effectively space-division multiplex the data cache.

Another example of integrated resource management involves accessing the disk. While QoS is a well-known idea in the networking community, it is not well understood in the area of file systems. It would be interesting to define the path abstraction such that it can support a disk at one end and provide a constant video rate at the other end. This is difficult because disk access times have a high variance, but fortunately, the same buffering strategy that can be used to hide network jitter can be used to hide variations in disk latency.

3.1.2 Caching

The model of paths just presented gives the impression that data always flows from one end of a path to the other as quickly as possible. In practice, however, it is sometimes desirable to cache data within the path so as to avoid having to traverse a high latency connection in the future. We will, therefore, explore the question of how to best integrate caches into the path abstraction.

Consider, for example, a path on a client system that is used to transfer file data. The obvious thing to do is to set up one path per open file. This does not work well if the file is cacheable, since some blocks will come from the cache and others from the network. One possible solution is to splice paths, so that one connects the client to the cache and another connects the cache with the network. Another option is to support parallel paths, one for blocks in the cache and another for blocks that are not in the cache. This implies having to multiplex between paths depending on the cache status of the block. Also, blocks that are read from the network have to be put into the cache. This could be a side-effect of the application-to-network path, or yet another path between the network and cache. The latter might make sense for a machine with a network interface that has a good DMA engine—Scout could DMA both to the application and to the cache. Furthermore, it might be necessary to flush a block out of the cache to read a new one, requiring one path to wait on another.

The path composition for writes is likely to be different from that for reads. With a write-through cache, blocks are probably written into the cache as a side-effect of the application-to-network path. This organization makes little sense for a write-back cache, however, because writes from the application to the cache are not immediately propogated to the network. Instead, file blocks are aged in the cache with the hope that some will die and not be written to the network at all. For this reason a write-back cache is likely to splice paths: the cache is a sink for paths connected to the applications and a source for paths connected to the network. The cache can be thought of as a switchboard for connecting these paths together. The complexity of the path interconnection is dependent on the type of file service being provided. In a conventional file system, the many paths into the cache are multiplexed onto the few paths out to the servers, but a single path in to the cache only connects to a single path out. In a striped file system like Zebra [25], there will be one path out of the cache per storage server, so that a single path in to the cache will be connected to many paths out.

In general, our goal is to build a toolkit of cache modules for Scout. These modules will differ in their caching strategies and how their paths are composed, allowing the cache structure to be tailored to the specific type of data being transferred and the architecture on which the cache is implemented. For example, data for NFS, Zebra, and the World-Wide Web could be cached differently, thereby enhancing the application-specific configurability of Scout.

3.1.3 Path-Centric Optimizations

An explicit path abstraction enables several optimizations. One style of optimization suggested by the very definition of paths is to minimize the performance impact of re-scheduling the CPU and crossing protection boundaries, or in some cases, to eliminate scheduling and protection boundary costs altogether (i.e., the entire path runs uninterrupted in supervisor mode). This is a coarse-grained optimization that can be summarized as removing heavy-weight OS mechanisms from the path. Scout will also explore finer-grained optimizations that are targeted at the sequence of modules that make up a path.

The key observation is that while operating systems are quite large, only a small fraction of that code accounts for the vast majority of instructions actually executed. It is therefore generally accepted that programmer effort should be concentrated on code that has a high probability of being executed—the so called *critical path*. Unfortunately, the typical "by hand" approach to critical path programming has several problems. First, critical paths tend to snake through large amounts of non-critical code. This decreases I-cache efficiency in that significant amounts of non-critical (and hence unexecuted) code is loaded into the I-cache. For example, Digital Equipment Corporation estimates that operating system code running on the DEC Alpha (21064) has a cache efficiency of 50%—only half the instructions loaded into the I-cache are ever executed [42]. Second, the scattering of the critical path almost randomly through an address space increases the likelihood that two parts of the same path will compete for the same location in a direct mapped I-cache. In both cases, poor I-cache utilization hurts performance [30]. A final problem with hand-coded critical paths is that the compiler knows nothing about them, and because of this, cannot correctly predict branches or do optimal register allocation.

Scout, therefore, adopts a systematic approach to identifying and optimizing critical path code—it makes critical paths, defined at the basic block level, explicitly visible to both the application and the compiler. When an application creates a Scout path, the critical path blocks will be linked together to provide a seamless critical path running from the device to the application. Segregating critical path code into contiguous memory will increase the likelihood that cache lines hold useful instructions (because the *density* of the code is high) and it will decrease the likelihood of cache collisions (because the critical path code is likely to be small, and hence, fit entirely in the cache). Furthermore, because critical path information will correctly indicate branch predictions, instruction pre-fetches will be more likely to correctly load both the cache and the pipeline.

While segregating critical path code from extraneous code will be a straightforward static optimization for our compiler, that is not enough. Because paths are composed at run-time—for example, when a network connection is opened—it will be necessary to reorganize the code for the critical paths at run-time into contiguous blocks. Fortunately, by splitting the work between the compiler and a run-time linker, this task will be both manageable and efficient. The compiler will generate the code *chunks* for each potential component of a path. These chunks will contain relocation information so that a run-time linker can put them in memory at any location by simply copying the instructions and adjusting addresses. Once a path is dynamically specified, the component pieces will be efficiently relocated into contiguous memory locations for optimal I-cache performance. In other words, a run-time *code relocation* facility will allow critical path code to be moved within the instruction space dynamically. Note that this optimization will only be done on critical path code, thereby reducing the danger of code explosion.

Finally, not all the code blocks that make up the critical path will be written by the programmer. Scout will also use run-time code generation to generate highly optimized code for small sequences of critical path code. Run-time code generation is the creation of executable code by an executing process; it enables programs to create specialized instruction sequences based on runtime information. For instance, the Synthesis Kernel [29] was able to generate highly optimized code sequences for reading and writing files tailored to dynamic information that was determined at file open time. Similarly, we have experimented with dynamic code generation for network protocols in Morpheous [2]. Scout will build on these two efforts.

3.2 Tools

Scout defines a collection of tools that will be created by isolating small, well-defined types of operating system functionality. From several instances of a given type of functionality, we will generalize and create a specification language. We will then build a *small compiler* that will generate an efficient OS component from a program written in the specification language. Note that these tools are designed to support the construction of arbitrary operating systems, not just Scout. That is, these tools will be as much a tangible result of this work as is Scout itself.

The following are four examples of such tools that we already recognized as being valuable. We expect to discover and implement more as this work proceeds.

3.2.1 Universal Stub Compiler

Writing correct, portable, and efficient byte swapping and alignment-sensitive code by hand is not an easy task. Scout, like most operating systems, requires byte swapping code in three distinct places: interfacing with devices with non-native byte-orders and alignments, reading and writing protocol headers, and presentation layer processing. Traditional presentation layer stub compilers such as ASN.1 or XDR are not flexible enough to generate the type of byte swapping code required for device interfaces and some network headers. Even when flexible enough, the poor performance of their generated stubs makes them unsuitable for use in Scout.

We have therefore designed and are implementing a new special-purpose stub compiler, called the Universal Stub Compiler (USC), that automatically generates stubs to convert a C data structure with one user-defined format to a C structure with another user-defined format [35]. USC is general enough to generate all byte swapping and alignment-sensitive code in Scout. In particular, USC automatically generates code from a concise specification, generates nearly optimal code, is protocol independent, provides nearly unlimited access to network data, and is easily portable.

3.2.2 Universal Protocol Compressor

Scout must scale across networks with very different performance characteristics. This leads to conflicts when designing communications protocols: the data transferred can either be packed as tightly as possible for good performance on low bandwidth networks, or packed in such a way as to make host processing simpler to support high bandwidth networks. This is a fundamental tradeoff in network protocol design. If an application can produce network data more quickly than the network can transfer that data, then the throughput of the system can be improved by compressing the data prior to transmission. On the other hand, if the network can accept data faster than the application can produce it, then the throughput of the system can be improved by processing the data as little as possible before sending. This optimization trades processor cycles for bandwidth. Scout attempts to resolve this problem by creating a *Universal Protocol Compressor* (UPC) which can be used to generate the code necessary to compress arbitrary structured network data in an efficient fashion.

While many different data compression techniques have been widely studied, most fail to fully exploit the structure and redundancy of network protocols to achieve maximal compression. A recent study shows that compression algorithms that view network data—in the case of this study, X-window messages—as a simple *byte-stream*, rarely achieve excellent compression rates [14]. The study reveals that compressing different message types separately (i.e., treating a mouse message differently from an image message) would help compression ratios. Additionally, utilizing knowledge about how the fields of different message types are likely to change can aid a smart compressor (e.g., rather than sending the mouse's coordinates, send the delta from the last value sent). These observations led to the compression of X-window messages that is twice as effective as previous unstructured techniques. Unfortunately, the approach is not useful for other structured network data because it was created by hand for only X-window messages.

In contrast, UPC automates and optimizes the compression of structured protocol data. From annotated specifications of data structures (given in a C-like syntax similar to that used by USC) UPC will generate a suite of compressors—one for each data structure. The annotations will include directives to indicate how each field should be encoded before compression. For instance, some fields will be encoded as-is, and others will be encoded as a delta from the last transmitted value for that field. Some may not be encoded at all if the values are redundant! While these annotations may be generated by the developer, we will also provide a tool that infers from network traces how each value can be best encoded in the future.

UPC will be a tool that generates compressors optimized for a given set of annotated data structures. Because of its flexibility, UPC will be able to handle network headers, X-window messages, and so on, without any special effort from implementors. Automatic generation of compressors will reduce the penalty of poorly designed (redundant) network data, increase the effectiveness of compression, and ease the incorporation of compressors into network systems.

3.2.3 Integrated Layer Processing

Both of the preceding tools manipulate the I/O data; i.e., load, process, and store each word of the data being transmitted. Other examples of data manipulations common in network communication include computing checksums, doing *forward error correction* (FEC), and performing data encryption. These functions are becoming more and more important as the gap between processor and memory speeds widens. *Integrated layer processing* (ILP) is a strategy that mitigates the cost of accessing network data multiple times[13]. The idea is to load each word of data once, perform all the computations required by the connection while the word remains in the CPU's registers, and then store the word back to memory.

We have previously engineered a general mechanism for realizing the ILP strategy, and investigated its performance limits [1]. We plan to apply these results to Scout by providing a "meta-tool" that allows the various data manipulation tools we build to be integrated. That is, USC and UPC will be designed so that they can be composed together on a path in a such a way that the I/O data is loaded and stored only once. Similarly, tools that compute a checksum and implement FEC will be designed to work in concert with USC.

3.2.4 Dynamic Code Generator

As described in Section 3.1.3, Scout will use run-time code generation to produce highly optimized path-related code sequences. Unfortunately, taking advantage of dynamic code generation has many difficulties.

Because binary instructions are generated, programs using dynamic code generation must be retargeted for each new machine—a potentially substantial programming effort. Differing memory subsystems (e.g., split instruction and data caches) present additional retargeting difficulties because code is generated in data space and then executed in instruction space. Furthermore, dynamic code generation must be efficient since the code generation time will be incurred by the executing program.

Scout, therefore, includes a *Dynamic Code Generator* (DCG) that portably and efficiently generates executable code at run-time [19]. DCG client programs specify dynamically generated code using the compact, machine-independent intermediate representation (IR) of the `lcc` compiler [22]. The machine-independent intermediate representation specifies a small, but rich, set of operators that are sufficient to express all C language constructs at nearly a machine-level, without sacrificing portability. Binary code is selected and emitted using the BURS tree pattern-matching technology [39, 23]. The code generation interface is small and easy to use—clients specify expression trees for a desired chunk of code and DCG returns a function pointer callable from the client program. A prototype implementation of DCG is also very efficient—the creation of IR and subsequent translation to binary instructions takes approximately 350 cycles per generated instruction. We are automating code generator retargeting by developing simple machine specification languages and preprocessors. To our knowledge DCG is the only proposed retargetable dynamic code generator to emit binary instructions directly.

3.3 Low-Level Optimizations

A major research thrust of Scout will be to carefully examine and re-evaluate the low-level abstractions, mechanisms, and implementation techniques found in most operating systems. This section illustrates our strategy for optimizing Scout by focusing on two particular places where significant improvements seem possible: the compiler used to build Scout, and how Scout implements synchronization.

Although improved performance is an obvious motivation for these optimizations, it is also likely that they will help to make the code more predictable. That is, they will reduce the variance in how long the sequence takes to execute. For example, the synchronization technique described in Section 3.3.2 not only produces faster code, but also reduces the standard deviation among a set of runs by almost an order of magnitude.

3.3.1 Compiler Optimizations

Many of Scout's low-level optimizations take the form of compiler optimizations. Traditional compiler optimizations are inadequate to handle the idiosyncratic demands of operating system code sequences. Somewhat more surprisingly, traditional compiler optimizations also appear to be inadequate to handle new high-performance host architectures. Note that although many of these optimizations could also be applied to application code, they are unique in that they are most strongly motivated by the characteristics of the OS. The following is a list of optimizations that we are exploring:

Register Allocation at Procedure Calls: Global register allocation is a fundamental component of any optimizing compiler. By convention, compilers must treat certain registers as callee-saved and others as caller-saved across call boundaries. While these conventions hold for user-level procedures, interrupt handling procedures must treat *all* registers as callee-saved. To be able to write handlers in C, it is necessary to indicate to the compiler what the calling conventions are for calls to a particular procedure. For example, this would allow an interrupt handler that simply counts clock ticks to save registers only when an exceptional condition occurs (e.g., the counter reaches a certain threshold); no registers need be saved in the common case.

Last Call Optimization: Operating systems—especially those incorporating object-oriented techniques—are frequently designed with many small procedures, each of which do a small, well-defined task and then call another such procedure. This structure is certainly common in protocol processing. A byproduct of this structure is that ultimately there is a long chain of procedure returns that serve only to unwind the call stack and hand control back to a routine waiting to do useful work. A *last call* optimization, which is common in functional and logic programming translators [4], can be employed to minimize the overhead of this program structure. This will make it possible to avoid certain procedure call prologue and epilogue code, as well as the chain of jumps that occur at return time.

Instruction Selection and Scheduling: Instruction schedulers must optimize for loads and stores (memory traffic) with respect to the pipeline and cache constraints, and, to a lesser degree, for branches with respect to branch-prediction mechanisms. Moreover, multiple-instruction issue machines like the Alpha present difficult concerns about how to group instructions for simultaneous execution for optimal performance [5]. While instruction schedulers know how to handle RISC-style delayed-loads [40], scheduling for cache misses is not currently done well. A simple attack is to separate loads from the instructions consuming their values by at least as many cycles as a cache miss. Unfortunately, matters become a more complicated if the architecture limits the number of simultaneously outstanding cache misses, as it is necessary to know (or guess) which access is most likely to miss.

3.3.2 Lock-Free Synchronization

As in any OS, coordinating asynchronous events is a fundamental problem in Scout. The atomic sequence—a sequence of instructions that needs to execute without interference—is the basic building-block to tackle this problem. Atomic sequences can be either realized with lock-based or with lock-free synchronization. Scout will use lock-free synchronization as it avoids the danger of dead-lock and because it is applicable to kernel modules that need to communicate with interrupt handlers. For multi-processors, atomic sequences using lock-free synchronization have to be constructed out of hardware primitives, such as load-linked/store-conditionally, that are often a poor match for the problem at hand. The overheads are therefore often higher than those for lock-based synchronization. Fortunately, the same need not be true for uni-processors. The classic uni-processor solution to lock-free synchronization is to disable interrupts while executing an atomic sequence. The drawback of this solution is that it requires manipulating a system's interrupt priority level while in privileged mode.

Scout will use a novel synchronization technique [31] that is not only applicable throughout the OS (not just in privileged mode), but according to preliminary measurements on the Alpha platform, also performs significantly better than the classic technique of disabling interrupts. The idea of this technique is to optimistically assume that an atomic sequence will be executed free of interference, and should this assumption be violated, recover from it. Compared to lock-based synchronization, this means that the overhead per execution of an atomic sequence is very small (possibly zero) but that interference incurs an additional overhead that may be larger than that for lock-based synchronization. A lock-free solution out-performs a lock-based scheme as long as atomic sequences are executed more frequently than there is interference, a common and reasonable assumption.

In other words, instead of *preventing* interference, our technique *recovers* from interference. After detecting that an atomic sequence has been interrupted, there are two possible recovery paths. The partially executed atomic sequence could be rolled back to the beginning or it could be rolled forward to the end. After recovery, rollback will leave the system in a state that is as if the atomic sequence had not been executing at all [7]. On the other hand, rollforward will leave the system in a state that is as if the atomic sequence had been executed to the end already. Either solution guarantees atomicity. However, not all atomic sequences can be rolled back (consider, for example, an atomic sequence that first fires a rocket and then increments a counter). In practice, this limitation is severely aggravated due to efficiency concerns. Rollforward does not have such an inherent limitation. Scout will therefore use and study rollforward as a potentially more viable technique to achieve lock-free synchronization.

4 Related Systems

This section discusses several complete systems that are particularly relevant to Scout. Individual ideas and techniques have already been cited where appropriate in the body of the paper.

The use of run-time code generation in operating systems was explored in the Synthesis Kernel [29], as were some of the techniques we plan to use to improve cache utilization. However, the Synthesis Kernel differs from Scout in that it was implemented entirely in assembler, and the cache optimizations were all done by hand. Scout will leverage both the compiler and the path abstraction to automate these techniques.

The Plan 9 operating system [38] shares with Scout the characteristic of being implemented from scratch on a RISC processor, and like Scout, it involved the design of a new C compiler [43]. This C compiler had minor extensions, and except for showing some concern about callee-saved registers, does not duplicate any of the work described in this paper.

In [8], a set of quality of service extensions to the Chorus operating is described. These extensions are designed to allow incoming data streams to maintain the same kind of QoS guarantee in the host as provided by the network. Unlike Scout, this work was done in the context of Chorus and hence no explicit path abstraction is possible. This work also does not attempt to schedule such low level hardware features as the cache or memory bus.

Scout has definite application as an operating system for embedded and real-time systems. Although there are many examples of such systems—for example, iRMX [46], RT-Mach [44], and QNX [26]—Scout's path abstraction provides the right specialized framework—one tailored to support communication—to enable a configuration that can be easily and efficiently optimized. Note that while our particular focus is more on soft real-time than hard real-time, Scout's effort to make OS performance scale predictably with processor performance should also be useful in the construction of hard real-time systems. Using Scout, such systems would then be able to schedule for worst case behavior that more closely tracks processor performance.

Hopefully the differences between Scout and other recent experimental operating systems (e.g., Amoeba [32], V [12], Chorus [41], Mach [3], QNX [26]) are obvious at this point—they can all be traced to the difference between taking a communication-oriented perspective and a compute-oriented perspective. On the other hand, these two models are really duals of each other: compute-centric systems certainly support communication, and vice versa. Perhaps more tangibly, the difference has to do with how these systems treat networking—while Scout provides all the necessary tools to construct high-performance inter-machine communication, the communication protocols themselves are not part of Scout. In contrast, these other systems provide some form of inter-machine communication as an integral part of the system. In fact, those operating systems postulate that intra-machine communication is simply a degenerate form of inter-machine communication. This is difficult to justify, considering the vast differences in the respective problem domain. For example, machine failures, heterogeneity, and interoperability are non-issues for intra-machine communication. On the other hand, there are many different ways in which intra-machine communication can occur, depending on efficiency, safety, and flexibility requirements. Given these differences, Scout makes no attempt to unify intra- and inter-machine communication. It concentrates on the former, leaving the latter to high-level, configurable software.

Finally, we compare Scout with other new efforts to re-define the structure of the operating system. Three notable examples include Spring developed at Sun Labs [24], SPIN from the University of Washington [6], and the Mach enhancements being done at the University of Utah [10, 21]. On one level, these efforts are complementary to ours: the resource allocation mechanisms, the software tools, and the compiler optimizations developed for Scout could be used in these other systems. On another level, however, there is an important difference in philosophy between Scout and these other systems. Systems like Spring and SPIN support a very dynamic view of extensibility; they allow a single system to be tailored, at run-time, to serve many masters. In contrast, Scout takes a rather static view of configurability: the OS for a particular system is configured at build time, and it does not change while the system is running. New data connections can be dynamically opened and closed, but once an instance of Scout is constructed for use on a network camera, it cannot be dynamically extended to support a database application. Although Scout can be configured as a more general-purpose OS supporting multiple independent users—for example, we plan to write a Unix emulator for Scout—its real advantages become apparent when Scout is specialized for a small number of long-lived applications that are known *a priori*.

5 Research Plan

A prototype of Scout is currently being implemented on Alpha-based workstations. We are using a port of the x-kernel running stand-alone on Alphas as the departure point. This port is finished, and we are now refining the path abstraction and implementing the tools and optimizations described in earlier sections. The compiler optimization work is being done in `gcc`.

This Alpha-based implementation will demonstrate Scout's viability as an operating system for a multimedia workstation capable of receiving, filtering, and displaying video data. In this case, we plan to configure Scout with the X-Window protocol, the NFS protocol, and AudioFile protocol, as well as some example video filtering modules. In addition, we plan to port Scout to different platforms, and demonstrate how it can be specialized for different applications:

Network Devices: We will port Scout to the LANai boards being built for the NetStation system-area network [20]. Here, Scout will be configured to support a video camera, and thus will encode, compress, and transmit video frames.

Scalable Storage Server: We will port Scout to a set of diskful workstations connected to a high-speed network, and configure them to stripe file data across the set of servers [25].

Cluster Computing: We will port Scout to a set of of workstations connected by a high-speed network, and configure them to support the run-time system of the C* data parallel language [45].

An important question is how we will evaluate the successes and failures of Scout. On the one hand, most of the software tools and low-level optimizations can be evaluated quantitatively; we will measure how much they contribute to latency and throughput, as well as how much variance they introduce into the system. In the case of the path abstraction, particularly those aspects having to do with integrated resource allocation, a quantitative evaluation is much more difficult. Our belief is that the path abstraction will enable a more comprehensive method for allocating and accounting for resources than is currently possible; that is, functionality is improved. It may also be the case, however, that there are some measurable effects; e.g., Scout can effectively handle more frames per second or higher resolution video than existing operating systems on the same hardware.

We envision two outlets of technology transfer for this work. First, we envision Scout, itself, being used on the kinds of systems identified above. We will distribute Scout to the research community for this purpose. Second, we will produce a set of tools and techniques that can be applied to operating systems and network software in general. In particular, we plan to distribute software tools like USC and DCG, as well as our `gcc` enhancements.

Finally, regarding the relationship between Scout and x-kernel, our plan is to leverage as much of the x-kernel as possible in the design and implementation of Scout. For example, Scout will borrow interfaces and protocol implementations from the x-kernel whenever possible. Looking at this issue in the other direction, we expect to incorporate many of the ideas developed in Scout back into the x-kernel, which we will continue to support on several different platforms.

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] M. B. Abbott and L. L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, Feb. 1993.

[3] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference*, July 1986.

[4] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[5] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 241–255, 1991.

[6] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN: An extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Department of Computer Science, University of Washington, Feb. 1994.

[7] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–237, Oct. 1992.

[8] G. Blair, A. Campbell, G. Coulson, D. Hutchinson, M. Papathomas, and P. Robin. On the implementaiton of a quality of service controlled ATM based communicaitons system in Chorus. Technical Report MPG-94-11, Lancaster University, 1994.

[9] 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.

[10] J. Carter, B. Ford, M. Hiber, R. Kuramkote, J. Law, J. Lepreau, D. Orr, L. Stoller, and M. Swanson. Flex: A tool for building efficient and flexible systems. In *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, Oct. 1993.

[11] 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.

[12] D. R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, Mar. 1988.

[13] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, Sept. 1990.

[14] J. Danskin. Higher bandwidth X. Submitted to *ACM Multimedia 94*.

[15] 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.

[16] 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.

[17] 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.

[18] P. Druschel, L. L. Peterson, and N. C. Hutchinson. Beyond micro-kernel design: Decoupling modularity and protection in Lipto. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, pages 512–520, Yokohama, Japan., June 1992.

[19] D. R. Engler and T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[20] G. Finn. An integration of network communication with workstation architecture. *Computer Communication Review*, 21(5):18–29, Oct. 1991.

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

[22] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software—Practice and Experience*, 21(9):963–988, Sept. 1991.

[23] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, Apr. 1991.

[24] G. Hamilton, M. L. Powell, and J. J. Mitchell. Subcontract: A flexible base for distributed programming. In *14th Symposium on Operating System Principles*, pages 69–79, Dec. 1993.

[25] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. In *14th Symposium on Operating System Principles*, pages 29–43, Dec. 1993.

[26] D. Hildebrand. An architectural overview of QNX. In *Proceedings of the Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, Apr. 1992.

[27] 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.

[28] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, 1989.

[29] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[30] S. McFarling. Program optimization for instruction caches. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[31] D. Mosberger, P. Druschel, and L. L. Peterson. Fast and general software solution to mutual exclusion on uniprocessors. Technical Report 94-07, Department of Computer Science, University of Arizona, Mar. 1994.

[32] S. J. Mullender, G. van Rossum, A. S. Tanebaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[33] NIST. *NII Forum*. Gaithersburg, MD, Mar. 1994.

[34] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[35] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, Aug. 1994.

[36] 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.

[37] 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.

[38] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *Proceedings UKUUG Conference*, July 1990.

[39] T. A. Proebsting. Simple and efficient BURS table generation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 331–340, June 1992.

[40] T. A. Proebsting and C. N. Fischer. Linear-time optimal code scheduling for delayed-load architectures. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 256–267, June 1991.

[41] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems Journal*, 1(4):305–370, Dec. 1988.

[42] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Burlington, Massachusetts, 1992. Order number EY-L520E-DP.

[43] K. Thompson. A new c compiler. Technical report, AT&T Bell Laboratories.

[44] H. Tokuda and T. Nakajima. Evaluation of real-time synchronization in real-time mach. In *USENIX Symposium Proceedings*, Monterey, Calif., Nov. 1991.

[45] C. J. Turner, D. Mosberger, and L. L. Peterson. Cluster-C*: Understanding the performance limits. In *Proceedings of the Scalable High Performance Computing Conference*, May 1994.

[46] C. Vickery. *Real-Time and Systems Programming for PCs*. McGraw-Hill, 1993.

[47] 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.

[48] 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.