Experiences Understanding Performance in a Commercial Scale-Out Environment

Robert W. Wisniewski[1], Reza Azimi[2], Mathieu Desnoyers[3], Maged M. Michael[1], Jose Moreira[1], Doron Shiloach[1], and Livio Soares[2]

[1] IBM T.J. Watson Research Center
[2] University of Toronto
[3] École Polytechnique de Montréal

Abstract. Clusters of loosely connected machines are becoming an important model for commercial computing. The cost/performance ratio makes these scale-out solutions an attractive platform for a class of computational needs. The work we describe in this paper focuses on understanding performance when using a scale-out environment to run commercial workloads. We describe the novel scale-out environment we configured and the workload we ran on it. We explain the unique performance challenges faced in such an environment and the tools we applied and improved for this environment to address the challenges. We present data from the tools that proved useful in optimizing performance on our system. We discuss the lessons we learned applying and modifying existing tools to a commercial scale-out environment, and offer insights into making future performance tools effective in this environment.

1 Introduction

For the past decade mainstream commercial computing has been moving from uniprocessor computing systems to multiprocessor ones. During the first phase of the commercial multiprocessor revolution, shared-memory multiprocessors (SMPs) became pervasive. SMPs of increasing size, with processors of increasing clock rate, offered ever more computing power to handle the needs of even large corporations. SMPs currently represent the mainstream of commercial computing. Companies like IBM, HP, and Sun invest heavily in building bigger and faster SMPs. These large SMPs are also called *scale-up* systems.

Computational needs, however, have continued to rise, and more recently, there has been increased interest in clusters of loosely coupled systems for commercial computing. These clusters are also called *scale-out* systems. Computer manufacturers have made it easier to deploy scale-out solutions with rack-optimized and blade servers. Scale-out has been the only viable alternative for large scale scientific computing for several years, as observed in the evolution of the TOP500 systems [3], and they are now becoming more popular for commercial computing. For the past year, we have been working on the problem of demonstrating a scale-out system with superior performance in commercial applications. In order to optimize such systems it is important to understand their performance.

A commercial scale-out (CSO) environment presents several challenges to understanding the performance of applications. We describe these in Section 2. In Section 3 we describe the tools we developed, ported, and used to understand the performance of our environment. Using these tools we obtained a better performance of our system and used that to guide a directed optimization. We present this data in Section 4. During the work of understanding performance and tailoring the tools for our commercial scale-out environment we learned valuable lessons and gained insights into what future performance tools should offer to be effective in this environment. We present these in Section 5. We present related work in Section 6 and conclude in Section 7.

2 Understanding performance in commercial scale out

Understanding performance in a commercial scale-out environment has two challenges similar to large parallel scientific environments. The first is that the large number of processing elements generate a large number of trace streams with a tremendous amount of data. It is thus important to develop automatic techniques for determining which parts of which traces should receive manual attention. For longer running applications it is important to have techniques to limit the amount of data.

The second challenge is that to correlate events from different machines a synchronized time is needed. Some parallel tool developers perform a linear shift between the first and last event for each trace. This is only sufficient for traces of smaller time windows collected on higher-end systems with more accurate clocks. In the CSO space, with commodity hardware the clocks often do not drift linearly and these techniques are not sufficient.

The two additional challenges that commercial workloads introduce are the complexity of the software stack and a large number of simultaneous threads of execution on a single processing element. The stack complexity manifests itself by requiring multiple loosely related coordinating processes. When multiple coordinating processes are executing it becomes difficult to understand the causality relationship between observed system behavior and the process or group of processes causing the behavior. As a simple example, if both a database and web server are running on a machine, and the observed behavior is that we are running short of page cache space, given our knowledge of these applications, we can probably conclude that it is the database causing the problem. However, if the characteristics of the running applications are not clearly understood in isolation, or if there is any interaction between them, it becomes much more difficult to understand particular system behavior. Related to this issue is understanding the interrelations between processes on different machines. For example, why did process X on machine 1 start process Y on machine 2.

Typical commercial workloads generate hundreds or thousands of concurrently executing threads in each machine. The threads do not necessarily all belong to a single process or logical unit of computation. It is therefore not meaningful to combine the performance of all the threads, nor is it practical to analyze them on a cases-by-case basis. In the scope of this work we made modifications to the tools described in Section 3 to handle these issues.

3 Performance tools

A major difficulty in a commercial scale-out environment is the number of different software pieces interacting together. Profiling tools like `gprof`, `oprofile`, and `nmon` show only a breakdown of the CPU time, but do not identify which resource a program is waiting for when it does not have the CPU. More information can be extracted by using an instrumentation approach.

The Linux Trace Toolkit Next Generation (LTTng) [1], is a trace-based tool that extracts information in a unified manner from all execution layers in the software stack (from hypervisor through user space) with minimal impact on the application performance and system behavior. The Linux Trace Toolkit Viewer (LTTV) uses the data output from LTTng, merges the data collected from each software stack layer, and organizes them in data structures that permit the identification of the producer of these events (which node, process, thread) and classification of the execution context in which the event occurs (process context or in which system call, trap, interrupt, softirq).

We have added features to LTTng to support commercial scale-out environments. We added PowerPC-specific instrumentation and tracing support for the Java language. This support was implemented using a JNI interface that calls a C handler which in turn calls the LTTng user space tracer to record the information. Once the information is available, it is important to be able to identify the information source. This has been made possible by adding *thread branding* events that are triggered when a new Java thread starts. The thread brand event records information about the name of the main thread function and specific thread information. The analysis and visualization tool, LTTV, has been extended to include thread brands into its representation of the system state, and allows filtering on them. To ameliorate start-up time for the many large files, we added a new precomputation module.

To complement the functionality of LTT, we designed a performance monitoring facility that provides an easy-to-use interface to the hardware Performance Monitoring Unit (PMU) of the PowerPC processors. This facility uses statistical sampling to continuously identify microprocessor bottlenecks. It has been implemented as a kernel module that performs hardware performance counter (HPC) multiplexing, does PC and data sampling, and calculates a stall breakdown model.

The key to achieve acceptable overhead during run-time monitoring is to minimize the frequency of user-kernel protection boundary crossings. In our implementation, the sampling module is fully implemented inside the Linux 2.6 kernel. As a result, except for infrequent control operations (such as initializa-

Miss Event	**Effect**	**Description**
I-cache miss	Empty reorder buffer	In-flight instructions after mispredicted branch are flushed
Branch misprediction	Empty reorder buffer	In-flight instructions after branch are flushed
Data cache miss	Retirement stops	Delay in the Load-Store-Unit (LSU) due to data cache miss.
Address translation misses	Retirement stops	Address translation (e.g., TLB) miss
LSU basic latency	Retirement stops	Delay in one of the LSUs to finish execution of an instruction
Rejections	Retirement stops	One of the units rejects an instruction – it must be reissued
FPU latency	Retirement stops	Delay in one of the FPUs to finish the computation
Integer latency	Retirement stops	Delay in one of the integer units to finish the computation
Other sources	Retirement stops	Usually results in flushing the pipeline

Table 1. Types of miss events with their potential effect in the micro-architecture function.

tion or reset), there is no interaction between the user code and the performance monitoring module.

We alleviate the problem of having limited number of physical counters by dynamically multiplexing [6] the set of hardware events counted by the HPCs using fine-grained time slices. The sampling module assigns each group of events a fraction of g cycles out of a *multiplexing round* R that is the time period in which all HPC groups have a chance to be scheduled. At the end of each HPC group's time slice, the sampling engine automatically assigns another HPC group to be counted by the hardware PMU. The value that is read from an HPC after g cycles is scaled linearly, as if that event had been counted during the entire R-cycle period. As a result, the user program is presented with N *logical* HPCs on top of n *physical* HPCs, where N can be an order of magnitude larger than n. Our earlier experimental evaluation [6] demonstrated that the statistical distance between the sampled and real rates of hardware events is small.

We use the hardware performance counters to calculate a Cycle-Per-Instruction (CPI) breakdown that attributes CPU cycles to the different hardware components that caused them. We restrict our approach to a breakdown of *stall cycles*. A stall cycle is a processor cycle in which no instruction completes (retires). When no stall occurs, the CPU throughput, in terms of IPC, is fairly close to the pipeline width and is fairly application-independent.

The key idea behind the stall breakdown model is that most bottlenecks can be detected by speculatively attributing a *source* to each stall. There are two major categories of such stalls:

– *Empty Reorder Buffer:* This implies that the front-end has not been able to feed the back-end in time. Assuming the micro-architecture is designed and tuned properly, such situations happen mostly when there is an I-Cache miss, or when a branch misprediction occurs.

– *Completion Stops:* In this case, the reorder buffer is not empty, but the oldest instruction bundle in the reorder buffer cannot retire. This happens mainly because one or more of its instructions in the bundle have not yet finished (i.e., they are waiting for an functional unit to provide the results).

We call the hardware events that can cause a stall *long-latency events*. The long-latency events we consider in this study are listed in Table 1 along with the type of stalls they cause and the potential effect they may have. By taking all sources of stalls into account, the following formula can be used to speculatively characterize the potential CPU bottlenecks at each phase in the program execution:

$$\mathrm{CPI}_{\mathrm{real}} = \sum_{i=0}^{n} \mathrm{Stall}_i + \mathrm{CPI}_C$$

where, Stall_i is the number of stalls caused by long-latency event i in the monitoring period, and CPI_C is number of *completion cycles* in each of which at least one instruction is completed. In fact, CPI_C can be used as an estimate for the CPI that can be achieved by an *ideal* hardware in which all the long latency events are removed and performance is solely determined by the program dependences and the width of the pipeline.

4 Experimental results

Our Commercial Scale-Out (CSO) environment was built using PowerPC blades to run the Nutch [8] web search application workload. The basic building block of the cluster is a BladeCenter-H (BC-H) chassis. Each BC-H chassis has 14 blade slots and is coupled with one DS4100 storage controller with a 2 Gbs Fiber Channel. Of the 8 chassis in our cluster, 4 are filled with JS21 blades. These are quad-processor (dual-socket, dual-core) PowerPC 970 blades, running at 2.5 GHz, with 8 GiB of memory each. The chassis are interconnected through two nearest-neighbor networks: a 4 Gbs Fiber Channel network and a 1 Gbs Ethernet network. Each DS4100 consists of dual redundant RAID controllers and 14 SATA drives of 400GB each.

Nutch is an open-source distributed web search application built on top of the Lucene search library [2]. The query engine consists of one or more front-ends and one or more back-ends, as shown in Figure 1. The front-ends provide a web interface for queries. Each back-end is responsible for a data partition, a subset of the complete set of documents. When a front-end receives a query, it broadcasts the query to all back-ends. Each back-end responds to the front-end with the top n (e.g., 10) documents in its data partition that match the query. The front-end collects the responses from all the back-ends to produce a single list of the top n documents with the best overall matches. For each document in the top n list, the front-end asks the corresponding back-end to return representative snippets of the document text. The front-end then responds to the requester with the query results. The Nutch architecture is similar to the Google search engine [7].

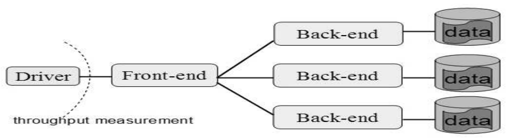


Fig. 1. Nutch distributed query environment.

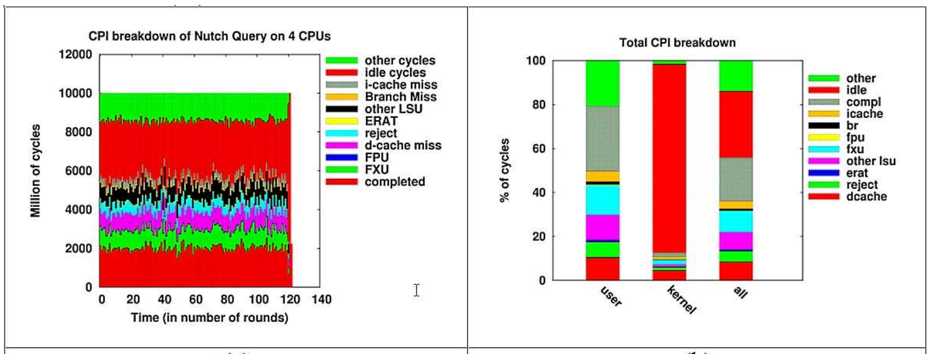
We took two approaches in understanding the performance of our application. The first was to use HPCs to understand the CPI-breakdown of our application. The second approach was to use LTT to examine and categorize time spent in different OS services.

Our measurements with HPCs show that the CPI for query fluctuates between 1.5 and 2.0 during a run, with an average of 1.7. This is far from the PowerPC 970 peak CPI of 0.2, but it is within what would be expected from previous experience with SPECcpu2000 benchmarks where we observed a CPI of 1.53. To better understand what aspects of the hardware were limiting the CPI we used the tool we described earlier to classify stalls. Figure 2(a) shows the stall breakdown with each bar representing a 1 second analysis period. In each second there are a total of 10 billion processor cycles executed by 4 processors running at 2.5GHz. Figure 2(b) shows the average over time of the data in Figure 2(a), separated for cycles in user mode, kernel mode, and total.

Instructions complete on only 20% of the cycles. Because instructions in the PowerPC 970 complete in bundles, multiple instructions can complete per cycle. The number of instructions executed per second (10 billion cycles/second + 1.7 cycles per instruction = 5.9 billion instructions/second), shows that the average bundle size is approximately 3 instructions (out of a maximum of 5). The non-stall CPI for query, computed by dividing the number of non-stall (completed) cycles by the number of instructions is 0.34. Again, this is similar to the non-stall CPI for SPECcpu2000 where we observed a CPI of 0.35.

The data shows that for a significant number of cycles (25%), the processor is idle. In principle, we should be able to reduce that idle time by increasing the load on the node. This is accomplished by increasing the number of tasks on the node. There is of course a balance, because an excessive number of threads causes a slowdown. In more extensive experiments, we found that we could keep the idle time down to between 10-15%, and are investigating how to drive this even lower.

Finally, we observe that the number of cycles wasted on the I-cache or on branch mispredictions is relatively small, stalls due to the fixed-point units account for 10% of the cycles, and stalls because of the memory hierarchy (D-cache, reject, ERAT, and other LSU) represent approximately 20% of the cycles. The

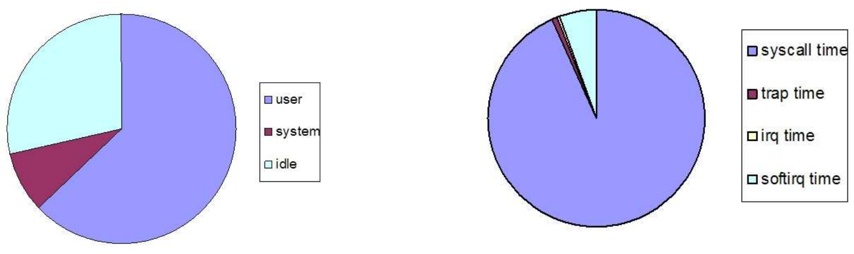


(a) (b)

Fig. 2. Hardware performance counter data

fraction of cycles wasted in the memory hierarchy is similar to the fraction of cycles doing useful work. Thus, a perfect memory system would at most double the performance of the existing processors. Alternatively, for an equivalent amount of chip area, a commensurate benefit could be obtained by doubling the number of processors and maintaining the memory hierarchy per processor.

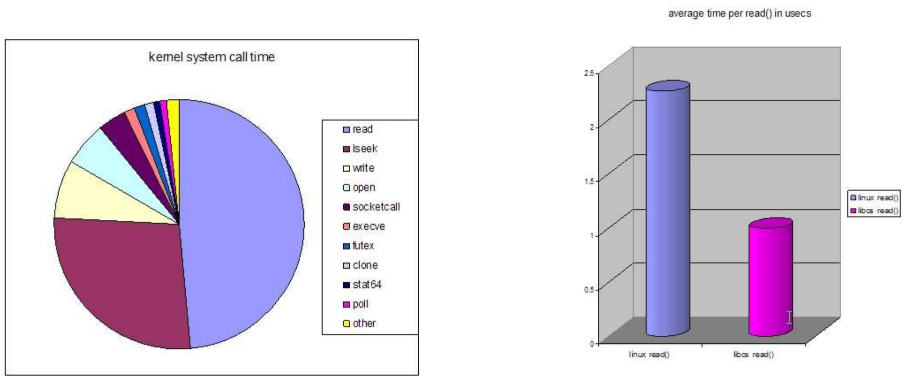
Using LTT to sample this workload at various points throughout its execution we determined that its behavior characteristics are stable over time. The following figures (Figures 3-4) are taken over a ten second snapshot during the portion of query running on a JS21 back-end blade as described earlier. Each of the data points was sampled several times and the below figures represent the median point. Figure 3(a) shows an overall breakdown of time in the system. As shown, the application (user mode) takes 70% of the CPU time, the system 20%, and 10% remains idle. One of our goals was to find bottlenecks in the system stack and to optimize for those. Thus, we examined the system closer with LTT. Figure 3(b) shows the breakdown into the major categories of system time. Figure 3(a) indicates that system call time dominates system activity for this application.



(a) (b)

Fig. 3. Time breakdown for overall system (a) and for kernel (b).

As system call time was critical to this application, we wanted to ascertain which system call was having the greatest impact on system performance. We thus used LTT to breakdown the system call time into its various components. As seen in Figure 4(a) `read()` is one of the largest components of system time. Based on the impact of `read()` time, we optimized the `read()` system call in our libOS environment. The optimization targeted for `read()` was to perform a specialized caching. We were able to achieve this because the libOS environment can be tailored to a specific application. This capability demonstrates the strength of a libOS environment [4]. Figure 4(b) shows a comparison of the modified `read()` libOS versus original Linux system. Using performance monitoring as provided by LTT, we were able to identify system calls having the significant impact on performance and target them for optimization. After optimizing `read()` we were able to improve the overall performance of our Nutch/Lucene search application.



(a) (b)

Fig. 4. Time breakdown for system calls (a) and comparison between original Linux and modified libOS (b).

5 Lessons learned and future tools requirements

The first trace data for our commercial scale-out system revealed a much larger than expected number of processes and threads simultaneously active on each processor. This was a stark contrast to scientific computing systems where on each processing element there is a single, or small number of, executing threads. We started questioning the plausibility of using a trace-based analysis for this environment.

We tried using other techniques such as log (as generated by the application of interest) analysis, and aggregated statistics such as provided by `top`. The log based approach applies just to the application. It does not help significantly with system understanding, and there is a tremendous amount of data generated by the logs. In tracking down one particular problem we could not use the logs because they perturbed the system too much and generated so much data we could not store it. There was no tool, other than `grep` and `split` (the files were too big to load into any editor) designed to process the logs.

In light of these issues, we decided to make modifications to our trace-based analysis to make it more useful for commercial scale-out. The first capability added was process branding to identify how and why a process is created. We could now identify processes that were started to create other worker JVMs upon receiving requests from the master node. This clarified one of the puzzles as to why some Java processes did nothing but wake up occasionally and go back to sleep without doing anything. We now had a good indication that it was not because they were blocked waiting for some resource, rather they just didn't have anything to do and were waiting for work. This highlights another difference in programming models from scientific to commercial computing. It is important that tools be able to recognize why a process is not computing, even when the reason occurred outside the time window over which the trace was collected. Another feature we added was the ability to only view a single process tree. This allowed us to examine threads of execution that were related to a particular higher-level computation. These few features made a meaningful difference in the understanding we could get from a trace-based analysis of a commercial workload, but still much more could be done. While there are some commercial workloads that behave similar to scientific workloads, and vice-versa, the experiences described above represent general tendencies for scientific and commercial computing.

Based on our experience, we have developed a set of recommendations for future performance tools:

– **Process branding:** An automatic way to determine how and why processes were started. For example, for a daemon that forks processes in response to requests from a master or other cooperating processes, we need the ability to track that relationship.
– **Time synchronization:** Add events into the trace generation and put analysis capability into the post-processing tools allowing timestamps collected on different machines to be adjusted on a fine granularity table-based manner. This would allow all events in a CSO system to have consistent timestamps and be displayed on a single timeline.
– **Automatic idle thread determination:** The inability to overlap I/O, poor use of synchronization primitives, etc., causes performance loss because the processor becomes idle. We need the ability to identify why the processor is not fully utilized, and in particular, to understand why a thread is unable to run at a given time.
– **Cross-machine logical causality:** Allow tools to display on a single timeline processes from one machine that have been created as a result of a process running on a different machine. This requires time synchronization and thread branding.

– **Tree-based causality:** Provide the ability to select a given process and see all processes that have been created because of actions taken by this process. Should work across machines.
– **Selective aggregation of different performance data:** Currently tools such as LTT can provide statistical information about how much time a processor spent executing particular system calls. This capability is provided on a process by process basis. What is needed is the ability to provide it on a tree-based causality basis.

6 Related Work

Barroso *et. al.* [7] describe Google's search architecture. This is similar to Nutch distributed query in partitioning data and designating web servers (front-ends) and index and document servers (back-ends). However, the former maintains separate index and document servers, while in the Nutch architecture, back-ends are responsible for both searching the index and providing the relevant snippets from the document data partitions.

The original LTT project [16] has been followed by LTTng [9]. The latter project reused work done in IBM's K42 [11, 15] on atomic tracing algorithms. The current work on with LTTng targets the Xen hypervisor and supports analysis of the interactions between the hypervisor and operating system.

Software multiplexing of HPCs is implemented for PAPI [10] at user-level using OS signals [12, 13]. Due to the large overhead of switching HPCs at user level (signal delivery plus kernel/user context switches), the granularity of multiplexing must be large which in turn results in high sampling error. DCPI [5] uses statistical sampling of the HPCs to identify system-wide hot spots and pipeline stalls at the instruction level. A major simplifying assumption in DCPI is that there is a fixed distance between the instructions that causes HPCs to overflow and the overflow exception. However, this assumption has been shown not to be realistic in modern processors with deep and wide pipelines. A simpler alternative for stall breakdown is to use fixed penalties for long-latency operations such as cache misses [14]. This approach though, significantly overestimates the actual penalties as it does not take potential overlapping of concurrent operations into account. Our approach is more accurate because it exploits existing hardware support to measure the stalls that actually occur.

7 Conclusions

In this paper we described the challenges a commercial scale-out environment poses for performance understanding, including a complex execution stack with multiple loosely related processes executing concurrently and the requirement for a large number of threads of execution per single processing element. We used LTTng and hardware performance counters to investigate performance in such an environment. This process was enlightening in terms of what the needs are for performance tools in the commercial scale-out space. We shared the lessons we learned as part of going through this process and then provided a set of recommendations for future performance tools for commercial scale-out environments.

References

1. Lttng home web page. http://ltt.polymtl.ca.
2. The lucene home page. http://lucene.apache.org/.
3. Top 500 supercomputer sites. http://www.top500.org/.
4. Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Byran Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: A library operating system for a jvm in a virtualized execution environment. In *VEE (Virtual Execution Environments)*, San Diego CA, June 13-15 2007.
5. J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandervoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium of Operating Systems Principles (SOSP)*, Saint-Malo, France, Oct. 1997.
6. Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS International Conference on Supercomputing*, Cambridge, Massachusetts, June 2005.
7. Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, March/April 2003.
8. Mike Cafarella and Doug Cutting. Building Nutch: Open source search. *j-QUEUE*, 2(2):54–61, April 2004.
9. Mathieu Desnoyers and Michel R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium) 2006*, July 2006.
10. J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *Proceedings of Workshop Parallel and Distributed Systems: Testing and Debugging (PATDAT), joint with the 19th Intl. Parallel and Distributed Processing Symposium (IPDPS)*, Niece, France, Apr. 2003.
11. The K42 operating system, http://www.research.ibm.com/k42/.
12. Wiplove Mathur and Jeanine Cook. Improved estimation for software multiplexing of performance counters. In *Proceedings of the 13th Intl. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Atlanta, GA, Sept. 2005.
13. John M. May. MPX: Software for multiplexing hardware performance counters in multithreaded systems. In *Proceedings of the Intl. Parallel and Distributed Processing Symposium (IPDPS)*, San Francisco, CA, Apr. 2001.
14. Harvey J. Wassermann, Olaf M. Lubeck, Yong Luo, and Federico Bassetti. Performance evaluation of the SGI Origin2000: a memory-centric characterization of lanl asci applications. In *Proceedings of ACM/IEEE Conference on Supercomputing (SC)*, San Jose, CA, November 1997.
15. Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing*, Phoenix Arizona, November 17-21 2003.
16. Karim Yaghmour and Michel R. Dagenais. The linux trace toolkit. *Linux Journal*, May 2000.