Recent Trends in Operating Systems and their Applicability to HPC[*][†]

Arthur Maccabe and Patrick 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
Scalable Computing Systems Department
Sandia National Laboratories
P.O. Box 5800; MS 1110
Albuquerque, NM 87185-1110
(rbbright, rolf)@cs.sandia.gov Email

Abstract

In this paper we consider recent trends in operating systems and discuss their applicability to high performance computing systems. In particular, we will consider the relationship between lightweight kernels, hypervisors, microkernels, modular kernels, and approaches to building systems with a single system image. We then describe how the Catamount lightweight kernel can be extended to support the Xen hypervisor API. This will, in turn, support use of Linux on the compute nodes of a large scale parallel system while minimizing the effort needed to support both, a lightweight OS and a full-featured OS.

1 Introduction

In 2000, Rob Pike (noted for his role in the development of the Plan 9 and Unix operating systems), presented a talk in which he proclaimed that "Systems Software Research is Irrelevant." [9] Much of his argument is based on his observations regarding operating systems aimed at desktop systems. The dominance of Microsoft and the need to support a large number of changing standards (e.g., TCP/IP, HTTP, HTML, XML, CORBA, POSIX, NFS, etc.) are two of the more significant reasons for the decline in interest in new operating systems.

One could also argue that research in infrastructure always tends to be marginalized (remains irrelevant) until it is demonstrated that the infrastructure is fundamentally broken or that other approaches might present substantial benefits. Recently, there have been several observations indicating that operating systems for very large scale systems are broken. In particular, full featured operating systems have been shown to limit application scalability [8, 6]. The alternative approach, the lightweight operating systems deployed on IBM's Blue Gene/L and Cray's XT3 do not provide many of the features that application developers have come to expect. To quote Rob Pike, these operating systems fail to "honor a huge list of large, and often changing, standards:..."

In the current environment, companies developing the largest systems face an unappealing dilemma: they can adopt a full-featured operating system at the risk of hindering application scalability, or they can adopt a lightweight operating system that does not support the broader application community. The possibility of supporting both a full-featured operating system and supporting a lightweight operating system is financially infeasible. In this paper, we ar-

[*]This work was supported in part by the US Department of Energy's Office of Science in the FAST-OS 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.

gue for an approach that we believe will enable support for both, a lightweight and a full-featured operating system, without significantly increasing budgets needed to support systems software.

The approach we advocate is to transform a lightweight operating system (like Catamount [7] for the XT3) into a virtualization layer that can support scalable applications directly, but can also support a full-featured operating system (like Linux). In the next three sections, we review the design approaches embedded in microkernels (Mach [10] and L4 [5]), hypervisors (VMware [11] and Xen), and lightweight operating systems (IBM's CNK and Cray's Catamount), respectively. In the following section, we examine the benefits that Linux provides in an effort to identify why Linux might provide a desirable environment for applications developed for high end computing systems. In the sixth section, we describe how Catamount could be extended to provide a virtualization layer capable of running Linux while retaining the properties that make this operating system viable for scalable applications.

2 Microkernels

The microkernel approach was popularized with the development of Mach in 1985. Figure 1 presents a graphical interpretation of the microkernel approach. The microkernel runs in privileged mode, everything else runs in user (non-privileged) mode. The microkernel provides the features needed to create and run processes and to communicate between processes. Services that would traditionally be part of a monolithic operating system, e.g., the file system, are implemented in server processes. Rather than making direct calls to operating systems services, applications send request messages to server processes.

The microkernel approach presents several benefits. From a security perspective, microkernels offer the potential for a much smaller privileged code base. From an implementation perspective there are fewer services that must be implemented in the microkernel, making it easier to optimize implementations of these services. Because microkernels only implement a minimal set of required services, users can develop implementations of services that are optimized to the needs of specific applications.

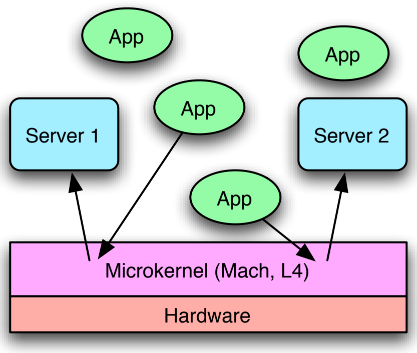


Figure 1: Microkernel Software Architecture

One of the most interesting developments was the observation that it was possible to run the full-featured operating system as a server on top of the microkernel with relatively simple modifications to the microkernel and the full-featured operating system. This was demonstrated by developing a Unix server that ran on top of the Mach microkernel [1]. This, in turn, enabled the execution of applications written (and compiled) for Unix without modification.

The first generation of microkernels, as exemplified by Mach, had a significant overhead for traditional system calls. More recent, second generation microkernels as exemplified by L4, do not incur the same overheads.

3 Hypervisors

While Mach showed that it was possible to run a full-featured operating system as a application level service, Mach was really designed to run applications and services that were written for Mach. Hypervisors, on the other hand, have been developed with the explicit goal of running full-featured operating systems as applications. Hypervisors do this by constructing a virtual machine for each operating system. Full virtualization involves full emulation for a specific set of hardware. This approach is exemplified by VMware and does not require any modification of the full-featured operating system, as long as that operating system is capable of running on the emulated hardware. In paravirtualization, as exemplified by Xen [2], the hypervisor provides an API for an idealized virtual machine. This machine is accessed using procedure calls rather than direct manipulation of device registers. The operating systems that are run on this type of hypervisor must be modified to use the API provided by the hypervisor.

Figure 2 presents a graphical interpretation for the Xen software architecture. In paravirtualization, the CPU is not usually emulated, applications and operating systems must be compiled for the processor used by the hypervisor. The MMU (Memory Management Unit) which controls memory access, can be emulated or accessed through procedure calls to update and install page tables. Access to I/O devices presents a particularly difficult problem. Xen provides a virtualized Ethernet device as part of each virtual machine. Access to other devices is provided through a special, privileged operating system running on Xen, called "Domain 0," which has direct access to the device registers for all of the devices in the underlying system. This strategy means that Xen is not complicated by the need to include device drives for a wide range of devices. This complexity can be relegated to a full-featured operating system, like Linux, that already has drivers for a wide range of devices.

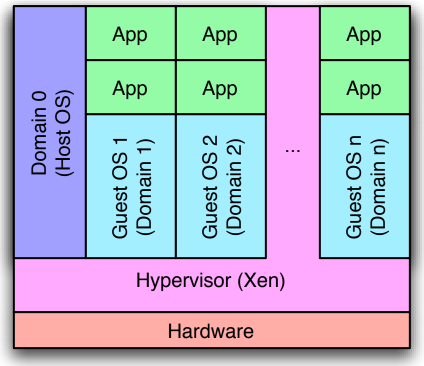


Figure 2: The Xen Software Architecture

4 Lightweight Kernels

Two of the largest systems currently available, IBM's Blue Gene/L and Cray's XT3 (Red Storm), use "lightweight" kernels on their compute nodes. These systems are based on a partitioned system architecture [4]. The nodes in the system are partitioned into groups to support different functional needs. Typical partitions include I/O nodes (which support access to storage devices), service nodes (which support user logins), and compute nodes (which are used to run computations). This partitioning allows for specialization, different types of node may have different types of hardware and may run different software stacks. For example, both the XT3 and Blue Gene/L run Linux on their I/O nodes but they run lightweight kernels on their compute nodes.

The lightweight kernel on the compute nodes provides a minimal environment needed to support computations. These kernels do not directly support I/O operations or many of the other services that applications will need. In the case of the Compute Node Kernel (CNK) used in IBM's Blu Gene/L, these services, when requested by an application, are forwarded to an I/O node. Figure 3 presents the software architecture for Blue Gene/L. The application runs on top of the CNK which runs on the hardware of the compute node. When the application makes a request, its execution is trapped by the CNK. The CNK handles some requests locally, while others are forwarded to an I/O node. This strategy is very similar to the microkernel approach to running a full-featured operating system in user space.

In the case of Catamount, the lightweight OS used in Cray's XT3, requests for extended services are handled by "wrapper" libraries that transform the request into a Remote Procedure Call (RPC) to the appropriate type of node. Figure 4 presents the software architecture of Catamount. As can be seen by this illustration, Catamount consists of three separate components: the wrapper library, the Process Control Thread (PCT) and the Quintessential Kernel (QK). The separation between the PCT and the QK reflects a separation between policy and mechanism. The QK provides the underlying mechanisms needed to support computation (for example, building and manag-

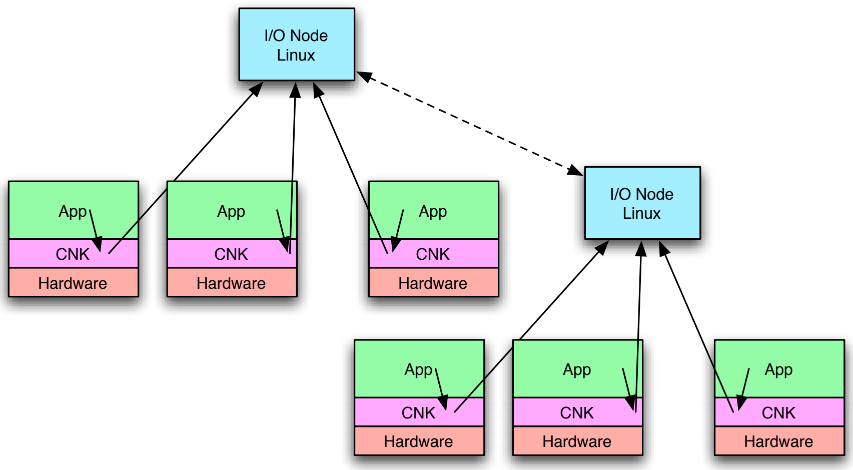


Figure 3: The Blue Gene/L Software Architecture

ing address spaces and interprocess communication). The PCT provides the policy needed to manage and support application processes (for example, process creation and scheduling and name services).

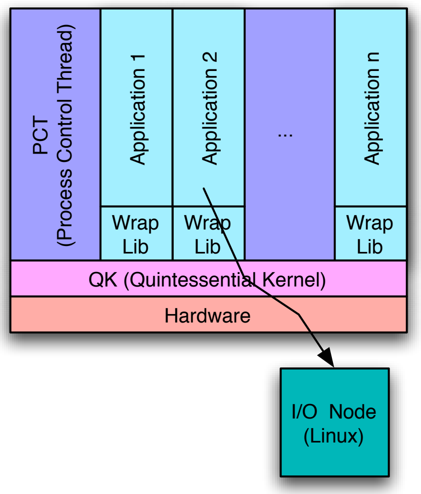


Figure 4: The Catamount Software Architecture

5 Linux

In recent years, Linux has emerged as the platform of choice for developing parallel scientific applications. There are several reasons for the popularity of Linux. From an application developer's perspective, Linux provides a wide range of familiar services. From the perspective of a library writer, Linux provides a target platform that is used by a large group of users. This means that library developers can reach a large percentage of their potential market when they build on Linux. This, in turn, grows the collection of services that are available in Linux. From system integrator's perspective, Linux provides a very large collection of device drivers, indeed 1.4 of the 2.6 million lines of code in the 2.4 kernel are device drivers. It is not too much of an exaggeration to say that Linux runs on just about any platform you might want to use. The availability of device driver for a wide range of devices has been a driving factor in the rapid emergence of Linux clusters. Figure 5 presents a graphical interpretation of the "ecosystem" surrounding Linux.

In network design, the structure shown in Figure 5 is referred to as an "hourglass design." Like an hourglass the structure narrows at the waist. In this case, the waist constricts to a single entity, Linux (in the case of modern networking, IP provides the single

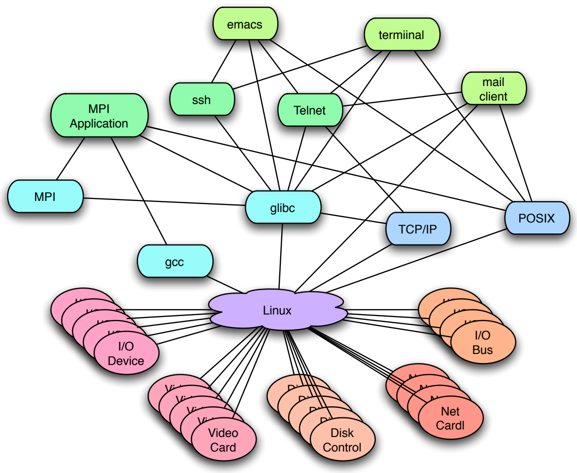


Figure 5: Linux Services

entity at the waist). The narrow waist defines a (relatively) small interface to be implemented on a variety of platforms. This interface, in turn, defines a common platform that can be used to implement additional functionality.

In the context of very large scale systems, the benefits associated with the vast collection of available device drivers are not as significant as they might be in other contexts. First, the compute nodes of these systems have very few devices; perhaps, an I/O bus, a serial port (for the monitoring system), and an interface to a high performance network. Second, a significant complexities associated with developing device drivers is the need to deal with hardware that does not completely follow the specification. Because they are typically based on specialized implementations using commodity components, if there is any complexity due to a mismatch between the hardware and its specification, the solutions needed are likely to be unique to the system being developed (and will not benefit to any significant extent from the solutions encoded in the Linux device drivers).

In contrast to the device drivers, applications **are** likely to benefit from the software ecosystem supported by Linux. These applications can take advantage of communication libraries, numerical libraries, compilers, and debugging tools that have been developed for Linux. In some cases, these applications may also benefit from the availability of commodity networking protocols that can be used to interface with remote services (e.g., visualization servers). While they may benefit from some of the Linux ecosystem, it is clear that these applications will not benefit from large parts of this ecosystem. For example, applications running on the compute nodes of a large system are unlikely to need a text editor, terminal emulator, or mail client. Figure 6 presents the software architecture of Linux as this architecture is likely to be used on the compute nodes of a very large system.

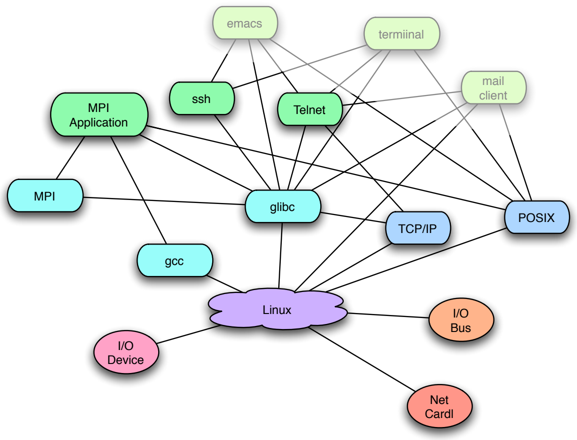


Figure 6: Linux HPC Services

6 Linux on Catamount

Given the richness of the Linux software ecosystem, running Linux on the compute nodes of a large scale system has clear benefits. However, other researchers have observed that full-featured operating systems an have a negative impact on application scalability [8, 6]. This conflict might be resolved by supporting both Linux and a lightweight operating system on the compute nodes of the very large system. This strategy introduces two problems. First, it may be difficult to enable Linux and the lightweight kernel to co-exist simultaneously on a single system. Second, the costs associated with supporting two operating systems on the same platform may make this approach infeasible.

Lightweight operating systems frequently make simplifying assumptions to minimize work that must be done during program execution. Catamount in particular, makes several important assumptions regarding the authenticity and trustworthiness of messages [12] to improve the performance of message passing and enhance application scalability. These assumptions will not be valid if Linux is allowed to coexist on a single system. As a minimum, this would require modification to the underlying communication mechanisms of Linux.

The second problem, the economic feasibility of supporting two operating systems, is likely to be the more significant. Supporting an operating system for the compute nodes of a large scale system is not a trivial task. As new applications are ported to the system, these applications are likely to test the systems software in new ways frequently exposing software issues that were previously unknown. The manpower needed to diagnose and fix these problems is significant. Unless the two operating systems share a common low-level code base, there will not be any carry over from one system to the other, requiring independent sets of resources to support each system.

To avoid these problems, we describe how Catamount, the lightweight operating system for Cary's XT3 system, could be transformed into a paravirtualization layer capable of supporting both the direct execution of"scalable applications" and full-featured operating systems like Linux. Figure 7 illustrates the basic strategy for running Linux on Catamount. In principle, the QK provides the hardware virtualization functionality and the PCT provides the Domain 0 functionality.

To more fully explore this approach, we need to consider the mechanisms used in Catamount. Figure 8 illustrates these mechanisms. The QK deals with hardware interrupts (timer interrupts and network interrupts) and exports two APIs: one for the PCT and another for applications. The application-QK API includes message passing (through Portals [3]), quit quantum (to wait for an event), and handlers for illegal instructions and illegal addresses

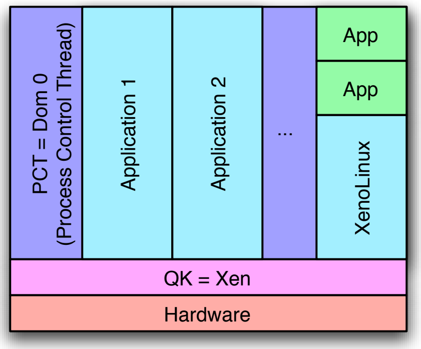


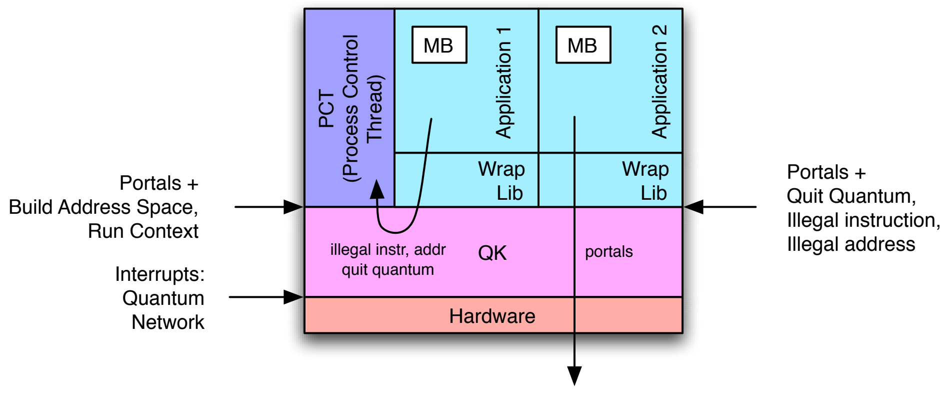
Figure 7: Linux on Catamount

(both of these reflect the exceptions to the PCT). The PCT-QK API includes message passing (so that PCTs can communicate with one another and with service nodes), functions to build address spaces and a function to run a context (an address space plus program counter).

Communication between the PCT and an application is accomplished through a mailbox (MB) which resides at a known address in the application. Applications place a request in the mailbox and invoke the quit_quantum QK call. When it is run, the PCT examines the mailbox for most recently run application to see if the application has a PCT request.

Porting Linux to a new (virtual) hardware environment typically involves the introduction of a new target architecture. Supporting a new architecture requires a great deal of effort. Rather than creating a new Linux target architecture, we propose to use XenoLinux [2] a port of Linux that runs on Xen, leaving the burden for supporting this architecture on the Xen/Linux community. This only requires that we support the Xen hypercalls used in XenoLinux. Table 1 summarizes these calls.

For the most part, these calls are easily implemented by extending the API provided by the PCT, with minimal modifications to the QK and no modifications to XenoLinux. Figure 9 presents a more realistic strategy for running Linux on Catamount. A "PCT Wedge" library declares the required mailbox and translates the Xen hypercalls into PCT re-



PCT (Process Control Thread)
MB
Application 1
MB
Application 2
Wrap Lib
Wrap Lib
Portals + Build Address Space, Run Context
Interrupts: Quantum Network
Portals + Quit Quantum, Illegal instruction, Illegal address
illegal instr, addr quit quantum
QK
portals
Hardware

Figure 8: Mechanisms used in Catamount

Hypercall	**Use**
set_callbacks	establish normal and "failsafe" event handlers
sched_op_new	one of yield, block, shutdown, or poll
mmu_update	update page table entries (can also use direct modification)
stack_switch	change the stack
fpu_taskswitch	next attempt to use floating point causes trap
memory_op	increase or decrease current memory allocation
event_channel_op	inter-domain event-channel management
physdev_op	BIOS replacement

Table 1: Hypercalls on Xen

quests. While the PCT can easily provide the needed functionality, it does not currently provide this functionality for Catamount applications. In particular, while the PCT can create address maps and run contexts, it does not make this functionality available to applications. To do so, will require some extensions to the PCT, indicated by the "Xen Wedge." The QK should not require any modifications. Address faults and exceptions encountered by XenoLinux (for example, during an attempt to modify an MMU register) would be forwarded to the PCT, where they can be handled in the appropriate manner.

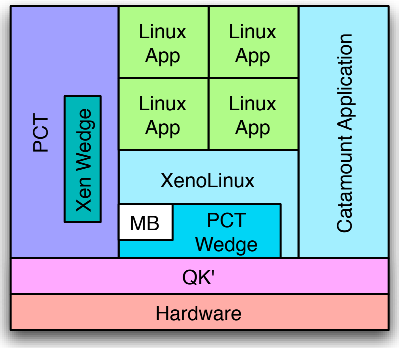


Figure 9: A More Realistic Approach to Running Linux on Catamount

7 Conclusion

In this paper, we have advocated using Catamount as a virtualization layer to support the use of full-featured operating systems, specifically Linux, on large scale systems. This can be accomplished with minimal changes to the QK and extensions to the PCT. Using this virtualization layer, application developers who choose to can take advantage of the Linux software ecosystem. Because there are minimal changes to the QK, application developers who do not need this functionality should not experience any limitations in application scalability.

All software evolves over time and Linux has had a particularly rapid rate of change over its relatively short history. By emulating the Xen hypercall API, our approach minimizes the need to track changes for future releases of Linux. In particular, application developers should be able to use the XenoLinux kernel, or any OS that has been ported to the Xen API, without modification. Only changes to the Xen API need to be tracked and incorporated into the Catamount software base.

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–112, 1986.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of teh 19th Sympoium on Operating Systems (SOSP-19)*, October 2003.

[3] 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.

[4] D. S. Greenberg, R. Brightwell, L. A. Fisk, A. B. Maccabe, and R. Riesen. A system software architecture for high-end computing. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. ACM Press and IEEE Computer Society Press.

[5] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-Kernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 66–77, New York, Oct. 5–8 1997. ACM Press.

[6] 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.

[7] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, May 2005.

[8] 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.

[9] R. Pike. Systems software research is irrelevant. Slides of a talk. Available from the author's Homepage: `http://cm.bell-labs.com/cm/cs/who/rob/`, Feb. 2000.

[10] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Orr, and R. Sanzi. Mach: a foundation for open systems (operating systems). In IEEE, editor, *Workstation operating systems: proceedings of the Second Workshop on Workstation Operating Systems (WWOS-II), September 27–29, 1989, Pacific Grove, CA*, pages 109–113, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1989. IEEE Computer Society Press.

[11] Virtualization: Architectural considerations and other evaluation criteria. `http://www.vmware.com/pdf/virtualization_considerations.pdf`, 2005.

[12] S. R. Wheat, A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup. PUMA: An operating system for massively parallel systems. *Scientific Programming*, 3:275–288, 1994.