Micro Benchmarks for Multiprocessor Memory Hierarchy Performance

by

Cristina-Ana-Maria Hristea

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science and Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

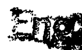
© Massachusetts Institute of Technology, 1997. All Rights Reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by ....
Arvind
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur Smith
Chairman, Departmental Committee on Graduate Theses

OCT 29 1997



Micro Benchmarks for Multiprocessor Memory Hierarchy Performance

by

Cristina-Ana-Maria Hristea

Submitted to the Department of Electrical Engineering and Computer Science

on May 23, 1997, in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

Abstract

In uniprocessor systems, the increasing performance gap between the processor and the memory imposes a memory bottleneck for many important scientific and commercial applications. In cache-coherent multiprocessors, contention for shared resources and the effects of the cache coherency protocol can adversely affect the performance of the memory system, so that applications may experience an even greater memory bottleneck.

Traditionally, memory performance has been characterized in terms of latency and bandwidth. The purpose of this thesis is to refine the traditional definition of memory performance to address the characteristics of modern processors. Previously, bandwidth could be computed from cache line size and latency. In modern systems, because of lock-up free caches and memory pipelining, *pipelined memory bandwidth* has become an important metric. With the ability of modern processors to support critical word restart, latency is defined in terms of two distinctive performance parameters: *back-to-back latency* and *restart latency.*

As more sophisticated techniques are used in microprocessors to hide memory latency and increase bandwidth, measuring memory performance has become increasingly difficult. In cache-coherent multiprocessors this task is complicated further by cache coherency and contention. This thesis presents a micro benchmark suite to measure memory performance in the context of cache-coherent multiprocessors. An analytic model based on queueing theory is developed to estimate memory performance in cases which can not be measured experimentally. The results for the SGI Origin 2000 multiprocessor and some results for the SUN Ultra Enterprise 4000 multiprocessor are presented. The benchmark suite has been used to optimize the memory system performance of the SGI Origin 2000 multiprocessor.

Thesis Supervisor: Arvind
Title: Professor of Computer Science and Engineering

Acknowledgments

First and foremost, I thank Daniel Lenoski who first introduced me to the wonders of computer architecture. As a mentor since my sophomore year, he has given me time and feedback, and has shared with me some of his vast knowledge and revolutionary ideas. Many of the ideas in Chapters 2 and 3, and some in Chapter 5 are borrowed from him. Also, during the great rush before the thesis deadline, he dedicated his evenings and weekends to revising my thesis and giving me helpful suggestions on how to improve it. This thesis would not have been possible without Dan.

Then, I thank my thesis supervisor at SGI, John Keen for his continuous support in the thesis effort. During the writeup stage, his numerous and careful reviews of the thesis lead to considerable improvements in its presentation.

At SGI, I also thank Marty Deneroff, Rick Bahr and Woody Lichtenstein for making the 6A program with SGI possible. Thanks go to the Origin team for feedback on the Origin data, in particular, to Doug Solomon and Jim Laudon. Many thanks to the lab assistants, to 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 R10000 and Origin software.

Thanks to Rafael Saavedra too for sharing insights into micro benchmarking and to Larry McVoy for participating in some of the Origin experiments.

At Boston University, I thank Glenn Bresnahan for access to the Origin machine. At MIT, many thanks to James Hoe for support and feedback on the thesis proposal and on a version of this thesis, and to professor Arvind for support and permission to use the Sun multiprocessor.

I thank and hug my parents, Rozalia and Nicolae, and my brother, Ionut for their warm and loving support. Thanks and hugs go to Mike Chow too for his friendship all throughout the thesis effort.

Table of Contents

1 Introduction.....7
- 1.1 Motivation.....7
- 1.2 Previous Work.....9
- 1.3 Approach.....11
- 1.4 Thesis Outline.....12

2 Uniprocessor Memory Performance.....13

3 Uniprocessor Micro Benchmarks.....19
- 3.1 Pipelined Memory Bandwidth.....19
- 3.2 Back-to-back Memory Latency.....21
- 3.3 Restart Memory Latency.....22
- 3.3 Results on the Origin 2000.....25
- 3.5 Results on the UE4000.....28

4 Cache Coherency.....29
- 4.1 Micro Benchmark Implementation.....32
- 4.2 Results on the Origin 2000.....29
  - 4.2.1 Reads.....32
  - 4.2.2 Writes.....33
- 4.3 Discussion of Results.....35
  - 4.3.1 Reads.....35
    - 4.3.1.1 Requestor and Owner Share the Processor Bus.....37
    - 4.3.1.2 Requestor and Owner on Different Processor Buses.....40
  - 4.3.2 Writes.....41

5 Contention.....45
- 5.1 Micro Benchmark Implementation.....45
- 5.2 Results on the Origin 2000.....48
- 5.3 Analytic Model.....49

6 Conclusions.....59
- 6.1 Accomplishments.....59
- 6.2 Future Work.....60

Appendix A
- A.1 Acquiring Laboratory Data.....63

Appendix B
- B.1 Derivation of Quadratic Contention Equation.....65

References.....67

List of Figures

Figure 1.1: A general architecture for a cache-coherent multiprocessor 8
Figure 2.1: Cache contention 14
Figure 2.2: Cache conflicts between loads 15
Figure 3.1: The restart latency micro benchmark 23
Figure 3.2: Origin block diagram 25
Figure 3.3: Origin topology 27
Figure 4.1: MESI protocol 30
Figure 4.2: Read of a (a) Dirty (b) Clean-exclusive line 36
Figure 4.3: Proc. bus transactions on a read from (a) Dirty (b) Clean-exclusive 40
Figure 4.4: Back-to-back latency of writes to shared in terms of D 42
Figure 4.5: Back-to-back latency of writes to shared in terms on N 43
Figure 5.1: Contention micro benchmark 47
Figure 5.2: The open queueing model 50
Figure 5.3: The closed queueing model 51
Figure 5.4: Restart latency under memory contention 57
Figure 5.5: Restart latency under processor bus contention 58

List of Tables

Table 3.1: The cycle breakdown of a back-to-back load 26
Table 3.2: Restart latency on Origin systems of various sizes 27
Table 3.3: Restart and back-to-back latency on UE 4000 28
Table 4.1: Back-to-back latencies of reads under various CC cases on Origin 32
Table 4.2: Restart latencies under various CC cases on Origin 33
Table 4.3: Pipelined memory b/w under various CC cases on Origin 33
Table 4.4: Back-to-back latencies of writes to lines shared in 2 other proc. 34
Table 4.5: Back-to-back latencies of writes to lines shared in 3 other proc. 34
Table 4.6: Back-to-back latency of a write wt sharers at most 2 hops away 35
Table 4.7: Back-to-back latency of a write wt sharers at most 3 hops away 35
Table 4.8: Cycle breakdown of a read from local clean-exclusive 37
Table 4.9: Cycle breakdown of a read from local dirty-exclusive 39
Table 5.1: Restart latency under memory contention 48
Table 5.2: Restart latency under proc. bus contention 49

Chapter 1

Introduction

1.1 Motivation

Many real-world applications have working sets which do not fit into the largest caches of modern microprocessors. With the increasing performance gap between processors and memory systems [4], these applications are facing a memory bottleneck. For example, for a miss rate of 4% and a miss penalty of 50 processor cycles, 68% of the time is spent servicing memory requests[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 [5], [9], [29]. While these techniques can dramatically improve the performance of some applications, they do not operate as effectively in others.

A shared-memory multiprocessor consists of a number of processors which see a large shared address space that is directly accessible by load and store operations. The processors may have private caches. The general architecture of a shared-memory multiprocessor with caches is shown in Figure 1.1. *P0* through *Pn* represent the processors of the system. Memory banks *M0* through *Mq* provide the storage for the shared address space while caches *C0* through *Cn* provide fast access to copies of the memory data to each individual processor. A coherence protocol is required to keep the caches consistent. The

1. The miss rate is defined as the ratio of the number of cache missing instructions to the total number of memory loads and stores. A superscalar processor executing one load/store per cycle, and blocking on cache misses is assumed.

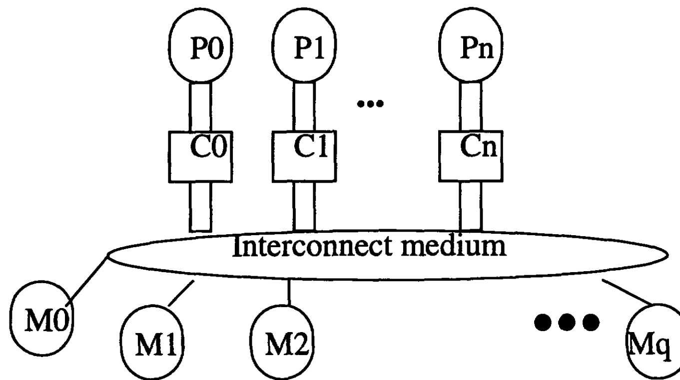


Figure 1.1: A general architecture for a cache-coherent multiprocessor.

interconnect medium could be a bus, a network or some other arrangement, such as a hierarchy of buses.

In such shared-memory multiprocessors, contention for shared resources and the effects of the cache coherency protocol can adversely affect the performance of the memory system, so that applications may experience an even greater memory bottleneck. Because of queueing delays, the memory performance perceived by a given processor can degrade considerably under contention. 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 a memory access incurred by a given processor.

Traditionally, memory performance has been characterized in terms of latency and bandwidth. Memory latency is the time from when the processor requests data to when the memory responds. In cache based systems, the memory latency would include the time from when the processor stalls on the cache miss to the time it receives the entire cache line from memory and restarts execution. Memory bandwidth is the rate at which a given processor can access memory. Memory bandwidth would be computed as the cache line size divided by latency as defined above.

This thesis refines the traditional definition of memory performance to address the characteristics of modern processors. Typically, the miss data represents only a small fraction of the cache line received from memory. In modern processors, on a given miss, the perceived latency could include the penalty of receiving either an entire cache line, or only the miss data. In light of these considerations, for modern processors, two distinctive latency parameters are of importance: *back-to-back latency*, which represents the time until an entire cache line has been received by the processor, and *restart latency* which represents the time until the requested fraction only of the cache line has been received. While previously bandwidth could be computed from the cache line size and latency, in modern systems, because of memory pipelining and the ability of the processors to support multiple outstanding requests to memory, this is no longer the case. With the decoupling of bandwidth from latency, *pipelined memory bandwidth* has become an important memory performance metric.

In light of the sophistication of current processors and systems, and the complexity of the performance space in cache-coherent multiprocessor systems, memory performance has also become difficult to measure. This thesis introduces a methodology for measuring memory performance in the context of modern microprocessors and cache-coherent multiprocessor systems.

1.2 Previous Work

For the past years, normal practice has been to select a set of real user applications and compose them into suites of benchmarks to evaluate computer systems. Such suites are the SPEC benchmarks [25], Perfect Club [1] and the SPLASH benchmarks [24]. While repre-

sentative of some real applications, these benchmarks do not shed light onto specific machine performance bottlenecks and onto the performance of other applications which are not very similar to the given benchmarks and are not included in the benchmark suite.

Another approach to measuring performance is based on micro benchmarks, which are small benchmarks each measuring a particular aspect of the system under controlled conditions. After having introduced it as a general philosophy for performance characterization based on an abstract high-level machine model [21], Saavedra developed a memory micro benchmark useful for measuring and analyzing memory hierarchy performance on real machines. His micro benchmark reveals the impact of cache performance and the cost of local and remote memory accesses [22]. One limitation of this micro benchmark is its performance metric which does not compute either the memory latency, or the pipelined memory bandwidth. In addition, by measuring contention in terms of the interaction between various processors, instead of the bandwidth consumption produced by these processors, the micro benchmark cannot estimate the memory performance under contention conditions selected by the user. Finally, the micro benchmark does not take into account the impact of cache coherency on memory performance.

McVoy's lmbench [17] is a popular suite of portable micro benchmarks which measure latency and bandwidth of data movement among the processor and the memory, network, file system and disk. However, lmbench is restricted to uniprocessor performance characteristics, and its performance metrics assume a simplistic view of the microprocessor and memory system designs.

McCalpin's STREAMS micro benchmarks [16] measure the memory bandwidth obtainable by user codes. Targeted at sustainable unit-stride memory bandwidth measurements, STREAMS is useful for predicting performance on unit-stride vectorizable code. However, STREAMS does not offer a complete solution to memory performance since it

does not measure memory latency. In addition, it does not measure the pipelined memory bandwidth under specific cache coherency and contention conditions.

1.3 Approach

Methodologies for performance analysis can be experimental, emulation-based or analytic [18], [19], [7], [6]. On the one hand, the analytic and emulation-based methods tend to be too simplistic or too complex. When simplistic, they ignore important architectural details, and when too complex, they are generally hard to develop or use. In addition, they are only useful when they can be validated against real systems. By contrast, experimental methods account for all the hardware features and performance factors at no cost to the developer or user of the particular methodology. On the other hand, the analytic and emulation-based methods can be used in the absence of real hardware, and can provide insights into cases which may be hard to reproduce experimentally.

Our methodology, which combines experimental and analytic performance tools, is based on micro benchmarks and queueing theory. Micro benchmarks allow users and hardware designers to characterize the memory performance of different machines in a uniform way, while providing a level of detail which is usually associated with hardware monitors [22]. The queueing model predicts memory performance under contention levels which cannot be reproduced experimentally.

The micro benchmark suite presented in this thesis addresses memory hierarchy performance only. While the performance of the processor, disk, graphics subsystem, the I/O and the operating system is also relevant to evaluating the overall performance of a computer system, we chose to focus on memory subsystem performance because of its increased complexity and importance. Our micro benchmarks overcome the limitations of

previous work. To achieve this, they 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.

The suite is aimed at memory system designers, application writers, and end-users. Designers can use the suite to validate the predicted performance of the memory system and 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.

1.4 Thesis Outline

The thesis is organized as follows. Chapter 2 details the variations in memory performance as measured by a single processor. Chapter 3 presents the micro benchmarks used to capture these effects. Chapter 4 introduces cache coherence and its effects on latency and bandwidth. Chapter 5 adds the complication of queueing within the memory subsystem. In all of these chapters, performance results of the benchmarks on the SGI Origin 2000 [10], [11] are given, along with selected results on the SUN Ultra Enterprise 4000 (UE4000). Chapter 6 presents a discussion of future improvements in the micro benchmarks and concludes the thesis.

Chapter 2

Uniprocessor Memory Performance

Traditionally, memory performance has been characterized in terms of latency and bandwidth. Memory latency is the time from when the processor requests data to when the memory responds. Memory bandwidth is the rate at which a given processor can access memory.

Processors using caches move memory data in cache line chunks, that are typically 32-128 bytes in size. For a given cache miss, a *critical word* is the fraction of the cache line representing the data which the processor requested from the memory. Until recently, processors would stall on the instructions dependent on the cache miss, until the entire cache line had arrived from memory and filled the appropriate line in the cache. In these systems, memory latency included the penalty of the cache line fill. With the larger cache line sizes of current microprocessors supported by innovations in bursty DRAM technology, the fill time has become a considerable fraction of the overall latency. For example, on a system utilizing 128 byte cache lines and capable of filling at 800MB/sec, the fill time after the initial critical word has been received is on the order of 160 ns. Assuming a critical word access time of 200 ns, this represents 44% of memory latency. In this thesis, this memory latency which includes the penalty of the entire cache fill is referred to as *back-to-back latency.*

To reduce memory latency, an optimization has been introduced in modern processors, known as *critical word restart,* which allows them to restart the computation of dependent instructions upon receiving the critical word [29]. In these processors, memory latency

includes the time to receive the critical data word but does not include the penalty of the entire cache fill. We refer to this measurement as *restart latency.*

While critical word restart allows processors to continue computation upon receiving a fraction of the cache line, it does not, by itself, prevent stalls of any cache accesses issued upon restart and which conflict with the on-going cache fill. To resolve such conflicts, modern processors must implement a mechanism to select, on a given clock cycle, between servicing new memory requests from the processor and servicing on-going cache fills. This selection mechanism is shown in Figure 2.1. A time diagram describing the succession of events which occur when a fill and a miss detection event conflict at the cache is given in Figure 2.2.

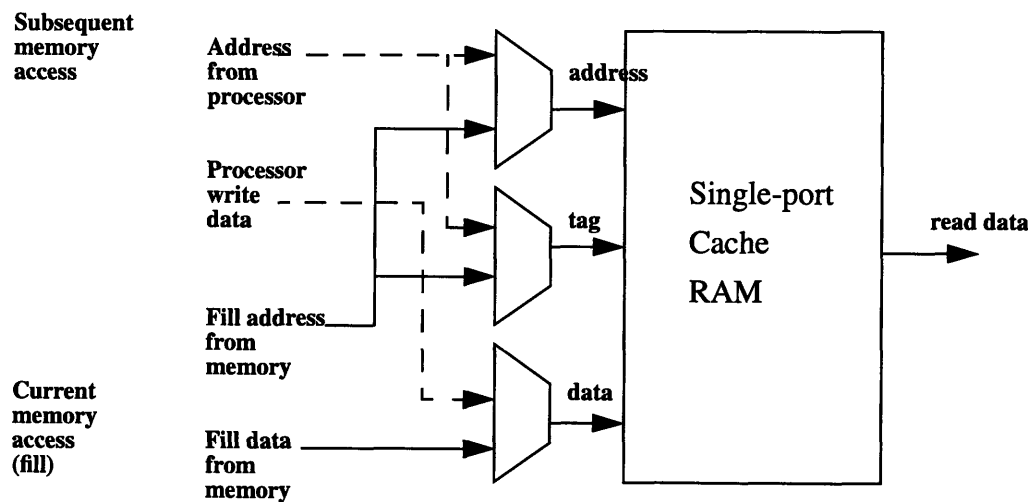


Figure 2.1: Cache contention. A cache mechanism selects between servicing new memory requests from the processor and servicing fills of current memory requests.

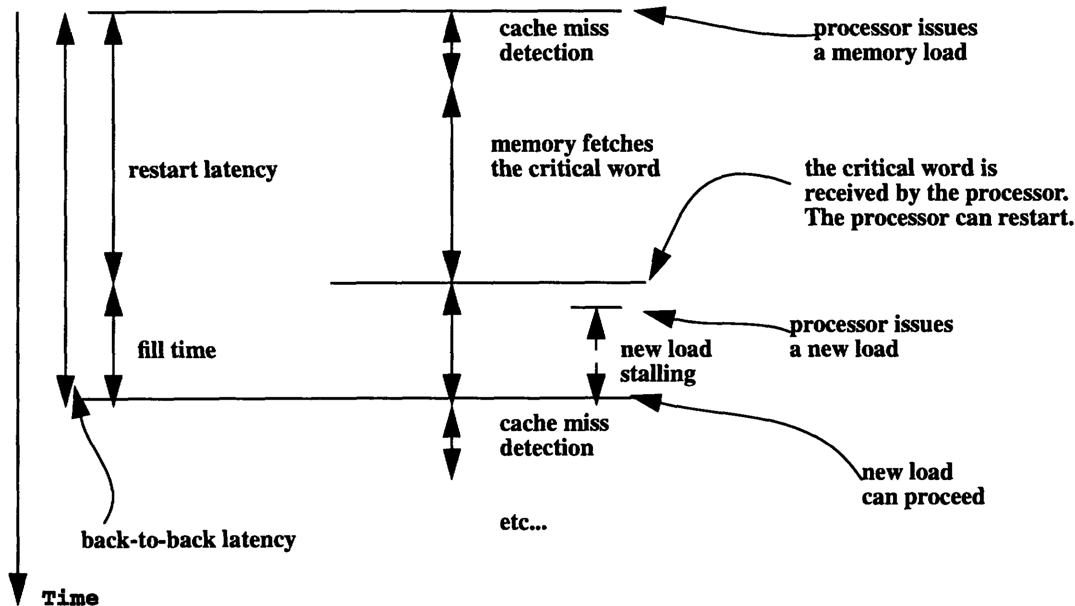


Figure 2.2: Cache conflict between two loads. The second load must stall at the cache until the fill corresponding to the first load completes.

To prevent such stalls, additional processor features are required which would allow it to interleave cache miss detection with cache fills. Some of these features are:

1. a multi-level cache hierarchy, in which the first level implements critical-word restart and uses smaller cache line sizes than the other caches in the hierarchy.
2. cache refill buffers which use idle cache cycles to complete the cache fills.
3. a dual port cache.

On systems which do not feature these optimizations, the worst latency incurred by a miss which interferes with some previous miss at the cache includes the fill time of the previous miss. Consequently, this latency effectively equals the back-to-back latency.

Until recently, systems were capable of moving only one cache line during one back-to-back latency time unit. With the introduction of lock-up free caches [9], which allow

processors to issue new requests while previous requests are still outstanding, modern systems can overlap memory fetches, and hence transfer more than one cache line in a latency time unit. As a consequence of this, a higher memory bandwidth can be achieved. To exploit this processor optimization however, applications have to ensure that the memory accesses are not dependent. On systems with lock-up free caches, the memory bandwidth is limited by one of two factors:

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

While previously bandwidth could be computed from the cache line size and latency, in modern systems, because of pipelining, this is no longer the case. With the decoupling of the bandwidth metric from latency, and the variations in the latency perceived by a single processor, all these three metrics -- back-to-back latency, restart latency and pipelined bandwidth -- are important for characterizing the performance of a memory system.

For a given application, any parameter of memory performance may be appropriate, or they may all apply in various phases of the program. For example, for many scientific applications written in FORTRAN, pipelined memory bandwidth is the most important performance parameter since the data access patterns of these programs consist of array references which are typically not inter-dependent. For applications in which the cache misses cannot be overlapped either because they are infrequent or because of inter-dependencies, restart latency and back-to-back latency are the most important performance metrics. Their relative importance depends upon the data access patterns and fill time. For example, on systems with a fill time on the order of 150 ns, running at 200MHz and executing one memory load or store per cycle, the fill is hidden if the processor executes 30 or

more load or store instructions before the next miss[1]. Likewise, back-to-back latency is appropriate if the processor executes much fewer than 30 load or store instructions in between memory misses. More generally, back-to-back latency pertains to codes with bursty memory accesses, while restart latency pertains to codes in which the memory accesses are isolated from one another. Given that a miss every 30 load or store instructions is a 3.3% miss rate which characterizes many applications [4], it is likely that restart latency penalties are seen more frequently across applications than back-to-back latency penalties[2].

In conclusion, based on the features of current architectures and the data access patterns of various applications, memory performance can be characterized in terms of the following parameters:

1. **back-to-back latency**: the time to service a cache miss, assuming that the instructions before and after the miss are also cache misses. This performance parameter includes the cache fill penalty, either because the processor is unable to restart upon receiving the critical word or because of conflict with the previous miss at the cache. Back-to-back latency is measured in time units, typically in nanoseconds (ns).

2. **restart latency**: the time to receive the critical word, assuming all previous cache misses have completed and consequently no interference with other misses in the systems is observed. Restart latency is measured in time units, typically in nanoseconds (ns).

3. **pipelined bandwidth**: the sustained rate at which a single processor can issue requests to the memory system and place the memory data into its caches. Pipelined band-

1. This argument assumes the processor blocks on cache misses. For a non-blocking processor, the number of instructions 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, this number is ~(30 + 30/4) = ~38

2. See footnote 1. For a non-blocking processor, the miss rate would have to be 1 in (30 + 30/4) or ~2.7%.

width is typically measured in megabytes per second (MB/s).

The following chapter describes our methodology to measure each of these memory performance parameters.

Chapter 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 processor execution and memory operations. Micro benchmarks aimed at these three performance metrics are presented in the following three sections. The micro benchmarks in this chapter measure the performance of memory reads only. Since most processors do write allocations, no micro benchmarks for measuring write performance are necessary, given that write performance is similar to read performance. The micro benchmarks also do not measure the impact of writebacks on memory performance. We consider supporting such measurements as part of our future work. The results of running our micro benchmarks on the SGI Origin 2000 and SUN UE 4000 are described in the fourth and fifth sections respectively.

3.1 Pipelined Memory Bandwidth

To measure memory pipeline performance, a micro benchmark must cause multiple memory accesses which are all cache misses. The following simple reduction loop achieves this goal:

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

int
do_dummy(int j) {
return j;
}
```

The *StartTime()* and *EndTime()* routines are calls to the Unix routine *gettimeofday*, which reads the system clock. To compensate for the coarse system 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.

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 line size). At this point, the execution time of the loop is relatively constant because each memory access is a cache miss and the same number of memory accesses are performed for all strides. A more detailed discussion of this argument is contained in [22].

The code given above does not compute the peak memory bandwidth but the sustained memory bandwidth which is achievable by user codes. This sustained memory bandwidth can be limited by the processor rather than the memory subsystem itself, in particular by the maximum number of requests that the processor can have outstanding simultaneously to the memory subsystem.

In our code, the 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 [17] includes such a benchmark. Saavedra's micro benchmark [22] measures pipelined memory performance on read-modify-write operations at various strides. On machines with write-allocate caches, such operations can generate three memory references (i.e. one for the read, one to allocate the write cache line in the cache, and one to writeback the data). John McCalpin's STREAMS benchmarks [16] computes memory pipeline performance on four unit-stride floating-point vector codes. With modern high-performance microprocessors these floating-point kernels are dominated by cache misses, hence the performance results are the same as 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 the loads dependent upon the value from the previous load, the cache misses cannot be overlapped at the memory. A simplified version of this code is shown below:

```
double
BackToBackLatency(int *arry, int num_reads1) {
    int i, *j;
    StartTime();
    j = arry;
    for (i = 0; i < num_reads; i++) {
        j = (int *)*j;
    }
    EndTime(time);
    return ( time/num_reads );
}

void
initialize_memory(int *arry, int num_reads,
            int stride)
{
    int i, *next, size;
    size = num_reads * stride;
    for (i = 0; i < size; i = i + stride) {
        next = &arry[i + stride];
        arry[i] = next;
    }
}
```

The dependency of each load on the previous load's data guarantees that no two memory accesses are in the memory pipeline simultaneously at any given time[2]. 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[3]

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

---

1. Suggested values for *num_reads* are on the order of millions.
2. To limit the amount of system interference at the memory, the micro benchmark must be executed on lightly-loaded systems.

3. The methodology and ideas presented in this section were suggested to me by Daniel Lenoski.

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 [29], and the SUN UE 4000 has a full/empty 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 3.1.

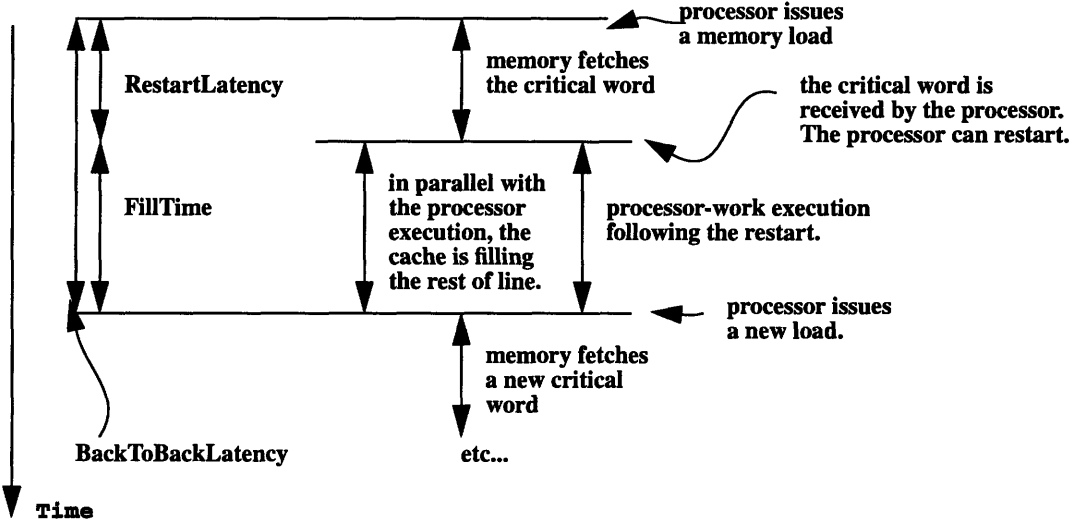


Figure 3.1: The restart latency micro benchmark and the derivation of restart latency from back-to-back latency and fill time.

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 below iterates over the processor-work time to identify the proper refill overhead:

```
double
RestartLatency(int **arry, int num_reads, double error, int dummy)
{
   int last_time = 0, cur_time = 0, num_iters = -1, *j, i;

   while ((cur_time <= last_time + error) || (last_time == 0)) {
      last_time = cur_time;
      num_iters++;
      StartTime();
      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;
```

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* and

error are on the order of millions and tens of milliseconds respectively. To limit the amount of system interference at the memory, the micro benchmark must be executed on lightly-loaded systems.

To estimate the time spent in the memory system only (including the cache) one must subtract the overhead of the load issue instruction which typically amounts to 1 processor cycle.

3.4 Results on the SGI Origin 2000

The Origin 2000 has a nonuniform memory access (NUMA) architecture [15], in which the nodes of the system include both processors and memory, and are connected by means of a scalable Craylink network [10], [11]. This architecture is shown in Figure 3.2. 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. The Origin 2000 comes with MIPS R10000 processors, running at 195MHz.

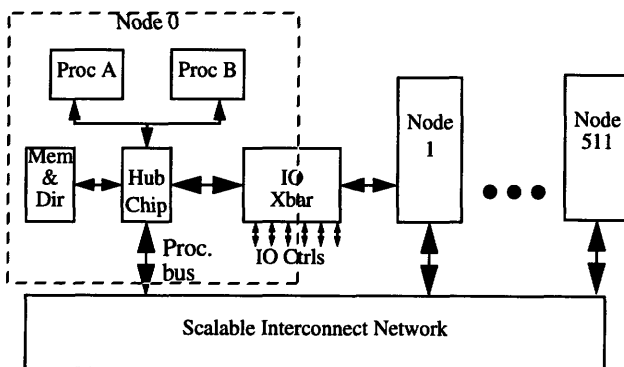


Figure 3.2: Origin block diagram

The secondary cache runs at 2/3 of the processor frequency (130MHz) and the processor bus at 1/2 of the processor frequency (98MHz).

The secondary cache 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. The procedure to acquire the logic analyzer data is described in Appendix A. A breakdown of this latency is given in Table 3.1.

Table 3.1: The cycle breakdown of a back-to-back load.

Proc. Bus Cycles	Activity
10	R10000 proc. finished filling in the L2 cache with the previous load's data; Subsequently, R10000 proc. detects L2 cache 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 cache line onto the proc. bus; R10000 proc. places the L1 and L2 cache lines into the L1 and L2 caches; R10000 detects an L1 cache 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 line being larger than the L1 cache line by 12 double words, where a double word corresponds to the size of the data path connecting the processor to the memory subsystem. The extra cycle is the result of an additional bus turn-around overhead in the back-to-back latency micro benchmark.

The topology of two Origin 2000 systems [2] is shown in Figure 3.3.

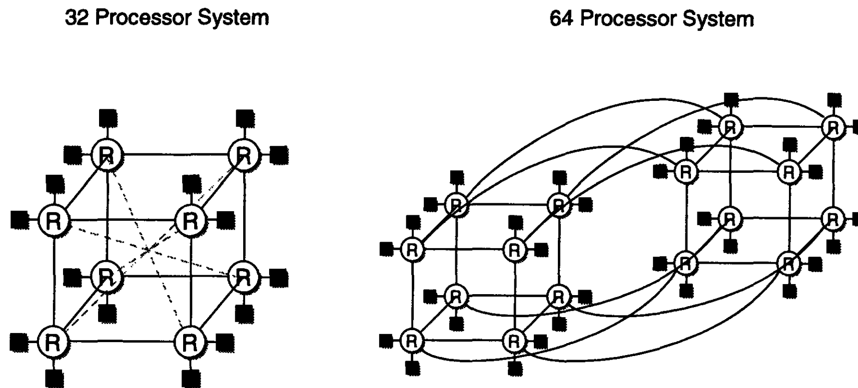


Figure 3.3: Origin topology, a hypercube. The nodes labeled R represent routers.

When the home is the closest remote memory, back-to-back latency takes 687 ns (67 processor bus cycles), while restart load latency takes 564 ns (55 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 3.2. The average values in the table assume a uniform

Table 3.2: Restart Load Latency on Origin Systems of various sizes

System size (CPU Count)	worst-case number of routers to home memory	worst-case latency (ns)	% home memory that is local	avg. number of routers to home memory	avg restart latency (ns)
2	0	338	100%	0	338
4	1	564	50%	0.5	451
8	2	759	25%	1.25	628
16	3	862	12.5%	1.875	719

distribution for the home memory locations of the requested data across the nodes of the system.

Finally, the pipelined memory bandwidth results are 536 MB/s for local memory and 417 MB/s ns for the closest remote node.

3.5 Results on the SUN UE4000

The SUN UE4000 processor is bus-based (hence all memory is equally far away from any processor). The cache size measured is 64 bytes. The latency results, which are shown in Table 3.3, show a difference of only 18 ns between back-to-back latency and restart latency. This result corresponds to our expectations since the data path size is 16 bytes, running at 84 MHz, and hence the cache fill penalty after the critical word has been received, which is on the order of 1-2 system bus cycles, is between 12-24 ns. In terms of system bus cycles, the back-to-back latency is ~26 and the restart latency is ~25 respectively.

Table 3.3: Restart and Back-to-back load latencies on UE4000

Restart Latency (ns)	Back-to-back Latency (ns)
295	313

Chapter 4

Cache Coherency

Caching increases performance by reducing the effective memory latency. However, when multiple processors are allowed to maintain local copies of the memory data, 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

To measure the effects of coherency, the micro benchmarks are modified to establish a particular caching state prior to the execution of the memory test. While there are many different coherence protocols, we focus on the Illinois [20] or MESI [26] protocol because it is one of the most commonly used. As shown in Figure 4.1, 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 invalid state as *unowned.* The exclusive state is also known as *clean-exclusive*, since it implies that the memory is up-to-date since the given processor has not written to the particular line. The coherency protocol affects the performance of a memory load/store when the requested cache line is in the modified state, and in some systems, when in the clean-exclusive state. In addition, the coherency protocol affects the performance of a store operation when the line is shared because an exclusive copy must be obtained by first invalidating the other cached copies.

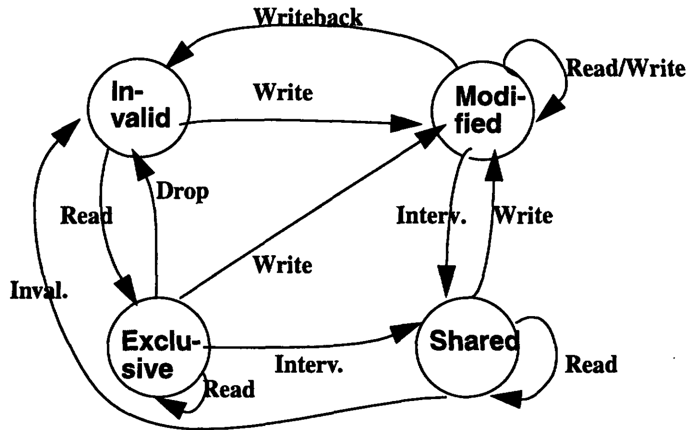


Figure 4.1: The cache line states and the cache line state transitions of the MESI protocol.

The micro benchmarks measure the performance of a "master" processor M which executes a set of memory operations. The other processors in the system, which we call P1, P2, etc. are used to establish the cache coherency state of the data accessed by M. To use these processors, threads are created and placed on the various processors using library thread routines [27].

To initialize the dirty state:

• P1 simply writes the data to be accessed by M.

To initialize the clean exclusive state:

• P1 writes the data to be accessed by M
• Then P1 writes an array of size equal to the secondary cache size. This second operation moves the data out to memory[1].
• Subsequently, P1 reads the data and hence establishes the clean exclusive state.

Finally, to initialize the shared state:

1. A direct-mapped cache or a least-recently-used (LRU) cache replacement algorithm is assumed.

- P1 first writes M's data.
- Subsequently, P2, P3, etc. read it. This sets the state to shared.

Depending on the virtual-to-physical address mapping scheme, different array addresses may be mapped onto the same cache location. By removing sections of the data from the cache which the micro benchmark assumes to be present in the cache, this limits the ability of the micro benchmark to measure the memory performance under the desired cache coherency conditions. To counteract this problem, the initialization routine is modified to request pages from the operating system which are large enough as to hold the entire experimental data. On systems where this is not possible, depending on the virtual-to-physical address mapping scheme, the user might have to reduce the size of the array if intended to be placed in the cache in its entirety. Moreover, the cache coverage aimed at removing data from the cache is to be done using arrays of sizes much greater than the cache size. Care must be taken in reducing the array size as not to increase timing sensitivity to system load.

The coherence state is re-initialized before every "while" loop iteration in the micro benchmark measuring the restart latency. This is done to preserve the consistency of the cache coherency state across successive iterations.

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);
}
```

The store corresponds to the second instruction in the loop. By setting the address of the write to the value of the previous write, the first loop instruction ensures that the successive stores do not overlap.

To verify that the initialization routine produced the correct coherence state on the Origin system, we counted the number of cache misses, write-backs and invalidates caused during M's execution. These checks relied on the ability of the processor to report the count of various processor events such as those we mentioned above. In addition, the 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.

4.2 Results on the Origin 2000

4.2.1 Reads

These are the cache coherency results for the unowned, exclusive and modified cases. In a MESI protocol, such as the one on the Origin 2000, the performance of a read to a line in the shared state is the same as when the line is in the unowned state, and hence is not included with our results. The following table lists the back-to-back latencies:

Table 4.1: Back-to-back latencies of reads under various CC cases on Origin.

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

a. The "home" is the memory storing the directory information associated with the requested cache line.
b. The "owner" is the processor which is caching the line requested by the read, either in the modified or the exclusive state.
c. By "remote 1" is meant a processor in the closest remote node to the requesting processor.

The following table lists the relevant restart latency values[1]:

Table 4.2: Restart latencies of reads under various CC cases on the Origin 2000

Home	Owner	Unowned (ns)	Exclusive (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 some of the bandwidth results:

Table 4.3: Pipelined Memory B/w under various CC cases on Origin 2000

Home	Owner	Unowned (MB/s)	Exclusive (MB/s)	Modified (MB/s)
local	local	536	381	304
remote 1	local	417	298	287

4.2.2 Writes

On processors using write-allocation (i.e., a write consists of reading the line from memory prior to writing it into the cache), writes and reads to memory incur the same penalty, except in the case of shared lines which, for writes, require some invalidations. While in

1. Only the results in the first row were measured experimentally. The rest of the results were derived from the corresponding back-to-back latency results and the fill times of each cache coherency state. These fill times were measured by subtracting the restart latency from the back-to-back latency results corresponding to the case where both the home and the owner are local. Notice that the fill time for the clean exclusive case is much smaller than the fill time for the unowned and dirty cases. In the clean exclusive case, the requestor cannot use the critical word until an acknowledgment from the owner has been received,

Table 4.4: Back-to-back latency of writes to lines shared in 2 other processors

Location of sharers	Back-to-back latency (ns)
local, remote 1	1320
remote 1, remote 1 (sharers on the same node)	1320
local, remote 2[a]	1438
remote 1, remote 2	1436
remote 2, remote 2 (sharers on the same node)	1438
remote 2, remote 3	1542
remote 3, remote 3 (sharers on the same node)	1542
local, remote 4	1672
remote 2, remote 4	1674

a. By "remote x", where x>1 is meant a processor located x-1 routers away.

bus systems, the invalidates are broadcasted through the common bus, in a NUMA multiprocessor such as the Origin 2000, separate invalidations must be sent to each of the sharers. As a consequence, in a NUMA multiprocessor, the latency of a write to a shared line depends on the number and the relative locations of the sharers. Table 4.4 and Table 4.5 show the back-to-back latency of writes to lines shared in 2 and 3 other processors respectively.

Table 4.5: Back-to-back latency of writes to lines shared in 3 other processors

Location of sharers	Back-to-back latency (ns)
local, remote 1, remote 1	1355
remote 1, remote 1, remote 2	1435
remote 1, remote 2, remote 2 (sharers are on different nodes)	1620

Table 4.6 and Table 4.7 show the variations in back-to-back latency of writes to shared lines, with the number of sharers located at some maximum distance N. While varying N,

the distance D to the furthest away sharer(s) is kept constant, first at the value 2 and then at the value 3.

Table 4.6: Back-to-back latency of a write to shared line; sharers at most 2 hops away

N[a]	Back-to-back latency (ns)
1[b]	1435
2	1620
3	1663
4	1664

a. The total number of sharers is 3 in the first three experiments and 4 in the last experiment.
b. 1 refers to the one sharer at the furthest distance 2. The coherency state of the line is not clean exclusive since there are other processors sharing the line, but which are located closer to the master.

Table 4.7: Back-to-back latency of a write to shared line; sharers at most 3 hops away

N[a]	Back-to-back latency (ns)
1	1542
2	1738[b]
3	1743

a. The total number of sharers is 2 in the first two experiments and 3 in the last experiment.
b. This result is considerably larger than the result measured in the case where only 1 sharer is located at D=3. While some difference between these results is expected, the measured ~200 ns difference is, intuitively, too large. We have not yet been able to explain this rather surprising result on the Origin 2000.

4.3 Discussion of Results

4.3.1 Reads

In the Origin 2000, the caches and the main memory are kept coherent through a directory structure implemented in hardware [10], [11]. 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 exclu-

sive 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 the figure below:

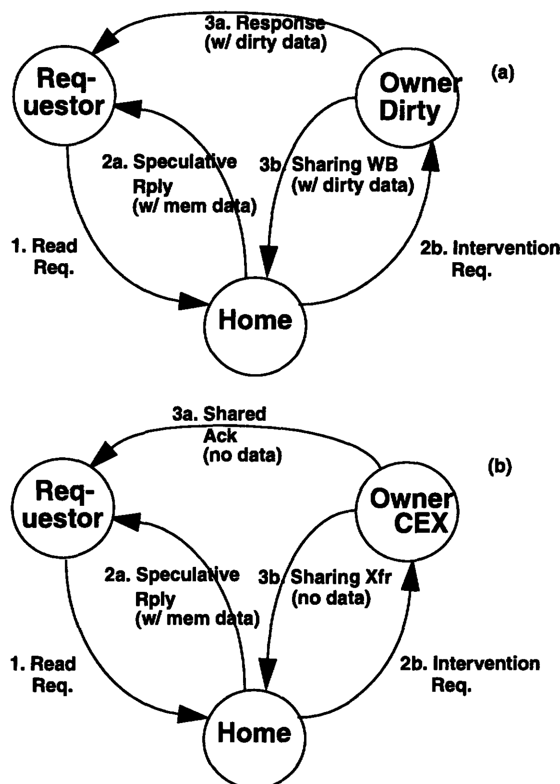


Figure 4.2: Read of a (a) Dirty (b) Clean-exclusive line

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 as measured by our micro benchmarks: 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 can be considerably longer. First, the owner has to wait until an intervention arrives from memory before it can look up the necessary information in its cache. Since the processing of the intervention cannot happen in parallel with the memory fetch, this overhead is considerable on systems with slow processor intervention processing. Secondly, in the dirty case, the data must traverse the processor bus one additional time in the form of a speculative response from memory since no mechanism (such as a dirty bit line on the snoopy buses) exists to disable this response.

4.3.1.1 Requestor and Owner Share the Processor Bus

The cycle breakdown gathered using the logic analyzer (see Appendix A), of a read from clean-exclusive which was and which is shown in Figure 4.3(b), is given in Table 4.8. While in the unowned case it takes a minimum of 10 processor bus cycles to issue the read

Table 4.8: Cycle breakdown of a read from local clean-exclusive

proc. bus cycles	activity
11	Requestor detects the L2 cache miss and requests processor bus
2	Bus granted
4	Requestor issues read request
24	Intervention to owner from memory
1	Spec. header
16	Spec. data from memory
3	Ack from owner
5	Requestor grant taken away; the acknowledgment from the owner into proc. interface outbound queue; the acknowledgment into the proc. interface inbound queue
66	total

request, in the clean exclusive and dirty-exclusive cases it takes a minimum of 17 cycles. The difference comes from the fact that the requestor must ask and wait for a bus grant

before each read. When the processor interface (which connects the processor bus to the Hub) does not need the bus, it releases it to the processor that owned the bus last. As a consequence of the back-to-back nature of our tests, in the exclusive and modified cases where the requestor and owner share the processor bus, it is given to the cache line owner which used the bus last while responding to the preceding memory request. The 7 processor cycle overhead comes from the following:

1. the requestor must request the bus from the processor interface (1 processor bus cycle).
2. the requestor must wait for the bus grant (2 processor bus cycles).
3. finally, the requestor must note it has received a bus grant and consequently place the read request on the bus (4 processor bus cycles).

The table also shows that processing an intervention to a clean-exclusive line takes 20 processor bus cycles (40 processor cycles, or 27 secondary cache cycles) in the R10000. The latency of this operation is surprisingly high since the operation should mainly involve comparing the tags and reading the cache line state.

Although the overhead of intervention processing is high on the Origin 2000, the snooping scheme was not implemented for the reasons discussed above, and also because the incidence of reads from lines in which the owner is local in real applications was considered small relative those in which the owner is not local.

The cycle breakdown gathered using the logic analyzer (see Appendix A), of a read from dirty-exclusive which is illustrated in Figure 4.3(a), is presented in Table 4.9.

Table 4.9: Cycle breakdown of a read from local dirty-exclusive

# proc. bus cycles	Activity
11	Requestor detects the L2 cache miss and requests proc. bus
2	Proc. bus granted
4	Requestor issues request
24	Intervention to owner from memory
1	Speculative header
16	Speculative data from memory
6	Owner requests proc. bus
2	Proc. bus granted
16	Dirty data from owner into proc. interface outbound queue
2	Dirty data goes from proc. interface outbound queue into proc. interface inbound queue
16	Dirty data from proc. interface inbound queue onto the proc. bus
100	Total

Reading a line present in the modified state in the other local processor is expensive because the entire cache line must traverse the processor bus 3 times and in a sequential fashion: once from memory to the requestor, a second time from the owner into the pro-

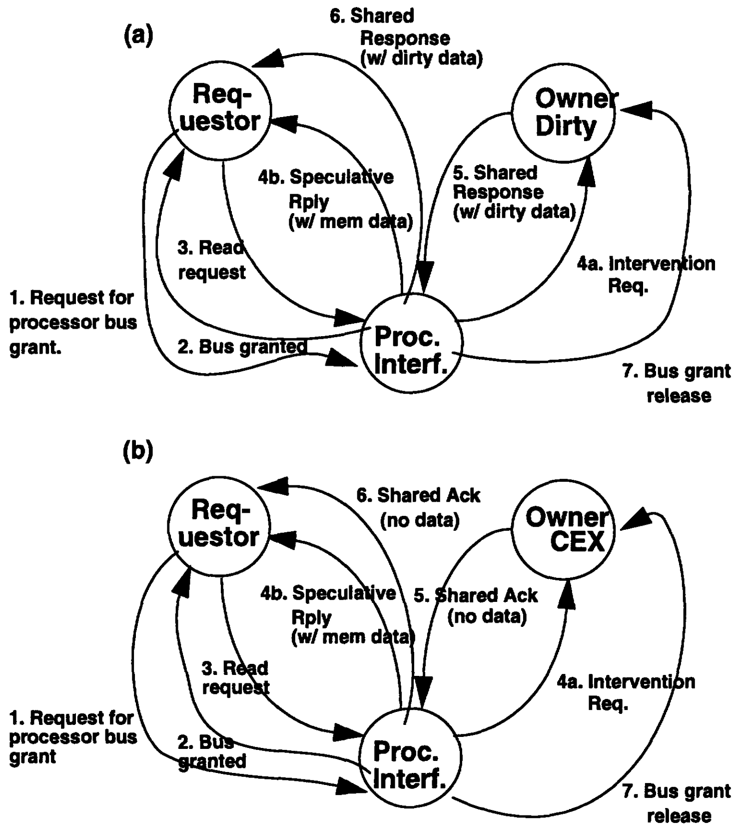


Figure 4.3: Proc. bus transactions on a (a) Dirty Data Read (b) Clean-Exclusive Data Read. In both (a) and (b) the requestor and owner processors share the same proc. bus.

cessor interface outbound queue and lastly from the processor interface inbound queue to the requestor.

4.3.1.2 Requestor and Owner on Different Processor Buses

When the owner is placed on a remote node, less contention at the local processor bus is incurred. However, since the requestor must wait for an acknowledgment from the remote owner and the network penalty is comparable to the bus contention penalty, no improvements in the overall results are noticed. These results are shown in Table 4.1 and Table 4.2. Finally, the worst penalty is paid when the owner is local and the home is remote. Under these conditions both processor bus contention and network latency penalties are incurred.

4.3.2 Writes

To write a line which is shared in other caches, the processor sends a read-exclusive request to memory. If the request succeeds, the memory sends invalidations to the sharers and an exclusive reply with a count of the invalidates pending to the requestor. Once the requestor receives all the invalidate acknowledgments, it fills the cache line with the memory data and sets its state to modified.

Since the overhead from processing the invalidates is insignificant compared to the network latency, the latency of a read-exclusive to a shared line does not depend on the number of sharers, but mainly on the maximum number of hops D from M to a sharer, and on the number of sharers N on different nodes located at distance D from M.

As shown in Figure 4.4, the latency increases linearly with D as it depends primarily on the number of routers traversed by the invalidate to the furthest remote sharer[1].

For small systems (32 processors), when N is small, the performance drops with increases in N, while for large N, it stays fairly constant. Figure 4.5 shows the dependency with N of back-to-back latency for writes to shared lines.

1. While varying D, N was held constant to 2.

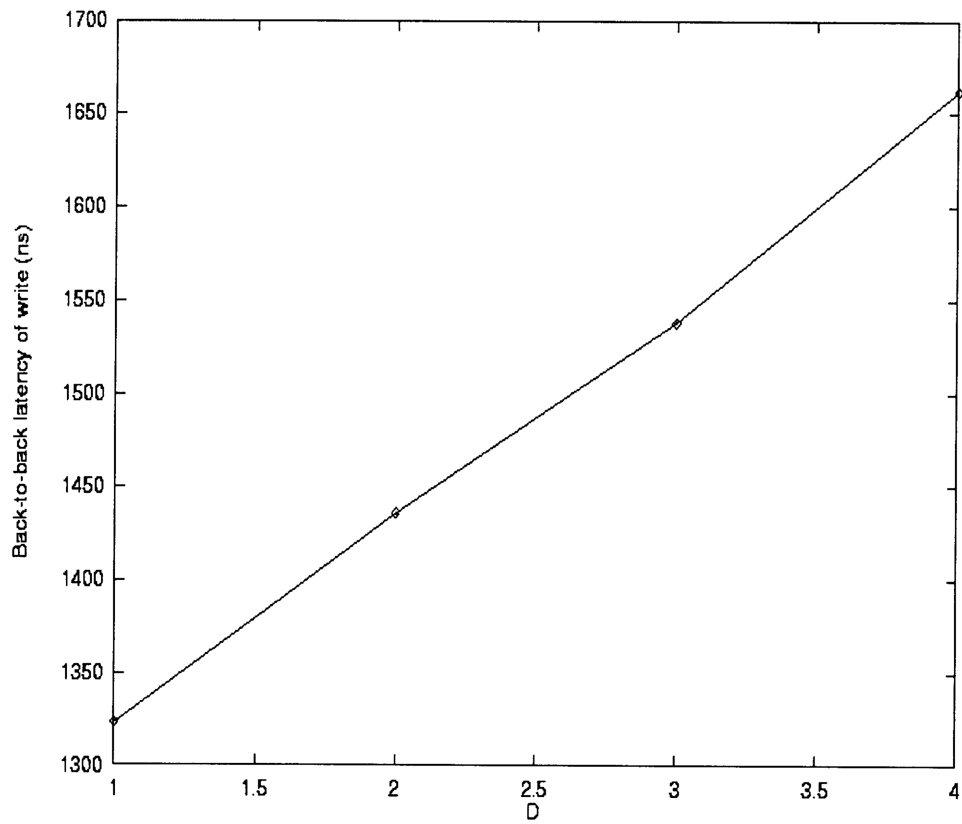


Figure 4.4: Back-to-back latency of writes to shared lines in terms of the distance (D) to the furthest sharer on the Origin 2000.

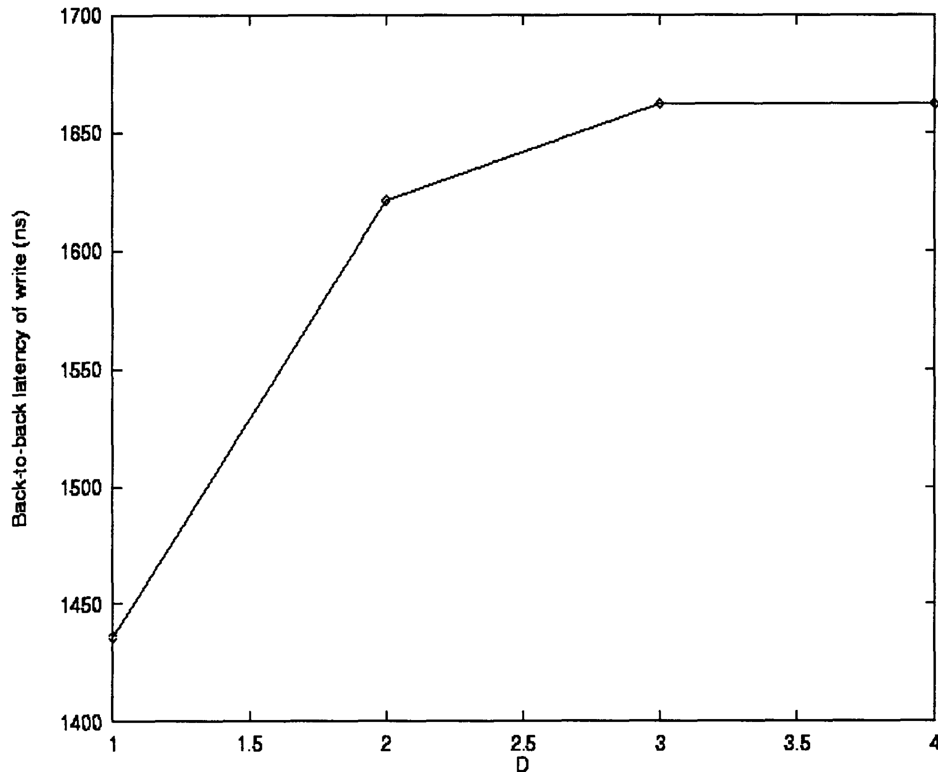


Figure 4.5: Back-to-back latency of writes to shared lines in terms of the number of sharers (N) located on different nodes at maximum distance 2 on the Origin 2000.

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.

The experimental results presented are relevant for Origin systems of sizes smaller than 32 processors. While results for larger systems are required for a thorough understanding of the impact of cache coherency on memory performance, such results could not

be acquired because, until recently, no Origin systems larger than 32 processors were available.

Chapter 5

Contention

Large-scale parallel applications typically induce contention when run on shared memory multiprocessors. It is easy to see how such contention can occur at any physically shared resource, such as: a shared processor bus, the memory, or the network. By introducing contention at user-specifiable system locations and measuring its impact on memory performance, our micro benchmark suite provides users with additional insight into machine performance on real multiprocessor applications and allows comparison of various architectures.

Predicting the performance of an arbitrary user application on a shared-memory multiprocessor requires knowledge of the memory performance seen by a processor under arbitrary levels of contention. While user applications can be written to generate some contention conditions, no application exists which generates any arbitrary contention value under which performance can be measured.

In light of these considerations, we have created a methodology, based on experimental and analytical tools, for estimating the memory performance for arbitrary contention values. The experimental aspects of this methodology are presented in Section 5.1 and the analytical aspects are presented in Section 5.3 of this chapter. Section 5.2 details the experimental results acquired on the Origin 2000.

5.1 Micro Benchmark Implementation

The memory test is modified to allow various threads to interfere with the 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 and independently of each other. All the 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 to be generated, between successive memory accesses, 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 5.1 illustrates the above discussion:

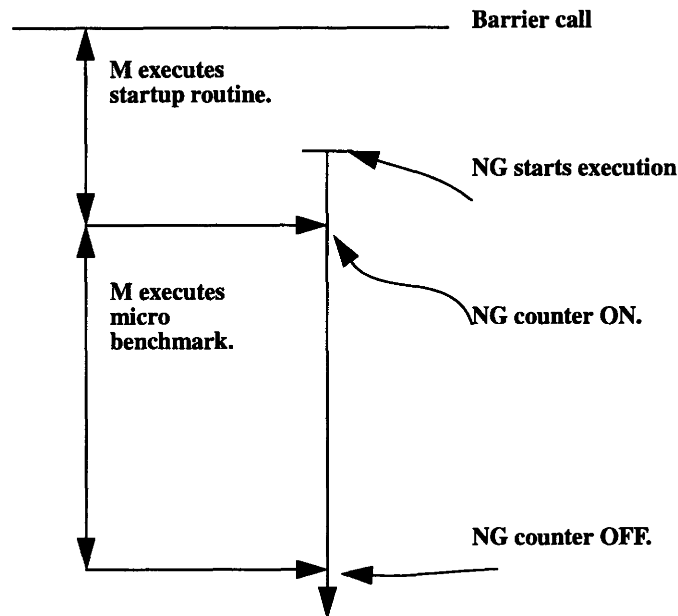


Figure 5.1: Contention micro benchmark.

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.

On shared-memory multiprocessors, the most important location of contention is the memory system. In a distributed shared memory multiprocessor with multiple processors connected to one node, another important contention location is the processor bus [14], [11], [13]. To restrict bandwidth consumption to a particular system location, system rou-

tines for process and data placement are used [27]. 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 processes are placed on different nodes and the data on the same physical memory.

5.2 Results on the Origin 2000

The following table lists the memory contention results on the Origin 2000[1]. The NG contention values, and the master restart latency and throughput were all computed during the execution of the micro benchmark.

Table 5.1: Restart latency under memory contention on the Origin 2000.

NG Contention (MB/sec)	Master Restart Latency (ns)	Master throughput (MB/sec)
0	338	271
41	371	253
121	421	231
233	483	207
336	572	181
484	915	122
538	1462	80

1. Only the restart latency results are presented. The corresponding back-to-back latencies can be computed by adding the fill time (measured previously) to restart latencies.

These are the processor bus contention results on the Origin 2000. As in the memory micro benchmark, the NG contention and the master restart latency and throughput were computed at run time.

Table 5.2: Restart latency under processor bus contention on the Origin 2000.

NG Contention (MB/s)	Master Restart Latency (ns)	Master throughput (MB/sec)
0	338	271
13.9	368	255
120	431	227
316	586	178
386	786	139

5.3 Analytic Model

The experimental technique presented in Section 5.1 can be used to measure the memory performance seen by a processor under some finite number of contention values. These values are not set by the user however, but measured at run time. To compute the memory performance under arbitrary, user-selected contention values, some analytic model is required instead. By assuming the M/D/1 model under closed queueing conditions for each contended resource, an expression of memory performance in terms of NG contention at the resource is derived. From this equation, memory performance under any arbitrary contention values can be computed. The peak sustained bandwidth for each resource is also derived, iteratively, based on the curve fit of the micro benchmark data and the analytic data model, evaluated for the particular value of peak sustained bandwidth.

A general diagram describing a queueing model with a single resource is shown in

Figure 5.2.

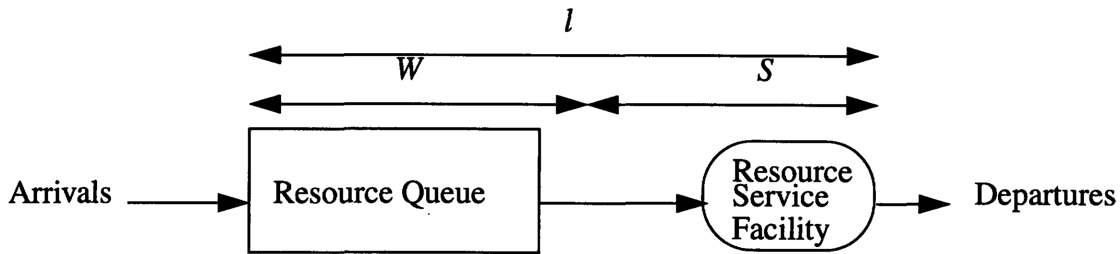


Figure 5.2: The open queueing model. l represents the average response time; W the wait time in the queue; S the service time.

In a queueing system, an important performance measure is the average response time at the resource l which includes the waiting time in the queue W and the service time S:

$$l = W + S \tag{5.1}$$

The wait time W varies with the server utilization U, the type and the model of the particular queue.

Depending on the distribution of arrival and service times, different queueing models apply. Under the conditions of our experiments, the service time is deterministic since all the clients receive the same amount of processing time at a given resource[1]; in light of the independent thread execution, the arrivals can be approximated by randomly distributed exponential variables. Hence the M/D/1 queueing model applies [3].

Queueing systems can be of two types: closed or open. As shown in Figure 5.3, a closed queueing system models networks in which a fixed and finite number of customers

1. Some resources, such as the shared-processor bus see two types of clients: requests, which require a short processing time (for example, a memory request consisting of a memory address, delivered in 1 bus cycle) and replies which require a long processing time (for example, a memory reply including large data typically delivered over multiple bus cycles). Requests are paired with replies. For such resources, each pair of requests and replies can be approximated by one single client whose service time is equal to the sum of the service times corresponding to the request and the reply. Consequently, the deterministic service model still applies.

are present, trapped such that none may leave and no others may enter [8]. By contrast, an open queueing system which is shown in Figure 5.2 permits external arrivals and has infinite populations. While the open queueing model is appropriate for large multiprocessors which generate and support the service of many simultaneous requests to the resource facilities, most computer systems resemble closed queueing systems. First of all, they contain no infinite populations since the number of pending requests to any resource facility is finite. Secondly, the nature of the arrival process depends upon the number of customers already in the system.

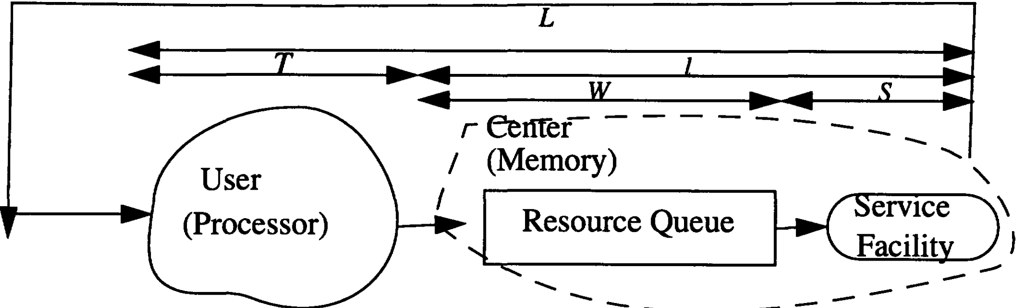


Figure 5.3: The closed queueing model.

The closed queueing model operates as follows: when a user (processor) makes a request for service, the request "enters" the center (memory) and proceeds to receive service according to the scheduling algorithm for the particular resource. During this time, the user either "goes to sleep" (i.e. cannot generate any new requests) or potentially generates a new request. When finally the request is complete, the response is fed back to the user at which point it "wakes up" if asleep and then begins to generate a new request. The time spent by the user in generating this new request is referred to as his "thinking time" T. For simplicity, T includes the time from when the user begins to generate the request to

when the request arrives at the center.

For a closed queue, the important performance metric is L, the average total latency in the system, which includes the average response time at the queue l and the thinking time T:

$$L = T + l \qquad (5.2)$$

Our aim in this section is to derive an expression for the latency seen by the master L, in the context of the M/D/1 closed queueing model, and in terms of NG contention, which we denote by *Contention(NG)*[1]. Initially, we derive an expression for L in terms of *Contention(NG)* and S. Then we compute S from the expression of L by a least square approximation to the experimental data for L.

To express L in terms of S and *Contention(NG)*, we first derive l in terms of L, NG contention and S, and then T in terms of S and the average total latency under no NG contention, which we assume to be known from the experiments, and which we denote by *L(Contention(NG) = 0)*.

According to (5.1), l depends on the wait time at the queue W and the service time S. To derive the expression of W in terms of L, *Contention(NG)* and S, we first express W as a function of L, S, *Contention(NG)* and U, the utilization of the resource. Since no closed form solution to W in terms of these parameters has been found in the available literature, the necessary expression is derived in this thesis. The derivation relies on the fact that the wait time W depends on the instant queue length A upon customer arrival at the queue, in the following manner [12]:

1. Although the "streaming" traffic pattern generated by the NGs may cause the total latency seen a NG request to be different from the total latency seen by a M request (since the thinking times T may differ), we can assume that the NGs act as N-1 master processors, and hence all N requests see the same average total latency L.

$$W = SA \tag{5.3}$$

The instant queue length A is computed on condition that some customer is arriving at the queue and so cannot itself be in the queue. To compute A, a characteristic of closed queueing systems is used, mainly that the instant queue length equals the average queue length Q in a system where the arriving customer is not included. Consequently, A is computed by subtracting the contribution of the arriving customer from Q. Since all customers are equal, they also contribute equally to the average queue length, hence the contribution of the arriving customer is Q/N where N is the total number of customers in the system[1]. Therefore,

$$W = S\left(Q - \frac{Q}{N}\right) \tag{5.4}$$

Since, from [12], for a M/D/1 queue, the average queue Q at a server under U utilization is:

$$Q = \frac{1}{2}\frac{U}{1-U} , \tag{5.5}$$

the expression of W in terms of U, S and N is:

$$W = S\left(\frac{1}{2}\frac{U}{1-U}\right)\left(1 - \frac{1}{N}\right) \tag{5.6}$$

To compute N, we use the Little Theorem:

$$N = L(Throughput(NG) + Throughput(M)) \tag{5.7}$$

where *Throughput(NG)* is the throughput[2] generated by the NGs and *Throughput(M)* is the throughput generated by M. Since M executes one request during L time, *Throughput(M)* is given by the following equation:

---

1. See footnote on page 49.
2. Throughput is defined as the number of requests generated by a processor per second.

$$Throughput(M) = \frac{1}{L} \tag{5.8}$$

Since each requests generates a data transfer of *CacheLineSize* bytes which contribute to contention, *Throughput(NG)* can be derived from *Contention(NG)* as follows:

$$Throughput(NG) = \frac{Contention(NG)}{CacheLineSize} \tag{5.9}$$

where *CacheLineSize* must be expressed in units of megabytes.

By substituting (5.8) in (5.7) and then substituting *N* in (5.6), *W* in terms of *L*, *Contention(NG)*, *U* and *S* becomes:

$$W = S\left(\frac{1}{2}\right)\left(\frac{U}{1-U}\right)\left(1 - \frac{1}{\left(\frac{LContention(NG)}{CacheLineSize} + 1\right)}\right) \tag{5.10}$$

Consequently, from (5.1) and (5.10), the expression of *l* in terms of *L*, *Contention(NG)*, *U* and *S* is:

$$l = S\left(\frac{1}{2}\right)\left(\frac{U}{1-U}\right)\left(1 - \frac{1}{\left(\frac{LContention(NG)}{CacheLineSize} + 1\right)}\right) + S \tag{5.11}$$

U and can be computed from the total contention and the peak sustained bandwidth through the resource:

$$U = \frac{Throughput(NG) + Throughput(M)}{PeakSustainedBandwidth} \tag{5.12}$$

where:

$$PeakSustainedBandwidth = \frac{1}{S} \tag{5.13}$$

By substituting (5.13) in (5.12), the expression of *U* in terms of *L*, *Contention(NG)* and *S* is:

$$U = S\left(\frac{Contention(NG)}{CacheLineSize} + \frac{1}{L}\right) \tag{5.14}$$

Hence, by substituting (5.14) in (5.11), the formula of *l* in terms of *L*, *S* and *Contention(NG)*:

$$l = S\left(\frac{1}{2}\left(\frac{S\left(\frac{Contention(NG)}{CacheLineSize} + \frac{1}{L}\right)}{1 - S\left(\frac{Contention(NG)}{CacheLineSize} + \frac{1}{L}\right)}\right)\left(1 - \frac{1}{\left(\frac{LContention(NG)}{CacheLineSize} + 1\right)}\right) + 1\right) \tag{5.15}$$

After some simplifications, *l* can finally be expressed as:

$$l = S \cdot \left(\left(\frac{1}{2} \cdot \frac{S}{1 - S\left(\frac{Contention(NG)}{CacheLineSize} + \frac{1}{L}\right)}\right)\frac{Contention(NG)}{CacheLineSize} + 1\right) \tag{5.16}$$

To compute *T* using (5.2), we evaluate *L* and *l* for *Contention(NG) = 0.* By noting that, when *Contention(NG)=0, l(Contention(NG)=0)= S* (since no wait in the queue is incurred), *T* is computed as:

$$T = L(Contention(NG) = 0) - S \tag{5.17}$$

where *L(Contention(NG) = 0)* is assumed to be of known value, as measured experimentally.

By substituting (5.15) and (5.17) into (5.2), the implicit expression of *L* in *Contention(NG)* and *S* can be derived as:

$$L = L(0) + \left(\frac{1}{2} \cdot \frac{S^2}{1 - S\left(\frac{Contention(NG)}{CacheLineSize} + \frac{1}{L}\right)}\right)\frac{Contention(NG)}{CacheLineSize} \tag{5.18}$$

where *L(0)* is a shorthand notation for *L(Contention(NG) = 0).*

After simplifications, (5.18) reduces to a quadratic equation in *L* (see appendix B). The roots of the equation are:

$$L_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (5.19)$$

where:

$$a = 1 - S \cdot \frac{Contention(NG)}{CacheLineSize} \quad (5.20)$$

$$b = L(0)\left(S \cdot \frac{Contention(NG)}{CacheLineSize} - 1\right) - \left(\left(\frac{1}{2} \cdot S^2\right) \cdot \frac{Contention(NG)}{CacheLineSize}\right) - S \quad (5.21)$$

$$c = L(0)S \quad (5.22)$$

L is set to the larger of the two roots, $\frac{(-b) + \sqrt{b^2 - 4ac}}{2a}$, so as to satisfy the inequality: $L \geq S$. This step completes the derivation of the average master latency L in terms of *Contention(NG)* and S.

By iterating over service times and choosing the value which results in the smallest least square error to the actual data, one can estimate the actual resource service time. Finally, by replacing in the expression of L the unknown service time S with the actual service time, one can derive the expression of L in terms of *Contention(NG)*. From this equation, one can compute the delays through the system on all contention values.

To compute the least square error per sample, formulae (5.23) and (5.24) are used:

$$LeastSquares = \sum_{i=1}^{i = NumSamples} (r(i) - m(i))^2 \quad (5.23)$$

where *NumSamples* is the number of contention values on which the benchmark data is gathered, *r(i)* is the benchmark data corresponding to sample *i,* and *m(i)* is the model data for sample *i.* Specifically, for the experiments run on the Origin 2000 and dedicated to memory contention studies, which are shown in Table 5.1, *NumSamples = 7*, *r(1)* through *r(7)* are the results stored in the second column of this table, and *m(1)* through *m(7)* represent the L values evaluated for each *Contention(NG)* value in the first column of the same

table.

To compute the error per experimental sample, the *LeastSquares* result is normalized as follows:

$$error = \frac{\sqrt{LeastSquares}}{NumSamples} \tag{5.24}$$

Applied to the Origin 2000, this methodology leads to the following results:

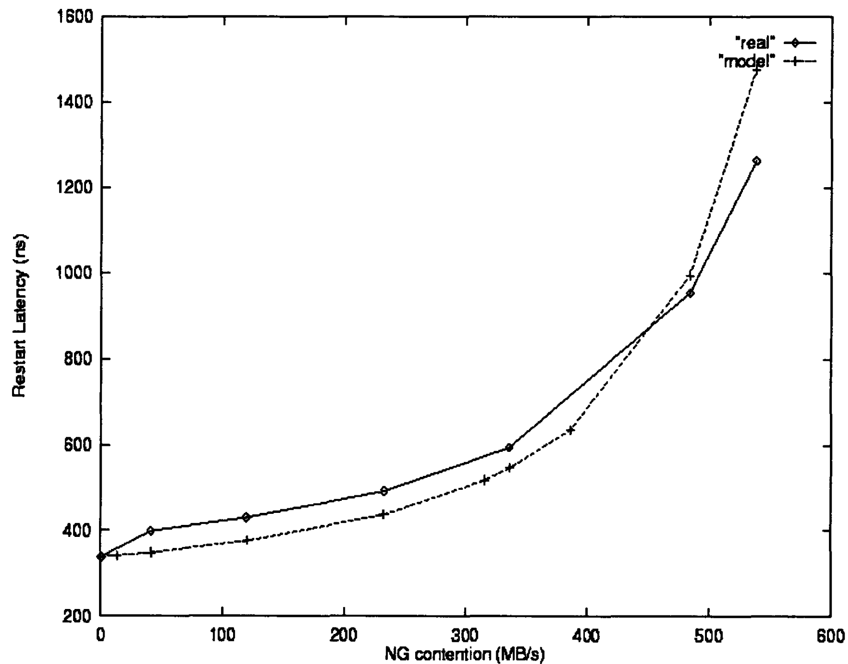


Figure 5.4: Restart Latency under memory contention and the best-fit curve implied by the closed M/D/1 model with a memory service time of 195 ns.

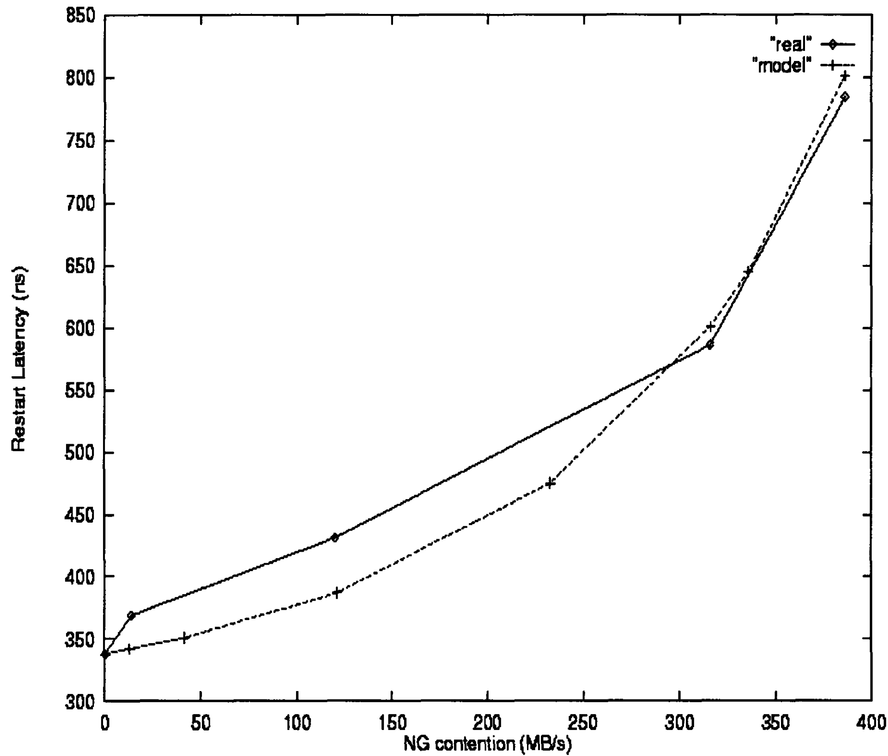


Figure 5.5: Restart Latency under processor bus contention and the best-fit curve implied by the M/D/1 closed queueing model with a processor bus service time of 215 ns.

Best fit based on least square computations implies a memory service time of 195 ns and a processor bus service time of 215 ns. The minimum least square errors are 44 ns/sample for the memory and 28 ns/sample for the processor bus respectively. The service times imply a peak sustained memory bandwidth of 657 MB/sec and a peak sustained processor bus bandwidth of 594 MB/sec. These results match the numbers seen in cycle-by-cycled based simulations within 5%.

Chapter 6

Conclusions

6.1 Accomplishments

In light of the new processor developments, memory latency has become two distinctive performance parameters: restart latency and back-to-back latency. Furthermore, pipelined memory bandwidth has replaced memory bandwidth, a performance metric traditionally computed as the ratio of cache line size to memory latency. With the sophistication of current processors and the complex performance space of cache-coherent multiprocessors, memory performance has also become more difficult to measure. This thesis has refined the current definition of memory performance so as to adequately address the characteristics of modern processors. In addition, it has introduced a methodology for measuring memory performance, in light of this refinement and in the context of modern cache-coherent multiprocessors.

The methodology is based on three micro benchmarks, each measuring one of the three parameters of memory performance. 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 is greater than restart latency by ~40%. The pipelined bandwidth to local memory is 536 MB/s, in contrast to 271 MB/s which would result if simply dividing the cache line size (128 bytes) by the back-to-back latency (472 ns).

To study the effects of cache coherency, a particular caching state was established prior to the execution of the memory micro benchmark. On the Origin 2000, the cache

coherency protocol increases the penalty of a memory access significantly. For example, for a read to a line with a local home, the restart latency is 338 ns when the state of the line is unowned (hence no extra cache coherency operations are required in order to service the miss), as opposed to 892 ns when the state of the line 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 line).

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. A methodology based on the M/D/1 closed queueing model was derived to compute the peak bandwidth through each studied resource and the memory performance under contention levels that cannot be reproduced experimentally. As shown by experiments on the Origin 2000, the resource utilization adds delays and reduces the bandwidth seen by a given processor. For example, the restart latency increases by 330% when the memory utilization changes from 40% to 94%.

6.2 Future Work

Our current methodology assumes a number of system features, such as a cache which is either direct-mapped or has a LRU replacement algorithm[1] and a MESI cache coherency protocol. As future extensions to our work, we envision enhancing the portability of the micro benchmarks, for example, by supporting additional cache coherency protocols and cache replacement algorithms.

Introducing additional system resources, such as the network and the cache controller, as cases in the micro benchmark dedicated to contention studies, and measuring the

1. In the routine which initializes the cache coherency state, it is assumed that some given data can be flushed from a cache by writing a large array into the cache (of size greater than the cache size).

impact of writebacks on memory performance are some other possible extensions of this work.

A current limitation of our methodology dedicated to cache coherency studies is that the entire experimental data must fit in the cache. By limiting the size of the experimental data to fairly small values, this methodology leads to inconsistencies of results across runs. To overcome this difficulty, currently the user must run the cache coherency micro benchmark a large number of times and select the smallest of the observed results. At this stage, we are not aware of a more elegant approach to overcoming this limitation.

The application of micro benchmarks to macro benchmark performance prediction remains an open issue. While research by Saavedra [23] has shown that, for simple uniprocessor applications, such predictions can be made within 10% only of the actual run times, no similar achievements in the area of multiprocessor applications have yet been demonstrated [28]. However, with the new results our work has generated in measuring multiprocessor memory performance, we believe constructing such a successful 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.

Appendix A

Acquiring Laboratory Data

To verify the micro benchmark results on the Origin 2000, a logic analyzer was installed at the SGI facility in Mountain View, CA. The logic analyzer was connected to a processor bus in a 2-node Origin 2000 system. The logic analyzer was monitoring events at the given processor bus. Some of the events monitored were: requests for memory data and for bus grant, acknowledgements and writebacks by each processor on the node, data replies, bus grant release to each of the processors, and interventions from memory.

To trigger the logic analyzer, a trigger instruction was executed, which caused some trigger value to appear at the processor bus. The trigger value was defined as follows:

```
#define TRIGGER_VALUE 0xfefefefefefefefeUL
```

A 64-bit value was chosen for *TRIGGER_VALUE* to reduce the probability that the trigger value occurs at the processor bus as a result of some operation other than the trigger instruction.

The micro benchmarks were modified to execute the trigger instruction after a few loop iterations. The trigger instruction consists of a write to an uncached variable:

```
*uncached_trigger = TRIGGER_VALUE;
```

Prior to the execution of the micro benchmark, the uncached variable was initialized as follows:

```
/* to allocate uncached memory */
uncached_trigger = memalign(16*1024, sizeof(long long));
cachectl(uncached_trigger, 16*1024, UNCACHEABLE));
```

Appendix B

This appendix derives the quadratic contention equation in L.

From (5.18),

$$L = L(0) + \left(\frac{1}{2} \cdot \frac{S^2}{1 - S\left(\frac{Contention(NG)}{CacheLineSize} + \frac{1}{L} \right)} \right) \frac{Contention(NG)}{CacheLineSize} \qquad (B.1)$$

We define the following constant A:

$$A = S \cdot \frac{Contention(NG)}{CacheLineSize} \qquad (B.2)$$

Then (B.1) becomes

$$L = L(0) + \frac{\frac{SA}{2}}{1 - A - \frac{S}{L}} \qquad (B.3)$$

Hence,

$$(L - L(0))\left(1 - A - \frac{S}{L}\right) = \frac{SA}{2} \qquad (B.4)$$

Further simplification leads to the following:

$$L(1-A) + \frac{L(0) \cdot S}{L} - S - L(0)(1-A) - \frac{SA}{2} = 0 \qquad (B.5)$$

Define constants a, b and c as follows:

$$a = 1 - A \qquad (B.6)$$

$$b = -S - L(0)(1-A) - \frac{SA}{2} \qquad (B.7)$$

$$c = L(0)S \qquad (B.8)$$

An implicit equation of L in terms of a, b and c follows from (B.5):

$$aL + \frac{c}{L} + b = 0 \quad \text{(B.9)}$$

Hence,

$$a(L)^2 + bL + c = 0 \quad \text{(B.10)}$$

q.e.d.

References

[1] Cybenko, G., Kipp, L., Pointer, L., and Kuck, D., *Supercomputer Performance Evaluation and the Perfect Benchmarks*, University of Illinois Center for Supercomputing R&D Technical Report 965, March 1990.

[2] Mike Galles. Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER chip. In *Hot Interconnects '96,* 1996.

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

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

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

[6] Cristina Hristea. The Performance of the Origin 2000 Multiprocessor. SGI Technical Report, August 1995.

[7] Cristina Hristea, Daniel Lenoski, John Keen. Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro Benchmarks. Submitted to *International Conference on Supercomputing '97*, May 1997.

[8] Leonard Kleinrock, *Queueing Systems*, Volume 1: Theory, John Wiley & Sons, New York, 1975.

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

[10] 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.

[11] James Laudon, Daniel Lenoski. System Overview of the SGI Origin 200/2000 Product Line. In *Proceedings of COMPCON* Spring `97: *Forty-Second IEEE Computer Society International Conference*, February 1997.

[12] Edward Lazowska, et. al., *Quantitative System Performance*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1984

[13] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148-159, May 1990.

[14] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

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

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

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

[18] Arun K. Nanda, Lionel M. Ni. MAD Kernels: An Experimental Testbed Study Multiprocessor Memory System Behavior. *IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 2,* February 1996.

[19] Robert W. Numrich. GET/PUT and F-- on T3D, T3E, SN0, SN1 and Beyond. Cray Research Inc., June 1996.

[20] 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.

[21] Rafael Saavedra, R. H. Smith, E. Miya. Machine Characterization Based on an Abstract High-Level Language Machine, *IEEE Trans. on Comp.* Vol.38, No.12, December 1989, pp. 1659-1679.

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

[23] Rafael H. Saavedra, Alan J. Smith. Performance Characterization of Optimizing Compilers. UCBerkeley Technical Report USC-CS-92-525, August 1992.

[24] Singh, J.P., Weber, W-D., and Gupta, A.. SPLASH: Stanford Parallel Applications for Shared Memory, Tech. Rep. CSL-TR-91-469, Stanford University, 1991.

[25] SPEC, *SPEC Newsletter: Benchmark Results*. Vol.2, Issue 1, Winter 1990.

[26] 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.

[27] Steve Whitney, John McCalpin, Nawaf Bitar, John L. Richardson, and Luis Stevens. The SGI Origin software environment and application performance. In *Proceedings of COMPCON Spring '97: Forty-Second IEEE Computer Society International Conference*, pages 165-170, February 1997.

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

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