

Right-Weight Kernels: an off-the-shelf alternative to custom Light-Weight Kernels*

Ronald G. Minnich, Matthew J. Sottile^{*} Sung-Eun Choi[†] Erik Hendriks[‡] Jim McKie[§] ^{||}

ABSTRACT

DOE has put forth a considerable effort into light-weight kernels for high performance computing, yet there has been a lack of acceptance and use, perhaps due to the limited support for hardware and software. The arguments for light-weight kernels have been based on the problem of *interference*, i.e. changes in application performance that occur when the operating system pre-empt the application. Nevertheless, using existing, well supported operating systems for HPC systems has been quite successful. The problems with the standard operating systems remain, however, although their impact on applications is still not quantified. At LANL, we have undertaken a research program to determine whether Linux and/or Plan 9 can be used to realize the benefits of light-weight kernels while maintaining the benefits of a full-featured operating system. Specifically, we are evaluating measures that quantify what is "good" in a Light-Weight Kernel. We are using this knowledge to modify Linux and Plan 9 to make them competitive with custom light weight operating systems, in essence, a Right-Weight Kernel. This paper represents both a summary of early results as well as a description of work in progress.

1. INTRODUCTION

The problem, in short, is simple: given a parallel system with a large number of nodes, an application would like to completely own the nodes while it is computing, in order to ensure that aggregate activities, i.e. activities involving all the programs on all the nodes owned by the application, proceed at full speed. Anything less than complete owner-

*This research was funded by the Mathematical Information and Computer Sciences (MICS) Program of the DOE Office of Science and the DOE ASCI Program. LANL LA-UR-06-1324. The software used in this work was in part developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago.

^{||}Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36.

^{*}Los Alamos National Lab, Los Alamos, New Mexico 87545, rminnich@lanlg.gov

[†]Cray Inc., 411 First Avenue S., Suite 600, Seattle, WA, 98104.

[‡]Google, Inc., Central Way Plaza, 720 4th Avenue, Ste 400, Kirkland, WA 98033

[§]Bell Laboratories 600 Mountain Avenue Murray Hill, NJ 07974

ship of all the nodes during these activities will cause the application to run poorly.

This problem has been observed in the real world. It was first documented by Ron Mraz at IBM[4], was later observed on the Red Storm and CPlant machines, and more recently on machines such as ASCI Q[5]. Different researchers have developed a number of different approaches to solving the interference problem. On ASCI Q, the problem was largely resolved by removing unnecessary daemons. On the IBM SP/2s the problem was resolved by making simple local scheduling decisions.

On ASCI Red, the resolution was drastic[9]. A full-featured operating system from Intel, OSF/1-MK ADh, was replaced by a very simple operating system with few features at all, called Puma[10]. This kernel was called a Light-Weight Kernel, or LWK.

To paraphrase an old saying, where we are is a function of where we were. In order to understand why we build HPC systems as we do, we should know what the intellectual underpinnings of our current systems are. In the next section, we provide a short background of kernel development.

1.1 A short history of kernels, or how we got where we are today

The time-sharing systems we take for granted today were developed as a response to the high cost of computing hardware; their use is a purely economic decision. Computers cost a lot, and time-sharing allows organizations to amortize that cost over many users. In a very real sense, time-sharing is a sacrifice that users make in order to access a shared common resource they could not individually afford. In a perfect world, all users would have all the computers they wanted at no cost. Time-sharing is a result of our imperfect circumstances.

Time-sharing also exacts a cost. In an ideal world, each application would get the machine exactly when it was ready, and would have dedicated use of the machine until it could sleep for a time while waiting for something else to complete, such as I/O. Unfortunately we do not live in an ideal world: applications do not run as fast in a time-sharing environment as they would on the raw hardware, simply because they do not always get the machine when they are ready to use it. The actual run time exceeds the ideal run time. This growth in run time is characterized by a measure called the *expansion*

sion factor, which in typical large-machine installations can be as high as an order of magnitude: an application, given dedicated access to the machine, would run 10 times faster than it does in the shared environment.

With the advent of Symmetric Multi Processors (SMPs) and parallel programs, time sharing got complicated. A program that ran on, say, 8 CPUs, might not run well unless it ran on all 8 CPUs. Each of the 8 parts of the application had to be simultaneously scheduled onto 8 CPUs, or not scheduled at all. If, by some chance, one of the 8 CPUs got handed to some other process, then the other 7 CPUs would sit idle while waiting for the other process to get a CPU again. This problem is one of the earliest examples of interference, and led to the development of so-called *gang scheduling*, i.e. the scheduling of multiple CPUs as a single unit.

In short, in a time-sharing environment, the non-dedicated nature of the machine makes itself apparent to each application in reduced performance: applications and OS activities interfere with each other, reducing the peak performance of each application turn. For the rest of this paper we will refer to this phenomenon as *interference*. An example of interference is the SMP case, where interference in just one CPU leads to a complete cessation of progress in the other seven CPUs until the eighth CPU is returned to the application.

Time-sharing systems are implemented with an operating system kernel, or kernel, a sophisticated descendant of the executives from the old days. Time-sharing Kernels are not without their merits. Time-sharing kernels provide:

- a protection boundary for processes, ensuring that they do not damage each other or the kernel,
- standard file system I/O, so that processes can open, read, and write files
- prefetching of data, and coalescing of file system writes so that maximum efficiency is achieved for file system I/O
- network protocols, network drivers and network security models, so that applications can safely access network resources without fear of destroying data, or revealing it to others
- Concurrency for process I/O, so that processes need not wait on data to be completely written out before the process proceeds on a computation

Operating systems kernels successfully manage a degree of concurrency that few programmers are capable of handling. In general, applications writers that attempt to manage data as efficiently as the kernel will fail. For the cost of abstraction of the kernel, there is a gain in efficiency that can not be ignored.

1.2 Clusters and other distributed memory machines

Clusters and similar machines represent a step back from time-sharing systems. The nodes are dedicated to an application at a time. It might seem that this dedicated operation would eliminate the interference problem, but it does

not. There is still significant non-application activity on the node. The operating system is scheduling internal activity that is triggered by clock interrupts. There are daemons on the system that can become active in response to a network packet.

The interference will typically happen if all the application processes are participating in a global operation (e.g. MPI collective) and one application is held up due to interference. The situation is exactly like the SMP situation described above. On a large distributed-memory machine, with 1000s of processors working together, having one processor suddenly unavailable can wreak havoc with the parallel performance of the application. In fact, going back to our 8-CPU example above, we can see that the problem will be 128 times worse, and hence the allowable outage will be 128 times smaller. This time can become very small indeed, as we shall see.

Consider, again, the case where 999 processes arrive at a barrier, and one is late. The laggard can not be too late, or the parallel throughput of the application will suffer. On today's machines, 'too late' is a very small number indeed. For example, if a 1000-node application hits this barrier once every few milliseconds, the allowable time for 'too late' is measured in microseconds.

The Cluster community approach: get rid of part of the kernel

The cluster community has taken an entirely different approach. The first step, starting with the SCRAMNet[3] system in 1987 and continuing on to Myrinet, Quadrics, and other systems, was to completely bypass the section of the operating system concerned with networking. Applications directly accessed hardware to make bits move. This mode of operation is a lot like the very earliest days of computers, where not a bit of I/O moved unless the application directly tickled a bit of hardware. It involves the application in the lowest possible level of hardware manipulation, and even requires application libraries to replicate much of the operating systems capabilities in networking, but the gains are seen as worth the cost.

The LWK approach: get rid of most of the kernel

LWKs take the process even further. LWKs eliminate almost all the capability of the kernel, providing an environment much like the executive environment mentioned above. Once an application starts, it owns the node. The LWK provides only the most basic functions for I/O. In essence, LWKs are a retreat back to the earliest type of operating system, removing capability in an attempt to provide better parallel performance.

LWKs eliminate most of the functions that people take for granted with a kernel. There is no file system. There are no sockets. In many LWKs, there is no paged memory, and hence no virtual memory. There is no security model in an LWK. In fact there is so little of the function that we associate with a kernel that applications users are sometimes unsure of whether to call it a kernel at all.

1.3 Do LWKs go too far?

The question is, do LWKs go too far? Some anecdotal evidence indicates they do. We have been told that many potential users of the ASCI Red machine opted for the Unix clusters instead, simply because the LWK did not support enough operations to allow their applications to run.

Users often say they would like to get the OS out of the way. But as we have seen, they want the OS out of the way until they need it for such things as:

- Shared libraries.
- Application fault management and debug support.
- A reliable file system.
- Sockets.
- Security.

The items mentioned above are just a few of the OS capabilities that the programmers know about. As we have mentioned, there is a lot of magic under the hood of a conventional kernel that supports concurrency, non-blocking I/O, secure communications, high performance network file systems, and the host of other things that programmers don't always know they're taking for granted.

1.4 What is the quantitative case for LWK?

We should also ask whether there is a quantitative case for LWK. As it happens not much has been done to quantify the arguments for LWK. It is true that very good scalability has been shown on the ASCI Red machine, as well as on the IBM BG/L machine. The problem is that these systems are all-or-nothing propositions; there is no option at present of running a comparable Linux system on them and even if there were, there is not a rigorous, quantitative, reproducible measure with which to compare the two types of operating systems.

The types of issues that come up include:

- What role do clock interrupts play? How do we measure the impact and what units do we use to quantify the impact? How do we measure an operating system against this yardstick – what is the measure of an operating system's worth?
- What role do architectural features such as Paging hardware, Translation Lookaside Buffers, and so on play? For example, for Linux, if we run the processor in a mode such that only 4 Mbyte pages are used, how would this impact the kernel's performance in HPC environments?
- What is the tradeoff, for applications, of having a file system present to support concurrency and caching vs. the overhead of having that file system there? We know that the performance of Sage, an important application component, can improve as kernel components are removed[5]. But Sage is only 1/3 of the application of which it is a part: how much harm is done

to the other 2/3 of the application by removing the file system?

- Is the interference problem the same over time, or does it change the longer a system runs, and if so, why?

These and many other questions wait to be answered. There is much anecdotal evidence, but what we need are rigorous measures and definitions that will allow us to compare systems realistically. It has proven impractical to compare LWKs and regular kernels on the same hardware, at least to date; therefore, generating quantitative measures, well grounded in sampling theory, is a next best step.

2. RIGHT-WEIGHT KERNELS

The research we are undertaking at LANL is grounded in several fundamental assumptions:

- we are not taking for granted that LWKs are necessary
- we believe that a properly configured off-the-shelf OS, such as Linux or Plan 9, can be used in HPC
- we can use simulation tools to recreate HPC environments, and perturb the systems in a way that will allow us to recreate OS interference phenomenon

For our use of off-the-shelf operating systems, we are taking a two-pronged approach.

- The first track is to take Linux, a well supported, existing operating system in wide use in DOE HPC systems and see if, with specific modifications, it can be made light-weight enough to compete with true light-weight kernels.
- The second track is to take Plan 9, an existing commercial, open-source operating system which is much lighter-weight than Linux and designed from the ground up to be a distributed system, and see if that operating system can compete with true light-weight kernels. Plan 9 is used in hard-real-time environments, such as routers, and the interference problems resolved in these systems are very similar to those we must resolve for HPC.

Before we can begin to answer the question of "compete with," we will need metrics. In what way should a kernel compete with a light-weight kernel, and what defines success? In other words, in the HPC context, what is *good*, and can we move the terms of *good* from the qualitative to the quantitative domain?

2.1 What is good?

There has not been any quantification of what makes an operating system sufficiently *light-weight*. What is light-weight is also defined by a particular application or class of applications. Clearly, the printer daemons should not be running on cluster compute nodes, but a file system daemon may be reasonable for applications that output large amounts of

```

start = sample(real-time-clock);
/* perform small work quantum
*/
end = sample(real-time-clock);

delta = end - start;
/* store the delta to a file */

```

Figure 1: A defective microbenchmark

data throughout an entire program run. The question is what is considered reasonable with respect to the impact and perturbation that context switches and other daemons cause. In general, the operating system needs to perform its job, but just how little can it do? For example, on ASCII Red, the PUMA light-weight operating system simply sets the node up for the application and never interrupts it again. PUMA provides no real file system environment; some users found this quite acceptable, others did not.

As we will show below, simply removing components, performing global scheduling, and trying to minimize clock interrupts is not sufficient. In some cases, there is interference on our systems that has not been quantified in research performed to date, and in some cases has been erroneously attributed to the operating system. In other cases, removal of OS capabilities may improve one segment of an application but lower performance in another segment – as in the case of removing file systems. The quantitative measure of interference is not simple, and an optimal node configuration may actually change during different phases of the application.

Sandia has done some preliminary work in this area. In particular, it has been identified that applications suffer greatly when interrupts on the individual nodes are not globally synchronized. Specifically, applications seem to run best when the operating system schedules an application concurrently on all the nodes. As clusters grow in size, applications will need to tolerate even more asynchrony. Can something be done about this at the library and runtime system level?

In order to answer these questions, we are performing a study of relevant applications to determine why they are so sensitive to unsynchronized scheduling. We are looking at support libraries and runtime systems to determine what can be done to minimize the effects of asynchrony on an application. Why are these applications so sensitive to unsynchronized scheduling?

2.1.1 Difficulties with simple measurements

Measuring OS interference is more difficult than it first seems. We have found that simple measurements are prone to serious failures[8]. In this section we provide a flavor for the type of issues that come up and how they can mislead performance analysts.

Shown in Figure 1 is a simple-minded microbenchmark for evaluating OS interference.

The problem with this benchmark is that it violates several rules of sampling. The time base is relative, not absolute. The start of the sample interval is not fixed. The well-known

```

base = sample(real-time-clock);
start = sample(real-time-clock);

/* for good sampling,
* make sure we start at
* an aligned time
*/
while (not_time_interval_aligned(start))
    start = sample(real-time-clock);

end = base + Delta;
while(sample(real-time-clock) < end)
/* perform a small work quantum,
* not too small;
* it should be longer in time than
* cache-line-flush,
* for example
*/

/* increment a counter */

end = sample(real-time-clock);

/*Store start, end and amount
* of work done to a file */
/* for the next iteration,
* increment end by Delta */

```

Figure 2: A correct microbenchmark

consequence of these two mistakes is that signals can be aliased and in some cases missed. We have measured both of these problems in practice. In one case, on a machine with two intel Pentium 4 processors, we ran two compute intensive codes to model complete OS interference. We then ran this benchmark shown in Figure 1. The benchmark got very little time on the machine, but also presented ideal results – as though there were no interference at all! Clearly, this error-prone benchmark is not appropriate for general use.

We built a new benchmark based on more solid ground. This new benchmark is shown in pseudo-code in Figure 2.

This benchmark has proven to provide very accurate data. We can use standard signal processing tools with the data produced by this benchmark. In fact, we can not only show interference exists, we can provide extremely accurate spectral data, i.e. we can determine both the frequency and amplitude of the interference phenomenon. We can compare the relative interference of, e.g., a standard Linux workstation and a simple, stripped-down compute node.

Shown in Figure 3 is a raw data plot from the benchmark, running on a machine with a full Plan 9 desktop environment. We are showing, with time on the X axis, the amount of work done per unit of time, over a period of time. Normally, a lot of work gets done, so that the points plotted are uniformly high. Occasionally, some other task interferes with our program, and a correspondingly low point can be seen as a downward spike. Interference can therefore easily be seen as the downward pointing spikes; their period can be inferred by referring to the time (X) axis; and some intuitive idea of the degree of interference can be inferred from the thickness of the line. There is a lot of information in

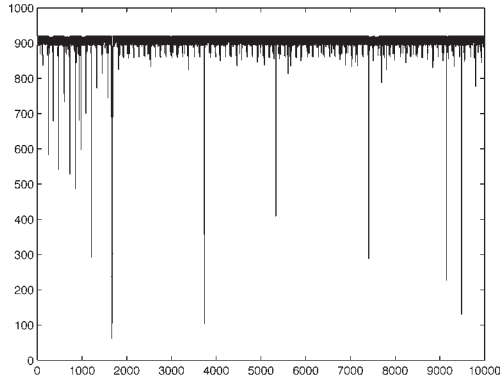


Figure 3: Plan 9 scheduler adaptation to process startup

this graph.

This trivial plot has a surprise built in. The process starts up and there is a great deal of interference. As the process continues to run, the interference is reduced. We do not completely understand why this is happening. We have not seen this behavior on Linux, MacOS X, or Windows XP.

As we mentioned, because our sample time basis is so stable, much more sophisticated processing is possible. Shown in Figure 4 are two series of graphs. The raw data graph is shown in the left column. The right column shows the result of processing to make view spectral information easier. The raw data was run through a Hamming filter, and then a cepstrum graph created. The cepstrum is the forward Fourier transform of the spectrum, and can be very useful for revealing data not always available in a simple spectral graph, i.e. data that might be hidden by harmonics.

We show the machine at boot time, console login time, and running gnome desktop. The graphs are normalized to the same scale. Not surprisingly, the boot-time interference is very uniform and small; the gnome environment is quite noisy, as revealed by the increasing number and amplitude of the frequency components on both graphs. It is also interesting to note that the full Plan 9 desktop environment is at most as noisy as the Linux console environment. This lends support to our earlier conjecture that Plan 9 may be a suitable system for computer nodes.

3. SIMULATOR RESEARCH

As discussed earlier, the key problem with OS interference with applications is the idleness induced on processors experiencing less interference than those they are interacting with, either via collective or point to point operations. In addition to quantifying the interference present on a system via microbenchmarks such as the one shown above, it is useful to analyze real application runs to explore their sensitivity to noise. We have been constructing the Chama simulation engine for this purpose[7].

The Chama tool is both a simulator and analyzer. Given a

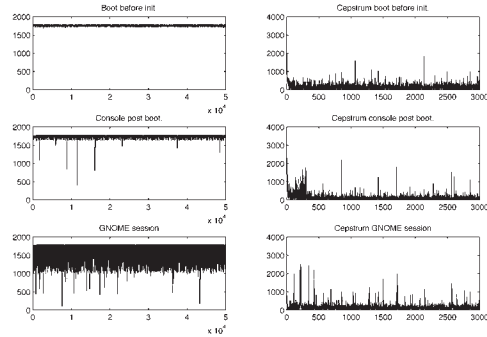


Figure 4: Workstation interference spectrum and cepstrum

parallel application based on the MPI programming library, we perform runs using real-world input files, thus causing the application to behave as it is expected to with respect to workloads and message passing patterns. Each message passing operation is logged to a tracefile for input into the Chama tool after the application completes execution. The overhead for such tracing is undeniable, and perturbs the application to some degree. Fortunately, it is consistent across nodes, and has been written to be low impact so as to avoid traces being too perturbed to represent real, non-traced runs.

Given a trace, we then reconstruct the message passing pattern in a manner similar to what users see within trace visualization tools such as the Intel Trace Toolkit and Paraver. Instead of visual examination of the messaging, we use this message passing pattern to define a directed graph that represents message flow within the parallel program, with event timings on each processor annotating the graph nodes. We then propagate perturbations along this graph to represent operating system interference, latency fluctuations, and other performance factors of interest. As these perturbations are propagated through, they are used to change the annotations based on the trace time stamps. The blocking semantics of each event are used to determine how far, and to what magnitude, simulated perturbations are passed through the graph.

To date, this tool has been used to gather statistics about the parallel programs and identify which MPI operation class (reduces vs gathers vs etc...) were the most prone to skew and noise, allowing the programmer to get a better sense of where tuning is required. Most recently, the tool has supported a 1.6 million event trace of a 32 proc ASCI FLASH run. This tool will be released as open source in April, 2006.

4. OPERATING SYSTEMS

4.1 Linux

Linux is arguably the most popular operating system for high performance computing today. Consequently, support for compilers, debuggers, and messaging libraries is pervasive. Standard desktop installation of Linux such as Red Hat include countless numbers of tools running countless

numbers of daemons. Most of these daemons are wholly unnecessary for cluster compute nodes.

LANL has performed some initial measurements on Pink, a 1024-node BProc cluster. BProc is a set of Linux kernel modifications that provides a single-system process space across an entire cluster [2]. One of the added benefits of running BProc is that the user space software running on the compute (*slave*) nodes is extremely light weight. There is only a single daemon running — the BProc slave daemon. While an application is running, this daemon is almost completely idle. The only interruption from another process is a tunable heartbeat built into the BProc protocol. This happens *once a minute*, an eternity in computer time, and can often be made much less.

The significant interruptions that remain are generated by the kernel. For example, Linux on x86 architectures has a timer that interrupts the kernel at 100hz. There are also other kernel threads which perform internal book keeping activities, flush blocks to disk and so on. Some of these may not be necessary on a cluster node. In other cases, it may be possible to reduce the impact on parallel applications.

4.2 Plan 9

Plan 9 [6] is an operating system developed at Bell Labs in the early 1990's. We are using Plan 9 as a testbed HPC operating system because its architecture represents an interesting hybrid of LWK and commodity OS ideas. The kernel itself is small, simple, and does not contain the usual components found in operating systems like Unix, such as file systems. In Plan 9, the only components hardwired into the kernel are those which are absolutely necessary for basic operations (*devices*); other components, which can be on the same or other computer, are run outside the kernel (*servers*). Devices include hardware controllers, process management, and network protocol stacks. Servers include traditional functions such as file systems.

While Plan 9 has all the capabilities of a traditional kernel architecture, it is also, in many ways, simpler than LWKs. There are over 80 system calls in the LWK for the BG/L CNK. Plan 9 has only 40. Plan 9 thus represents a point in the design space somewhere between Unix-like systems and the LWKs such as Puma and the BG/L Computational Node Kernel, or CNK. The one thing that Plan 9 has that LWKs do not is a clock interrupt, which places Plan 9 permanently outside the circle of LWKs, it being a cardinal principle of LWKs that a process should never be interrupted.

As part of continuing work with Plan 9, Bell Labs has recently added new capabilities to better perform hard-real-time tasks needed for wireless routers. The problems of OS interference on a router have an interesting similarity to the problems we see in HPC. When processing packets in these routers, it is very important that the packet processing proceed without interference from the operating system. We are finding that the causes, and remedies, are often similar: recent work to improve Plan 9s performance as a wireless router seems to have applicability to its use as an HPC platform.

Plan 9 allows individual users to make policy decisions about

where components should be placed. For example, users can, if they wish, run a file system on the same node as the application. Or they may create a socket, and bind the socket to a portion of their name space, and run the file server elsewhere. A user might decide, for example, that having a local file system on the node would optimize some phase of an application; another user might decide that the file system would be best run outside the node. There is thus a high degree of customization possible in the user's runtime environment, and it is all controllable by the user.

5. CONCLUSION

LANL and Bell Labs have undertaken a program to answer an important question: can off-the-shelf operating systems continue to be used for HPC systems which, in future, are expected to have up to one million CPUs? Will the general-purpose aspects of these systems, which make them useful for desktops and web servers, fatally impair their use in HPC?

To aid in our comprehension of the question, we are developing measurement tools that allow us to provide reproducible, quantitative measures of these operating systems; and a simulator which allows us to do full application simulation, including interference simulation.

We can then apply the lessons we learn from these tools to our two candidate operating systems, Linux and Plan 9. We have already learned some important lessons from our measurement of these systems. Linux systems have very different interference patterns 24 hours after being booted. The Plan 9 driver for the Intel EEpro 1000 chips has a periodic queue flush operation that causes very visible interference when measured with our tools.

In the end, we hope to learn how to configure a kernel that is the right weight for HPC — i.e., a Right-Weight Kernel.

6. REFERENCES

- [1] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, 1999.
- [2] Erik A. Hendriks. BProc: The Beowulf distributed process space. *16th Annual ACM International Conference on Supercomputing*, June 2002.
- [3] Stephen Menke, Mark Moir, and Srikanth Ramamurthy. Synchronization mechanisms for scamnet+ systems. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 71–80, New York, NY, USA, 1998. ACM Press.
- [4] Ronald Mraz. Reducing the variance of point-to-point transfers in the IBM 9076 parallel computer. In *Supercomputing*, 1994.
- [5] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asc q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

- [6] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.
- [7] M. Sottile, V. Chandu, and D. Bader. Performance analysis of parallel programs via message-passing graph traversal. In *Proceedings of IPDPS 2006, Rhodes Island, Greece*, 2005.
- [8] M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 371–377, Washington, DC, USA, 2004. IEEE Computer Society.
- [9] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: an operating system for massively parallel systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume II, pages 56–65, Wailea, HI, USA, 1994.
- [10] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: an operating system for massively parallel systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume II, pages 56–65, Wailea, HI, USA, 1994.