SANDIA REPORT

SAND2009-2660
Unlimited Release
Printed April 2009

An Extensible Operating System Design for Large-Scale Parallel Machines

Rolf Riesen, Kurt Ferreira

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-2660
Unlimited Release
Printed April 2009

An Extensible Operating System Design for Large-Scale Parallel Machines

Rolf Riesen, Kurt Ferreira
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
{rolf,kbferre}@sandia.gov

Abstract

Running untrusted user-level code inside an operating system kernel has been studied in the 1990's but has not really caught on. We believe the time has come to resurrect kernel extensions for operating systems that run on highly-parallel clusters and supercomputers. The reason is that the usage model for these machines differs significantly from a desktop machine or a server. In addition, vendors are starting to add features, such as floating-point accelerators, multicore processors, and reconfigurable compute elements. An operating system for such machines must be adaptable to the requirements of specific applications and provide abstractions to access next-generation hardware features, without sacrificing performance or scalability.

Contents

Introduction

Large-scale, high-performance clusters and supercomputers used for scientific parallel computing require specialized operating systems (Brightwell, Maccabe, and Riesen 2003). Usually, these machines run a single, parallel application that is spread across many or all the nodes of a system. Each process that is part of that application is assigned to a CPU (core) and "owns" it for the duration of the run. That means that during that run, no other processes that are not part of that application will be assigned to these CPUs. Multiple applications *space-share* a parallel machine.

OS-noise has been identified as a major culprit that inhibits scalability (Petrini, Kerbyson, and Pakin 2003; Ferreira, Brightwell, and Bridges 2008). Parallel applications exchange messages and often need to wait for the data before they can proceed. If one of the processes is delayed because the local operating system is busy running other processes or doing housekeeping tasks, it will delay the entire application. As parallelism increases, the likelihood that any one operating system instance is not currently running the parallel application process, increases as well. That means the parallel application is slowed down as it is run on more nodes and resources are wasted as hundreds or thousands of nodes wait for the straggler.

All the resources of a node: CPUs, network interfaces, and memory, are allocated to processes of the same application. While memory protection between processes is still useful for debugging, it is no longer strictly necessary. In fact, it might be performance beneficial to let processes running on multiple cores freely share the memory of a single node (Brightwell, Pedretti, and Hudson 2008). Policies for process control should also be determined by the application itself. Therefore, within a node, less protection is needed than what typical operating systems provide.

However, some protection mechanism still need to be enforced by the operating system. For example, the application should have full control over the network interface (OS bypass) so it can be managed as efficiently as possible, but trusted header information, such as the source node ID and process ID of a message, must be under operating system control. In other words, the operating system should let an application manage the nodes that have been allocated to it, while still protecting the resources of the machine that belong to other applications.

Many clusters, especially the larger ones, and most supercomputers employ a parallel file system whose storage devices are external to the machine or attached to dedicated I/O nodes. Compute nodes do not have local disks. Most other peripherals that are supported by desktop operating systems are missing as well. In fact, the main peripheral accessible to the application is a high-speed network interface sometimes directly attached to the memory bus. All I/O operations, remote memory access, and explicit message passing are handled by that one device.

This architecture limits the number of devices an operating system must support. Fur-

thermore, many modern network interfaces are intelligent and interact with the application directly. Copying data through the operating system would have disastrous effects on network performance.

Because of these characteristics, parallel applications running at large scale have very specific demands of an operating system. In addition, the hardware to build clusters and supercomputers is changing and requires adaptation from the application and the operating system. The operating system is expected to match an application to the hardware it runs on as efficiently as possible.

Some of the new hardware features that require applications and operating systems to adapt are here already. One example is the use of multicore processors. Other features are not in production use yet, but are being discussed as potential performance booster for next-generation systems. Examples include attaching graphic processing units (GPUs) or other specialized processors, such as IBM's cell architecture, to general purpose CPUs to accelerate floating-point intensive calculations. More exotic devices, for example Field Programmable Gate Arrays (FPGAs) that can be reconfigured on the fly for a specific application need, or Processor in Memory (PIM) devices that could help alleviate memory bus throughput demands, are on the horizon.

An operating system must provide abstractions that unify several of these technologies and make them accessible to portable applications. Such applications cannot be re-written for every possible new feature a vendor offers. Rather, it is the operating system's role to manage the new resource, and make them available to the application.

A look at the list at `www.top500.org` reveals that the number of processors built into the five-hundred fastest systems in the world, is increasing every year. With a higher component count the likelihood of a hardware failure increases. This reduces the Meantime Between Failures (MTBF) for large scale applications.

A similar argument can be made for the software side. The likelihood that a subtle timing bug is triggered increases with the number of code instances running. A small operating system kernel is easier to debug, test, and reason about than a multi-million line, full-featured operating system. Furthermore, it is easier to restart or migrate a small operating system in case of a hardware failure or an early warning.

An earlier HotOS paper (Hunt, Larus, Tarditi, and Wobber 2005) listed the challenges for next-generation systems as: dependability, security, system configuration, extension, and multi-processor programming. While that paper was written in the context of desktop operating systems, many of the points the authors make also apply to operating systems for high-end parallel computing. In particular, system configuration and multi-processor programming are areas that need to be addressed. We think that kernel extensions allow us to write small, simple, and scalable operating systems with the flexibility to adapt to the demands of future architectures and applications.

For these reasons we believe the time has come to revisit extensible operating systems and apply some of the techniques and lessons learned in the 1990's to high-end, parallel

computing. We envision a very small kernel that provides base services and can be extended by the runtime system of the machine or by the application itself. Some of these extensions adapt the kernel to a given machine and are probably inserted during boot time by a trusted entity. Less trusted extensions can be inserted by the applications. These are only needed while the application is running and are meant to provide a better impedance match between the application and the underlying operating system and hardware.

We will explain our design ideas in the next section and then discuss why we think these ideas are beneficial to high-end parallel computing platforms. We will look at related work and provide a summary at the end of the paper.

Nimble

Work on a kernel called Kitten that builds on our experiences with lightweight kernels (Wheat, Maccabe, Riesen, van Dresser, and Stallcup 1994a; Maccabe, Bridges, Brightwell, Riesen, and Hudson 2004; Riesen, Brightwell, Bridges, Hudson, Maccabe, Widener, and Ferreira 2009) for massively parallel machines is currently under way. The goal of the Kitten project is to efficiently use the multicore resources that have begun to appear in modern parallel machines. In this section we describe Nimble, an extension infrastructure for a lightweight kernel such as Kitten.

Extensibility

Core operating system tasks, such as initializing devices, and simple process and memory management, will be provided by the lightweight kernel. In contrast to a full-features desktop or server operating system, lightweight kernels provide only rudimentary services. There is a single, or at most a couple, of processes running on each CPU core and there is no support for demand paging, kernel-level threads, TCP/IP, dynamic libraries and many other amenities that desktop users expect but limit performance and scalability.

Nevertheless, some applications may be willing to make performance and scalability sacrifices for a given feature. At other times in may not be practical to rewrite as service or a library on which an application depends. Additionally, providing just one more feature may not compromise performance or scalability, while making all of them available will cripple the operating system and the machine it runs on.

Therefore, we propose to use kernel extensions to customize the operating system to the specific hardware it is running on and adapt it to the currently running application. There are two types of extensions. *Trusted* kernel extensions have direct access to the hardware and are used by the system administrator and the runtime system to adapt the kernel to the hardware. This is typically done at boot time and is akin to loadable kernel modules that Linux provides.

Application-inserted kernel extensions are generally not trusted and do not have direct access to hardware or other privileged resources. However, remember that most of the node resources have been allocated to the application already. The kernel must enforce access policies to resources outside the node, but most of the node resources are managed by the application itself. This is typically done through a library that inserts a kernel extension on behalf of the application.

For example, an application-inserted kernel extension can augment the basic process scheduler present in the lightweight kernel to allow for active messages to force a context switch to a user-level handler as soon as they arrive. An application that consists of a single process per CPU core could run with an infinite time quantum.

An application could use kernel extensions to dedicate a CPU core to handle message-passing traffic from the network interface and run compute-intensive processes on the other cores. An application that is not communication intensive may use an interrupt-driven kernel extension to handle network interface requests, and use all CPU cores for computation.

Latency-sensitive operations, such as remote memory accesses or collective message-passing operations, could benefit from a kernel extension that handles some network requests in the kernel on behalf of the application instead of incurring a full context switch to run a user-level handler.

Instead of the kernel providing a slew of mechanisms, it provides only basic services and the ability to insert extensions that provide new mechanisms and set policy on behalf of the application.

Implementation

In the 1990's several methods to extend kernels were investigated. One that has not had very much attention is interpretation. An interpreter is relatively easy to write and it is easy to shield other parts of the kernel from code that is interpreted. For code from a trusted source, Nimble will disable access and privilege checks in the interpreter. This will yield a small performance advantage, but more importantly, it will allow trusted extensions to access and manipulate protected resources.

Another reason we are considering an interpreter is that we expect most extensions to be small, simple, and to only run for brief moments. For example, a process scheduler that needs to pick the next process to run from a pool of less than a handful, does not require a lot of instructions. The code to do the actual context switch is written in C, already resides in the lightweight kernel, and can be called by the interpreter. Code to initiate a data transfer through the network interface already exists as well. A kernel extension makes a single subroutine call to start the data transfer.

Techniques for fast interpretation have been studied for a long time and are still under investigation for today's byte code interpreters. One such technique, called threaded code (Bell 1973), is one we intend to pursue.

During code insertion the extension is threaded. Each instruction in the extension is converted to a subroutine call into the interpreter. This makes interpretation a two-step process. First, the instructions are examined and translated. In the second step, during the execution of an extension, execution proceeds along a thread that consists of various subroutines the interpreter provides.

When Nimble starts executing an extension, for each original statement in the extension, control flow will be redirected into the appropriate subroutine inside the interpreter. The subroutines contain the code that encapsulates the semantics of a given statement in the kernel extension.

The subroutines are written in C and compiled into Nimble. We intend to have two versions of most subroutines. One that performs access checks and another which does not.

Threaded code reduces the interpretation overhead to one or two assembly instruction per interpreted statement in the kernel extension. While many statements will be simple, such as loading a value, many are more complex and the overhead of interpretation will be negligible.

There are variations on threaded code. The method we described above is called subroutine threading. During the translation step the statements to be interpreted are translated into a series of CPU "call subroutine" instructions. It can be argued that this technique is not interpretation, since the call instructions are simply executed after the original statements have been translated. Depending on the CPU architecture, direct threaded coded may be faster. Direct threaded code is just a compact list of addresses. The interpreter reads these addresses and calls the subroutine where these addresses point to. Either of these techniques are faster than interpreting byte code (Ertl 1993; Klint 1979; Dewar 1975).

Nimble will add mechanisms to the lightweight kernel to insert and remove extensions, to call them for specific events, such as interrupts or application traps into the kernel, allow one extension to call another, and to limit the running time of an extension.

Discussion

Operating system kernel extensions have been extensively studied in the 1990's, but have not really caught on in main-stream desktop operating systems. We believe that one reason for that is that much of the extensibility was aimed at improving operating system performance by avoiding unnecessary kernel-to-user-level-transitions by executing user code in the kernel.

There are other techniques for that and machines have gotten fast enough for many tasks that were considered to be in need of improvement in the 1990's. We believe that kernel extensions have a place in high-end parallel computing for several reasons. One is that speed is still of primary concern here and that inefficiencies in the operating system can severely limit the scalability of a parallel machine. The other reason is that next-generation supercomputers will employ technologies – starting with multicore processors to attached floating-point engines – that will be difficult to exploit in a general purpose way.

Each application and programming model has its own specific needs. If a hardware resource and an access policy can be customized for a specific application, performance and scalability benefits will follow. In the type of machines we are considering, this is possible because they are space shared and whole sets of nodes are allocated to an application. Letting the application manage these resources is more efficient than providing general purpose mechanisms and policies. We need an operating system that can be used to manage the machine as a whole (allow for node allocation for example), but gets out of the way when an application wants to make use of the resources allocated to it.

Therefore, we want to give each application the opportunity to set its own resource allocation policy and add the specific features it needs in an operating system, while not being burdened by services it does not need and that could limit performance or scalability. Allowing applications to insert user-level code into the kernel seems an ideal way to achieve this flexibility.

Allowing user-level code to execute inside a lightweight kernel makes sense because the usage model and the requirements for high-performance parallel computers are quite different from the needs of a desktop user or even a server farm. Some aspects of embedded computing apply as well, but these systems, once deployed, are more static in nature than the application mix run on a parallel computer.

Related work

The idea of executing user code, an extension, inside the kernel has seen several incarnations. Sand-boxing, also called software-based fault isolation (Wahbe, Lucco, Anderson, and Graham 1993), is the idea of limiting data accesses to a certain segment of main memory. This is done by inserting code before potentially unsafe instructions that sets (or checks) the upper n bits of the address to a value that corresponds to a segment which the code is allowed to access. For our approach this may be too limiting, since we do want to make some memory mapped devices accessible to extensions, while preventing access to memory-mapped registers that must be protected.

The SPIN project (Bershad, Chambers, Eggers, Maeda, McNamee, Pardyak, Savage, and Sirer 1994) used a trusted compiler to generate spindles that get inserted into the kernel. The spindles are digitally signed to ensure that they were generated by the trusted compiler. The runtime system for the chosen language and the cryptographic tools to verify the signatures would need to be available on each node. Depending on the source language chosen, this may be a significant amount of code that is statically linked with the parallel application.

Interpreters have been studied extensively and some of them have been embedded in kernels before. The BSD packet filter is one example (Mogul, Rashid, and Accetta 1987). Another is our work of adding a FORTH interpreter to the firmware of a Myrinet network interface (Wagner, Jin, Panda, and Riesen 2004). We used that to improve the performance of collective MPI operations such as broadcast.

Interpreters are often considered to be too slow for system services. However, our extensions are small and perform simple tasks. It should be possible to gather the most often used constructs and sequences into a virtual machine which can be optimized to execute efficiently (Pittman 1987; Proebsting 1995). Then, using indirect threaded code or direct threaded code techniques, build an extremely fast interpreter (Bell 1973; Dewar 1975; Kogge 1982; Ertl 1993).

Several operating systems have made use of extensions. We already mentioned SPIN. Global Layer Unix (GLUnix) (Vahdat, Ghormley, and Anderson 1994) used software-based fault isolation to move OS functionality into user level libraries. We want to move user code functionality into the kernel. The VINO kernel was designed to let applications specify the policies the kernel uses to manage resources. That is what we are interested in, but specifically for high-performance parallel environments, instead of database management systems for which VINO was designed.

In the μChoices operating system (Campbell and Tan 1995; Tan, Raila, and Campbell 1995) agents can be inserted into the kernel. These agents are written in a simple, flexible scripting language similar to TCL, and are interpreted. Agents batch a series of system calls into a single procedure that requires only one trap into the kernel to be executed. Agents use existing kernel services and do not extend the functionality of the kernel or provide services that are not available at user level. Agents are a simple optimizations to eliminate

the overhead of several system calls. We also need to mention the MIT Exo kernel (Engler, Kaashoek, and O'Toole 1995; Engler and Kaashoek 1995). It attempts to lower the OS interface to the hardware level, eliminating all abstractions that traditional operating systems provide, and concentrates on multiplexing the available physical resources. This is similar to our lightweight kernels which provide only very basic services and rely on user-level libraries to implement other services. Nimble is meant to extend this concept and push some of that functionality back into the kernel when it is needed at run time.

Methods to safely execute untrusted code in a privileged environment are compared in (Small and Seltzer 1996).

Conclusions and future work

Lightweight kernels have proven successful in the past on Intel's Paragon and ASCI Red at Sandia National Laboratories, on Cray's XT-3 Red Storm, and on IBM's Blugene/L series of machines. These lightweight kernels are small and scalable, allow applications to get most of the available memory (without demand-paging) and run on tens of thousands of nodes in parallel.

As more applications are being ported to these kinds of machines, the demand for additional features and services increases. A modern operating system must provide some of these features without compromising scalability or efficiency.

We are working on a new lightweight kernel called Kitten that carries our experiences with large-scale parallel machines forward to machines with potentially hundreds of cores per node. We have started the design of Nimble which will be integrated into Kitten. Nimble will provide the infrastructure to let applications extend Kitten's functionality. These extensions are meant to provide additional services and provide access to next-generation hardware features.

Epilogue

We submitted this paper to HotOS XII, 2009. Unfortunately, it got rejected. The reviewers had some interesting points which we would like to address here and should have had in the original paper.

Operating systems for supercomputers are not of great interest to the main-stream OS research community. When we mention high-performance parallel computing, Google, MapReduce (Dean and Ghemawat 2004; Dean and Ghemawat 2008), and Hadoop (Apache Hadoop Project Members 2009) come to the minds of our reviewers. That is unfortunate, but not something we can easily change. Google's use of Linux and commodity hardware is a form of distributed, rather than parallel, computing (Riesen, Brightwell, and Maccabe 1998), but the general OS community lumps distributed and parallel computing together.

In distributed computing, latencies are high and computations are much less coupled (dependent on each other) than in parallel computing. That means that some issues plaguing tightly coupled, highly-parallel machines, are no problem at all for loosely coupled clusters. OS noise (Petrini, Kerbyson, and Pakin 2003; Ferreira, Brightwell, and Bridges 2008), for example, is only an issue in high-end parallel computing. Extremely low-latency message passing, and highly efficient collective communications are also much more important in parallel computing than distributed computing. After all, if a system has to deal with disk access latencies, delays caused by wide-area networking stacks, and the latencies of messages crossing a room or even farther, then issues in the sub-microsecond range do not come into play.

Therefore, a general-purpose operating system like Linux is very suitable for distributed computing, but it may not be for parallel computing. The lack of knowledge and interest in systems software for massively parallel computing is not something we can change. However, the reviewers had more concrete comments and questions, that we should address. We do that in the sections below.

Are extensible systems leading us astray?

Two reviews mentioned (Druschel, Pai, and Zwaenepoel 1997) in which Druschel, Pai, and Zwaenepoel argue that extensible systems are not necessary and lead the OS research community away from more important research topics. They say that making extensions safe is not necessary. During development and research it is not necessary, because these extensions are not used by ordinary users on production systems. Later, these extensions can be built directly into the kernel and be treated like any other kernel modification.

The effort of taking a research extension and incorporating it into a production kernel is error prone and not trivial. A small research group may have the resources to develop the extension, but not the will and time to build it into a production kernel. If such an extension can be run safely inside a production kernel, it is much more likely to be used by end users.

Also, we envision that some extensions will be written by application developers. These application specific extensions may run in the kernel or the NIC and provide specialized services that are only useful to the currently running application. System software on the nodes allocated to this application, and the rest of the system, must be protected from malice and bugs of these application-specific extensions.

Finally, one reason that general-purpose operating systems are not suitable for high-end parallel computing is that they are riddled with services and code that is not (currently) needed and interferes with performance and scalability of these systems. Kernel extensions allow us to keep bloat under control by only loading those extensions that are required for a given application.

Do extensions add to OS noise?

One reviewer wondered whether it is possible that extensions add noise rather than reduce it. Each extension and the infrastructure needed to execute it, add overhead. Whether this overhead adds to noise is not clear. Usually noise is associated with asynchronous execution of code not directly related to the operation on hand. Extensions are meant to make responding to events, such as receiving a message, faster. We do not believe that the extensions we envision in this report would add to OS noise. However, it is certainly something that should be investigated.

Exokernel and Corey

An alternative approach to extensions, suggested by the reviewers, is to use an Exokernel (Engler, Kaashoek, and O'Toole 1995) or Corey (Boyd-Wickizer, Chen, Chen, Mao, Kaashoek, Morris, Pesterev, Stein, Wu, Dai, Zhang, and Zhang 2008) in a multicore system. The Exokernel does to provide the mechanisms we need to tailor an OS to the application needs in a massively parallel system. For things that must be done inside a kernel, the Exokernel even allows for downloadable code; i.e., kernel extensions! In many ways, application-specific handlers in the Exokernel are exactly the kernel extensions we propose in this paper. We go a step further and would allow OS developers to download extensions that are not subject to the protection mechanisms required for application-provided extensions.

Besides the ability to download unprotected extensions, the main difference between our approach and the Exokernel is the realization that in a massively parallel system all node resources are turned over to a single application. The Exokernel still assumes there are multiple processes sharing the resources of a node. While that is true for a few HPC applications, most of them simply need access to the floating-point units and the ability to communicate with other nodes.

As a matter of fact, in (Kaashoek, Engler, Ganger, Briceño, Hunt, Mazières, Pinckney, Grimm, Jannotti, and Mackenzie 1997) the Exokernel designers make many of the points

for kernel extensions we make. Given the specialized environment and usage of a massively parallel system, we need even fewer of the mechanisms an Exokernel provides. Going with the philosophy of removing as much code and mechanism from the kernel as possible, we are proposing a kernel that is even more lightweight than the Exokernel.

Some of the things Corey does, in particular when and when not to share data among cores, and what work to assign to a specific core, are some of the things our extensions are supposed to do. However, we believe that these decisions are better made at the application level, but they do need support from the OS to be implemented; i.e., policy on how to use local resources that have been assigned to a job should be made by the application. The mechanism to implement that policy needs to reside in the kernel to maintain safety and integrity of the system. Only kernel extensions provide the flexibility to deal with new architectural features quickly and customized to an application's requirements.

Could an Exokernel be used to implement the features Corey provides? If an Exokernel is flexible and fast enough to do that, then choosing an Exokernel over Corey is preferable, since Corey does not provide all the features we want for kernel extensions. Of course, if an Exokernel is unsuitable to replace Corey, then it is equally unsuitable to replace kernel extensions.

Kernel modules

Why can we not simply use kernel modules in existing systems such as Linux or Chorus? The reason is that kernel modules in Linux, in particular, are meant to add device drivers to Linux, not completely change the way the kernel handles common resources such as processor state and time slices, and memory. It is the same reason that drove the design of the Exokernel: once too much mechanism and policy is embedded in a kernel, it cannot be removed when it is not needed or gets into the way.

One question we received is how can extensions be kept small, so they still execute quickly under interpretation, when the kernel provides so few services? The majority of high-performance parallel applications need only a core set of services: floating-point operations and the ability to communicate with other nodes. Many of the additional services needed are either slim, handling specific messages in the kernel for example, or infrequently invoked, loading a dynamic library for example. Furthermore, interpretation is not the only solution we intend to pursue. For some extensions it may make more sense to build them into the kernel and simply load that version of the kernel before the application that needs this extension is started.

References

Apache Hadoop Project Members (2009, April). **Welcome to Apache Hadoop!** `http://hadoop.apache.org`.

Bell, J. R. (1973, June). **Threaded Code**. *Communications of the ACM 16*(6), 370–372.

Bershad, B. N., C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer (1994, February). **SPIN - An Extensible Microkernel for Application-specific Operating System Services**. Technical Report CSE-94-03-03, University of Washington.

Boyd-Wickizer, S., H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang (2008, December). **Corey: An Operating System for Many Cores**. In *OSDI'08: Proceedings of the 8th conference on Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA. USENIX Association.

Brightwell, R., T. Hudson, R. Riesen, and A. B. Maccabe (1999, December). **The Portals 3.0 Message Passing Interface**. Technical report SAND99-2959, Sandia National Laboratories.

> NOTE: There are newer version of this document available, but none have been published as tech reports yet. The techreport describes the motivation and requirements for Portals 3.0 and gives examples on how to use them.

Brightwell, R., A. B. Maccabe, and R. Riesen (2003, April). **On the Appropriateness of Commodity Operating Systems for Large-Scale, Balanced Computing Systems**. In *International Parallel and Distributed Processing Symposium (IPDPS '03)*.

Brightwell, R., K. Pedretti, and T. Hudson (2008). **SMARTMAP: operating system support for efficient data sharing among processes on a multi-core processor**. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*.

Campbell, R. H. and S.-M. Tan (1995, May). **μChoices: An Object-Oriented Multimedia Operating System**. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS V)*, pp. 90–94.

Dean, J. and S. Ghemawat (2004). **MapReduce: simplified data processing on large clusters**. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA, pp. 10–10. USENIX Association.

Dean, J. and S. Ghemawat (2008). **MapReduce: simplified data processing on large clusters**. *Commun. ACM 51*(1), 107–113.

Dewar, R. B. K. (1975, June). **Indirect Threaded Code**. *Communications of the ACM 18*(6), 330–331.

Druschel, P., V. Pai, and W. Zwaenepoel (1997, May). **Extensible kernels are leading OS research astray**. pp. 38–42.

Engler, D. R. and M. F. Kaashoek (1995, May). **Exterminate All Operating System Abstractions**. In *Proceedings of HotOS V*.

Engler, D. R., M. F. Kaashoek, and J. O'Toole, Jr. (1995, December). **Exokernel: An Operating System Architecture for Application-Level Resource Management**. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 251–266.

Ertl, M. A. (1993). **A Portable Forth Engine**. In *EuroFORTH '93 conference proceedings*.

Ferreira, K. B., R. Brightwell, and P. G. Bridges (2008, November). **Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection**. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*.

Hunt, G. C., J. R. Larus, D. Tarditi, and T. Wobber (2005). **Broad New OS Research: Challenges and Opportunities**. In *Proceedings of the 10th conference on Hot Topics in Operating Systems (HotOS IV)*.

Kaashoek, M. F., D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie (1997). **Application performance and flexibility on exokernel systems**. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, New York, NY, USA, pp. 52–65. ACM.

Klint, P. (1979). **Interpretation Techniques**. *Software Practice and Experience 11*, 963–973.

Kogge, P. M. (1982, March). **An Architectural Trail to Threaded-Code Systems**. *IEEE Computer 15*(3), 22–32.

Nice paper describing the step-by-step implementation of a threaded code system

Maccabe, A. B., P. G. Bridges, R. Brightwell, R. Riesen, and T. Hudson (2004, June). **Highly Configurable Operating Systems for Ultrascale Systems**. In *First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-1)*, pp. 33–40.

Mogul, J. C., R. F. Rashid, and M. J. Accetta (1987). **The Packet Filter: An Efficient Mechanism for User-Level Network Code**. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 39–51.

Petrini, F., D. Kerbyson, and S. Pakin (2003). **The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q**. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*.

Pittman, T. (1987, June). **Two-Level Hybrid Interpreter/Native Code Execution for Combined Space-Time Program Efficiency**. In *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pp. 150–152.

Proebsting, T. A. (1995, January). **Optimizing an ANSI C Interpreter with Superoperators**. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 322–332.

Riesen, R., R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira (2009). **Designing and Implementing Lightweight Kernels for Capability Computing**. *Concurrency and Computation: Practice and Experience 21*(6), 793–817.

Riesen, R., R. Brightwell, and A. B. Maccabe (1998). **Differences Between Distributed and Parallel Systems**. Technical report SAND98-2221, Sandia National Laboratories.

> NOTE: Originally written in 1996 but not published as a techreport until 1998. Therefore, some of the examples refer to systems not in use anymore. The report also pre-dates the popularity of COTS clusters. The report lists a set of criteria that distinguish MP (parallalel) systems from distributed systems. It notes that with the arrival of networks of workstations, the distinction gets blurred as the systems converge. The second part of the report gives an overview of the Puma kernel and Puma Portals, the predecessor to Portals 3 [Brightwell et al. 1999].

Small, C. and M. Seltzer (1996, January). **A Comparison of OS Extension Technologies**. In *1996 USENIX Annual Technical Conference*.

> Compares SFI, trusted compiler, and interpreters

Tan, S.-M., D. K. Raila, and R. H. Campbell (1995, August). **An Object-Oriented Nano-Kernel for Operating System Hardware Support**. In *Proceedings of the Fourth International Workshop on Object Orientations in Operating Systems*, pp. 220–223.

Vahdat, A., D. Ghormley, and T. Anderson (1994, December). **Efficient, Portable, and Robust Extension of Operating System Functionality**. Technical Report UCB CS-94-842, Computer Science Division, UC Berkeley.

Motivation for GLUnix

Wagner, A., H.-W. Jin, D. K. Panda, and R. Riesen (2004, September). **NIC-Based Offload of Dynamic User-Defined Modules for Myrinet Clusters**. In *IEEE Cluster Computing*, pp. 205–214.

Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham (1993, December). **Efficient Software-Based Fault Isolation**. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 203–216.

Wheat, S. R., A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup (1994a). **PUMA: An Operating System for Massively Parallel Systems**. *Scientific Programming 3*, 275–288.

This is an extended versions of the Hawaii paper (Wheat, Maccabe, Riesen, van Dresser, and Stallcup 1994b). It contains the new stuff about capabilities.

Wheat, S. R., A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup (1994b). **PUMA: An Operating System for Massively Parallel Systems**. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pp. 56–65. IEEE Computer Society Press.

THE Puma paper. Describes the kernel and portals.

DISTRIBUTION:

1	MS 1319	Rolf Riesen, 1423
1	MS 1319	Kurt Ferreira, 1423
1	MS 0899	Technical Library, 9536 (electronic)

Sandia National Laboratories