Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks[1]

Cristina Hristea
christea@{mit.edu, engr.sgi.com}
Massachusetts Institute of Technology
Cambridge, MA 02139

Daniel Lenoski
John Keen
{lenoski, jkeen}@sgi.com
Silicon Graphics, Inc.

1. In the proceedings of *SC97: High Performance Networking and Computing*

Abstract

Even with today's large caches, the increasing performance gap between processors and memory systems imposes a memory bottleneck for many important scientific and commercial applications. This bottleneck is intensified in shared-memory multiprocessors by contention and the effects of cache coherency. Under heavy memory contention, the memory latency may increase 2 or 3 times. Nonetheless, as more sophisticated techniques are used to hide latency and increase bandwidth, measuring memory performance has become increasingly difficult. Previous simple methods to measure memory performance can overestimate uniprocessor memory latency and underestimate bandwidth by tens of percent. This paper introduces a micro benchmark suite that measures memory hierarchy performance in light of both uniprocessor optimizations and the contention and coherence effects of multiprocessors. The benchmark suite has been used to improve the memory system performance of the SGI Origin multiprocessor.

1 Introduction

Many real-world applications have working sets which still do not fit into the large caches of modern microprocessors. With the increasing performance gap between processors and memory systems, these applications face a memory bottleneck [2]. For example, assuming a data miss rate of 4%, which is not uncommon, and a miss penalty of 50 processor cycles, the processor is stalled on memory 68% of the time[1]. To address this issue, modern microprocessors use a number of techniques to aggressively overlap cache misses with computation and subsequent memory operations. Some of these techniques include critical-word restart, lock-up free caches, deferred cache-writebacks and store buffers [3], [5], [15]. While these optimizations can dramatically improve the performance of some applications, they do not operate as effectively for others. Furthermore, the memory performance perceived by a given processor in a shared-memory multiprocessor can degrade considerably due to queueing delays. While beneficial for supporting processor caching and enforcing consistent views of memory by different processors, the cache coherency protocol can also increase the penalty of memory accesses.

In light of this memory bottleneck, measuring memory performance has become critical for predicting the performance of applications and for comparing the performance of different systems. However, with the sophistication of current compilers, processors, memory systems and cache controllers, measuring memory performance has become more difficult. Furthermore, given the variation in architectural designs across systems, creating a benchmark suite which is portable is challenging.

This paper introduces a portable micro benchmark suite that measures memory hierarchy performance. The novelty of this suite comes from its ability to handle both sophisticated processor optimizations and multiprocessor workloads. To maximize portability, the micro benchmarks are written in C and their dependence on system facilities is minimized. We believe this suite is useful to memory system designers, application writers, and end-users. Designers can use the suite to validate the predicted performance of the memory system and to find subtle performance bottlenecks. Application writers can use it to understand the critical path in their application kernels. Finally, end-users can use the benchmarks to predict the relative performance of various applications on different platforms.

The paper is organized as follows. Section 2 discusses aspects of memory performance as measured by a single processor. Section 3 presents the micro benchmarks used to measure these effects. Section 4 introduces cache coherence and its effects on latency and bandwidth. Section 5 adds the complications of queueing within the memory subsystem. In all of these sections, performance of the benchmarks on the SGI Origin 2000™ [6] and Sun Ultra Enterprise 10000™ [13] are given. Finally, Section 6 presents a discussion of future improvements in the micro benchmarks and concludes the paper.

2 Uniprocessor Memory Performance

While caches help to significantly improve the effective performance of the memory system, all applications experience some cache misses. The performance of the memory system when servicing these misses can significantly affect the overall performance of an application.

Assuming a two-level cache hierarchy, the following steps are typically required to service a cache miss[2]:

1. Wait until the controller of the primary (L1) cache completes a previous operation and then access the L1 cache tag and data.

---

1. A superscalar processor executing one load/store per cycle, with blocking cache misses is assumed.

2. Check the tag and determine that there is a L1 miss.
3. Wait until the controller of the secondary (L2) cache completes any previous operation and then access the L2 cache tag and data.
4. Check the tag and determine that there is a L2 miss.
5. Wait for the processor memory bus to free and send a request to memory for the missing block. Then wait for the response.
6. Pass the critical (miss) word onto the execution unit and possibly continue execution.
7. Complete filling of the L1 and L2 caches with the missing block.

Until recently, most microprocessors would stall from the time of the cache miss detection (step 2) until the missing block had been fetched from memory and loaded into the caches (step 7). In this model, no stall penalty would be paid for steps 3 and 5, because only one oustanding memory request was supported. With the increasing gap between the processor and memory speeds, reducing the effective memory access time has become paramount. One common improvement is to allow the processor to continue execution as soon as the critical word has been delivered. With this optimization, known as "critical word restart", the overhead of the cache fill (step 7) can be eliminated. A second step towards reducing the effective memory access time is to overlap the memory fetches of subsequent cache misses during the wait (step 5) of the first miss. This requires that the caches be lock-up free [5], and hence able to satisfy new requests while completing a cache miss, and that the subsequent miss addresses are not dependent upon the data returned by the first miss. If the miss is generated by a store or prefetch instruction, subsequent execution is not dependent upon the cache fill and continues naturally. If the miss is generated by a load instruction, some form of valid bit must be maintained on the target register to ensure that subsequent instructions receive the result of the load.

While these two optimizations can dramatically improve memory performance (both bandwidth and processor stall time), the effectiveness of these techniques depends upon the type and timing of the misses. In particular, the effectiveness of critical-word restart depends upon the delay to the next cache miss. If the next miss occurs during the fill, the stall in step 7 is simply transferred to steps 1 or 3. Likewise, memory fetch overlap is only possible if the processor can determine the succeeding cache miss and if there is no dependency between subsequent cache misses (e.g., array indexing versus pointer chasing).

Measuring memory performance in systems supporting these types of optimizations is difficult: not all misses incur the same timing penalties, and misses interact with one another. In order to understand these effects in more detail, we classify misses into three categories:

in-isolation misses are isolated in time from one-another so that the fill time (step 7) is less than the time to the next miss (thus steps 1 and 3 do not include any wait time).

back-to-back misses are dependent[3] and have minimal separation so that any contention for the caches at steps 1 and 3, and in the memory system in step 5 are seen by each miss.

pipelined misses are independent and have minimal separation so that performance is limited by whichever resource is the bottleneck during the cache fills.

Note that in-isolation and back-to-back misses represent the best and worst-case performance of the memory system when the processor is only generating one miss at a time. In-isolation represents the best-case in the sense that the processor only stalls on the cache miss detection and on the fetch of the critical word from memory. We also refer to this as *restart latency.* Back-to-back is the worst case in the sense that the cost of the entire cache fill and any interference between subsequent misses is paid by the processor. A back-to-back miss does not include the contention due to other processors, but does include the effects of the processor's interference with itself when generating new cache or memory accesses.

Pipelined miss performance is limited by one of two factors:

1. The rate limiting resource within the cache/memory system servicing the miss (cache, processor bus interface or memory).
2. The latency of a memory fetch (step 5) and the maximum number of outstanding requests that a processor can have pending to the memory system.

In either case, when pipelined operation is possible, throughput (i.e., bandwidth) becomes more important than latency itself.

Cache/memory system designs must trade-off optimizing these performance components among themselves and with other performance criteria such as hit rate. For example, a design that permits a high degree of pipelining might have a larger in-isolation or back-to-back latency due to the added complexity of the lock-up-free cache required to enable pipelining. Likewise, increasing cache block size can increase hit rates and effective memory bandwidth, but is likely to increase back-to-back latency.

For a given application, any of these three performance metrics may be most appropriate, or they may all apply in various phases of the program. For example, in many scientific applications written in FORTRAN, pipelined memory bandwidth is most important since the data access patterns of these programs consist of array references where addresses are not dependent. For applications in which memory accesses can not be overlapped either because they are infrequent or because of data dependencies, restart latency and back-to-back latency are most important. Their relative importance depends upon the data access patterns and fill time. For example, on a system utilizing 128 byte cache blocks and capable of filling at 800MB/sec (i.e. during step 7), the fill time after the initial critical word is received is on the order of 150 ns. If one assumes a 200MHz superscalar proces-

2. For simplicity, this analysis assumes that the cache can launch the read for the missing block before writing any dirty data back to memory (i.e. there is a write-back buffer).

3. A miss *m2* is said to be *dependent* on a miss *m1* when the address of *m2* is dependent on the data returned by *m1*.

sor executing one memory load or store per cycle, then the fill is hidden if the processor executes 30 or more loads or stores before the next miss[4]. In this case the misses will appear in-isolation. Likewise, the back-to-back latency is appropriate if the processor executes much fewer than 30 loads in between memory misses. Since many applications have a data miss rate of less than 3.3%, in-isolation misses are relevant to many applications.

3 Uniprocessor Micro Benchmarks

Measuring pipelined memory bandwidth, back-to-back latency, and restart latency requires micro benchmarks of increasing sophistication. The difficulties arise from the need to limit the amount and type of overlap between the processor execution and memory operations. Micro benchmarks aimed at these three performance metrics are presented in the following three subsections[5]. The results of running the benchmarks on the SGI Origin 2000 and Sun UE 10000 are described in the fourth and fifth subsections respectively.

3.1 Pipelined Memory Bandwidth

To measure memory pipeline performance, a micro benchmark must cause multiple memory accesses with independent addresses. The simple reduction loop in Figure 1 achieves this goal[6].

```
PipelinedBandwidth()
{
    StartTime(time);
    for (i = 0; i < size; i += stride) {
        j += arry[i];
    }
    EndTime(time);
    do_dummy(j);
    return ( size/time );
}
```

Figure 1 Pipelined bandwidth micro benchmark.

To compute the memory bandwidth limit, the code must be performed on increasing strides until each load generates another cache miss (i.e., the stride is greater than the cache block size). At this point, the execution time of the loop will be relatively constant. A more detailed discussion of this argument is contained in [11]. In addition, sustained memory bandwidth can be underestimated as a consequence of loop overhead and array address calculations. These effects can be minimized by unrolling the loop multiple times and binding the stride statically.

The reduction performed on the fetched data as well as the *do_dummy* call at the end of the routine ensure that aggressive compiler optimizations do not eliminate the otherwise "dead-code" in the main benchmark loop.

Similar micro benchmarks to measure memory bandwidth have existed for some time. Larry McVoy's lmbench [9] includes such a benchmark. Saavedra's micro benchmark measures pipeline memory performance on read-modify-write operations at various strides [11]. On machines with write-allocate caches, such operations can generate three memory references (i.e. one for the read, one to allocate the write, and one to writeback the data). John McCalpin's STREAMS benchmarks [8] computes memory pipeline performance on four unit-stride floating-point vector codes. With modern high-performance microprocessors, the floating-point codes are dominated by cache misses, hence the results are similar to Saavedra's for unit strides (provided the appropriate unit conversions are made between the two micro benchmarks).

3.2 Back-to-Back Memory Latency

To produce a non-pipelined reference stream, the back-to-back memory latency benchmark must defeat the aggressive overlap of cache misses supported by many microprocessors. McVoy's lmbench realizes this micro benchmark using linked-list pointer chasing. An array is initialized with pointer values at a given stride, and the time to traverse the list is then measured. By making all loads dependent upon the value from the previous loads, the cache misses can not be overlapped. A simplified version of this code is shown in Figure 2.

```
BackToBackLatency() {
    StartTime(time);
    j = arry;
    for (i = 0; i < num_reads; i++) {
        j = (int *)*j;
    }
    EndTime(time);
    do_dummy(j);
    return ( time/num_reads );
}

initialize_memory()
{
    for (i = 0; i < size; i=i+stride)
        arry[i] = &arry[i + stride];
}
```

Figure 2 Back-to-back latency micro benchmark.

4. This argument assumes the processor blocks on cache misses. For a non-blocking processor, the number of cycles would have to be greater than 30 to account for the instructions executed during the memory fetch. On a superscalar, non-blocking processor with an instruction queue of ~30 instructions and executing ~4 instructions/cycle, hence ~1 load each 4 instructions, this number is ~(30 + 30/4) = ~38

5. The uniprocessor micro benchmarks measure the performance of reads only. Although no measurements of uniprocessor write performance were gathered, the modification required by the micro benchmarks is presented in Section 4.2.2. We also did not measure the impact of writebacks on uniprocessor performance.

6. The *StartTime()* and *EndTime()* routines are calls to the Unix routine *gettimeofday* which reads the system clock. To compensate for the coarse clock resolution, which is on the order of milliseconds on a system such as the Origin 2000, a large number of reads must be performed, typically on the order of millions.

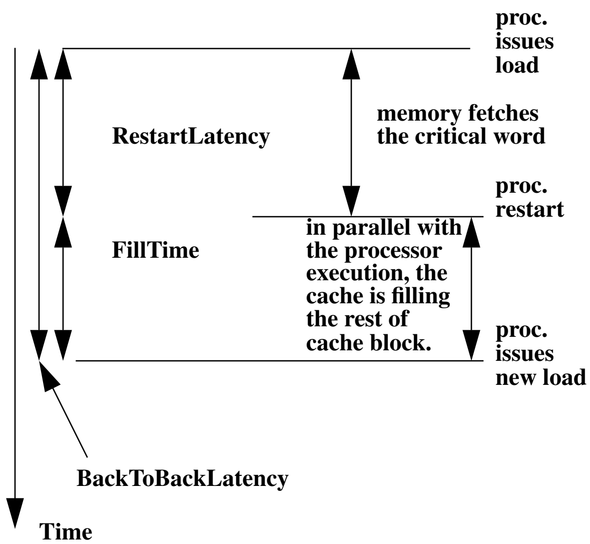

Figure 3 Restart latency micro benchmark and the derivation of restart latency from back-to-back latency.

The dependency of each load on the subsequent load's data guarantees that no two memory accesses are in the memory pipeline simultaneously at any given time. However, although no overlap at the memory exists between the cache misses, depending on the hardware, each memory access is likely to conflict with its successor at the processor cache. By including the penalty of the entire cache fill, the back-to-back latency may overestimate the latency perceived on a given miss.

3.3 Restart Memory Latency

Restart latency represents the minimum penalty incurred by the processor for a cache miss, assuming the corresponding cache fill is hidden by the execution of other processor instructions. The actual CPU stall time associated with a given miss could be less than the restart latency if the access to the target register is delayed, and if the processor provides a mechanism, such as a dynamic pipeline or a valid bit on the load register, to execute past the completion of the given memory operation. For example, the MIPS R10000™ has a dynamic pipeline [15], and the Sun UltraSPARC™ has a valid bit on the load register. To measure restart latency, the back-to-back latency test must be modified to interleave cache misses with instructions that do not interfere with the remaining cache fill. On microprocessors with two-level caches, this implies that the operands for the subsequent instructions are either in registers or the first-level cache. For measurement purposes, we break these overlapped instructions into a sequence of processor work units. If the processor-work is made dependent on the preceding memory access, and the succeeding memory access is made dependent on processor-work, then the activities of acquiring the critical word, executing the processor-work and issuing the subsequent load all happen sequentially. This is shown in Figure 3.

Because of the dependency between the succeeding load and the processor-work, when the processor-work is greater than the fill time, the issue of the succeeding load is delayed. These delays result in idle periods at the cache which, in turn, cause an increase in the total execution time. Thus, by iterating on the amount of processor-work until the execution time increases, the restart latency can be calculated as:

RestartLatency = BackToBackLatency - FillTime

where *FillTime* is the maximum amount of processor-work that can be executed between successive cache misses without an increase in the total execution time.

The code in Figure 4 iterates over the processor-work time to identify the proper refill overhead.

```
RestartLatency() {
    last_time = 0, cur_time = 0;
    while ((cur_time<=last_time) || (last_time == 0)) {
      last_time = cur_time;
      num_iters++;
      StartTime(cur_time);
      j = arry;
      for (i = 0; i < num_reads; i++) {
        j = (int *)*j;
        for (k = 1; k <= num_iters; k++)
            DO_WORK(dummy, j);
      }
      EndTime(cur_time);
    }
    return ( (last_time - (num_iters - 1) *
              WORK_COST) / num_reads );
}
```

DO_WORK is the following simple instruction:

```
#define DO_WORK(dummy, j)
    j = j + dummy;
```

Figure 4 Restart latency micro benchmark.

To avoid the overlapping of processor-work with the subsequent load, a dependency scheme on *j* is enforced between the two activities. The argument *dummy* is passed in the value 0 so as to prevent the modification of the address of the subsequent load. Care must be taken to invalidate the cache data between successive iterations to avoid reading from the cache. In order to compute the cost of each processor-work unit, *DO_WORK* itself is timed in a loop with a large number of iterations. Appropriate values for *num_reads* are on the order of millions. To limit the amount of system interference at the memory, the micro benchmark must be executed on lightly-loaded systems.

3.4 Results on the SGI Origin 2000

The Origin 2000 has a nonuniform memory access (NUMA [7]) architecture, in which the nodes of the system include both processors and memory, and are connected by means of a scalable Craylink network [6]. This architecture is shown in Figure 5. The Origin 2000 uses MIPS R10000 processors, running at 195MHz. The L2 cache runs at 2/3 of the processor frequency (130MHz) and the processor bus at 1/2 of the processor frequency (98MHz). The L2 cache block size was measured to be 128 bytes.

The back-to-back load latency to local memory is 472 ns (46 processor bus cycles). This result, which matches the result measured by McVoy's micro benchmark, was verified by gathering program traces on the logic analyzer. A breakdown of this latency is given in Table 1.

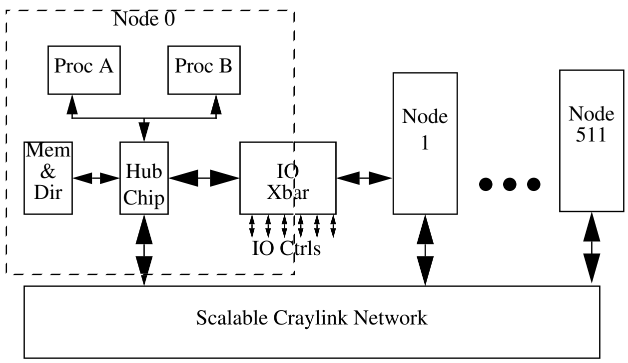

Figure 5 Origin block diagram.

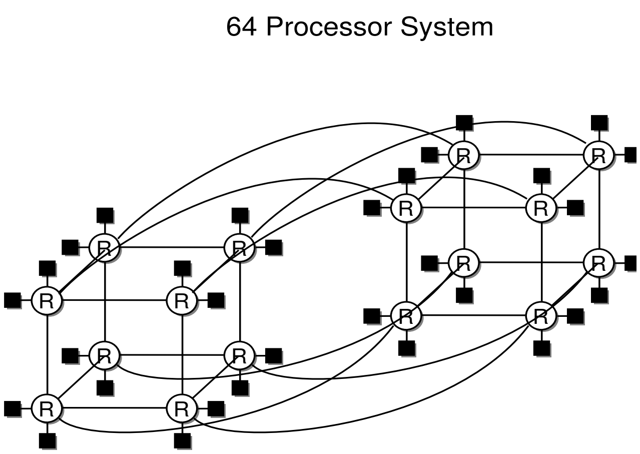

Figure 6 Origin topology, a hypercube. The vertices labeled R represent routers.

Table 1: The cycle breakdown of a back-to-back load.

Bus Cycles	Activity
10	R10K proc. finishes filling in the L2$ with the previous load's data; Subsequently, R10K proc. detects L2$ miss on the newly issued load and places a read request on the processor bus.
20	The hub gives the first double word from memory (the critical word) to the processor.
16	The hub places the rest of the L2$ block onto the proc. bus; R10K proc. places the L1$ and L2$ blocks into the L1$ and L2$; R10K detects an L1$ miss on the next load.
46	total

Restart load latency to local memory takes 338 ns. The 133 ns (13 processor bus cycles) difference between the back-to-back load latency and the restart load latency is mainly a consequence of the L2 cache block being larger than the L1 cache block by 12 double words, where a double word corresponds to the size of the data path connecting the processor to the memory subsystem[7]. The extra cycle is the result of an additional bus turnaround overhead in the back-to-back latency micro benchmark. Approximately three extra processor cycles (15.3 ns) are paid by each miss in the restart micro benchmark for issuing the load, detecting the L1 cache miss and sending the miss address to the L2 cache pins.

The interconnect topology of Origin 2000 systems [1] is a hypercube as shown in Figure 6. Because of its NUMA architecture, the Origin multiprocessor exhibits varying delays to memory, depending on the relative location of the data with the respect to the processor accessing it. When the home[8] is the closest remote memory, back-to-back latency takes 687 ns (67 processor bus cycles), while restart load latency takes 554 ns (54 processor bus cycles). For requests to further nodes there is a penalty of 103 ns (10 processor bus cycles) per round-trip for each router that must be traversed. A summary of restart latencies on various system sizes is given in Table 2.

7. The processor must wait until the L1 cache is entirely filled (not just the required doubleword) before it can resume execution.

8. The home is the memory storing the directory information associated with the requested cache block.

Table 2: Restart latency for reads measured on Origin 2000 systems of various sizes

System size (CPU count)	worst-case router hops	worst-case latency (ns)	% local mem.	avg. router hops	avg restart latency (ns)
2	0	338	100	0	338
4	1	554	50	0.5	446
8	2	656	25	1.25	524
16	3	759	12.5	1.875	629

The average values in the table assume a uniform distribution for the home memory locations of the requested data across the nodes of the system. The pipelined memory bandwidth results are presented in Table 3.

Table 3: Uniprocessor bandwidth for reads measured for various home locations on the Origin 2000

Number of hops to home	Pipelined bandwidth (MB/s)
0	554
1	508
2	486
3	481

3.5 Results on the Sun UE 10000

As shown in Figure 7, the Sun UE 10000 multiprocessor has a uniform memory access architecture with a two-level switch topology. All processors and memories on a module are connected through a many to one switch to a 16X16 crossbar implemented on the centerplane in the system. All the memory accesses go through the crossbar. The UE 10000 uses UltraSPARC-II processors, running at 250 MHz. The L2 cache runs at 1/2 of the processor frequency (125 MHz) and the processor bus at 1/3 of the processor frequency (83 MHz). The L2 cache block size was measured to be 64 bytes, and the L1 cache block size is 32 bytes.

Table 4: Uniprocessor memory performance on UE 10000

Restart Latency (ns)	Back-to-back Latency (ns)	Bandwidth (MB/s)
548	560	195

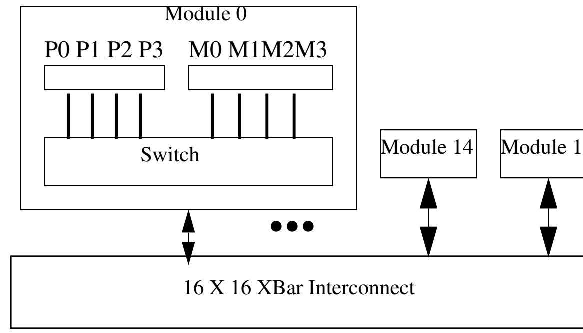

Figure 7 UE 10000 block diagram

The back-to-back latency is 560 ns (46.7 bus cycles) and the restart latency is 548 ns (45.7 bus cycles). The difference of 12 ns (1 processor bus cycle) between the back-to-back and restart latencies is a result of the L2 cache block size being larger than the L1 cache block size by 32 bytes which are moved from memory to the processor in 2 processor bus cycles. Approximately three extra processor cycles (12 ns) are paid by each miss in the restart micro benchmark for issuing the load, detecting the L1 cache miss and sending the miss address to the L2 cache pins.

4 Cache Coherency

Caching allows higher performance to be achieved by reducing the effective memory latency. However, when multiple processors are allowed to maintain local copies of shared-memory locations, a cache coherence protocol is required to ensure the processors have consistent views of memory. The coherence protocol adds complications to memory operations and, depending upon the caching state of the given memory location, can affect memory performance.

4.1 Micro Benchmark Implementation

While there are many different coherence protocols, we focused on the Illinois [10] or MESI [12] protocol because it is one of the most commonly used. This protocol is invalidation-based and uses four caching states: *modified, exclusive, shared,* and *invalid*. The modified state is also known as *dirty*, and the exclusive state as *clean-exclusive* since it implies that the processor with a cached copy of the block has not written to it. The coherence protocol affects the performance of a memory load/store when the requested cache block is in the modified state, and in some systems, the clean-exclusive state. In addition, the coherency protocol affects the performance of a store operation when the block is shared because an exclusive copy must first be obtained by means of invalidating the other cached copies.

To measure the effects of coherency, the uniprocessor micro benchmarks are modified to establish a particular caching state for a group of consecutive virtual addresses prior to their access in a uniprocessor micro benchmark[9].

9. A LRU replacement algorithm is assumed.

```
CCPerformance() {
    StartTime(T_cc_setup);
    for (i=0; i<num_groups; i++)                    |
        InitializeState of group i in slaves        | A
    EndTime(T_cc_setup);
    StartTime(T_total);
    for (i=0; i<num_groups; i++) {                  |
        InitializeState of group i in slaves        | B
        UniMB on group i in master                  |
    }                                               |
    EndTime(T_total);
    time = T_total - T_cc_setup;
    return(time / num_reads);
}
```

Figure 8 The cache coherency micro benchmark.

Processors called *slaves* are used for initialization, and a *master* processor is used for running the uniprocessor micro benchmarks. The group size must be small enough to ensure the presence of the data in the desired caches upon master's access, and also large enough to give good timing results (such as a page size).

As shown in the pseudocode of Figure 8, the entire array is covered twice (code sections A and B), and timing measurements are taken at both runs. The master performs the uniprocessor micro benchmark only when covering the array for the second time (code section B). This way, two parameters are measured: the performance of the cache coherency initialization routines (including the barrier calls necessary to precede the use of data by its initialization) *T_cc_setup*, and the combined performance of the initialization routines and the master accesses *T_total*. The memory performance is computed by subtracting *T_cc_setup* from *T_total*.

The code for *InitializeState()* is given in Figure 9. *UniMB()* is a routine which dispatches the desired uniprocessor micro benchmark. For accuracy, the time spent calling *UniMB,* which is included in *T_total* but not accounted for in *T_cc_setup* must be eliminated, by inserting the body of *UniMB* into the calling routine *CCPerformance*.

To verify that the initialization routine produced the correct coherence state, the number of cache misses, writebacks and invalidates caused by the master were counted on the Origin 2000. These counts relied on the ability of the processor to report the various processor events such as invalidates, writebacks and L1 and L2 cache misses. In addition, a logic analyzer was used to gather traces at the processor bus. We analyzed the traces by checking the state of the data as read from the processor cache.

```
InitializeState () {
    switch (cc) /*global var storing the CC state*/
     {
       case MODIFIED:
         for (i=0; i<size; i += stride)
            arry[i] = &arry[i + stride];
          break;
       case EXCLUSIVE || SHARED:
         for (i=0; i<size; i += stride)
            j += arry[i];
         do_dummy(j);
         break;
     }
}
```

Figure 9 State initialization in the cache coherency micro benchmark.

A shortcoming of our methodology is that, in the modified case, extra writebacks may happen during the first array coverage, which would lead to overestimating *T_cc_setup* and hence to underestimating the performance degradation of cache coherency for the modified state. We believe this shortcoming is significant only for systems with poor page coloring.

4.2 Results on the Origin 2000

4.2.1 Reads

These are the cache coherency results for the unowned, exclusive and modified cases. The following table lists the back-to-back latencies:

Table 5: Back-to-back latencies of reads under various CC cases on the Origin 2000

Home	Owner	Unowned (ns)	Clean-Ex. (ns)	Modified (ns)
local	local	472	707	1036
remote 1[a]	local	704	930	1272
local	remote 1	472	930	1159
remote 1	remote 1	704	917	1097

a. By "remote 1" is meant a processor in the closest remote node to the requesting processor.

The following table lists the restart latency values:

Table 6: Restart latencies of reads under various CC cases on the Origin 2000

Home	Owner	Unowned (ns)	Clean-Ex. (ns)	Modified (ns)
local	local	338	656	892
remote 1	local	570	879	1128
local	remote 1	338	879	1015
remote 1	remote 1	570	866	953

Finally, these are the pipelined memory bandwidth results:

Table 7: Pipelined bandwidth of reads under various CC cases on the Origin 2000

Home	Owner	Unowned (MB/s)	Exclusive (MB/s)	Modified (MB/s)
local	local	554	340	182
remote 1	local	508	298	178
local	remote 1	554	342	246
remote 1	remote 1	508	334	238

In the Origin 2000, the caches and the main memory are kept coherent through a directory structure implemented directly in hardware [6], [8]. Requests to memory are processed by the home of the requested data. The home reads the contents of the requested location while performing a directory lookup. In the case of a read request, if the directory state is exclusive with an owner different from the requestor, the home sends an intervention shared request to the owner and a speculative reply to the requestor. Depending on whether the owner has a modified copy or an exclusive or invalid copy, it sends a shared response (data) or a shared ack (no data) to the requestor. These two cases are illustrated in Figure 10.

While sharing the same bus, the two processors on an Origin node do not function like a snoopy cluster. Consequently, caches do not change state by monitoring the bus for requests by the other processor, but only as a result of a memory intervention. This has several implications on performance. On the positive side, uniprocessor performance is improved because there is no irrelevant snoop traffic. Further, requests for remote memory are forwarded directly by the local processor without having to wait for a local snoop. On the negative side, if the data is cached in the other local processor, latency is considerably longer as shown in Table 6.

Since the requestor can not access the critical word until it receives the acknowledgment from the owner, the back-to-back and restart latencies to cache blocks in the clean-exclusive state take almost the same amount of time. Notice however that they are not congruent. After being stored in the processor buffers, the critical word along with the rest of the L1 cache block are transmitted to the L1 cache upon receiving the acknowledgment from owner. This activity takes around 40 ns.

In the exclusive and modified cases, the pipelined memory bandwidth is better when the owner is remote because twice as much processor bus bandwidth is available than when the owner is local.

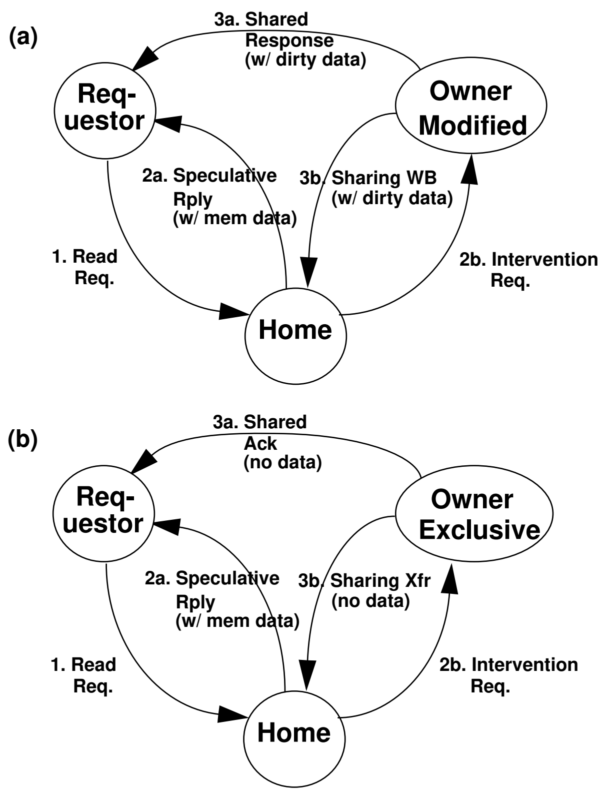

Figure 10 **Read of (a) Dirty Data and (b) Clean-Exclusive Data.**

4.2.2 Writes

To write to a block which is shared in other caches, the processor sends a read-exclusive request to memory. If the request succeeds, memory sends invalidations to the sharers and an exclusive reply with invalidates pending to the requestor. Once the requestor receives all the invalidate acknowledgments, it fills the cache block with the memory data and sets its state to modified. To measure store latency, the test kernel is modified as follows:

```
for (i = 0; i < num_reads; i++) {
    j = (int *)(*j);
    *j = (int)j+stride;
}
```

Since the overhead from processing the invalidates is insignificant compared to network latency, the latency of a read-exclusive to a shared block does not depend on the number of sharers, but mainly on the maximum number of hops D from M to a sharer. As shown in Figure 11, this latency increases by about 110 ns for each additional hop.

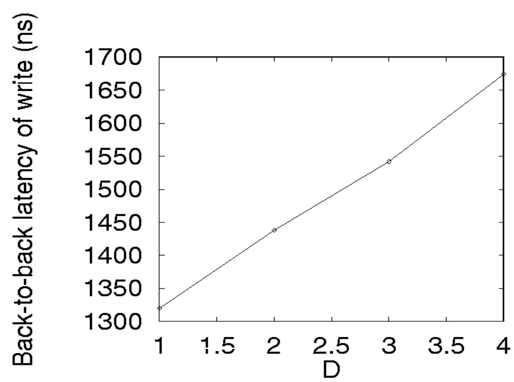

Figure 11 **Back-to-back latency of writes to shared lines in terms of the distance D to the furthest sharer on the Origin 2000.**

The performance does not depend on the number of sharers located on a particular node. This is a consequence of the Origin design in which only one invalidate message per node is used.

4.3 Results on the UE 10000[10]

Table 8: Cache coherency results measured on UE 10000

	Unowned	Modified
Pipelined b/w (MB/s)	195	154
Back-to-back lat. (ns)	560	742
Restart lat. (ns)	548	730

In the UE 10000, because of snooping, the owner of modified data can service interventions earlier than in the Origin 2000 where it must first wait for a request from memory. Consequently, the modified case incurs only additional 182 ns (15.2 bus cycles) over the unowned case.

5 Contention

Large-scale parallel applications inevitably induce contention when run on shared memory multiprocessors. Contention can occur at any physically shared resource, such as a shared processor bus, memory, or the network. By generating contention at user-specifiable system locations and measuring its impact on performance, our micro benchmark suite provides users with yet another insight into machine performance on real applications.

5.1 Micro Benchmark Implementation

The memory test is modified to allow various threads to interfere with some master processor M at various system resources. In this work, we studied the impact of contention at the memory and the shared-processor bus.

To generate contention levels which are representative of loads on real applications, a number of threads are created and placed on separate processors so as to be run concurrently with M and independently of each other. All these threads, which we call noise generators or NGs, run the same code.

A data access pattern is defined as the amount of interleaving between memory accesses and instruction execution (work) in the cache by some processor. To vary contention, the NGs are run under various data access patterns (along with running M). The contention value each pattern generates is unknown to the user and measured at run time. The more work the NG does between memory accesses, the less contention it produces in the system. The NGs access memory at strides which increase with the desired contention. In addition, when reduced contention is desired, the NGs execute a routine which accesses register or cache data only. To vary the contention level, the number of interfering threads is also varied, from 1 to the maximum number of available processors in the system (when the memory contention is studied) or on the shared node (when the processor bus contention is studied).

To estimate contention, the NGs keep counters of the memory blocks transferred. To avoid overestimating the amount of contention, M flags the NGs upon start and finish. The NGs turn the counters on upon M's start and off upon M's finish. M must start after the NGs and finish before them. To cause M to start after the NGs, a barrier call is executed early in the program and a start-up routine is executed by M prior to the memory test, which, by generating a large number of accesses to memory, incurs a long delay and hence, ensures that all the NGs have started execution by the time M enters the memory test. To cause M to finish before the NGs, the NGs are forced to cover larger data sets than the data set accessed by M during the execution of the memory test. Figure 12 illustrates the above discussion:

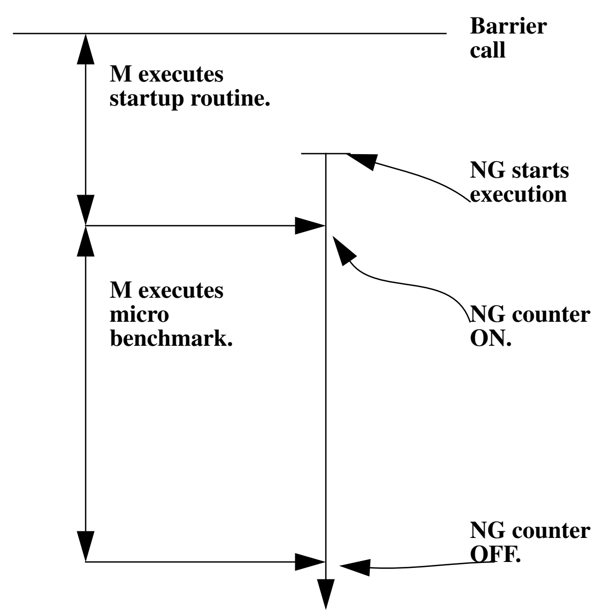

Figure 12 Contention micro benchmark.

The flag, which is checked periodically by the NGs, is implemented as a shared variable with two possible values: OFF and ON. When the flag is OFF, the NG leaves the counter, a private variable, unchanged. When the flag is ON the counter is incremented by the number of bytes moved since the last check. To compute this increment, the NG keeps track of the bytes moved before the previous check and subtracts it from the total number of bytes moved.

10. The Exclusive state is not supported by this multiprocessor, hence only results for the Unowned and Modified states are presented.

5.2 Results on the Origin 2000

On current shared-memory multiprocessors, an important point of contention is the memory. Because of its architecture, another important contention location on the Origin 2000 is the processor bus. To restrict bandwidth consumption to a particular location, system routines for process and data placement are used. To restrict contention to the processor bus, the NGs and M are placed on the same processor bus and the data on different physical memories. Similarly, to restrict contention to some physical memory, the NGs and M are placed on different buses and the data on the same physical memory.

Table 9 and Table 10 list the restart latencies for a range of contention conditions enforced at the memory and processor bus respectively on the Origin 2000 multiprocessor.

Table 9: Restart latency under memory contention on the Origin 2000

NG Contention (MB/sec)	Master Thruput (MB/sec)	Total Contention (MB/s)	Memory Utilization (%)	Restart Latency (ns)
0	271	271	41	338
41	253	294	44	371
121	231	352	53	421
233	207	440	66	483
336	181	517	78	572
538	80	618	93	1462

Table 10: Restart latency under processor bus contention on the Origin 2000

NG Contention (MB/sec)	Master Thruput (MB/sec)	Total Contention (MB/s)	Proc. bus Utilization (%)	Restart Latency (ns)
0	271	271	43	338
121	227	348	55	431
316	178	494	78	586
386	139	525	83	786

The memory utilization data assumes a peak sustained memory bandwidth of 667 MB/s. The processor bus utilization data assumes a peak sustained processor bus bandwidth of 631 MB/s. These assumptions will be justified in the next section.

5.3 Discussion of Results

This section outlines a general methodology for analyzing the results on the contention micro benchmark. As expected, latency grows nonlinearly with the increasing NG contention *N*. If we assume a M/D/1 queueing model for the contended resource (random arrivals, deterministic service [4]), latency can be expressed in terms of resource utilization *U* as follows:

$$L(U) = L(0) + S\left(\frac{1}{2} \cdot \frac{U}{1-U}\right)$$

where *L(0)* is the latency under no contention and *S* is the average service time of the contended resource. The resource utilization can be derived from the resource contention (which includes the contention generated by the master) in the following way:

$$U = \frac{\mathrm{TotalContention}}{\mathrm{CacheLineSize}} \cdot S$$

L(0) can be deduced from the latency under utilization by M only (no NG contention) as follows:

$$L(0) = L(U^{(N=0)}) - \left(\frac{1}{2} \cdot \frac{U^{(N=0)}}{1-U^{(N=0)}}\right) \cdot S$$

One can compute *L(U)* by expressing it in terms of *S* and iterating over service times *S* to find the value which results in the smallest least square error to the actual data. Applied to the Origin 2000, this methodology leads to the coefficient values listed in Table 12. The queueing model results plotted against the experimental data are shown in Figure 13.

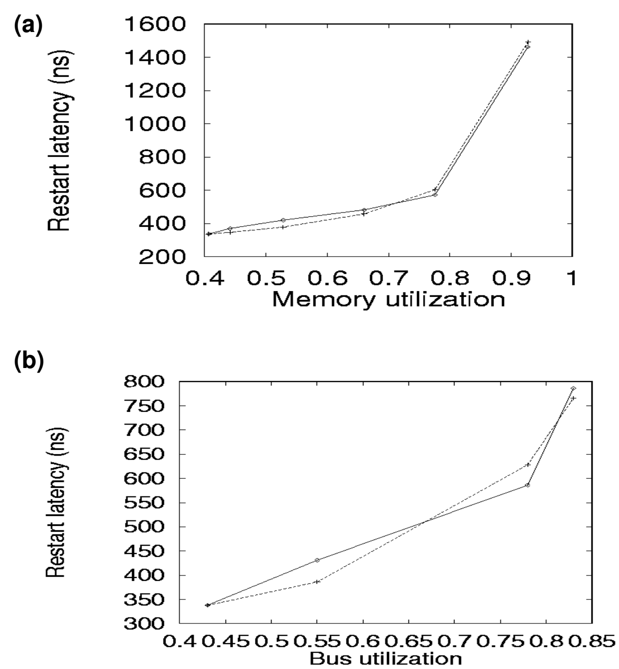

Figure 13 (a) Memory (b) Processor bus contention on the Origin 2000. The dashed line is derived from the queueing model. The continuous line is the experimental data.

The least square errors are 28 ns/sample for the memory and 32 ns/sample for the processor bus respectively. The service times imply peak sustained memory bandwidth of 667 MB/sec[11] and peak sustained processor bus bandwidth of 631 MB/sec. These results match the sustained bandwidths seen in cycle-by-cycle based simulations within 5%.

5.4 Results on the UE 10000

Table 11 lists the restart latencies for a range of contention conditions enforced at one memory module of a UE 10000 multiprocessor.

Table 11: Restart latency under memory contention on the UE 10000.

NG Contention (MB/sec)	Master Thruput (MB/sec)	Total Contention (MB/s)	Memory Utilization (%)	Restart Latency (ns)
0	114	114	13	548
189	113	302	34	553
312	108	420	47	583
387	105	492	55	595
453	101	554	62	623
528	97	625	70	651
694	79	773	87	799
805	45	850	96	1411

The utilization coefficients *L(0)* and *S* are given in Table 12 and the plot of the queueing model data against the experimental data is given in Figure 14. Best fit based on least square computations with a least square error of 32 ns/sample implies a memory service time of 72 ns (6 bus cycles). In turn, this implies a peak sustained memory bandwidth of 889 MB/sec[12].

Table 12: Utilization coefficients for the Origin 2000 and UE 10000 systems

	L(0)	S
Origin memory	273	192
Origin proc. bus	261	203
UE 10000 mem.	543	72

11. These numbers refer to the performance of a single memory node. The impact on memory performance of OS tools for data placement across multiple memories is not measured by our experiments.

12. Like for the Origin 2000, these numbers refer to the performance of a single memory module and do not take into account the impact on memory performance of OS tools for memory data placement.

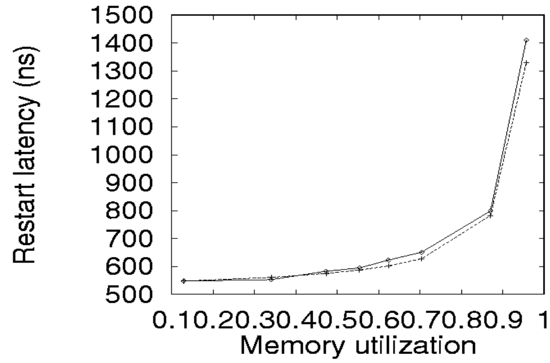

Figure 14 Memory contention experimental data (continuous line) and the queueing model (dashed line) for the UE 10000.

6 Conclusions

The increasing performance gap between processors and memory systems imposes a memory bottleneck for many important scientific and commercial applications. Consequently, understanding memory performance has become important for predicting the performance of real applications and for comparing different systems. With the sophistication of current processors and the complex operations of cache-coherent multiprocessors, memory performance has become more difficult to measure. This paper has introduced a memory benchmarking methodology which can handle both sophisticated uniprocessor and multiprocessor workloads.

The methodology is based on three micro benchmarks, each measuring one of the three parameters of memory performance: restart latency, back-to-back latency and pipelined bandwidth. As revealed by our experiments, each parameter offers a unique result and insight into memory performance. Back-to-back latency can be significantly larger than restart latency. For example, on the Origin 2000, back-to-back latency to local memory is greater than restart latency by ~40%. Likewise, the pipelined bandwidth to local memory is 554 MB/s, in contrast to 271 MB/s which would result if simply dividing the cache block size (128 bytes) by the back-to-back latency (472 ns).

The memory access architecture affects memory latency significantly. While scalable uniform memory access machines such as UE 10000 maintain constant latency across all ranges of expandability, NUMA machines such as Origin 2000 achieve a lower latency to close memories but a higher one to the memories which are far away. For example, the restart latency is 548 ns on the UE 10000. On the Origin 2000, it is 338 ns to local and 554 to the closer remote memory, and it increases by about 100 ns for each additional hop to the memory.

Pipelined memory bandwidth depends primarily on the cache block size and the ability of the processor to overlap memory accesses. On a 16 processor Origin 2000 system which uses MIPS R10000 processors and 128 byte L2 cache block sizes, a pipelined memory bandwidth of about 500 MB/s can be achieved to any memory. The UE 10000, which uses UltraSPARC-II processors and 64 byte cache block sizes sustains a pipelined memory bandwidth rate of 200 MB/s.

To study the effects of cache coherency, a particular caching state was established prior to accessing each memory block. On the Origin 2000, the cache coherency protocol increases the penalty of a memory access significantly. For example, the restart latency increases by 164% and the pipelined memory bandwidth drops by 67% when the state of the block is dirty in the other local processor (in which case some extra cache coherency operations are required to obtain the most updated version of the cache block). Similarly on the UE 10000, for the same cache coherency state, the restart latency increases by 33% and the pipelined memory bandwidth is reduced by 21%.

To measure the impact of contention, the micro benchmarks were extended to allow various levels of interference to be introduced and measured at controlled system resources. As shown by experiments on both Origin 2000 and UE 10000, the resource utilization adds delays and reduces the bandwidth seen by a given processor. For example, when the memory utilization is ~70%, the restart latency increases by 45% on the Origin 2000 and ~20% on the UE 10000.

6.1 Future work

The application of micro benchmarks to macro benchmark performance prediction remains an open issue. While research by Saavedra [11] has shown that, for simple uniprocessor applications, such predictions can be made within 10% of the actual run times, no similar achievements in the area of multiprocessor applications have yet been demonstrated [14]. However, with the new results our work has generated in measuring multiprocessor memory performance, such a methodology may be possible. In addition, the ability of our micro benchmarks to handle sophisticated processor optimizations may allow for further improvements in uniprocessor macro benchmark prediction to be achieved.

7 Acknowledgments

We thank the Origin team for feedback on the Origin data. In particular, we thank Doug Solomon, Jim Laudon, Mike Galles and Mike Woodacre. We also thank Mi-Suk Huh, Jim Laudon, Mike Woodacre and Mike Galles for help in setting up the laboratory equipment. Thanks also to Narayan Ganapathi, John Richardson, Luis Stevens, Ben Fathi, Rolf Mueller and Steve Whitney for assistance with the R10K and Origin software. Thanks to Rafael Saavedra for sharing insights into micro benchmarking and to Larry McVoy for participating in some of the Origin experiments. At Boston University, we thank Glenn Bresnahan for access to the Origin machine. At MIT, we thank James Hoe for support and feedback on the final draft of this paper, and professor Arvind for permission to use the Sun multiprocessor. Finally, at Sun Microsystems we thank Brian Whitney, Greg Papadopulos, Shahin Khan, Brad Carlile and Anne Lavery for the experimental results on the UE 10000.

References

[1] Mike Galles. Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER chip. In *Hot Interconnects ‘96.*

[2] J. Hennessy and N. Jouppi. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer* 24(9):18-29, Sept. 1991.

[3] John Hennessy, David Patterson. *Computer Architecture: a Quantitative Approach*. Morgan-Kaufman, San Mateo, California, 1990.

[4] W.H. Highleyman, *Performance Analysis of Transaction Processing Systems*, Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1989.

[5] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proc. 8th Int. Symp. on Computer Architecture*, pages 81–86, May 1981.

[6] James Laudon, Daniel Lenoski. The SGI Origin 2000: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[7] Daniel Lenoski, Wolf-Dietrich Weber. *Scalable Shared-Memory Multiprocessing.* Morgan-Kaufman, San Mateo, Cali fornia, 1995.

[8] John D. McCalpin. Sustainable Memory Bandwidth in Current High Performance Computers. *http://reality.sgi.com/mccalpin_asd/*, October 1995.

[9] Larry McVoy, Carl Staelin. lmbench: Portable tools for performance analysis. *http://reality.sgi.com/lm_engr/index.html*, 1995.

[10] M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. 11th Int. Symp. on Computer Architecture*, pages 348-354, May 1984.

[11] Rafael Saavedra, R. Stockton Gaines, Michael Carlton. Characterizing the Performance Space of Shared Memory Computers Using Micro Benchmarks. July 1993.

[12] P. Sweazy and A. Smith. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In *Proc. 13th Int. Symp. on Computer Architecture*, pages 414-423, June 1986.

[13] The Ultra Enterprise 10000 Server. *http://www.sun.com/servers/datacenter/products/starfire/wp.html.*

[14] Stephen J. Von Worley, Alan J. Smith. Microbenchmarking and Performance Prediction of Parallel Computers. UCBerkeley Technical Report CSD-95-873, May 1995.

[15] Kenneth Yeager, The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, 16(2):28-40, April, 1996.