

Highly Configurable Operating Systems for Ultrascale Systems*

Arthur B. Maccabe and
Patrick G. Bridges
Department of Computer
Science, MSC01-1130
1 University of New Mexico
Albuquerque, NM 87131-0001
maccabe,bridges@cs.unm.edu

Ron Brightwell and
Rolf Riesen
Sandia National Laboratories
PO Box 5800; MS 1110
Albuquerque, NM 87185-1110
rbbbrigh,rolf@cs.sandia.gov

Trammell Hudson
Operating Systems Research,
Inc.
1729 Wells Drive NE
Albuquerque, NM 87112
hudson@osresearch.net

ABSTRACT

Modern ultrascale machines have a diverse range of usage models, programming models, architectures, and shared services that place a wide range of demands on operating and runtime systems. Full-featured operating systems can support a broad range of these requirements, but sacrifice optimal solutions for general ones. Lightweight operating systems, in contrast, can provide optimal solutions at specific design points, but only for a limited set of requirements. In this paper, we present preliminary numbers quantifying the penalty paid by general-purpose operating systems and propose an approach to overcome the limitations of previous designs. The proposed approach focuses on the implementation and composition of fine-grained composable *micro-services*, portions of operating and runtime system functionality that can be combined based on the needs of the hardware and software. We also motivate our approach by presenting concrete examples of the changing demands placed on operating systems and runtimes in ultrascale environments.

1. INTRODUCTION

Due largely to the ASCI program within the United States Department of Energy, we have recently seen the deployment of several production-level terascale computing systems. These systems, for example ASCI Red, ASCI Blue Mountain, and ASCI White, include a variety of hardware architectures and node configurations. In addition to differing hardware approaches, a range of usage models (e.g., dedicated vs. space-shared vs. time-shared) and program-

*This work was supported in part by Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

ming models (e.g. message-passing vs. shared-memory vs. global shared address space) have also used for programming these systems.

In spite of these differences and other evolving demands, operating and runtime systems are expected to keep pace. Full-featured operating systems can support a broad range of these requirements, but sacrifice optimal solutions for general ones. Lightweight operating systems, in contrast, can provide optimal solutions at specific design points, but only for a limited set of requirements.

In this paper, we present an approach that overcomes the limitations of previous approaches by providing a framework for configuring operating and runtime systems tailored to the specific needs of the application and environment. Our approach focuses on the implementation and composition of *micro-services*, portions of operating and runtime system functionality that can be composed together in a variety of ways. By choosing appropriate micro-services, runtime and operating system functionality can be customized at build time or runtime to the specific needs of the hardware, system usage model, programming model, and application.

The rest of this paper is organized as follows: section 2 describes the motivation for our proposed system, including the hardware and software architectures of current terascale computing systems and the challenges faced by operating systems on these machines, and presents preliminary numbers and experiences to outline the scale of this problem. It also presents several motivating examples that are driving our design efforts. Section 3 describes the specific challenges faced by operating systems in ultrascale environments, and section 4 presents our approach to addressing these challenges. Section 5 describes various related operating system work, and section 6 concludes.

2. MOTIVATION

2.1 Current and Future System Demands

Modern ultrascale systems, for example the various ASCI machines and the Earth Simulator, have widely varying system-level and node-level hardware architectures. The first terascale system, ASCI Red, is a traditional distributed memory massively parallel processing machine – thousands of nodes, each with a small number of processors (2). In contrast,

the ASCI Blue Mountain machine was composed of 128-processor nodes, while ASCI White employs 16-way SMP nodes. We also expect additional hardware advances such as multi-core chips and processor-in-memory chips to be available in similar systems in the near future.

In addition to hardware, the approach from a programming model standpoint has varied as well. The lightweight compute node operating system on ASCI Red does not support a shared-memory programming model on individual compute nodes, while the other platforms support a variety of shared memory programming constructs, such as threads and semaphores. This has led to the development of mixed-mode applications that combine MPI and OpenMP (or pthreads) to fully utilize the capabilities of systems with large numbers of processors per node. Applications have also been developed for these platforms that extend the boundaries of a traditional programming model. The distributed implementation of the Python scripting language is one such example [12]. Advanced programming models, such as the Global Address Space model, are also gaining support within the parallel computing community.

Even within the context of a specific programming model such as MPI, applications can have wide variations in the number and type of system services they require and can also have varying requirements for the environment in which they run. For example, the Common Component Architecture assists in the development of MPI applications, but it requires dynamic library services to be available to the individual processes within the parallel job. Environmental services, such as system-level checkpoint/restart, are also becoming an expected part of the standard parallel application development environment.

The usage model of these large machines has also expanded. The utility of capacity computing, largely driven by the ubiquity of commodity clusters, has led to changes in the way in which large machines are partitioned and scheduled. Machines that were originally intended to run a single, large parallel simulation are being used more frequently for parameter studies that require thousands of small jobs.

2.2 Problems with Current Approaches

General-purpose operating systems such as Linux provide a wide range of services. These services and their associated kernel structures enable sophisticated applications with capabilities for visualization and inter-networking. This generality unfortunately comes at the cost of performance for *all* applications that use the operating system because of the overheads of unnecessary services.

In an initial attempt to measure this performance difference, we compared the performance of the *mg* and *cg* NAS B benchmarks on ASCI Red hardware [19] when running two different operating systems. We use Cougar, the productized version of the Puma operating system [24], as the specialized operating system, and Linux as the general-purpose operating system. To make the comparison as fair to Linux as possible, we have ported the CplantTM version of the Portals high-performance messaging [1] layer to the ASCI Red hardware. Cougar already utilizes this Portals for message transmission.

Figures 1 and 2 show the performance of these benchmarks when running on two different operating systems. Linux outperforms Cougar on the *cg* benchmark with small numbers of nodes because Cougar uses older, less optimized compilers and libraries, but as the number of nodes used increases, application performance on Linux falls off. Similar effects happen on the *mg* benchmark, though *mg* on Cougar outperforms *mg* on Linux even on small numbers of nodes despite using older compilers and libraries. A variety of different overheads cause Linux's performance problems on larger-scale systems, including lack of contiguous memory layout and the associated TLB overheads and suboptimal node allocations due to limitations with Linux job-launch on ASCI Red.

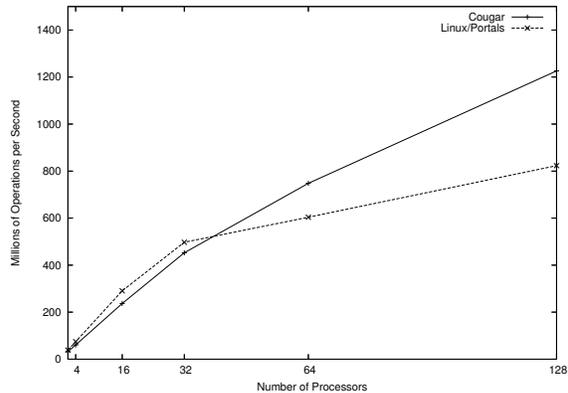


Figure 1: CG Performance on Linux and Cougar on ASCI/Red Hardware

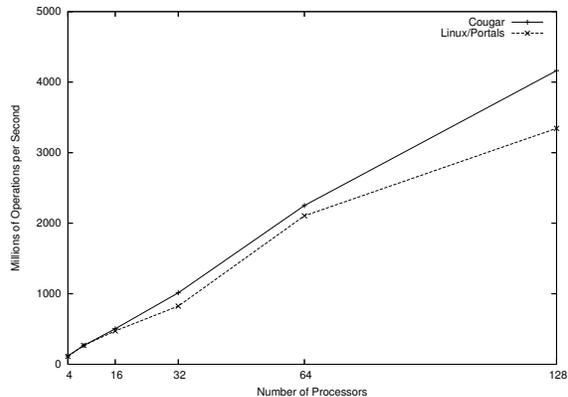


Figure 2: MG Performance on Linux and Cougar on ASCI/Red Hardware

Such operating system problems have also been seen in other systems. Researchers at Los Alamos, for example, have shown that excess services can cause dramatic performance degradations [15]. Similarly, researchers at Lawrence Livermore National Laboratory have shown that operating system scheduling problems can have a large impact on application performance in large machines [11].

2.3 Motivating Examples

The changing nature of demands on large scale systems present some of the largest challenges to operating system

design in this environment. We consider changing demands in several areas along with specific examples from each are to motivate our work.

2.3.0.1 Changing Usage Models.

As large-scale systems age, they frequently transition from specialized capability-oriented usage for a handful of applications to capacity usage for a wide range of applications. Operating systems for capability-oriented systems often provide a restricted usage model (dedicated or space-shared mode) and need to provide only minimal services, allowing more operating system optimizations. Operating systems for capacity-oriented systems, in contrast, generally support much more flexible usage models, such as timesharing, and must provide additional services including TCP/IP inter-networking and dynamic process creation.

2.3.0.2 Changing Application Demands.

Applications have varying demands for similar operating system services depending on their needs. Correctly customizing these services can have a large impact on application performance. As a concrete example, consider four different ways for a signal to be delivered to an application indicating the receipt of a network packet:

- Immediate delivery using interrupts (e.g. UNIX signals) for real-time applications
- Coalescing of multiple signals and waiting until some other activity (e.g., an explicit poll) causes an entry into the kernel, thereby minimizing signal handling overhead.
- Extending the kernel with application-specific handler code for performance-critical signals
- Forking a new process to handle each new signal/packet (e.g. `inetd` in UNIX)

2.3.0.3 Changing Hardware Architecture.

Operating system structure can present barriers to hardware innovation for ultrascale systems. Operating systems must be customized to present novel architectural features to applications and to make effective use of new hardware features themselves. Existing operating systems such as Linux assume that each machine is similar to a standard architecture, the Intel x86 architecture in the case of Linux, and in doing so limit their ability to expose innovative architectural features to the application or to use such features to optimize operating system performance. The inability of current operating systems to do so presents a significant impediment to hardware innovation.

Consider, for example, operating system support for parcel-based processor-in-memory (PIM) systems [20]. Operating systems for such architectures must be flexible enough to perform scheduling and resource allocation on these architectures and make effective use of this hardware for its own purposes. We specifically consider the use of a PIM as a dedicated OS file cache that makes its own prefetching, replacement, and I/O coalescing decisions. Processes that access files would send parcels to this PIM, which could immediately satisfy them from a local cache, coalesce small

writes together before sending the request on to the main I/O system, or aggressively prefetch data based on observed access patterns. Doing such work in a dedicated PIM built for handling latency-sensitive operations would free the system's heavyweight (e.g. vector) processors from having to perform the latency-oriented services common in operating systems.

2.3.0.4 Changing Environmental Services.

Finally, consider the variety of shared environmental services that operating systems must support, such as file systems and checkpointing functionality. New implementations of these services are continually being developed, and these implementations require changing operating system support. As just one example, the Lustre file system [2] is currently being developed to replace NFS in ultrascale systems. Lustre requires a specific message-passing layer from the operating system (i.e., Portals), in contrast to the general networking necessary to support NFS but in return provides much better performance and scalability. Similarly, checkpointing services require a means to determine operating system state and network quiescence. Finally, these services are often implemented at user-level in lightweight operating systems; in these cases, the operating system must provide a way to authenticate trusted shared services to applications and other system nodes.

3. CHALLENGES

There are four primary factors that influence the design of operating systems for ultrascale computing systems: application needs, system usage models, architectural models (both system-level and node-level architectures), and the design of shared services like file systems.

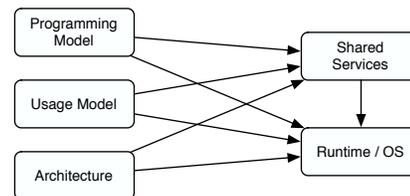


Figure 3: Factors Influencing the Design of Operating Systems

3.1 Application Needs

Applications present challenges at two levels. First, applications are developed in the context of a particular programming model. Programming models typically have a basic set of services that must exist. For example, in the explicit message passing model, it is necessary to allow for data to be moved efficiently between local memory and the network. Secondly, applications themselves may require extended functionality beyond the minimal set needed to support the programming model. An application developed using a component architecture may require system services to enable the use of dynamic libraries.

While lightweight operating systems have been shown to support the development of scalable applications, this approach places an undue burden on the application developer. Given any feature typically associated with modern

operating systems (e.g., UNIX sockets), there is at least one application that could benefit from having the feature readily available. In the lightweight operating system approach, the application developer is required to either implement the feature or do without. In fact, this is the reason that many of the terascale operating systems today are based on full-featured operating systems. The real challenge is to provide features needed by a majority of applications without adversely affecting the performance and scalability of other applications that do not use these features.

Advanced programming models strive to provide a high-level abstraction of the resources provided by the computing system. Describing computations in terms of abstract resources enhances portability and can reduce the amount of effort needed to develop an application. While high-level abstractions offer significant benefits, application developers frequently need to bypass the implementations of these abstractions for the small parts of the code that are time critical. For example, while the vast majority of the code in an application may be written in a high-level language (e.g., FORTRAN or C), it is not uncommon for application developers to write a core calculation, such as a BLAS routine, in assembly language to ensure an optimal implementation. The crucial point is that the abstractions implemented to support advanced programming methodologies must allow application developers to drop through the layers of abstraction as needed to ensure adequate performance. Because we are interested in supporting resource constrained applications, providing variable layers of abstraction is especially important.

Finally, because the development of new programming models is an ongoing activity, the operating and runtime system must be designed so that it is relatively easy to develop high-performance implementations of the features needed to support a variety of existing programming models as well as new models that may be developed.

3.2 System Usage Models

The system usage model defines the places where the principal computational resources can be shared by different users. Common usage models include: dedicated systems in which these resources are not shared; batch dedicated system in which the resources are not shared while the system is being used, but may be used by different users at different time; space-shared systems in which parts of the system (e.g., compute nodes) are not shared, but multiple users may be using different parts of the system at the same time; and time-shared systems in which the resources are being used by multiple users at the same time.

Sharing requires that the operating system take on the role of arbiter, ensuring that all parties are given the appropriate degree of access to the shared resources – in terms of time, space, and privilege. Here the challenge is to provide mechanisms that can support a wide variety of sharing policies, while ensuring that these mechanisms do not have any adverse impact on performance when they are not needed.

3.3 Architectures

Architectural models present challenges at two levels: the node level and the overall systems level. An individual com-

pute node may exhibit a wide variety of architectural features, including: multiple processors, support for PIM, multiple network interfaces, programmable network interfaces, access to local storage, etc. Variations in systems-level architectures may require different levels of operating system functionality on the compute nodes. As an example, limited access to the compute nodes may eliminate the need for some functionality related to protection.

3.4 Shared Services

Finally, the implementation of shared services, for example file systems, will depend on the three previously mentioned factors and will place requirements on the operating system. As an example, the operating system may need to maintain user credentials in a secure fashion while an application is running so that these credential can be trusted by the shared file system.

4. APPROACH

In the context of the factors discussed in the previous section, a “lightweight operating system” reflects a minimal set of services that meet the requirements presented by a small set of applications, a single usage model, a single architecture, and a single set of shared services. The Cougar operating system, for example, represents a lightweight with the following bindings: application needs are limited MPI and access to a shared file system, the system usage model is space sharing, the system architecture consists thousands of simple compute nodes connected by a high performance network, and the shared services include a parallel file system which relies on Cougar to protect user identification.

Our goal is to develop a framework for building operating and runtime systems that are tailored to the specific requirements presented by an application, the system usage model, the system architecture, and the shared services. Our approach is to build a collection of *micro-services* and tools that support the automatic construction of a lightweight operating system for a specific set of circumstances.

4.1 Micro-Services

At a minimum, each application will need micro-services for managing the primary resources: memory, processor, communication, and file system. We can imagine several implementations for each of these micro-services. One memory allocation service might perform simple contiguous allocation; another might map physical page frames to arbitrary locations in the logical address space of a process; another might provide demand page replacement; yet another may provide predictive page replacement. A processor management service may simply run a single process whenever a processor is available, and another might include thread scheduling.

There may be dependencies and incompatibilities within the micro-services. As an example, a communication micro-service that assumes that logically contiguous addresses are physically contiguous (thus reducing the size of a memory descriptor) would depend on a memory allocation service that provides this type of address mapping. There will also be dependencies between micro-services and system usage models. For example, a communication service that provides direct access to a network interface would not be compatible with a usage model that supports time sharing on a node.

In addition to micro-services that provide access to primary resources, there will be higher-level services layered on top of the basic micro-services. As an example, a micro-service might provide garbage collected dynamic allocation, another might provide first fit, explicit allocation and de-allocation (malloc and free) for dynamic memory allocation. Other examples include an RDMA service or a two-sided message service layered on top of a basic communication service.

Finally, we will need “glue” services: micro-services that enable combinations of other services. As an example, consider a usage model that supports general time-sharing among the applications on a node. Further, suppose that one of the applications to be run on a node requires a memory allocator that supports demand page replacement and another application requires a simple contiguous memory allocator. A memory compactor service would make it possible to run both applications on the same node.

4.2 Tools

In addition to micro-services, we will need to develop tools that analyze combinations of micro-services to determine an optimal set. Some of these tools will be needed to ensure that the required micro-services are available, others to ensure that applications are isolated from one another within the context of a usage model.

We cannot burden application programmers with all of the micro-services that provide the runtime environment for their applications. Application programmers should only be concerned with the highest-level services that they need (e.g., MPI) and the general goals for lower-level services. Here, we envision a tool that will take as input a set of the top-level services used by an application and produce a directed graph of the permissible lower-level services for the required runtime environment. Nodes of this graph will be weighted by the degree to which the micro-service represented by the node meets the goals of the application developer. We plan to base some of our work on tools for composing micro-services on existing tools, such as the Knit composition tool developed at the University of Utah in the context of the Flux project [17]. Other tools will be needed to select particular services in the context of a system usage model. These tools will also need to ensure that the services selected meet the sharing requirements of the system.

4.3 Signal Delivery Example

To illustrate how our micro-services approach can be used to address the challenges presented by ultrascale systems, we consider the signal delivery example presented toward the end of Section 2. Because signal delivery may not be needed by all applications, micro-services associated with signal delivery would be optional and, as such, would not have any performance impact on applications that did not need signal delivery.

For applications that do require signal delivery, we would need a collection of “signal detector” micro-services that are capable of observing the events of interest to the application (e.g., the reception of a message). These micro-services would most likely run as part of the operating system kernel. To ensure that they are run with sufficient frequency,

the signal detector micro-services may place requirements on the micro-service used to schedule the processor.

The signal detector micro-services would then be tied to one of several specialized “signal delivery” micro-services. The specific signal delivery micro-service will depend on the needs of the application. An immediate delivery service would modify the control block for the target process so that the signal handler for the process is run the next time the process is scheduled for execution. A coalescing signal delivery service would simply record the relevant information and make this information available to another micro-service that would respond to explicit polling operations in the application. A user defined signal delivery service could take a user defined action whenever an event is detected. Finally, a message delivery service could convert the signal information into data and pass this to the micro-service that is responsible for delivering messages to application processes. The runtime level could then include a micro-service that would read these messages and fork the appropriate process.

5. RELATED WORK

A number of other configurable operating systems have been designed including microkernel systems, library operating systems, extensible operating systems, and component-based operating systems. In addition, configurability has been designed into a variety of different runtime systems and system software subsystems, including middleware for distributed computing, network protocol stacks, and file systems.

5.1 Configurable Operating Systems

Most standard operating systems such as Linux include a limited amount of configuration that can be used to add or remove subsystems and device drivers from the kernel. However, this configurability does not generally extend to core operating system functions, such as the scheduler or virtual memory system. In addition, the configuration available in many subsystems such as the network stack and the file system is coarse-grained and limited; entire networking stacks and file systems can be added or removed, but these subsystems cannot generally be composed and configured at a much finer granularity. In Linux, for example, the entire TCP/IP or Bluetooth stack can be optionally included in the kernel, but more fine-grained information about exactly which protocols will be used cannot easily be used to customize system configuration.

Other operating systems have allowed more fine-grained configuration. Component-based operating systems such as Scout [13], the Flux OSKit [4], eCos [16], and TinyOS [8], on the other hand allow kernels to be built from from a set of composable modules. Scout, for example, is built from a set of *routers* that can be composed together into custom kernels. eCos and TinyOS provide similar functionality in the context of embedded systems and sensor networks, respectively. The Flux OSKit provided a foundation for component-based OS development based on code from the Linux and BSD kernels, focusing particularly on allowing device drivers from these systems to be used in developing new kernels. Unlike our proposal, however, none of these systems have concentrated on customizing system functionality at the fine granularity necessary to take full advantage of new hardware environ-

ments or optimize for the different usage models of ultrascale systems.

Microkernel and library operating systems such as L4 [6], Exo-kernels [3], and Pebble [5], for example, allow operating system semantics to be customized at compile-time, boot-time, or run-time by changing the server or library that provides a service, though this composability is even more coarse-grained than the systems described above. Such flexibility generally comes at a price, however; these operating systems may have to use more system calls and up-calls to implement a given service than a monolithic operating system, resulting in higher overheads. It also can result in a loss of cross-subsystem optimization opportunities. In contrast, our approach seeks to decompose functionality using more fine-grained structures and to preserve cross-subsystem optimization opportunities through tools designed explicitly for composing system functionality.

5.2 Configurable Runtimes and Subsystems

A variety of different systems have also been built that enable fine-grained configuration of system services, generally in the realm of protocol stacks and file systems. In contrast to our approach, none of these systems seek to use configuration pervasively across in an entire operating system.

Coarse-grained configuration on network protocols stacks has been explored in System V STREAMS [18], the *x*-kernel [10], and CORDS [21]. Composition in these systems is layer-based, with each component defining one protocol layer. Similar approaches have been used for building stackable file systems [7, 25].

More fine-grained composition of protocol semantics has been explored in the context of Cactus [9], [22], Ensemble [23], and Rwanda [14]. Cactus's event-based composition model, in particular, has influenced our approach to building; in fact, we are using portions of the Cactus event framework to implement our system. To date the Cactus project has focused primarily on using event-based composition in network protocols, not the more general operating system structures as described in this paper.

6. CONCLUSIONS

In this paper, we have presented an argument for a framework for customizing an operating system and runtime environment for parallel computing. Based on the results of preliminary experiments, we conclude that the demands of current and future ultrascale systems cannot be addressed by a general-purpose operating system if high-levels of performance and scalability are to be maintained and achieved. The current methods of using specialized lightweight approaches and generalized heavyweight approaches will not be sufficient given the challenges presented by current and future hardware platforms, programming models, usage models and application requirements. To address this problem, we presented a design for a framework that uses micro-services and supporting tools to construct an operating system and associated runtime environment for a specific set of requirements. This approach minimizes the overhead of unneeded features, allows for carefully tailored implementations of required features, and enables the construction

new operating and runtime systems to adapt to evolving demands and requirements.

7. REFERENCES

- [1] R. Brightwell, T. Hudson, R. Riesen, and A. B. Maccabe. The Portals 3.0 message passing interface. Technical report SAND99-2959, Sandia National Laboratories, December 1999.
- [2] Cluster File Systems, Inc. *Lustre: A Scalable, High-Performance File System*, November 2002. <http://www.lustre.org/docs/whitepaper.pdf>.
- [3] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, 1995.
- [4] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, Saint-Malo, France, 1997.
- [5] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 267–282, Monterey, CA, USA, 1999.
- [6] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ -kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.
- [7] J. Heidemann and G. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994.
- [8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.
- [9] M. A. Hiltunen, R. D. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, 1999.
- [10] N. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.
- [11] T. Jones, W. Tuel, L. Brenner, J. Fier, P. Caffrey, S. Dawson, R. Neely, R. Blackmore, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of SC'03*, 2003.
- [12] P. Miller. Parallel, distributed scripting with python. In *Third Linux Clusters Institute Conference*, October 2002.

- [13] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–168, 1996.
- [14] G. Parr and K. Curran. A paradigm shift in the distribution of multimedia. *Communications of the ACM*, 43(6):103–109, 2000.
- [15] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of SC'03*, 2003.
- [16] Redhat. eCos. <http://sources.redhat.com/ecos/>.
- [17] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for system software. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 347–660, 2000.
- [18] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, 1984.
- [19] Sandia National Laboratories. *ASCI Red*, 1996. <http://www.sandia.gov/ASCI/TFLOP>.
- [20] T. L. Sterling and H. P. Zima. The gilgamesh MIND processor-in-memory architecture for petaflops-scale computing. In *International Symposium on High Performance Computing (ISHPC 2002)*, volume 2327 of *Lecture Notes in Computer Science*, pages 1–5. Springer, 2002.
- [21] F. Travostino, E. M. III, and F. Reynolds. Paths: Programming with system resources in support of real-time distributed applications. In *Proceedings of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, 1996.
- [22] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr. A framework for protocol composition in Horus. In *Proceedings of the 14th ACM Principles of Distributed Computing Conference*, pages 80–89, 1995.
- [23] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. A. Karr. Building adaptive systems using Ensemble. *Software Practice and Experience*, 28(9):963–979, 1998.
- [24] S. R. Wheat, A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup. PUMA: An operating system for massively parallel systems. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 56–65. IEEE Computer Society Press, 1994.
- [25] E. Zadok and I. Badulescu. A stackable file system interface for Linux. In *Proceedings of the 5th Annual Linux Expo*, pages 141–151, Raleigh, North Carolina, 1999.