Operating System Issues for Petascale Systems*

Pete Beckman Kamil Iskra Kazutomo Yoshii Susan Coghlan
Argonne National Laboratory
Mathematics and Computer Science Division
9700 South Cass Avenue
Argonne, IL 60439, USA
{beckman, iskra, kazutomo, smc}@mcs.anl.gov

ABSTRACT
Petascale supercomputers will be available by 2008. The largest machine of these complex leadership-class machines will probably have nearly 250K CPUs. These massively parallel systems have a number of challenging operating system issues. In this paper, we focus on the issues most important for the system that will first breach the petaflop barrier: synchronization and collective operations, parallel I/O, and fault tolerance.

Keywords
petascale, synchronicity, noise, parallel I/O, fault tolerance

1. INTRODUCTION
It is very hard to predict the future. Science fiction writers had difficulty imagining a general-purpose computer. The utility and pervasiveness of the Internet were also overlooked. Many events that experts believed would be extraordinarily hard or impossible, such as beating a human at a game of chess, have turned out to be possible. Likewise, topics that seemed relatively straightforward, such as autonomous vehicle piloting, have proven quite challenging. The same is true for petaflop computing.

In February 1994, a series of workshops began to explore how petaflop computing might be reached. After this initial meeting, interested researchers met yearly in Bodega Bay; those sessions have been generally referred to as the *Bodega Bay Petaflops Workshops*. At that time, the CM-5 from Thinking Machines was dominant, occupying four of the top five slots in the Top500 list [12], with the largest machine bragging 1024 CPUs. Cray vector machines, such as the Y-MP, also retained honorable positions. However, imagining petaflop computing—which was 10,000 times faster than machines of the day—was daunting.

While many of the predictions from those petascale workshops were indeed insightful, some of the most critical design issues for petascale machines were overlooked. While participants spent time imagining new computer languages, processing-in-memory, architectures creation and management of one million threads of control, and the future capabilities of such a monstrous machine, such as the reconstruction of DNA of extinct species, they overlooked issues that have now become obvious, such as electrical power.

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Power consumption is now a critical constraint for petascale systems. The total monthly bill is one issue, but possibly more important is how power and the required cooling affect density and floor space. There are practical limits to the size of a machine room.

Of course, with petascale machines now only two to three years from delivery, we have a tremendous advantage over the participants of the Bodega Bay workshops. The important issues are in clear focus. The most significant issues for petascale operating systems are synchronization, parallel I/O, and fault tolerance.

The remainder of this paper is organized as follows. A short introduction into petascale architectures and operating systems is provided in Section 2. Section 3 discusses how synchronization affects operating system design. The challenges of I/O are the topic of Section 4. Section 5 examines fault tolerance for petascale machines. Section 6 provides the conclusions and discusses the next step for petascale systems.

2. PETASCALE ARCHITECTURES
Three architectures seem likely to achieve petaflop speeds over the next five years. While architecturally distinct, from an operating system (OS) point of view they are actually quite similar. First are the commodity scale-ups. These machines use what are essentially commodity servers packaged, with varying levels of integration, with a high-speed interconnect. They are represented in the Top500 list as commodity Linux clusters, IBM JS20 Cluster, Dell PowerEdge Cluster, or Itanium2 Tiger4 Cluster. The largest of these machines has about 12,000 CPUs. Many are SMPs. To reach a petaflop, these architectures must get about 12 times faster, a feat likely with just one size doubling combined with the availability of four- and eight-core CPUs. Therefore, for this architecture we anticipate from 1,000 to 10,000 contexts, depending on the size of SMP node. The most common range will be 4- to 16-way SMP.

The most likely candidate for being the world's first petascale machine will be IBM's BG/P, the followup to the successful BG/L [9] architecture, and current Top500 leader. The BG/L at Lawrence Livermore National Laboratory (LLNL) is the fastest machine on the planet by a factor of almost 4.4 from its nearest architectural rival, a commodity scale-up. BG/L is really quite different in several important aspects. No part of the BG/L is sold as a departmental server, nor would it be successful as one. Its abilities stem *only* from its unprecedented scale. With 131,072 CPUs, it is 16 times bigger than the largest commod-

ity Linux cluster. Combining low-power (both electrically and computationally) with a tightly integrated high-speed interconnect, the LLNL BG/L has a peak theoretic speed of 367 TF and can deliver 280 TF on LINPACK [2]. The largest BG/P will more than double the CPU count and increase clock speeds and network links. Nodes will be 4-way SMP with cache-coherent memory. With approximately 70,000 contexts, the largest BG/P will stress the limits of the OS architecture.

The third type of possible petascale architectures is actually a subspecies of the commodity scale-up. Custom accelerators such as GRAPE and ClearSpeed can turn a commodity Linux platform into a petascale system. From an OS perspective, however, these machines are largely the same as their more vanilla counterparts. They simply have a special-purpose subroutine processor. Still, these architectures should not be underestimated: after all, GRAPE-4 [4] was the first system to break the teraflop barrier back in 1995. Vector machines fall into a similar regime from the OS perspective, looking a lot like a cluster with special floating-point capabilities.

From the OS, these three classes of architecture, all capable of achieving petaflop performance in the next five years, are similar. They handle the basic run-time system for the application, interface to the low-level messaging system, pass I/O back to the file system, and manage process startup and shutdown. Therefore, they must all address the same OS issues for petascale computation: synchronization effects, I/O challenges, and fault tolerance.

3. SYNCHRONICITY

Synchronicity is a fundamental requirement for many programs that use the Single Program Multiple Data (SPMD) programming model such as MPI. These programs often coordinate global activities with barriers or global reductions. Most often, the program runs as slow as the slowest processor. Therefore, small delays in processing, or "detours," can cause a node to arrive at a global synchronization point late, forcing the entire synchronization to be delayed. Even if long detours are unlikely for any given node, the size of a petascale system makes this a critical issue.

What are the sources of these delays, or "noise," as some researchers call it? Some come from the hardware itself. For example, clock skew on a large machine can affect communication links and influence parallel performance. At least one massively parallel architecture employs special hardware mechanisms to synchronize the machine, such as a single clock generator to drive the whole system, including network hardware at each node. On the other hand, each compute node on a commodity Linux cluster is driven by an individual clock generator, or even multiple clocks, one for each piece of hardware. This is a fundamental difference in design between a commodity cluster and a massively parallel machine. But even if the hardware is fully synchronized, software is usually the greatest source of detours and can therefore have a tremendous impact on parallel performance.

A typical OS such as Linux is designed for multiuser multitasking. It has quick response even under high load. The coders working on Linux are motivated to tune for responsiveness of the user interfaces. This design goal is actually a disadvantage and a source of overhead for computational usage.

For example, a typical OS tends to use small-size pages (such as 4 KB or 8 KB) for efficient memory use. During code execution, a CPU must translate a virtual address to a physical one for every memory access. Unfortunately, this is an expensive operation. To speed it, CPUs have a translation lookaside buffer (TLB). On most CPUs this buffer is fairly limited in size. A memory access is ten or more times slower when the TLB does not have the virtual to physical mapping handy. This buffer "miss" is costly; the mapping must be looked up outside the CPU. Thus, small-size pages slow applications that sweep big areas of memory. Unfortunately, this is exactly the behavior of many scientific applications that walk through large data structures. Because of differences in data decomposition and layout on a massively parallel machine, TLB miss rates may also differ across the machine. This situation can significantly affect synchronicity. To mitigate this, many compute node kernels use extremely large pages for user applications. If all the pages associated with an application can be mapped into the TLB, there will be no misses and hence no loss of synchronization.

A typical OS is designed for quick response; nearly any operation can be preempted. Since every process pollutes or modifies the cache in a different way, the harm caused by running other tasks is more than just the total time stolen from the application: it can also cause increased cache misses. Once again, the synchronization across the machine can be changed as some nodes load their operands from cache and some from main memory. A compute OS needs to reduce the number of cache polluters. Controlling the behavior of processes with special MMU settings is one method for addressing this problem.

Table 1 shows some examples of preemptive detours during a computation. The data is measured by a benchmark called Selfish, which measures all cycles not given to the application. Those cycles may have been used to process interrupts or processes such as the cron daemon. The table provides the ratio of the OS detour to the total CPU time and the maximum duration of a single detour.

In the table, the BG/L compute node (CN) is running BLRTS, a lightweight kernel developed by IBM, optimized for computation. The BG/L I/O node (ION) has exactly the same CPU hardware as the BG/L CN; however, the I/O nodes run a highly tuned Linux. Jazz is a standard commodity x86 Linux cluster with a relatively untuned Linux kernel.

From the table, one can see that BLRTS has almost no sources of delay to the benchmark application. Very few cycles are lost to other handlers. Interesting, the tuned Linux kernel running on the BG/L I/O nodes is also relatively quiet. The detours present on Jazz, however, are stark. Careful tuning of the Linux kernel can have a dramatic effect on the detours and therefore the loss of synchronization across a petascale system.

Table 1: Operating System Detour (Noise).

Node	CPU	OS	Ratio	Max [μs]
BGL CN	PPC (700 MHz)	BLRTS	3.2e–07	1.8
BGL ION	PPC (700 MHz)	Linux	2.0e–04	10
Jazz	Xeon (2 GHz)	Linux	1.2e–03	110

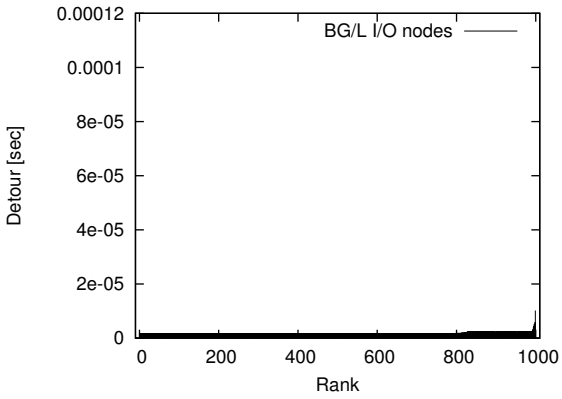

Figure 1: Detours on a BG/L I/O node.

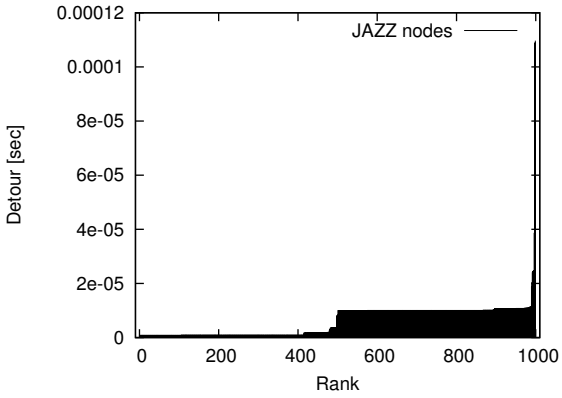

Figure 2: Detours on a Jazz node.

Figures 1 and 2 provide graphical representation of the detour results (detour samples are sorted by detour time) from a BG/L ION and a Jazz node, respectively. On both graphs, base activities come from timer updates. Although the application is interrupted at a relatively high rate, every 10 ms, this is not a big issue because the interval is fixed, so such interrupts are relatively easy to synchronize. The spikes close to the right edge of the graph are far more dangerous because they come from asynchronous detours. On a petascale system, even with the probability of such long delays being exceedingly small for each individual processor, such a detour would likely be the source of a delay for each global operation. In this case, for a petascale machine using the Jazz Linux kernel, every barrier would be at least 110 μs. Of course, zero interruption is ideal for parallel applications. In practice, however, if an application needs full operating system services, such as support for multiple processes or dynamically loaded libraries, the OS must support multitasking. Reducing OS interference and the maximum detour times is essential for petascale machines.

The current design of compute node operating systems is based on an idea that cuts or minimizes OS interference with parallel application. In that sense, there may be little room for further improvement. However, current compute node operating systems, being so lightweight and independent, have a tendency to perform redundant work. Such redundancy might be seen as part of the SPMD paradigm, but there is no reason why the operating system should not try to optimize it. For example, most of the time, there is no need to perform file open operations independently from each of the thousands of compute nodes. In order to avoid such situations, global resource management is required—a design that we hope to see in future operating systems.

A number of studies on OS noise or interference have been reported. Petrini et al. [6] discovered the source of performance problems on a supercomputer related to unrequired processes and tasks stealing cycles from the application. They developed a methodology to determine which sources of noise impact the performance and how to eliminate them. With their methodology, they improved the performance of the ASCI Q machine. Jones et al. [3] obtained a factor of 3 improvement in the execution times of collectives such as *allreduce* by adding parallel awareness to the operating system, consisting primarily of coscheduling parallel processes across the whole machine. Agarwal et al. [1] provided an initial theoretical study into the problem of the impact of noise on the scaling of collectives. They found that exponential noise is not much of a problem but heavy-tailed or Bernoulli noise can drastically reduce the performance. All of these studies show that the core issue is synchronicity of operating systems for petascale machines.

4. PETASCALE I/O

Data input/output can be a considerable problem on petascale machines. Stripped-down kernels running on compute nodes are ill-equipped to handle it. From the perspective of the storage server, handling I/O from 10K to 100K individual compute nodes would be daunting. Therefore, file I/O is typically offloaded onto dedicated I/O nodes, acting as "client reducers." As a trivial example, imagine a 100K machine in which all processors try to open a file for reading. The resulting file system storm would probably swamp any single-interface storage server. Furthermore, without intelligent file system semantics, 100K copies of exactly the same file could be pushed through the network. Parallel supercomputers must be paired with parallel storage servers, and I/O nodes must perform the impedance matching.

The amount of data that can be generated by a petascale machine is staggering. Consider the 64K-node BG/L machine at LLNL. Each of these nodes can put data on the collective network, used for communication with I/O nodes, at 2.5 Gbit/s. For the whole machine, that would result in 20 TB/s of aggregate I/O. If the internal communication network and backend file system could support these amazing rates, the entire 32 TB system could be checkpointed in about 2 seconds. However, supporting such an internal network is cost prohibitive. Naturally, compromises have to be made. There is but one I/O node for 64 compute nodes, decreasing the throughput by that factor. Moreover, each I/O node has a Gigabit Ethernet interface, capable of at most 100 MB/s. So in total, the I/O capability of the entire system is 100 GB/s, assuming perfect scalability within the Ethernet infrastructure.

It takes a large Ethernet switch to handle Gigabit traffic from 1024 I/O nodes. For BG/L machines, the ratio of 64 compute nodes to I/O nodes is considered to be *I/O poor*. Much richer configurations, up to 8 compute nodes per I/O node, are possible. However, no single switch currently on the market has 8192 Gigabit Ethernet ports, so such configurations are fully

usable only on smaller-scale machines.

Obviously, no single fileserver can currently handle data input in the range of 100 GB/s. Thus, file I/O must be parallelized. One could imagine a set of independent NFS-based fileservers handling the task. However, access to such scattered data would be rather inconvenient. Further, NFS was not designed for highly parallel access. Our experiences indicate that just a few dozen not particularly persistent parallel writers make an NFS filesystem unusable for interactive work by other users. A larger number of more aggressive writers will bring it to its knees.

A dedicated parallel filesystem has become a standard component for leadership-class architectures. It provides its users with a single directory tree, while internally storing the data on multiple fileservers. There are two schools of thought regarding the interface that a parallel filesystem should provide. A widespread view is that a POSIX-compliant interface should be provided, since that is what most users are familiar with. GPFS [7] and Lustre [10] are two examples of this approach. We do not share this view. POSIX consistency semantics might be appropriate for a mail server and many single clients accessing their respective mailboxes, but such semantics simply are not suitable for a parallel application. Strict POSIX compliance requires locking, increasing overhead and reducing the level of concurrency. The last thing that a petascale infrastructure should do is to trade performance for compliance to a standard irrelevant to its primary use. Further, POSIX cannot effectively describe noncontiguous I/O, which is fairly common in scientific applications. An alternative, the PVFS2 filesystem [11] does provide a POSIX-style interface to allow access using familiar UNIX tools, but it is not strictly compliant, and parallel applications are expected to take advantage of its native interface (directly or through, e.g., MPI-IO) for maximum performance and flexibility.

We point out that using to full advantage a third-party product such as PVFS2 turns out to be surprisingly difficult on a machine like BG/L, primarily because of its stripped-down, closed-source kernel. While we agree that it is a good idea to have a stripped-down kernel, we feel that (paraphrasing from Einstein) the kernel should be made as simple as possible, but no simpler. The BG/L kernel is so stripped down that it does not provide useful interfaces for shipping user data between compute nodes and I/O nodes, an extremely vital data path and one that is at the heart of operating system research for petascale platforms.

I/O nodes bridge two worlds, the realm of the compute nodes and the domain of the large parallel file system. For petascale architectures, this OS component is unique and requires special consideration and design. Lightweight compute nodes rely on the I/O nodes to handle file I/O. Should a large parallel computation want to call on a dynamically loaded library, compute nodes could generate an I/O function call storm, all of which would be simply to load the same library. The compute node and the I/O node must be carefully paired and matched. This symbiotic relationship makes the I/O node critical for petascale systems. Our current algorithms and techniques for collective operating system calls and impedance matching the I/O capabilities of the compute nodes to the parallel I/O system are primitive at best. Current experience suggests that even the simplest of I/O operations, performed by all compute nodes, can cause tremendous bottlenecks.

We anticipate that much research and development will be required to prevent I/O nodes and their paired compute nodes from creating what will essentially be denial-of-service attacks on the parallel file system. New caching strategies, collective I/O calls, and automatic I/O reductions and broadcasts must be added to I/O nodes for petascale machines to achieve scalability.

5. FAULT TOLERANCE

Leadership-class platforms are complex hierarchical systems. Machines such as IBM's Blue Gene and Cray's XT3 [8] are designed and built with impressive hardware fault detection and recovery mechanisms. If a chip overheats, it is automatically turned off. If a network link gets CRC errors, an alert is raised. A redundant power supply can automatically switch on when another fails. At extreme scale, however, two problems need special attention. First, not every component in these petascale architectures has sophisticated error detection or correction hardware. While memory often has additional fault detection, caches within the CPU as well as some data paths do not. Given common rates for modern CPUs of around one soft error per 25 years [5], a petascale machine with 72,000 cores might be expected to fail every 3.5 hours. Indeed, many of the largest supercomputers have endured significant problems associated with soft errors. Second, most large parallel applications have no real fault strategy: when one processor dies, the entire job aborts. Within the system software, operating system, middleware, and user code, very little work has been done to dynamically respond to fault detection for HPC systems. Petascale machines must address faults throughout the system—not only inside the hardware, but from the OS software all the way up to the application. We believe that two areas need enhanced capabilities within extreme-scale architectures: self detection/correction and virtual processors.

It is relatively easy for hardware, when faulty, to signal the OS. A fan that rotates too slowly, a checksum error on a link, or a disk timeout all trigger corrective actions within the OS. However, given the sheer number of unprotected registers and caches in a petascale system, the system must become smarter. Soft errors, or "upsets" that are an integral part of our current chip technology cannot cause catastrophic failures. The system must detect errors even when no hardware faults are signaled.

To do this, we need to adjust the OS strategy. On the Internet, buffer overflow attacks have finally caused programmers to actually check array bounds and buffer overrun conditions. Likewise, the crash of one component in a browser, such as the Acrobat Reader or the Flash Player, should not cause the entire browser—or worse yet, the entire machine—to falter. For petascale operating systems, we must take a similar tack and enhance our programming styles. Every single error code must be checked. Correctness validations must be built into OS routines. Additional checksums and state information must be saved. Much like a journaling file system, the OS must be able to recover from lost processes or subroutines. The OS must detect soft errors—not via a hardware interrupt, but from the internal consistency checks in the kernel. Why is such error checking so critical? Without more careful error checking in the OS, the application could continue for hours, or pos-

sibly days, with corrupt data. Sadly, many applications rarely check for internal data corruption. The OS could identify problems and raise red flags for the application, potentially saving many thousands of CPU hours. Petascale operating systems must carefully add internal consistency checks to provide a productive work environment to scientists.

Also important is how the petascale OS coordinates its fault response with other parts of the system. The most common and robust method for providing fault tolerance in scientific applications is the checkpoint/restart (CPR). Either user-initiated or system-initiated, a checkpoint writes out the important data required to restart the computation, either to additional memory on a different node or to a disk. User-level checkpoints have the distinct advantage of knowing the minimum amount of data required for a restart. System-level checkpoints are usually brute force, saving all of the user data as well as large swaths of OS memory associated with the process or its message buffers. Unfortunately, the layers upon layers of middleware and run-time software linked to the application make hiding a CPR event very difficult. Common practice is to stop all computation, rebuild all MPI communicators, back up to the last checkpoint, and resume computation. Several global synchronizations are required.

A better strategy for petascale operating systems is to virtualize the processor and the interfaces, allowing another processor or system node to temporarily host two virtual nodes. The CPR image would be restarted on the CPU that was most idle during the previous timesteps, or simply not used at the time. Then, the application could be notified and corrective action taken. For applications with no fault handling, the application would simply run slightly slower, with warning messages sent to the user's application console. Applications with fault awareness could be asked if they want to redistribute data, reducing the workload from the two nodes sharing a physical resource by half or more. In this way, faults could be handled more gracefully, causing global communication only should the application request that behavior—MPI communicators and all other internal functions would continue normal operation. In order to accomplish this, petascale operating systems must not only support CPR but also virtualize the communication interfaces so that, should memory be available, nodes can temporarily service two OS stacks.

6. CONCLUSIONS

There are no looming showstoppers for petascale computing. There are, however, several key issues for the OS components within petaflop computers, with the most significant of these being synchronicity. While it has been argued that special-purpose OS kernels are required for petascale operating systems, we do not believe that to be the case. Addressing synchronicity is straightforward. By focusing first on trimming a kernel such as Linux down, and reducing the latency and overhead of interrupts and frequency of timer interrupts, overall effects of drifting CPUs can be reduced to a handful of microseconds. More advanced techniques, taken from real-time OSes, could reduce OS drift even more. Using a global clock to synchronize timer interrupts or using a tick-less configuration for the kernel can provide even higher levels of synchronization. For the near future, these techniques should help systems scale to petaflop levels without difficulty.

Addressing the fault tolerance and I/O needs of petascale systems will be an ongoing issue that will last years. Since the current capabilities of I/O nodes in leadership-class architectures are still primitive and software fault tolerance is practically nonexistent within the OS, significant investment in research and development will be needed to achieve high-productivity machines in the presence of faults. We are encouraged to hear that Japan has recently started a new research program to build a 10-petaflop system. Such large-scale national projects can help catalyze independent research teams and build a stable forum for development. We believe a similar program in the United States could augment the research being done at all levels, from academia to the national labs and corporations.

7. REFERENCES

[1] S. Agarwal, R. Garg, and N. K. Vishnoi. The impact of noise on the scaling of collectives: A theoretical approach. In *Proceedings of the 12th International Conference on High Performance Computing*, volume 3769 of *Springer Lecture Notes in Computer Science*, pages 280–289, Goa, India, Dec. 2005.

[2] J. J. Dongarra and G. W. Stewart. LINPACK—A package for solving linear systems. In W. R. Cowell, editor, *Sources and Development of Mathematical Software*, Prentice-Hall Series in Computational Mathematics, Cleve Moler, advisor, pages 20–48. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[3] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Phoenix, AZ, Nov. 2003.

[4] J. Makino, M. Taiji, T. Ebisuzaki, and D. Sugimoto. GRAPE-4: A one-Tflops special-purpose computer for astrophysical N-body problem. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 429–438, Nov. 1994.

[5] S. S. Mukherjee, J. Emer, and S. K. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture*, pages 243–247, San Francisco, CA, Feb. 2005.

[6] F. Petrini, D. J. 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 the ACM/IEEE Conference on Supercomputing*, Phoenix, AZ, Nov. 2003.

[7] F. B. Schmuck and R. L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies*, pages 231–244, Monterey, CA, Jan. 2002.

[8] *http://www.cray.com/products/xt3/.*

[9] *http://www.research.ibm.com/bluegene/.*

[10] *http://www.lustre.org/.*

[11] *http://www.pvfs.org/pvfs2/.*

[12] *http://www.top500.org/.*