Preprint
UCRL-JC-146246

# Memory Benchmarks for SMP-Based High Performance Parallel Computers

*A.B. Yoo, B. R. de Supinski, F. Mueller, S.A. McKee*

This article was submitted to
The 29th International Symposium on Computer Architecture,
Anchorage, Alaska, May 25-29, 2002

**November 20, 2001**

*U.S. Department of Energy*
Lawrence
Livermore
National
Laboratory

DISCLAIMER

# Memory Benchmarks for SMP-based High Performance Parallel Computers *

Andy B. Yoo[1], Bronis R. de Supinski[1], Frank Mueller[2] and Sally A. McKee[3]

[1]Lawrence Livermore National Laboratory
Livermore, CA 94551
e-mail: {ayoo | bronis}@llnl.gov

[2]Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695
e-mail: mueller@cs.ncsu.edu

[2]School of Computing
University of Utah
Salt Lake City, UT 84112
sam@cs.utah.edu

## Abstract

*As the speed gap between CPU and main memory continues to grow, memory accesses increasingly dominate the performance of many applications. The problem is particularly acute for symmetric multiprocessor (SMP) systems, where the shared memory may be accessed concurrently by a group of threads running on separate CPUs. Unfortunately, several key issues governing memory system performance in current systems are not well understood. Complex interactions between the levels of the memory hierarchy, buses or switches, DRAM back-ends, system software, and application access patterns can make it difficult to pinpoint bottlenecks and determine appropriate optimizations, and the situation is even more complex for SMP systems. To partially address this problem, we have formulated a set of multi-threaded microbenchmarks for characterizing and measuring the performance of the underlying memory system in SMP-based high-performance computers. We report our use of these microbenchmarks on two important SMP-based machines.*

*This paper has four primary contributions. First, we introduce a microbechmark suite to systematically assess and compare the performance of different levels in SMP memory hierarchies. Second, we present a new tool based on hardware performance monitors to determine a wide array of memory system characteristics, such as cache sizes, quickly and easily; by using this tool, memory performance studies can be targeted to the full spectrum of performance regimes with many fewer data points than is otherwise required. Third, we present experimental results indicating that the performance of applications with large memory footprints remains largely constrained by memory. Fourth, we demonstrate that thread-level parallelism further degrades memory performance, even for the latest SMPs with hardware prefetching and switch-based memory interconnects.*

## 1 Introduction

The speed gap between CPU and main memory is already large, and it continues to grow: CPU speeds double about every 18 months, whereas main memory speeds double about every ten years. Worse, evidence indicates that this trend will

*This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

continue. This implies that the time to access the memory system will continue to dominate the performance of many applications. This is especially true in the realm of high-performance, parallel computers, where we have observed CPU utilization rates as low as about 5% — precisely because the memory system cannot provide data as fast as the application needs it. Using the memory efficiently is therefore crucial to achieving good system utilization. Unfortunately, several key issues governing memory system performance in current systems are not well understood. For instance, as user applications execute, they exhibit a variety of memory access patterns, including non-unit strided accesses and indirect accesses through pointers or indirection vectors. Traditional summary measures like cache hit rates can be useful, but alone they are insufficient for explaining *how* an application's behavior generates those statistics. Obtaining more accurate and meaningful measurements and projections of a memory system's performance requires understanding how that memory system behaves under a given set of access patterns.

In the symmetric multiprocessors (SMPs) that have become the dominant components of high performance computer (HPC) systems, several CPUs can access a shared main memory simultaneously. Applications running on HPCs are usually parallelized to take advantage of the SMPs' concurrent memory access capability. How well the memory system supports the concurrent accesses varies widely — for instance, bus-based systems offer little support for concurrent accesses to main memory, while switch-based systems claim to eliminate this problem. Unfortunately, no existing memory benchmarks specifically address measuring the effect of concurrent accesses on SMP memory system performance.

Here we report on a suite of microbenchmarks that we developed to measure the memory performance of SMP-based machines. To test the effect of concurrent memory accesses on memory performance, each benchmark is multi-threaded. Furthermore, the benchmarks are parameterized to perform either fixed or randomly selected, variably strided accesses, allowing them to be configured to emulate the varying access patterns of HPC parallel applications.

Portability was an important design goal, and thus we implemented the suite using only the standard libraries, commonly available on many systems. To the best of our knowledge, these are the first benchmarks specifically targeted for properly measuring the memory performance of SMP-based machines running parallel applications. In addition, we complement the benchmarks with a tool that can be used to investigate the characteristics of the underlying memory system, such as sizes, line lengths, and associativities of caches and TLBs. Since these characteristics directly determine the performance regimes of a machine's memory hierarchy, this tool can greatly reduce the time required to measure the full spectrum of memory system performance.

We have used our benchmarks to measure the memory performance of *a machine we'll call* **blue** *for purposes of blind review* and *a machine we'll call* **snow**, two SMP-based HPCs in operation at *location and real machine names omitted for blind review*. We find that the performance of applications with large memory footprints remains largely constrained by memory. Furthermore, our analysis indicates that concurrent accesses significantly decrease per-thread memory bandwidth, even for memory subsystems specifically designed to support such concurrent workloads for these applications. For example, on **blue**, we find that memory bandwidth drops by 30% to 40% as the number of threads increases from one to four.

The rest of the paper presents our four main contributions as follows. The details of our first two contributions, our tool to determine memory system characteristics, and our new microbenchmark suite to assess and to compare the performance of different levels in SMP memory hierarchies systematically and to determine key parameters, are described in Section 2. We then present a short study of the memory system characteristics of **blue** in Section 3. Results from our memory performance study on SMP-based HPCs at *location name omitted for blind review* are analyzed in Section 4. In Section 5, we then briefly discuss existing microbenchmark suites, and we provide concluding remarks and directions for future study in Section 6.

## 2 Benchmarks

Just as in uniprocessor systems, the performance of SMP machines — the building blocks for large, scalable, parallel computers — is limited by their memory performance. Existing memory benchmarks for uniprocessor systems are not suitable for SMP machines, since they fail to capture the effect of concurrent memory accesses exhibited by parallel applications. Furthermore, the underlying memory system architecture of SMP machines is different from (and usually more sophisticated than) that of uniprocessor machines. Nonetheless, even these advanced memory system designs do not work well for all applications. That is, applications exhibiting certain memory access patterns benefit the most from the more sophisticated memory systems of SMP machines.

Our benchmarks are multi-threaded to accurately measure the memory system performance when the memory is accessed concurrently by a parallel application. The number of threads can be set by the user. In addition, the benchmarks allow the user to specify different memory access patterns for these threads. We also provide a separate tool to obtain information about the memory system parameters, such as cache sizes and associativities, that interact with access patterns to greatly affect memory system performance.

### 2.1 Memory Characterization Benchmarks

Memory system performance is largely affected by the characteristics of the underlying memory system architecture. Therefore, having accurate information about the memory system characteristics can significantly reduce the effort required to capture the spectrum of a machine's memory system performance. Knowledge of the memory system parameters is essential to fully understanding the performance variations applications exhibit for different configurations (such as from varying the number of threads or problem or input data-set size). All the relevant information about data and instruction cache sizes, cache line length, TLB size, and so forth, is not always readily available for the HPCs we target. Many are custom-built, and therefore differ from their high-commodity cousins in both their composition and the availability of documentation. Although some of the memory system parameters can often be inferred using existing memory performance microbenchmarks, the process is usually time-consuming and worse, can give inaccurate information. Furthermore, many memory system characteristics are difficult or impossible to determine using these inference methods. In any event, the inference methods invert

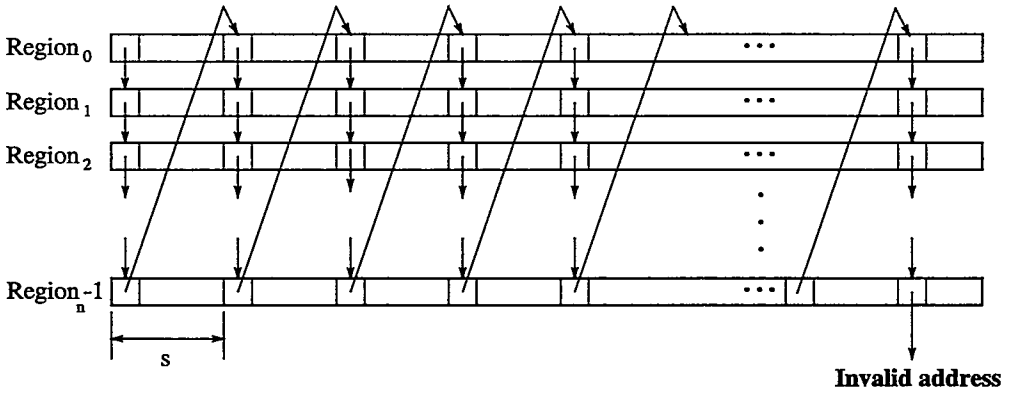



**Figure 1. A snapshot of the accesses for $a$ memory regions with stride $s$.**

the goal of using memory system characteristics to measure the spectrum of memory system performance quickly. For this reason, we have developed a tool specifically designed to quickly determine memory system characteristics, which then can be used to guide memory performance studies. Here we describe the composition of our tool that determines memory system characteristics. Section 3 walks the reader through an example use of our tool to deduce memory system parameters.

We have used the *performance application programming interface* (PAPI) [12] in developing this tool. PAPI provides a standard application programming interface (API) for accessing hardware performance monitors available on most modern microprocessors. These monitors are registers that track events, which are defined as specific signals or states related to the processor's function. By monitoring these events, we can facilitate the correlation between the structure of an application code and the efficiency of mapping that code to the underlying architecture. The events monitored by our tool include L1 and L2 data/instruction cache misses and TLB misses.

Our tool uses multiple executions of a simple test program to characterize fully some aspect of the memory system. Each execution measures the relevant event for that memory system aspect, *e.g.*, L1 cache misses or TLB misses. The simple test program has three parameters: memory region size (region_size), number of memory regions (region_count) and stride. The region size and the stride are specified in units of the size of an address on the machine.

The program performs a series of accesses through the set of memory regions. Accesses are interleaved across the memory regions and the distance between accesses within a memory region is the stride. Figure 1 illustrates the series of accesses performed by the program. Note that the accesses reduce to strided accesses when a single region is used.

The tool performs the series of accesses once (or more) to warm up the cache or load the TLBs. It then initializes and starts the performance monitors, performs the accesses again and then stops the performance monitors. We note that many memory system characteristics can be accurately determined by performing the accesses a single time; however, some characteristics can be obscured by accesses performed in the performance monitor routines themselves. As we discuss in Section 3, we compensate for this pollution effect by performing the accesses multiple times in these cases. Pseudocode for the entire program used by our tool is shown in Figure 2.

```
int region_size;   /* region size in pointers */
int region_count; /* number of regions */
char **memory;  /* an array of pointers */
int stride; /* a distance between two linked elements in memory */

memory = (char **) malloc (region_size*region_count*sizeof(char *));
naccesses = region_size/stride;
for(i=0;i<naccesses;i++)
   for(k=0;k<region_count;k++){
        p= (char **) &(memory[region_size*k+i*stride]);
        if(k+1 != assoc)
           next = &memory[region_size*(k+1)+i*stride];
        else
           next = &memory[(i+1)*stride];
        *p = next;
   }
memory[region_size*assoc+naccesses*stride] = -1;

p= (char **) &(memory[0]); while (p != -1) p=(char **) *p; /*cache warmup */

p= (char **) &(memory[0]);
Start performance monitors.
while (p != -1) p=(char **) *p;
Stop performance monitors.
```

**Figure 2. Pseudocode for simple test program.**

## 2.2 Parallel Memory Performance Benchmarks

We set out to design a set of programs to study the effects of parallel applications' concurrent memory accesses on memory system performance. An application can be parallelized using several different methods, alone or in combination. For instance, the application could be multi-threaded, either via OpenMP [14] or other compiler directives, or via a low-level threading library such as Pthreads [13, 17]. Alternatively, the application could use process-level parallelization with message passing communication, such as MPI [15], or interprocess communication through Unix shared memory. Our suite of microbenchmarks can test memory performance in the presence of deterministic concurrency (and thus memory contention) for applications that use either of the threading options or process-level parallelization via Unix shared memory. Note that the communication mechanism used in the process-level parallelization mechanism is not relevant for our microbenchmarks, since the operations do not involve interprocess (or inter-thread) communication. Our initial implementation uses OpenMP due to the ease of implementation provided by its directives. The results presented in Section 4 are from this version.

Although OpenMP is a standard set of compiler directives, implementations are not yet available on many platforms. In addition, many OpenMP implementations automatically use a significant level of optimization, which often requires long compilation times and complicates verification that the desired memory access pattern is generated. For this reason, we also ported our suite to Pthreads [13, 17]. Results with this implementation are consistent with those presented in Section 4. In

general, the synchronization overhead (described below) is slightly lower with this version, and so the measured performance is slightly higher when the same level of optimization is used.

In our multi-threaded implementations, each thread of a benchmark program is assigned a memory region, and the thread generates a given access pattern within that region. The memory accesses emulate various access patterns exhibited by parallel applications: the stride can be fixed, or a user can opt to use irregular strides computed by a random number generator. This facilitates modeling the behavior of typical multi-threaded parallel applications, where the threads perform similar accesses but on different memory regions.

Our benchmarks are derived from the portion of hbench:OS [5] that measures the performance of memory operations that read, write, read/write, copy, or clear memory locations. As in hbench:OS, our suite includes tests that measure the memory bandwidth and latency of these operations using the standard Unix wall-clock timer. Using the commonly available wall-clock timer, along with standard threading techniques like OpenMP and Pthreads, makes our benchmarks portable.

We extend the capabilities of hbench:OS in four ways, even for measuring uniprocessor memory system performance. First, we add support for strided access patterns to all of the bandwidth tests. Second, we add support for a random walk access pattern to the memory read latency test and the bandwidth tests. Third, we are in the process of adding support to allow the access patterns to be read in from compact address traces that can be generated directly from real applications [1]. This advanced access pattern support is important for determining the expected performance of a memory system for the real applications. For example, hardware prefetch mechanisms are generally limited to unit-stride access patterns (with respect to cache lines) [1]. Accurate testing of the memory system should demonstrate that applications with this access pattern will achieve significantly higher performance than those without (*e.g.*, applications that use indirection). Previous memory microbenchmarks fail to capture this property.

Finally, in addition to supporting a wide-range of access patterns, we have modified hbench:OS to perform measurements at the full range of access granularities. We achieve this by implementing a macro generator, a separate program that produces the actual code to perform the memory accesses such that the memory operations performed by the tests are encapsulated in a C preprocessor macro. This choice limits overhead while allowing the code that performs the accesses to reflect key parameters, such as the amount of data accessed (the region size parameter from hbench:OS), the stride between accesses, or the number of threads. This solution allows us to use straight-line code for moderate region sizes. Alternatively, our tests loop over a large number of memory operations for large region sizes, similarly to the single macro used by hbench:OS. This avoids negative performance effects from instruction cache misses. Further, we can increase the number of accesses per iteration when the region size is small; this supports accurate measurements for regions as small as a single integer. This capability allows us to perform inference studies on cache associativity, as discussed in Section 2.1. Furthermore, our macro generator allows us to optimize the implementation of our synchronization operations based on the number of threads in the Pthreads version.

1. Initialization. Set the number of threads and stride.
2. *iterations* = ONE_SECOND_WORTH /* determined previously */
3. do_mem_op(*iterations*)
4. Calculate the performance metric.

**Figure 3. Description of the benchmark programs.**

1. Create $n$ threads.
2. Set a thread variable to point to the beginning of a memory region assigned to the thread.
3. first synchronization operation.
4. start_time = current time.
5. second synchronization operation.
6. Repeat for $i$ times, where $i$ is the number of iterations requested by main function.
   perform a memory operation.
7. third synchronization operation.
8. end_time = current time.
9. fourth synchronization operation.
10. return (end_time - start_time).

**Figure 4. Description of `do_mem_op` functions.**

The basic procedure for gathering a data point is to run the macro generator for the parameters of interest, compile the generated test application, and then run the resulting test. All of this is easily encapsulated in shell scripts; for detailed studies, a single script can capture the procedure for multiple data points. Thus, we overcome the hbench:OS restriction to region sizes that are of multiples of 200 integers without reducing the usability of our suite.

Figure 3 shows the structure of the benchmarks, each instance of which contains a separate function that performs the memory operations being tested. These functions, specified as `do_mem_op` in Figure 3, have a single parameter that controls the number of iterations for which the given memory operations are executed. The structure of our `do_mem_op` functions, shown in Figure 4, is again similar to the mechanism used in hbench:OS. The key differences are that our version of each function is executed by multiple threads, and the timing operations are synchronized across these threads. In our OpenMP implementation, the threads are created simply by making the entire body of the function a parallel region; in the Pthreads version, the main thread creates $(n - 1)$ threads, and they all execute the `do_mem_op` function.

The synchronization operations around the timing calls ensure that the memory operations are performed concurrently by all threads. The first synchronization operation guarantees that the threads are ready to perform their memory operations when the timer is started, while the second guarantees that the operations that follow are performed after the timer is started. Similarly, the third synchronization ensures that the timer is not stopped until all of the threads have performed their memory operations. The fourth synchronization minimizes interference from any threads that complete their operations early. In the OpenMP implementation, all of the synchronization operations are essentially barriers (the second and fourth use the implicit

| Stride | 1 | 2 | 4 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|
| Cache Hit Ratio | 0.87 | 0.75 | 0.5 | 0.12 | 0.0 | 0.0 |

**Table 1. Determining Line Size.**

| Region Size (Bytes) | 1024 | 2048 | 4096 | 8192 | 16384 |
|---|---|---|---|---|---|
| Cache Hit Ratio | 0.82 | 0.87 | 0.87 | 0.34 | 0.0 |

**Table 2. Estimating Cache Size.**

barrier at the end of the OpenMP *single* construct). In the Pthreads implementation, the timer operations are performed in the middle of a barrier implemented with condition variables, which results in slightly less overhead.

In both our implementations, the overhead cost relative to the cost of the memory operations is low, since we ensure that the iteration parameter is large enough that the timing spans approximately one second total wall-clock time (much like in hbench:OS). Results for our benchmarks with a single thread are indistinguishable from those for running the corresponding hbench:OS (uniprocessor) benchmarks. Further, results for multiple threads when the region being accessed fits into the L1 cache are consistent with our single-thread results. Based on these results, we conclude that the synchronization overhead is not statistically significant.

## 3 Memory System Characterization

Many of the high-performance computers we target are custom-built, unique machines. For these one-of-a-kind systems, detailed information about data and instruction cache sizes, cache line length, TLB size, and so forth, may be difficult to obtain (essentially, one may have to track down the people who built that machine). Given this, we wanted a straightforward methodology for extracting the memory system parameters from the results of a set of microbenchmark performance data. Section 2.1 described the general operation of the benchmarks we designed to address this information gap, and here we describe an example in which we determine the details of the L1 caches on *company name omitted for blind review node type omitted for blind review* (*i.e. CPU type omitted for blind review* processors). We also discuss how to extend the procedure directly to capture information about other aspects of the memory system, such as the number of TLB entries.

As discussed in Section 2.1, our method uses performance monitors; the relevant event for determining L1 cache characteristics is L1 cache misses. In order to normalize across the parameters of the tool (in particular, region_size and region_count), our tool estimates the number of addresses accessed as the naccesses variable shown in Figure 2 and reports the L1 cache hit ratio as the monitored count divided by this value. Similarly, one can estimate TLB hit ratios by monitoring TLB misses.

We first deduce the cache line size (in TLB studies, we determine the page size, *e.g.*, the amount of data mapped by a single TLB entry). Our tool uses a single large region for this purpose; in our results, we use 1 MB for region_size. The key feature of this region size is that we are confident a priori that it exceeds the size of the L1 cache (for TLB studies, the region

| Number of Regions | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Cache Hit Ratio | 0.96 | 0.7 | 0.49 | 0.49 | 0.0 |

**Table 3. Determining Associativity.**

| Region Size (Bytes) | 1024 | 2048 | 2559 | 2560 | 2561 |
|---|---|---|---|---|---|
| Cache Hit Ratio | 0.99 | 0.93 | 0.0097 | 0.0 | 0.0 |

**Table 4. Determining Cache Size.**

size must exceed the size of memory mapped by the TLBs). As shown in Table 1, as we increase the stride parameter, our cache hit ratio estimate declines to zero at stride eight. From this, we are able to deduce correctly that the L1 cache line is 32 bytes (*i.e.* stride eight * four bytes per address). What happens is that for smaller strides, we access more than one entry per cache line, thus getting cache hits for those accesses (specifically, the number of hits per cache line is the difference between the line size and the stride).[1]

Next, we obtain a rough estimate of the L1 cache size (for TLB studies, of the amount of data mapped by the TLBs). In this experiment, we use a stride equal to the line size discovered as described above and again use a single region. We then sample power of two region sizes and observe the cache hit ratios. As shown in Table 2, we see that the cache hit ratio is zero when the region size is 16384 (*i.e.* 64 Kbytes) and dramatically decreases when the region size is increased from 16 Kbytes to 32 Kbytes. Thus, we can safely deduce that the L1 cache size is at least 16Kbytes, or 4K address values and that studies that use a total memory size (*i.e.* region size times number of regions) of this or less will fit into the L1 cache.

We can now deduce the L1 (or TLB) associativity. In this experiment, we use the region size determined above; this ensures that the memory region fits in cache, thus eliminating capacity misses. We set the stride to half of the actual cache line size and observe the cache hit ratios as we vary the number of regions. When the number of regions exceeds the cache associativity, accessing one of the sets in the cache generates more misses because the access pattern is causing additional cache conflicts. As we see in Table 3, the cache hit ration drops to zero when the number of memory regions is increased from four to five, from which we correctly deduce that the L1 cache of the *CPU type omitted for blind review* is four-way associative.

We can now determine the cache size precisely. In this experiment, we use a stride equal to the cache line size determined in the first experiment (*i.e.* eight) and set the number of memory regions to the associativity that we just determined (*i.e.* four). We then vary the region size. The results from our experiment are shown in Table 4; note that we actually repeat the monitored accesses ten times in this experiment in order to compensate for the cache pollution effects of the performance monitor accesses. Under this experiment, it is clear that when the region size is one-fourth (*i.e.* the inverse of the number of

[1]We assume the absence of hardware prefetching for this experiment. In the presence of hardware prefetching, non-uniform strides would have to be used, but the experiment would not change significantly.

regions) of the cache size or less, all of the regions will fit into cache. As we increase the region size, some of the accesses will suffer capacity misses; when total amount of memory accessed (the region size times the number of regions, *i.e.* four) exceeds the cache size by exactly one-fourth (again, the inverse of the associativity), then all of the accesses will suffer capacity misses.[2] From Table 4, we deduce that the 10240 (2560*4) address values, or 40 Kbytes is 25% more than the cache size. Thus, we correctly determine that the L1 cache of the *CPU type omitted for blind review* is exactly 32 Kbytes. From this, the L1 line size of 32 bytes and the four-way associativity, we can calculate that the L1 cache has exactly 1024 sets. Similarly, we can calculate the number of TLB sets in TLB studies (we note that the product of the sets and associativity, *i.e.* TLB entries, is the characteristic of greatest interest).

Some features cannot be determined this way due to the lack of an appropriate event. For example the *CPU type omitted for blind review* does not support the monitoring of L2 cache misses. In these cases, we must either determine a closely related event to serve as a proxy or we must resort to the inference techniques based on performance. We have also designed methods to infer the same range of memory system characteristics based on performance measurements [2].

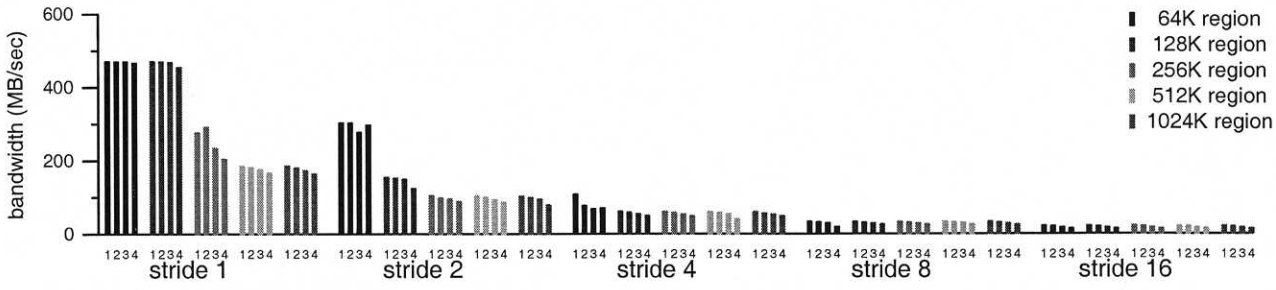
## 4 Case Study: Memory Performance Measurements

We have measured the memory system performance of two SMP machines at *location omitted for blind review*, **blue** and **snow**, using our benchmarks and report the results in this section. Although both of these machines are produced by *company name omitted for blind review*, they have several significant differences. **blue** has *CPU name omitted for blind review*, which are CISC-based and have relatively small L1 and L2 caches. **Snow** has *CPU name omitted for blind review*, which are RISC-based, include unit-stride hardware prefetching, have 128-way set associative L1 caches and significantly larger L2 caches. In addition, the rest of the memory systems of these machines highlight the contrast in design options available in SMPs – the nodes used in **blue** have a simple bus-based memory system while the nodes in **snow** connect the CPUs to main memory with a crossbar switch. Both use *CPU family name omitted for blind review* family processors, which allow us to use the same compiler to generate executables for fair performance comparison, and yet have different memory system architectures.

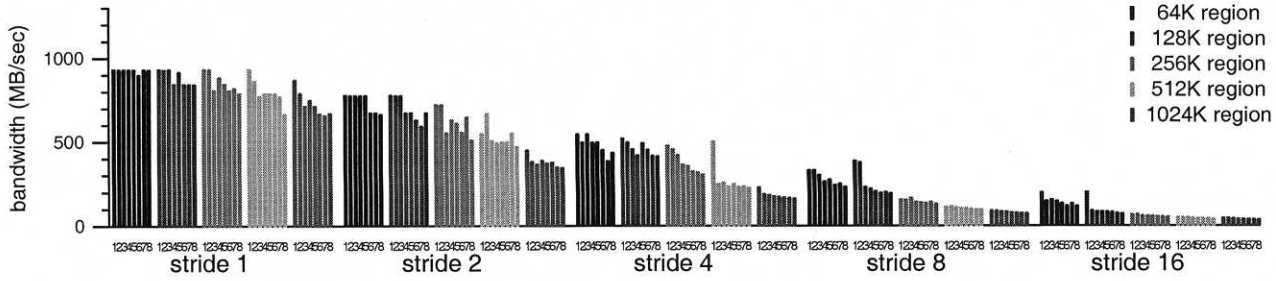
The results are shown in Figure 5 to Figure 9. We show only the results for test cases with large memory footprints in the figures because the memory accesses to a region with small memory footprints show little variation in performance regardless of parallelism. Reader should refer to [2] for more information.

Figure 5 plots the memory bandwidth measurements for read accesses. Obviously, those read operations with larger strides show poor performance compared to the ones with equal memory footprints (larger memory regions) but smaller strides due to lower cache utilization. Furthermore, higher memory accesses caused by cache misses increase the bus/switch contention by threads. There is some variation in the bandwidths of test cases whose memory footprints fit into the L2 cache, as we increase the number of threads (64 KB region with stride of two for **blue** and 128K region with stride of one for **snow**,

[2]We assume an LRU or pseudo-LRU cache replacement policy in this analysis. With random replacement or most recently used, the required property does not hold. We are investigating methods that work with any cache replacement policy.




(a) **blue**.

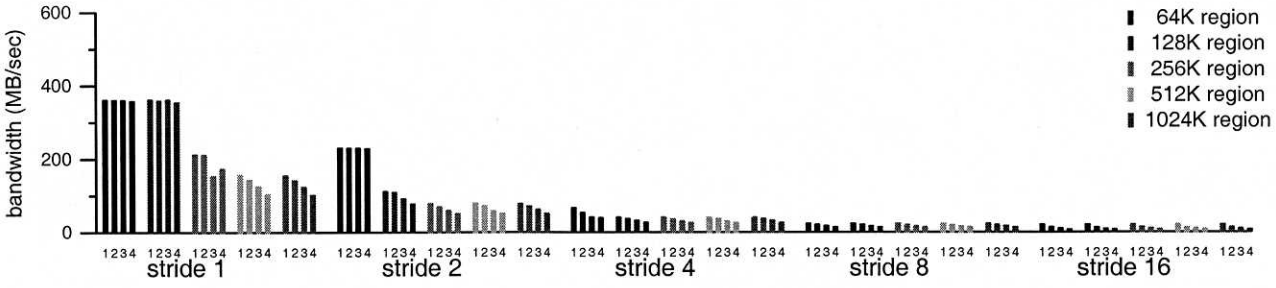



(b) **snow**.

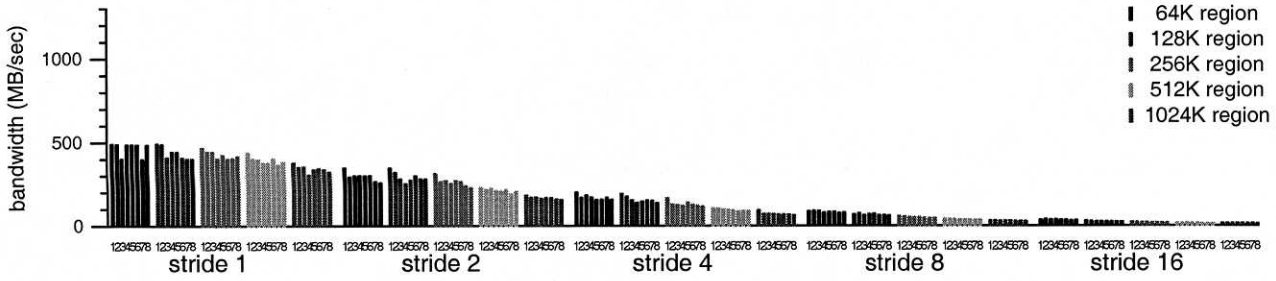
**Figure 5. Memory read bandwidths.**

for instance). The variance in these cases occur because the L2 caches in both machines are of *unified* type (*i.e.* shared between instructions and data). **Snow** exhibits far better performance than **blue** regardless of memory footprint sizes due to its advanced memory system architecture. Comparing the results for cases with equal memory footprint sizes on **blue** and **snow** reveals that the memory bandwidths on **blue** for these cases can be as low as 30% of those on **snow**. The number of threads that perform the memory read operations has a significant effect on memory bandwidth. Although it is not clearly visible in the graphs due to scaling, an analysis of the data reveals that memory bandwidth can drop as much as 40% for **blue** when four threads perform the read operations, and 30% for **snow** when eight threads perform the operations, compared to a single thread. Thus, we see that although the **snow** with its switch-based memory architecture can reduce memory contention and, thus, improve performance, the effect of concurrent accesses is still significant.

The results for the memory write bandwidth test cases are shown in Figure 6. The memory performance for write operations is similar to what we have observed in the memory read experiments: for the cases with large memory footprints, the memory access pattern has significant effect on memory performance, and **snow** exhibits better memory performance than **blue**. The memory bandwidth can drop as much as 50% for **blue** and 30% for **snow** for large memory footprints as we increase the stride. However, the results also show that the memory write operations are affected more by the number of threads, compared to the memory read operations, for which the size of memory footprints and the memory access pattern have more significant effect on memory performance. As the graphs show, the memory bandwidth decreases as the number of threads increases. The effect is more evident for smaller memory footprints with shorter strides [2]. This is due to *false sharing*. The data shown in Fig. 6 is collected without using any padding areas between memory regions; since the regions




(a) **blue**.




(b) **snow**.

**Figure 6. Memory write bandwidths.**


(a) **blue**. (b) **snow**.

**Figure 7. Byte operation bandwidths.**

are larger than a single cache line, we did not expect to observe any false sharing effects. However, our benchmarks provide an option to place paddings between memory regions. We present the data collected using the padding areas in Table 5 for comparison. Only the results for the stride of one and two are reported here due to space limitations. In these results, we see that the write bandwidth variations with threads access regions spaced by at least 4K bytes (the page size). From these results, we conclude that significant aspects of cache coherence are managed at a memory page granularity, contrary to our expectations for a bus-based snoopy coherence system.

The memory bandwidths for the read-write benchmark, in which the content of a memory location is read and written right back to the same location, is very similar to those of the write benchmark, and hence is not reported here due to space limitations.

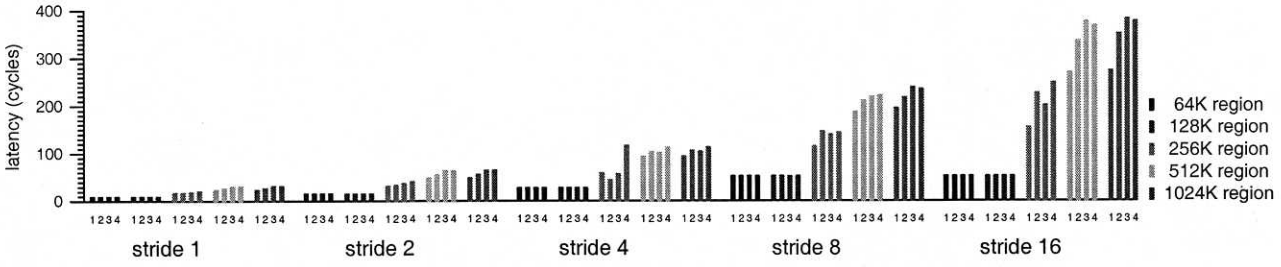
Figure 7 compares memory bandwidths of byte operations, `bzero` and `bcopy`, respectively. For the `bzero` function, **blue** shows higher bandwidths than **snow** for test cases with small memory footprints until the memory footprint exceeds the L2 size, after which **snow** achieves higher bandwidths [2]. This is probably due to the difference in the speed of CPUs;

| | | blue | | | | snow | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Stride | Region | Number of Threads | | | | Number of Threads | | | | | | | |
| Length | Size (KB) | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 1 | 1241.03 | 1253.36 | 1253.12 | 1226.42 | 844.12 | 844.99 | 845.12 | 844.54 | 844.47 | 844.47 | 844.53 | 842.53 |
| | 2 | 2508.80 | 1246.98 | 1258.25 | 1237.19 | 845.49 | 844.91 | 844.68 | 844.51 | 844.36 | 844.38 | 843.71 | 843.09 |
| | 4 | 1261.16 | 1261.23 | 1259.69 | 1251.31 | 844.68 | 844.69 | 844.57 | 844.59 | 843.49 | 843.76 | 843.92 | 840.09 |
| | 8 | 1262.54 | 1261.90 | 1261.42 | 1241.36 | 844.93 | 844.90 | 843.84 | 843.72 | 844.36 | 844.29 | 844.26 | 829.04 |
| | 16 | 1262.84 | 1262.87 | 1262.05 | 1242.49 | 845.16 | 844.55 | 844.00 | 844.42 | 844.10 | 844.74 | 843.57 | 841.56 |
| | 32 | 1243.89 | 1241.61 | 1241.12 | 1238.34 | 845.46 | 843.81 | 843.80 | 843.10 | 844.00 | 410.77 | 843.17 | 840.86 |
| 2 | 1 | 1253.93 | 1253.48 | 1252.62 | 1226.37 | 845.50 | 845.02 | 844.64 | 844.52 | 844.54 | 844.39 | 844.57 | 842.20 |
| | 2 | 1258.82 | 1257.91 | 1258.86 | 1235.00 | 845.32 | 844.32 | 844.37 | 844.93 | 844.78 | 844.32 | 843.59 | 841.67 |
| | 4 | 1261.91 | 1260.49 | 1255.95 | 1239.57 | 845.44 | 842.33 | 844.02 | 844.18 | 844.08 | 844.02 | 843.80 | 842.30 |
| | 8 | 1262.92 | 1261.97 | 1260.80 | 1242.68 | 845.42 | 844.25 | 844.32 | 844.38 | 844.44 | 843.85 | 844.27 | 841.62 |
| | 16 | 1226.96 | 1227.79 | 1226.80 | 1218.79 | 845.21 | 843.78 | 843.68 | 843.79 | 843.76 | 843.68 | 841.31 | 841.84 |
| | 32 | 229.65 | 229.41 | 229.33 | 227.69 | 346.06 | 346.10 | 275.52 | 345.61 | 265.02 | 264.71 | 292.92 | 263.94 |

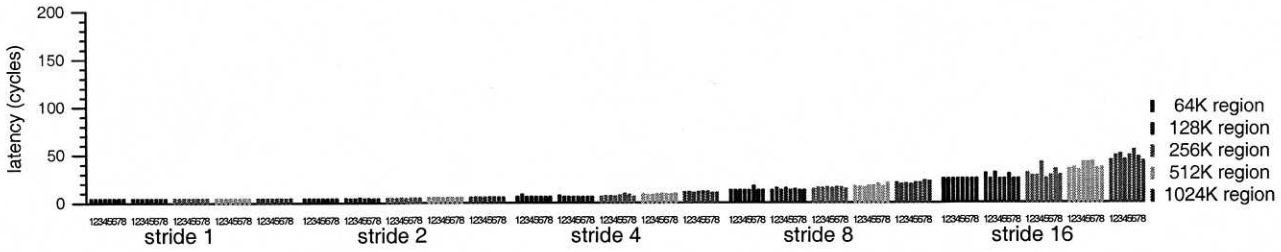
**Table 5. Memory write bandwidths with padded memory regions.**

**blue** operates on 332 MHz CPUs while **snow** has 222-MHz CPUs. That is, the `bzero` function only needs to clear locations in cache for the cases with small memory footprints and CPU speed dominates the performance in this case. This does not occur in `bcopy` tests, since the `bcopy` involves memory reads and writes. These operations can be considered to be memory read and write operations with stride of one. However, the figures indicate that the memory bandwidths achieved by these memory operations are considerably lower than those achieved by memory read and write operations. This is probably due to the way the `bzero` and `bcopy` functions are implemented. As expected, higher memory bandwidths were obtained by the `bzero` function, because the `bcopy` function also involves memory reads. For both operations, **snow** exhibits a large variance in bandwidth for the cases with large memory footprints. This is because we did not align the memory regions to measure the effect of misaligned memory regions on memory performance.

The memory read latency results for **blue** and **snow** are shown in Figure 8. For **blue**, the results are consistent with what we have observed in earlier experiments: the performance of the test cases with large memory footprints is significantly affected by the number of threads and the stride they use. For both **blue** and **snow**, the memory reads with larger strides exhibit higher memory latencies than those with smaller strides because of lower cache utilization among the test cases with the same memory footprint sizes. Regarding the effect of the number of threads, the memory latency increases as much as 40% as the number of threads increases from one to four. However, the number of threads has little effect on memory read latency on **snow** regardless of region sizes and strides, as shown in Figure 8. Comparing the results for the same region size

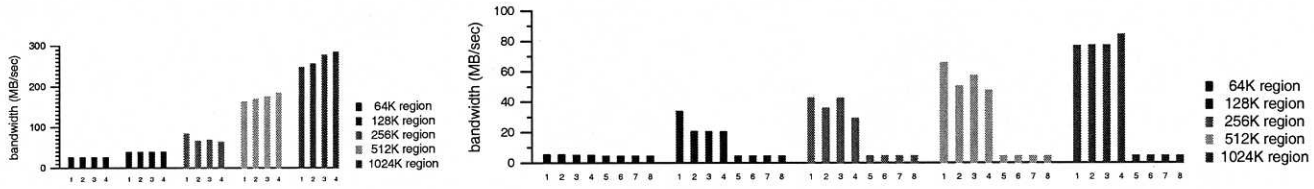



(a) **blue**.




(b) **snow**.

**Figure 8. Memory read latencies with fixed-length strides.**




(a) **blue**. (b) **snow**.

**Figure 9. Memory read latencies with randomly-selected strides.**

and stride, the increase in latency due to the increase in the number of threads is less than 5%. This is because **snow** can hide latencies using the hardware prefetching unit which fills the cache lines well in advance. In addition, its switch-based memory system can minimize the contention caused by several threads trying to access memory at the same time.

Figure 9 shows the latency of memory read operations when the memory locations are accessed randomly. Like other memory operations, latency is small for the test cases with small memory footprint [2]. In addition, the memory latency is affected less by the number of threads for **blue**, because, with a high probability, many of memory accesses fail to get the data from caches with the random accesses. Furthermore, our results reveal that the memory latency on **snow** is slightly higher than that of **blue** for the test cases whose memory footprints can fit into L1 cache. This is due to the fact that the memory contents prefetched are not likely to be used with the random accesses.

In summary, We find that the performance of applications with large memory footprints remains largely constrained by the characteristics of the underlying memory system. In addition, for these applications, the number of threads significantly affects memory performance, even for SMP machines that are specifically designed to handle concurrent memory accesses.

Finally, we also have found that the memory access pattern of an application still dominates its performance, regardless of memory footprint size of and the number of threads in the application.

## 5 Related Work

The benchmark suite and methodology we present here is limited to application programs, and focuses on memory performance, but we have found it to be both accurate and portable. We achieve portability by leveraging widely available libraries, including OpenMP, Pthreads, and PAPI. Many other benchmark suites have been used to evaluate aspects of memory performance in SMP systems. We restrict our discussion of these to those used for purposes most closely related to our goals in this study.

While the SPEC92 and SPEC95 [16] CPU benchmarks have been widely used to evaluate product performance for commercial computer systems and to evaluate new microarchitectural ideas in simulation, their working sets are too small to adequately stress high-performance memory systems. The SPEC 2000 CPU suite contains benchmarks that are more sensitive to memory bandwidth and latency, but these whole-program benchmarks are not designed to isolate performance aspects of the memory system in particular, and they are designed to run as single-threaded applications on a uniprocessor or single node of a multiprocessor system.

Given the limitations of existing benchmark suites to measure details of memory performance, a number of microbenchmark approaches have been developed. For instance, McCalpin's STREAM [10, 9] microbenchmarks measure sustainable memory bandwidth and the corresponding computation rate for simple, unit-stride vector kernels on uniprocessors, vector processors, shared-memory systems, and distributed-memory systems. The single-threaded STREAM benchmarks use data-sets larger than cache sizes in order to measure memory bandwidth for uncached accesses.

*lmbench* [11] and *hbench:OS* [5] are simple microbenchmark suites designed to evaluate OS performance. lmbench is a portable suite of microbenchmarks, a subset of the suite tests memory read and write bandwidth, bcopy bandwidth, and cached file read bandwidth, along with memory read latency on Unix platforms. The hbench:OS builds on lmbench, increasing its flexibility and precision in order to better study interactions between the system and the hardware architecture on which it runs.

The importance of using such tools to evaluate system and architectural interactions is highlighted in Brown and Seltzer's detailed study of the processor evolution of the x86 architecture: by using their hbench:OS suite, they find that the memory system continues to influence OS performance significantly, and that poor design choices for the memory subsystem can nullify the higher performance characteristics of modern processors [5].

de Supinski and May [7] implement the first Posix Threads (*Pthreads*) benchmark suite and use it to measure system performance aspects such as thread creation and the LogP [6] parallel computation performance of standard communication mechanisms on four different SMP platforms. Their results show that thread performance varies widely for different mech-

anisms and across the different platforms, revealing the opportunity for further optimizations in each system's support for threading.

Hardware performance monitors greatly enhance the quantity and accuracy of profiling data, expanding the set of events that can be measured in isolation or in relation to each other. For instance, Bhandarkar and Ding [4] use the Pentium Pro performance monitors to characterize the performance of that architecture according to CPI, cache misses, TLB misses, branch prediction, speculation, resource stalls, and micro-operations. They present their measurements for the SPEC benchmarks as averages per instruction over the application's entire execution (*e.g.*, for memory data references) or as averages per thousand cycles executed (*e.g.*, for cache and TLB misses). PAPI [12] and PCL [3] provide standard application programming interfaces for accessing the hardware performance monitors now included in most modern microprocessors; PAPI is becoming widely-accepted and is available on a large number of platforms. MPX [8] supports monitoring of conflicting events through multiplexing; it is built on top of PAPI and has been integrated into it.

## 6 Conclusions and Future Work

A new set of microbenchmarks for measuring the memory system performance of SMP-based computers is reported in this paper. The benchmarks, derived from hbench:OS [5], are multi-threaded to emulate the memory access behavior of parallel applications running on SMPs. Each benchmark measures the performance of a specific memory operation executed concurrently by a certain number of threads. Users can control the number of threads and the memory access pattern each thread follows in these benchmarks. Our benchmarks are implemented using only standard libraries that are commonly available on most systems, and hence are portable. In addition, the benchmarks are complemented by a tool that can be used to identify the characteristics of the underlying memory system.

We also have analyzed and reported in this paper the results from experiments in which we measured the memory system performance of some of SMP-based high performance computers at *location omitted for blind review* using our benchmarks. Our analysis reveals that the performance of applications with large memory footprints is largely constrained by the characteristics of the underlying memory system, and the number of threads significantly affects their performance, even when the memory system is specifically designed to support such concurrent workloads. Furthermore, our results demonstrate a significant false sharing effect when threads write data on the same memory page, a much larger coherence granularity than we anticipated, at least on the bus-based system. Finally, we have found that the memory access patterns of applications still dominates their performance, regardless of memory footprint size of and the number of threads in the applications.

Many modern CPUs offer hardware performance monitors that can be used to count data loads and stores, cache misses, and other events. Using these counters, we can obtain very accurate and detailed performance measurements. We have used PAPI [12], a standardized interface for accessing performance monitors, to implement a tool that determines memory system characteristics such as cache size, line sizes and number of TLB entries. Since it's results can provide guidance in

where different performance regimes will occur, this tool can greatly speed the process of measuring the full spectrum of performance of the memory system.

The initial implementation of our benchmarks use OpenMP library [14], an easy-to-program method for threading applications. However, OpenMP implementations are not available on many platforms and are still not always reliably implemented when they are. Further, they frequently force a high level of optimization on the user. Although this may be generally desirable, the optimizations can invalidate microbenchmark programs such as ours. To overcome these drawbacks, we have also implemented our benchmarks in Pthreads. We also plan an implementation that measures the effect of memory system contention with process-level parallelism that will use Unix shared memory for interprocess synchronization.

We have extended the range of memory access patterns that can be measured, even for uniprocessor systems. In addition, we have implemented a macro-generator that improves measurement accuracy through parameter-specific access operation code. This macro-generator approach allows fine grain measurements that clearly identify specific transitions between memory performance regimes. Further, it will allow us to extend the memory access pattern support even further; we are currently implementing a compact trace facility that will capture the access patterns of real applications and measure the baseline memory performance for those accesses with our microbenchmarks.

## References


[1] anonymous. title and publishing information omitted for blind review, Nov. 2001.

[2] *author names omitted for blind review*. Memory benchmarks for smp-based high performance parallel computers. Technical report, *institution name omitted for blind review*, 2001.

[3] R. Berrendorf and H. Ziegler. PCL – the performance counter library: A common interface to access hardware performance counters on microprocessors (version 1.3). Technical report, Central Institute for Applied Mathematics, Research Centre Jülich, GmbH, Jülich, Germany, Nov. 1999.

[4] D. Bhandarkar and J. Ding. Performance characterization of the Pentium Pro processor. In *Proc. of 3rd IEEE Symposium on High-Performance Computer Architecture*, pages 288–297, Feb. 1997.

[5] A. B. Brown and M. I. Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel $x86$ architecture. In *Proc. ACM SIGMETRICS*, pages 214–224, June 1997.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *Proc. 4th ASM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 1–12, May 1993.

[7] B. de Supinski and J. May. Benchmarking pthreads performance. In *Proc. 1999 Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1985–1991, June 1999.

[8] J. May. Mpx: Software for multiplexing hardware performance counters in multithreaded programs. In *Proc. 2001 Int'l Parallel and Distributed Processing Symp.*, 2001.

[9] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[10] J. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. `http://www.cs.virginia.edu/stream/`, 1999.

[11] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proc. 1996 USENIX Technical Conference*, pages 279–295, Jan. 1996.

[12] P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP User Group Conference*, June 1999.

[13] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., 1996.

[14] OpenMP Commitee. OpenMP. `http://www.openmp.org`.

[15] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*. The MIT Press, 1998.

[16] Standard Performance Evaluation Corporation. SPEC Benchmarks. `http://www.spec.org`, 2001.

[17] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, 1996. ANSI/IEEE Std 1003.1, 1995 Edition, including 1003.1c: Amendment 2: Threads Extension [C Language].