Hardware Performance Monitoring in Multiprocessors

by

Guy G. F. Lemieux

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 1996 by Guy G. F. Lemieux

Hardware Performance Monitoring
in Multiprocessors

Guy G. F. Lemieux

Master of Applied Science, 1996

Department of Electrical and Computer Engineering
University of Toronto

Abstract

Multiprocessors are often quoted as being capable of a 'peak performance,' but in practise it is difficult to utilize this potential. Consequently, software applications must be well tuned to run efficiently.

In this thesis, factors affecting performance in cache-coherent multiprocessors, particularly those which use a sequentially-consistent memory model, are analyzed. Cost-effective hardware features are suggested to measure these factors without intruding on software performance. Many of the proposed features can be readily incorporated into future processor designs, and others are easy to implement in external hardware. The measurements can be used to help tune a program's performance as well as other purposes such as simulation verification, workload characterization, or runtime performance decision support.

To demonstrate cost feasibility and other implementation concerns, the processor card hardware performance monitor developed for the NUMAchine multiprocessor is also described.

Acknowledgements

I am indebted to the many people who have helped me along in the course of developing this thesis. In particular, I wish to thank Stephen Brown and Zvonko Vranesic for the opportunity and their support and encouragement.

Next, I am very grateful for the sacrifices of my Grandmother, who worked hard to help support me throughout my university education. My love is with her always.

Of course, my friends have contributed to maintaining my sanity and providing distractions. Thanks, Kelvin, for the many great squash games! I have also enjoyed the company of Steve, Zeljko, Derek, Dan, Alex, Robin, Rob, Mitch, Dean, Ben, Gordon, Wing-Chi, and many, many others. Most importantly, I will always cherish Martha's love, support and companionship.

The helpfulness of discussions and criticisms with Naraig, Orran, Todd, and Keith is appreciated.

Financial support from OGS, ITRC, UofT, and Hewlett-Packard is gratefully acknowledged.

Table of Contents

Abstract .... **ii**

Acknowledgements .... **iii**

Table of Contents .... **v**

List of Figures .... **ix**

List of Tables .... **xi**

Chapter 1 ***Introduction*** **1**

1.1 Motivation 1
1.2 Objectives 2
1.3 Contributions 3
1.4 Overview 3

Chapter 2 ***Background*** **5**

2.1 Monitoring in Software 5
2.1.1 gprof 5
2.1.2 Mtool 6
2.1.3 Paradyn 7
2.1.4 QPT and CProf 8
2.1.5 MemSpy 8
2.1.6 Summary 9
2.2 Hardware Monitoring in Processors 9
2.2.1 Intel Pentium and Pentium Pro 10
2.2.2 DEC Alpha 21064 10
2.2.3 MIPS R4400 and R10000 10
2.2.4 Hewlett-Packard PA-RISC 13
2.2.5 Sun SPARC 13
2.2.6 IBM/Motorola/Apple PowerPC 604 14
2.2.7 IBM POWER2 14
2.2.8 Processor Summary 15
2.3 Hardware Monitoring in Systems 16
2.3.1 University of Toronto Hector 16
2.3.2 Stanford DASH 18
2.3.3 MIT M-Machine 20
2.3.4 Cray Research Inc. 20
2.3.5 Convex Computer Corporation 21
2.3.6 Kendall Square Research 23
2.4 Summary 24

Chapter 3 ***Multiprocessor Hardware Performance Monitoring*** **27**

3.1 Performance Losses: Software View 27

3.1.1 Inefficient Algorithms 27
3.1.2 Extra Parallel Work 28
3.1.3 Synchronization 28
3.1.4 Load Imbalance 28
3.1.5 Inefficient System and Library Calls 28
3.1.6 Amdahl's Law 29
3.1.7 Summary 29
3.2 Performance Losses: Hardware View 30
3.2.1 Instructions 30
3.2.2 Instruction Scheduling and CPI 30
3.2.3 Memory Stalls 31
3.2.4 Other Stalls 31
3.2.5 Exceptions and Interrupts 32
3.2.6 Branch Mispredictions 32
3.2.7 Special Cases 33
3.2.8 Summary 33
3.3 Processor View of Memory Stalls 34
3.3.1 Counting Cache Misses 35
3.3.2 Classifying Cache Misses 37
3.3.3 Measuring Cache Miss Latency 42
3.3.4 Summary: Processor View of Memory Stalls 48
3.4 Hardware Support for Software Tools 49
3.4.1 Timer Support 49
3.4.2 Basic Block Profiling 49
3.4.3 Pointing Losses Back to Code and Data 52
3.4.4 Summary: Hardware Support for Software Tools 52
3.5 Summary 52
Chapter 4 *Hardware Implementation* 55
4.1 Introduction to NUMAchine 55
4.2 Processor Card Organization 58
4.3 Monitor Organization 59
4.3.1 Local Bus Controller 59
4.3.2 Pipeline Status 61
4.3.3 Counters & Interrupts 62
4.3.4 SRAM Memory 64
4.3.5 Count & Increment 64
4.3.6 Latency Timer 64
4.3.7 Configuration Controller 65
4.4 Programmable Configuration 65
4.4.1 PhaseID Register and PhaseID Watch 65
4.4.2 Command Watch and Command Filter 67
4.4.3 Address Watch and Address Filter 67
4.4.4 General-Purpose Counter Modes 68
4.4.5 SRAM Counter Mode 71
4.4.6 Master Counter Enables 75

4.5 Summary 75

Chapter 5 *Conclusions* **79**

5.1 Contributions 79

5.2 Future Work 81

Appendix A *Memory Card Monitoring* **83**

Bibliography **85**

List of Figures

FIGURE 2.1. SUPERMON block diagram....................17
FIGURE 2.2. DASH performance-monitoring hardware....................19
FIGURE 3.1. An example of Amdahl's Law limitations on speedup....................29
FIGURE 3.2. Time-based basic block profiling without special hardware support....................51
FIGURE 3.3. Time-based basic block profiling with special hardware support....................51
FIGURE 4.1. The NUMAchine station....................56
FIGURE 4.2. The NUMAchine hierarchy....................56
FIGURE 4.3. Memory consistency state transition diagrams....................57
FIGURE 4.4. NUMAchine processor card datapath organization....................58
FIGURE 4.5. Processor card performance-monitoring subsystem....................60
FIGURE 4.6. Counters & Interrupts datapath, with write path highlighted....................63
FIGURE 4.7. MCC registers with one Count-Enable Circuit....................69
FIGURE 4.8. State machines to detect invalidation hits and misses....................71
FIGURE 4.9. SRAM address generation. The outlined portion is inside the MCC....................73
FIGURE A.1. Memory card organization....................83

List of Tables

TABLE 2.1. Intel Pentium performance-monitoring counter inputs.............................11
TABLE 2.2. DEC Alpha 21064 performance-monitoring counters and inputs 12
TABLE 2.3. MIPS R4400 externally observable events...12
TABLE 2.4. MIPS R10000 performance-monitoring counters and inputs....................13
TABLE 2.5. PowerPC 604 performance-monitoring counters and inputs.....................15
TABLE 2.6. Overview of performance-monitoring features on current processors......16
TABLE 2.7. CRAY Y-MP C90 performance-monitoring counters...............................22
TABLE 2.8. Convex Exemplar SPP-1200 performance-monitoring counter inputs 23
TABLE 2.9. KSR performance-monitoring counters..24
TABLE 4.1. Breakdown of 64-bit address space in NUMAchine 60
TABLE 4.2. Pipeline Status bits PS[6..5] specify one of three operating modes 62
TABLE 4.3. Pipeline Status bits PS[4..0] select which events to monitor.....................62
TABLE 4.4. Master Configuration Controller user configuration registers 66
TABLE 4.5. General-purpose counter events..70
TABLE 4.6. SRAM counter events 72
TABLE 4.7. SRAM counter modes...72
TABLE 4.8. Miss types encoded in the Miss Type Register..74
TABLE 4.9. Response State Register encoding 74
TABLE 4.10. Comparison of recommended versus implemented features 76
TABLE 4.11. Approximate cost of monitoring components 77
TABLE A.1. Memory state hit table 84

Chapter 1
Introduction

Measuring performance is important for tuning and understanding program behaviour. This thesis concentrates on the development of performance monitoring hardware in cache-coherent multiprocessors, what it should measure, and its uses. Additionally, a flexible implementation of the performance monitor for NUMAchine [Vranesic95] is described.

1.1 Motivation

Microprocessors have improved in speed at a tremendous rate in the last decade. The improvement has come from the combination of advances in silicon processing technology and architectural progress. However, the performance gap between main memory and microprocessors has widened, and it is apparent that this trend will continue. This speed discrepancy necessitates usage of cache memory. Cache memory is effective because programs often exhibit locality of reference. Yet some programs do not exhibit locality, and others that do have locality can fail to have requisite data in cache due to caching policies. Such programs exhibit significant performance loss, and hence cache behaviour is a prime candidate for performance tuning.

The problem of poor cache performance is worsened in shared-memory multiprocessors. Memory access times in multiprocessors are typically much longer than in uniprocessors due to longer paths through the communication network, additional arbitration logic, and network or memory contention. Cache effectiveness is further decreased with cache coherence, because a processor that modifies data must first remove any shared copies from the caches of other processors. Despite these performance problems, cache-coherent multiprocessors are attractive parallel systems because they provide a simple programming model.

Poor cache behaviour makes parallel applications difficult to scale and speedups are poor — even when other sources of performance loss, such as load imbalance, are minimized. To improve performance the programmer needs insight into the program's behaviour and the sources of performance loss. For example, knowing the number of references that miss in the cache gives the programmer a specific target to minimize. A more detailed breakdown, such as knowing which code segments suffered most of the misses, would be even more helpful.

To assist with performance tuning, a variety of previously developed tools can be used. Software such as gprof [Graham82] measures execution time spent in each procedure of a program being monitored, but its results can be misleading since it is not clear whether a procedure is simply inefficient or if caching problems occur. Another tool, Mtool [Goldberg93], provides more detail by breaking up procedure execution time into compute time,

memory overhead, and synchronization overhead, but it is still at the procedure level. Both gprof and Mtool are intrusive and can introduce a significant *probe effect*. Other tools, such as CProf [Lebeck94] and MemSpy [Martonosi95], go into greater detail by performing cache-level simulations to identify exactly where cache misses occur, but they exhibit enormous runtime overhead; MemSpy slows parallel execution by a factor of about 200 [Martonosi95]. These tools are helpful, but they can be too abstract, intrusive, or very slow.

In contrast to performance tuning software, dedicated hardware can be used to alleviate most of the problems. Performance monitoring hardware can be made *nonintrusive* and, consequently, allow applications to run at full speed. Also, collected data can be as fine-grained as required to assist tuning. Additionally, events that are unobservable or inconvenient to measure in software, such as peak memory contention or response time to invoke an interrupt handler, can be done effectively in hardware. Consequently, hardware can expand the variety and detail of measurements available with no impact on application performance, providing a 'best of both worlds' option that is not available with software-based approaches.

Although motivated by tuning, a hardware performance monitor has additional uses. For example, it can help programs adapt to their run-time situation by dynamically choosing whether prefetching will improve performance [Horowitz95]. Also, it can assist computer architects by collecting data normally obtained from slow system simulations. Similarly, collected data can be used to help verify mathematically-abstract system models, characterize workloads for the models, and even generate benchmark suites. For example, one supercomputer site is creating such a suite based on one year of data from a Cray Y-MP [Gao95].

1.2 Objectives

Although a hardware performance monitor adds to the cost of designing and constructing a multiprocessor, its use can add significant value. In this thesis, factors which influence multiprocessor performance are identified and, where possible, hardware features to measure these factors are proposed. The primary objectives of the hardware features are to identify the causes of software performance loss and aid multiprocessor research while maintaining low cost and remaining nonintrusive. There are many measurements which can be made, so prudence is required. When selecting what features are important to measure in hardware, the primary objectives are carefully considered. To demonstrate the low cost and feasibility of the hardware features, a portion of the hardware performance monitor implemented for NUMAchine, a cache-coherent shared-memory multiprocessor, is presented.

To aid in multiprocessor research the hardware monitor should collect enough data to obviate the need for some detailed architectural simulations, which often run too slowly to collect statistically reliable data. Additionally, the monitor should assist machine and workload characterization for higher level simulation models; typical characteristics are the latency of memory read operations under various conditions or the probability of a

read versus a write. These characteristics can be used to form powerful abstractions because they permit rapid testing of ideas using faster, high-level simulations. Also, the measurements are useful to verify the results from such simulations.

A hardware performance monitor has a special appeal to software developers who wish to do software tuning. In this application, the goal of the hardware is not to replace software tuning tools, but to improve their instrumentation so that data collection is done nonintrusively and in real time. Additionally, the collected data should be fine-grained enough to identify, as best as possible, the true bottleneck to be optimized.

In the development of performance-monitoring hardware, a secondary objective is to make performance data accessible to a running program. By doing this, it is hoped that new research into software that can make runtime decisions to improve performance will be made possible. To achieve this goal, it is important to move the monitoring features as close to the processor as possible so that performance data queries can be satisfied quickly.

1.3 Contributions

The contributions made by this work can be summarized as follows:

- comprehensive list of 23 performance measurements for multiprocessors,
- extending informing memory operations to include an external ***TRIGGER*** pin,
- using numerous counters selected by a ***PhaseID*** register to partition performance data,
- returning the memory state with memory responses to estimate coherence overhead,
- definition, detection, and measurement of invalidation and ownership misses, and
- cache conflict measurements, fine-grained timing hardware, and quantifying latency-hiding mechanisms on performance comprise other minor contributions.

Although the contributions are made in the context of multiprocessors, many of the measurements and hardware mechanisms can be applied to uniprocessors as well. The difference is that coherence misses are eliminated and miss latency is more regular (*i.e.*, the effects of contention and maintaining coherence are reduced or eliminated) in uniprocessors.

1.4 Overview

This thesis is organized as follows. Chapter 2 describes previous work in both software and hardware monitoring systems. Chapter 3 proposes a number of measurements that can be made, including both new ideas and those borrowed from previous research, which are consistent with the main objectives. As an implementation example, a portion of the NUMAchine performance monitor is shown in Chapter 4. Finally, conclusions are given in Chapter 5.

Chapter 2
Background

This chapter describes some performance-monitoring systems that have been constructed recently, concentrating on those for shared-memory multiprocessors. Both software and hardware systems are discussed, with the latter covering both research and commercial systems.

2.1 Monitoring in Software

Software approaches to performance monitoring can be broadly divided into intrusive profiling tools, such as gprof, Mtool and Paradyn, and simulation tools such as CProf and MemSpy. The operation of these tools is described below.

2.1.1 gprof

Although not intended for parallel programs, gprof [Graham82][Graham83] is described here because it is a widely used tool for performance analysis. A compiler with gprof support can automatically annotate object code to count the number of times a subroutine is called by each caller by storing counters in a sparse hash table. Additionally, added code samples the program counter at a regular rate during execution, typically 60 or 100 Hz, and a histogram is formed. At exit, this data is dumped to a file. The gprof program examines this file and the executable image (with the symbol table) to correlate program counter values with subroutine names. The resulting histogram shows the statistical proportion of time a program spent in each subroutine, but this does not include time spent waiting for subroutines it invoked.

Under the assumption that a subroutine always takes the same length of time, the subroutine counts are used to divide its running time among its callers. This information is presented to the user in two ways: first, subroutines are ranked by execution time, and second, by the call-tree hierarchy. The execution time ranking is useful for identifying slow routines, while the call-tree format shows which parent routine calls a subroutine most often. In this way, a programmer can optimize a routine to run faster and also improve the parent routine so that fewer subroutine calls are made. As a performance tuning tool, gprof is good for general code and algorithm development, especially when the time complexity of a routine is not known beforehand.

A disadvantage of gprof is that the general nature of the execution profile provides little insight into sources of performance loss. For example, if a floating-point matrix multiply routine was labeled as being very slow, it would be unclear whether the floating-point operations were slow or if poor cache behaviour was encountered. Additionally, the inserted bookkeeping code overhead can be intrusive; we have observed code in which the bookkeeping routine accounts for 30% of the run time. This amount of intrusion can cause

gprof data to be misleading. For example, consider that cache interference from gprof's inserted code can inflate the execution time of a procedure beyond any other and then misreport it as being the bottleneck. Additionally, Varley reports that when a routine is called by more than one parent, gprof can incorrectly attribute the running time among its callers because of the invalid assumption made above [Varley93]. Moreover, since sampling is done at a relatively slow rate, the program must run long enough to capture statistically reliable data. Consequently, although gprof is useful for general code and algorithm improvement, it does not provide enough insight into the source of performance loss and may occasionally report incorrect or misleading results.

A hardware performance monitor should strive to reduce the perturbation imposed by a profiler such as gprof, and at the same time ensure that execution time is attributed to subroutines and callers accurately. One possible way is to time routines with high-resolution hardware timers instead of program counter sampling.

2.1.2 Mtool

Similar to gprof, Mtool [Goldberg93] automatically instruments a program at the object-code level by timing the execution of important routines and adding counters to *basic blocks*, a term that describes small segments of straight-line code which have only one entry point and only one exit. In addition, it supports parallel programs by measuring synchronization overhead and extra parallel work not present in sequential code. Mtool further breaks up runtime by classifying execution time as either compute time or memory overhead. This taxonomy aids the programmer during code tuning.

Basic block counting allows Mtool to predict an ideal compute time. It does this by modelling the processor pipeline and assuming that memory references always hit in the cache. The difference between the ideal and actual running time is approximately equal to the amount of memory overhead. With the help of timing information, the memory overhead can be divided among the proper routines.

To reduce the intrusiveness of counting every basic block, Mtool uses powerful control-flow and loop induction analysis to identify where counters should be placed. As a result, counters are added to only a few basic blocks and other counts can be derived from these. For example, in a loop containing an if-else statement, only the loop iteration count and the else-path count are needed to derive the number of if-paths followed. To further reduce overhead, loops do not always need to increment a counter after every iteration because the initial or final value of the loop index can often be used. Additionally, Mtool uses feedback from an initial profile to find and instrument the less frequently taken control paths. These techniques reduce basic block counting overhead to between 1% and 5% on the SPEC and Stanford SPLASH benchmarks.

Mtool times loops and procedures via program counter sampling (in a manner similar to gprof), high-resolution hardware timers, or both. By using both, it tries to balance the drawbacks of sampling with the overhead of inserting timer probes. Using an initial basic block count profile, Mtool automatically selects memory-intensive loops and procedures that should be instrumented with timers. It also measures synchronization procedure calls

with timers, but only if they are available on-processor or if it is requested by the user. Otherwise, Mtool uses program counter sampling. Despite the variety of timing methods supported, the authors of Mtool encourage that high-resolution timers be made available on-processor with access latencies of 1-2 cycles because they are the preferred way to instrument code.

The aggregate timing information collected is separated into performance loss from *synchronization overhead*, *load imbalance*, and *extra parallel work*[1]. It is also used in conjunction with the ideal compute time to estimate memory overhead in loops and procedures. After optimizing where to place instrumentation code, the overhead of collecting the timing and basic block counts together is less than 10%.

As a performance tool, Mtool imposes little intrusion and helpfully categorizes performance loss for routines into compute time, memory overhead, synchronization overhead, and extra parallel work. However, the low intrusion is based upon establishing an initial basic block profile and performing complex control-flow analysis. Also, it is unclear if Mtool suffers from the multiple-caller problem experienced by gprof. Additionally, Mtool gives no insight into the sources of memory overhead; for example, was it caused by many cache misses? long memory latency? or false sharing? Both CProf and MemSpy, described shortly, apply insight into the memory overhead, but only at a significant performance loss. Before describing them, however, the Paradyn code-instrumenting tool will be outlined.

2.1.3 Paradyn

Paradyn [Miller95] is a performance tool for parallel applications similar to Mtool. It focuses on *dynamic instrumentation* [Hollingsworth94] of code, where a program is augmented at key points with short instrumentation subroutines, called *base trampolines*. The contents of the trampoline are changed dynamically at run time by a daemon process under the control of a master Paradyn process. The controller, which may be running on a different computer, collects data from the daemon and decides which base trampolines require more instrumentation. It tells the daemon to modify the base trampolines to call *mini-trampolines* which do conditional checking and counting.

The power of Paradyn is that it inspects free-running programs. In this regard, it can measure the performance of certain routines over time. To do this, the overhead of the base trampolines is kept small (less than 10%). This allows the Paradyn controller to adapt to program behaviour and perform more instrumentation at the points in the program that are consuming most of the processor cycles. This instrumentation methodology forms a part of the W^3 search model [Hollingsworth93] used by the authors to determine *where*, *when* and *why* a program is performing poorly. These W^3 questions shall be considered again later as hardware performance-monitoring features are developed.

1. These terms will be defined more clearly in Chapter 3.

2.1.4 QPT and CProf

As parts of the University of Wisconsin WARTS tool set, QPT [Ball94][Larus93] and CProf [Lebeck94] work together to give insight into cache performance on uniprocessors. QPT provides basic block counting in much the same way that Mtool does; it can even calculate the execution cost of procedures à la gprof. However, QPT can also generate a highly compressed trace file of memory references. The trace is used by CProf to perform detailed simulations of cache activity and accredit the misses back to the source code and data structures in which they occurred. If CProf is given the costs of a cache hit and a miss, it estimates the performance of a routine or the whole program.

In CProf, cache misses are classified by the widely-accepted taxonomy, first introduced in [Hill88], as either *compulsory*, *capacity* or *conflict* misses. Compulsory misses are caused by the very first reference and can be reduced only by reducing the total size of data used. Capacity misses occur because the cache is too small, but they can be prevented by decreasing the length of time data is required to stay in the cache by using blocking or loop fusion, for example. Finally, conflict misses occur because there are too few locations within the cache for placing particular data. Conflicts can often be reduced by placing the conflicting data close together; merging two vectors by interleaving the elements is a good example. By classifying the causes of misses and relating them back to the source code and data structures, CProf provides the programmer with sufficient insight to tune an application. In fact, the SPEC benchmarks tuned in [Lebeck94] were made between 1.03 (for eqntott) and 3.46 (for vpenta, a benchmark kernel) times faster, with an average improvement of 1.69.

Although CProf provides very useful information, the cache simulation is admitted to be very slow (no performance numbers were given for CProf, but QPT trace generation alone increases runtime by a factor of 5). Despite this, the key observation to make about CProf is that classifying misses and attributing them to the correct parts of the code and data structures is very useful to the programmer. Performance monitoring hardware should strive to make comparable measurements and allow the application to run at full speed.

For convenience, the combination of QPT and CProf shall be referred to as just CProf in the remainder of this thesis.

2.1.5 MemSpy

MemSpy [Martonosi95] produces information similar to CProf and also runs slowly, but it supports parallel applications and the approach to instrumentation is very different. In particular, MemSpy collects data at the procedure (rather than basic block) level and techniques such as *hit bypassing* and *trace sampling*, described in [Martonosi95], are used to speed up simulation. Despite these techniques, simulation of parallel programs is roughly 200 times slower after accounting for the loss due to the sequential simulation of parallel code.

MemSpy is driven by a separate program that traces memory references, procedure calls and returns, memory allocations, and synchronizations. Performance data is collected

at a procedural granularity and attributed to the proper code and data structures. Cache misses are categorized as compulsory, replacement (combining both capacity and conflict misses), and invalidation by simulating the activity of the caches. The invalidation misses, which are not measured in CProf, are a result of data sharing in a parallel program, *i.e.*, when a processor changes a shared datum, stale copies that are present in other processors' caches must be invalidated. If those processors re-reference that datum, a cache miss due to invalidation occurs.

MemSpy assumes cache misses take a constant amount of processor time, irrespective of whether invalidations were necessary to satisfy the miss or whether any memory or network contention exists. This assumption is false, because many current multiprocessors have non-uniform memory access (NUMA) times and some programs do create hotspots. Unfortunately, NUMA, contention, and invalidation overhead are important performance factors that make it difficult to design an efficient multiprocessor.

MemSpy is an important tool because it gives greater insight into why a program is slow than does Mtool, for instance. However, it is very slow and it uses an unrealistic simulation of memory performance.

2.1.6 Summary

As shown, software techniques for performance monitoring range from slightly intrusive to very slow; the finer the granularity, the slower the execution speed. In all of the tools described, the software does not account for operating system activity or system call effects. However, these effects can be significant because even a single system call can eject a significant amount of data from the cache. In contrast, the system call and subsequent cache misses would be visible to performance-monitoring hardware.

Software tuning benefits from basic block profiles, program counter sampling or procedure timing, and cache miss measurements. Additionally, it is very helpful to attribute cache misses to the proper data structures and code fragments and to classify the cause of the cache miss as precisely as possible. Fine-resolution clocks can be used for precise procedure timing to measure synchronization overhead, for example. Also, basic block profiling makes it possible to predict ideal compute time and, consequently, estimate memory overhead.

2.2 Hardware Monitoring in Processors

As system integration increases, monitoring hardware is able to observe less and less of the system. For example, SUPERMON [Bacque91] was able to trace every instruction reference because the instruction cache was on a separate chip, but all current microprocessors integrate this cache into the processor. In fact, current systems are even putting the second level cache in the same chip (the Alpha 21164 or Pentium Pro, for example). Fortunately, microprocessor designers also see the need for performance observation and

include dedicated measurement hardware in the latest designs. This section describes the measurement facilities present in recent processors.

2.2.1 Intel Pentium and Pentium Pro

Work by Mathisen [Mathisen94], which was later updated by Ludloff [Ludloff94] and Collins [Collins95], reverse engineered[2] some of the Pentium instruction set and revealed that there are two 40-bit registers dedicated for performance measurement. Each of these registers is capable of counting one of the 42 possible events listed in Table 2.1. Additionally, the counters can be arranged to count events or the number of CPU cycles spent performing the events. Unfortunately, a program must be in supervisor mode to access them, meaning a (costly) system call is usually required.

In addition to the performance counters, the Pentium processor also includes a timestamp counter and two performance monitoring *output* pins. The timestamp counter is accessible by a user instruction and can be used for accurate timing of routines. The output pins can be configured to toggle after any performance event or a counter overflow. By wiring one of these pins to an interrupt pin, software can be made reactive to performance data.

The Pentium Pro processor enhances the Pentium counter feature set and has even added a user-level instruction to read the performance counters. Further, it can internally generate an interrupt when a counter toggles or overflows, thus obviating one of the uses of the Pentium output pins.

2.2.2 DEC Alpha 21064

The DEC Alpha 21064 processor [Digital92] includes two 32-bit counters that can count a total of 17 different events. Interestingly, two of the events are for counting the number of cycles a dedicated pin is asserted. This allows external events to be counted with high-speed on-chip registers, and is ideal for giving software low-latency access to performance data. Unfortunately, the counters are only available in supervisor mode.

A summary of the countable events is given in Table 2.2. In addition to these counters, the 21064 includes a dedicated 32-bit timestamp counter which can be used for precision timing.

2.2.3 MIPS R4400 and R10000

Rather than provide on-chip counters, the MIPS R4400 [Heinrich94] brings pipeline status information to externally observable pins. Although it requires some hardware design effort, off-chip counters can be built to keep track of the 15 different pipeline events listed in Table 2.3. This feature is convenient because system designers can build 15 dedicated

---

2. This information is contained within Intel’s infamous Appendix H of the Pentium User’s Manual which is not publicly available. Consequently, the counter functions described here may be inaccurate.

Index	Event	Index	Event
0	data read	15	pipeline flushes
1	data write	16	instructions executed in both pipes
2	data TLB miss	17	instructions executed in the v-pipe
3	data read miss	18	bus utilization (clocks)
4	data write miss	19	pipeline stalled by write buffer overflow
5	write (hit) to modified or exclusive state lines	1A	pipeline stalled by data memory read
6	data cache lines written back	1B	pipeline stalled by write to modified or exclusive line
7	data cache snoops	1C	locked bus cycle
8	data cache snoop hits	1D	I/O read or write cycle
9	memory accesses in both pipes	1E	noncacheable memory reference
A	bank conflicts	1F	AGI (address generation interlock)
B	misaligned data memory references	20	unknown, but counts
C	code read	21	unknown, but counts
D	code TLB miss	22	floating-point operation
E	code cache miss	23	breakpoint 0 match
F	any segment register load	24	breakpoint 1 match
10	segment descriptor cache accesses	25	breakpoint 2 match
11	segment descriptor cache hits	26	breakpoint 3 match
12	branches	27	hardware interrupt
13	branch target buffer (BTB) hits	28	data read or data write
14	taken branch or BTB hit	29	data read miss or data write miss

TABLE 2.1. Intel Pentium performance-monitoring counter inputs.

hardware counters and capture all information about a program in one pass, compared to most other microprocessors which only provide two counters. The R4400 also provides a 32-bit cycle counter for high-precision timing. An interrupt can be raised when the cycle counter reaches the value stored in a compare register — this is very useful for high-resolution program counter sampling, for example.

Interestingly, the next-generation MIPS processor, the MIPS R10000 [MIPS95], removes the pipeline status pins and provides two dedicated on-chip 32-bit counters instead. The events these counters monitor, shown in Table 2.4, are similar to the R4400 events except for a few additions. First, a difference is drawn between issued instructions and graduated instructions. This distinction is important because not all instructions that are issued will graduate due to speculative execution. These measurements give feedback to the processor and compiler designers on the effectiveness of the dynamic and static scheduling of instructions. A second addition to the event table is the *way prediction* entry. The secondary cache is two-way set-associative, with the two ‘way’ tag comparisons done

Performance Counter 0		Performance Counter 1	
Index	**Event**	**Index**	**Event**
0,1	total instruction issues / 2	0	data cache miss
2,3	pipeline dry (no instructions ready for issue, caused by cache miss, misprediction, exception, delay slot)	1	instruction cache miss
4,5	load instructions	2	dual issues
6,7	pipeline frozen (no instructions issued due to resource conflict)	3	branch mispredictions
8,9	branch instructions	4	floating-point instructions
A	total cycles	5	integer operations
B	PALmode cycles	6	store instructions
C,D	total non-issues / 2	7	external input pin
E,F	external input pin		

TABLE 2.2. DEC Alpha 21064 performance-monitoring counters and inputs.

Index	Event	Index	Event
0	other integer instruction	8	other stall — write buffer full?
1	load instruction	9	primary instruction cache stall
2	untaken conditional branch	A	primary data cache stall
3	taken conditional branch	B	secondary cache stall
4	store instruction	C	other floating-point instruction
5	reserved	D	branch delay instruction killed
6	multiprocessor stall	E	instruction killed by exception
7	integer instruction killed by slip	F	floating-point instruction killed by slip

TABLE 2.3. MIPS R4400 externally observable events.

sequentially in time (in other products, they are done in parallel). This event measures the effectiveness of the *way prediction table* to predict which way to check first. The third addition is the inclusion of cache state events, such as counting external intervention or invalidate hits. These events may indicate performance loss due to multiprocessor data sharing conflicts and can be useful for tuning parallel programs[3].

The MIPS processors allow the operating system to grant user-level access to the performance counters and timer on a per-process level.

3. It will be shown in the Invalidation Misses portion of Subsection 3.3.2 that counting these is not as useful as it may seem.

Performance Counter 0		Performance Counter 1	
Index	**Event**	**Index**	**Event**
0	cycles	0	cycles
1	issued instructions	1	graduated instructions
2	issued load/prefetch/sync/cacheop	2	graduated load/prefetch/synch/cacheop
3	issued stores	3	graduated stores
4	issued store conditionals	4	graduated store conditionals
5	failed store conditionals	5	graduated floating-point instructions
6	decoded branches	6	quadwords written back from primary data
7	quadwords written back from secondary cache	7	TLB refill exceptions
8	correctable ECC errors in secondary cache	8	mispredicted branches
9	instruction cache misses	9	data cache misses
A	secondary cache misses (instructions)	A	secondary cache misses (data)
B	secondary cache misprediction from way prediction table (instructions)	B	secondary cache misprediction from way prediction table (data)
C	external intervention requests	C	external intervention hits
D	external invalidate requests	D	external invalidate hits
E	virtual coherency condition	E	stores or prefetches with store hint to CleanExclusive secondary cache blocks
F	instructions graduated	F	stores or prefetches with store hint to shared secondary cache blocks

TABLE 2.4. MIPS R10000 performance-monitoring counters and inputs.

2.2.4 Hewlett-Packard PA-RISC

Hewlett-Packard has deemed performance monitoring hardware important enough to reserve room in the PA-RISC instruction set architecture (ISA) [HP94] and to define a performance monitor coprocessor and interrupt. Two specific instructions are currently defined, `PMDIS` and `PMENB`, to disable and enable data collection respectively. The performance monitor coprocessor is left as an implementation-defined unit, but additional room is left in the ISA for future standardized extensions. Further details about the HP performance monitors are kept confidential, but some insight may be gained by looking at the Convex SPP systems described in Section 2.3.5.

2.2.5 Sun SPARC

The first SPARC implementation to contain performance monitoring functions is the SuperSPARC II [Sun95]. This implementation includes a high-precision cycle counter and an instruction counter. From [Sun95], it is unknown whether these can only be accessed at the supervisor level. The more recent UltraSPARC I design claims to contain ‘performance instrumentation’, but it is not clear whether it provides any additional features.

Like the R4400, the SuperSPARC I and II contain special-purpose pins that permit observation of pipeline events. A total of 10 signal pins are defined in SuperSPARC I and this is extended into 48 signals, divided into 4 groups and multiplexed onto 12 pins, in SuperSPARC II. Despite the larger number of signals, the functionality of these pins is roughly similar to the MIPS R4400 outputs.

2.2.6 IBM/Motorola/Apple PowerPC 604

The latest PowerPC chips, the 604 [Motorola94a][Motorola94b] and the forth-coming 620, contain two performance monitoring counters, a dedicated cycle counter, and 2 special address sampling registers. The counter inputs for the 604 are described in Table 2.5. The sampling registers are useful for applications like gprof which periodically sample the program counter, but the idea is extended to cover the most recently accessed data address as well.

The PowerPC 604 has two other interesting features. First, a performance monitoring interrupt (PMI) can be caused by the counters overflowing (defined as becoming negative) or by a threshold event, which is defined as a primary cache miss which is not serviced within *threshold* cycles, a programmable value between 0 and 63. When a PMI occurs, the sampling registers point to the instruction and most recently accessed data address. Second, there are four special counter events (9, A, 17, 18) to measure misses that took too long to be satisfied. The difference between counters 9 and 17, for example, is that event 9 is counted if the pin L2_INT is asserted, otherwise event 17 is counted. The pin is intended to signal that another L2 cache controller on a snoopy multiprocessor bus is sending the response, but it needn't be used that way. Another use for it would be to indicate that the memory was in contention or that the network was busy.

Unfortunately, these performance monitoring facilities are not defined in the PowerPC architecture, so it is not guaranteed whether they will be present in the future. Consequently, all performance counters are restricted to supervisor-only access.

2.2.7 IBM POWER2

Of all current processors, IBM's POWER2 contains the most performance monitoring features [Welbon94]. A total of 22 counters are available: one cycle counter, one 'correctable memory error' counter (which relates more to reliability-performance), and five counters in each of four functional units. The 20 functional unit counters can each observe one of 16 different items, for a total of 320 observable events. In addition, a special mode exists where 21 counters (all except the cycle counter) are dedicated to memory performance metrics of the storage control unit (SCU).

From [Welbon94], it is not known whether a user-level process has access to these counters, and precise details about the different counter events are not available.

Performance Counter 0		Performance Counter 1	
Index	**Event**	**Index**	**Event**
0	do nothing, hold value in counter	0	do nothing, hold value in counter
1	processor cycles	1	processor cycles
2	instructions completed per cycle	2	instructions completed
3	real-time counter extension	3	real-time counter extension
4	instructions dispatched	4	instructions dispatched
5	instruction cache misses	5	load miss cycles
6	data TLB misses	6	data cache misses
7	branches mispredicted	7	instruction TLB misses
8	reservations requested	8	branches completed
9	load data cache misses which took more than *threshold* cycles and were satisfied by another L2 cache	9	reservations obtained
A	store data cache misses which took more than *threshold* cycles and were satisfied by another L2 cache	A	mfspr instructions dispatched
B	mtspr instructions	B	icbi instructions
C	sync instructions	C	pipe-flush operations
D	eieio instructions	D	branch unit produced results
E	integer instructions	E	SCIU0 produced results
F	floating-point instructions	F	MCIU produced results
10	LSU produced result without exception	10	instructions dispatched to branch unit
11	SCIU1 produced results	11	instructions dispatched to SCIU0
12	FPU produced results	12	loads completed
13	instructions dispatched to LSU	13	instructions dispatched to MCIU
14	instructions dispatched to SCIU1	14	snoop hits
15	instructions dispatched to FPU		
16	snoop requests received		
17	marked load data cache misses which took more than *threshold* cycles and were **not** satisfied by another L2 cache		
18	marked store data cache misses which took more than *threshold* cycles and were **not** satisfied by another L2 cache		

TABLE 2.5. PowerPC 604 performance-monitoring counters and inputs.

2.2.8 Processor Summary

A summary of the performance-monitoring features found in current processors is presented in Table 2.6. Where information about a processor is not known, the entry is shown

as a question mark. As seen from the table, two performance-event counters and a timestamp counter are common. Unique features among processors include the Pentium's output pins, Alpha's input pins, PowerPC's sampling registers and threshold conditions, and POWER2's reliability counter. Positive trends are providing user-level access to counters and, as a result, instruction set architecture definitions for accessing them.

	Pentium	**Alpha**	**R4400**	**R10000**	**PA-RISC**	**Super-SPARC II**	**PowerPC**	**POWER2**
No. counters	2	2	0	2	?	1	2	21
Total No. Events	42	17	0	32	?	1	41	321+21
Events per Counter	42	9/8	0	16/16	?	1	24/20	16+1
Counts External Events?	n	y	n	n	?	n	n	n
Timestamp Counter?	y	y	y	y	y	y	y	y
User-level Access?	timer only	n	y	y	start/stop only	?	timer only	?
Performance Interrupt or Exception?	y (wiring required)	y	y (timer only)	y (timer only)	y	?	y	y
Externally Observable Events?	n	n	y	n	?	y	n	n
ISA Definition?	y	y	n	n	y	?	n	n

TABLE 2.6. Overview of performance-monitoring features on current processors.

2.3 Hardware Monitoring in Systems

This section describes hardware performance monitoring features that have been included in recent parallel computing systems. First, the Hector, DASH and M-Machine research systems are described. These are followed by commercial systems from Cray, Convex and KSR.

2.3.1 University of Toronto Hector

Hector [Vranesic91][Stumm93] is a shared-memory multiprocessor with a hierarchical organization. At the lowest level, a processor, I/O, and globally-addressable memory are coupled together. These processor-memory modules are connected by a bus to form a *station*, and stations are connected via a hierarchy of bit-parallel unidirectional rings obeying a slotted-ring protocol. The processors, 20 MHz Motorola 88000's, each have 16KB of instruction and data cache memory, but no cache coherence is provided in hardware. Per-

formance monitoring is done with a plug-in card that connects to either the instruction or data cache/memory management unit (CMMU) sockets.

Hector Hardware Monitoring: SUPERMON

By plugging into the CMMU socket, SUPERMON [Bacque91] is able to snoop on all processor-to-memory traffic. In particular, all cache activity is observable, including instruction fetches. This provides SUPERMON with the ability to perform exact program counter sampling, for example.

A block diagram of SUPERMON is shown in Figure 2.1. SUPERMON data collection is based upon two banks of 32K deep by 48-bit wide SRAM to be used as counters or for trace data storage. In trace mode, the memory is triggered by a programmable state machine which takes inputs from control signals and the virtual address. The trace data contains either the 30-bit address and a 16-bit timestamp or 8-bits of user-defined state data and a 40-bit timestamp. The state data is a function of address and control signals. The logic to control the state machine, control signals, and address trigger is all performed with additional cascaded SRAMs. Performance of the cascaded SRAM is maintained by inserting registers between them and pipelining the design. This approach was chosen over using PALs because SRAM provides good logic capacity, predictable performance and is easier to reprogram.

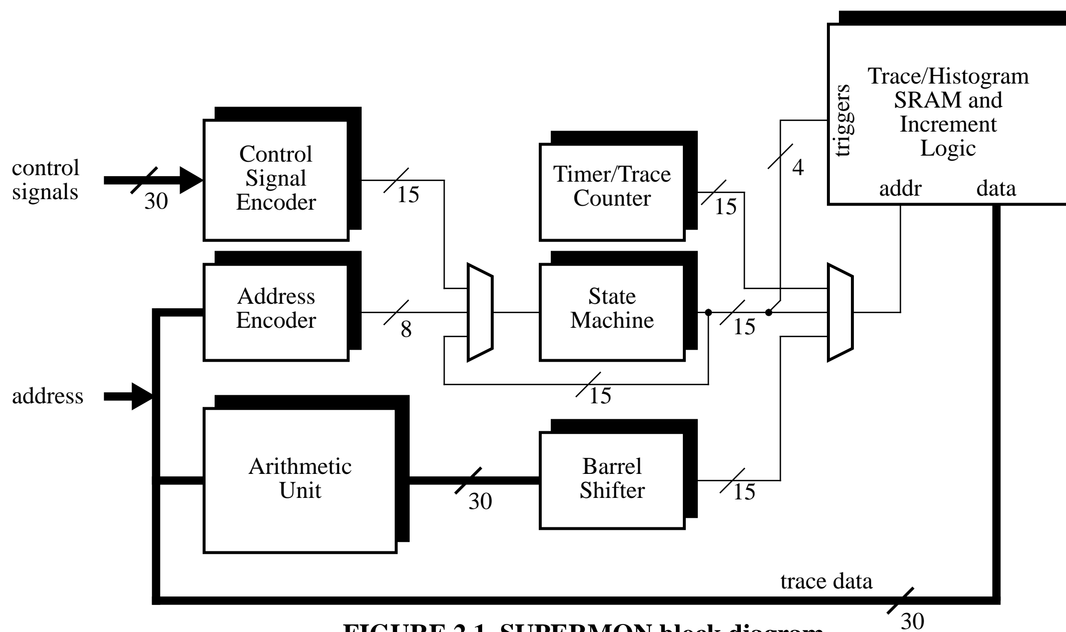

FIGURE 2.1. SUPERMON block diagram.

In histogram mode, the SRAM-based counter banks are interleaved because of speed; reading, incrementing, and writing a counter takes 2 cycles. The histogram counters are effective for counting:

- how much time is spent in a certain user-defined state (there are 32K possible states),

- histogram latency; how often an operation takes 1 cycle, 2 cycles, ..., up to 32K cycles, or
- histogram absolute addresses, addresses relative to a fixed offset, or sequential address differences.

By building a histogram of addresses relative to a fixed offset, it is easier to monitor accesses to data structures like an array which can have an arbitrary starting address. Additionally, an interesting feature is provided while histogram data on addresses is collected: the bucket size can be constant, by shifting the address through a barrel shifter, or it can be variable by re-encoding the address into a floating-point representation. This encoding extends the dynamic range of the histogram to cover the entire memory space while still providing fine-sized buckets in a region of interest (denoted by the fixed offset).

Using SUPERMON

SUPERMON is controlled by a 68000-based microcomputer board, called Gizmo; both the data-collection SRAM and the "programmable-logic" control SRAM are read or written in this manner. To program SUPERMON, a C program is compiled and run to generate the appropriate data that is loaded into the control SRAM via the Gizmo. Similarly, performance data is only available to the Gizmo computer. Fortunately, Gizmo has an Ethernet interface, so the performance data can be transferred to a workstation. This arrangement, however, effectively prevents a program running on Hector from examining its own performance data. This precludes the use of adaptive programs on Hector, so the data is only useful for off-line performance tuning of an application. A benefit of this architecture, though, is the ability of a single workstation to nonintrusively and continuously collect data from multiple SUPERMON nodes via the Ethernet. This can be useful to characterize the machine workload or to check the machine's performance over time.

2.3.2 Stanford DASH

DASH [Lenoski92] is a cache-coherent shared-memory machine. The lowest-level node consists of four 33 MHz R3000 processors connected to memory and an interconnection network interface over a 16 MHz bus. Each processor contains 64 KB of instruction and 64 KB of write-through data cache backed by a 256 KB write-back second level cache. The network interface holds cache directory information and uses a two-dimensional mesh interconnect. The DASH performance monitor is placed on the directory controller, one of the two network interface boards on the bus. The monitor consists of two banks of SRAM-based counters, a DRAM-based trace buffer, and a programmable controller. Unlike SUPERMON, the DASH monitor is organized to permit every processor in the machine read access to the collected data. A block diagram of the monitor is illustrated in Figure 2.2.

The SRAM-based counters take two cycles to operate: one to fetch and one to increment and write back. The two banks can be interleaved to count an event that may change every cycle, or they can be used independently to count less frequent events. Each bank is 16K deep and 32 bits wide, providing approximately four and a half minutes of continuous

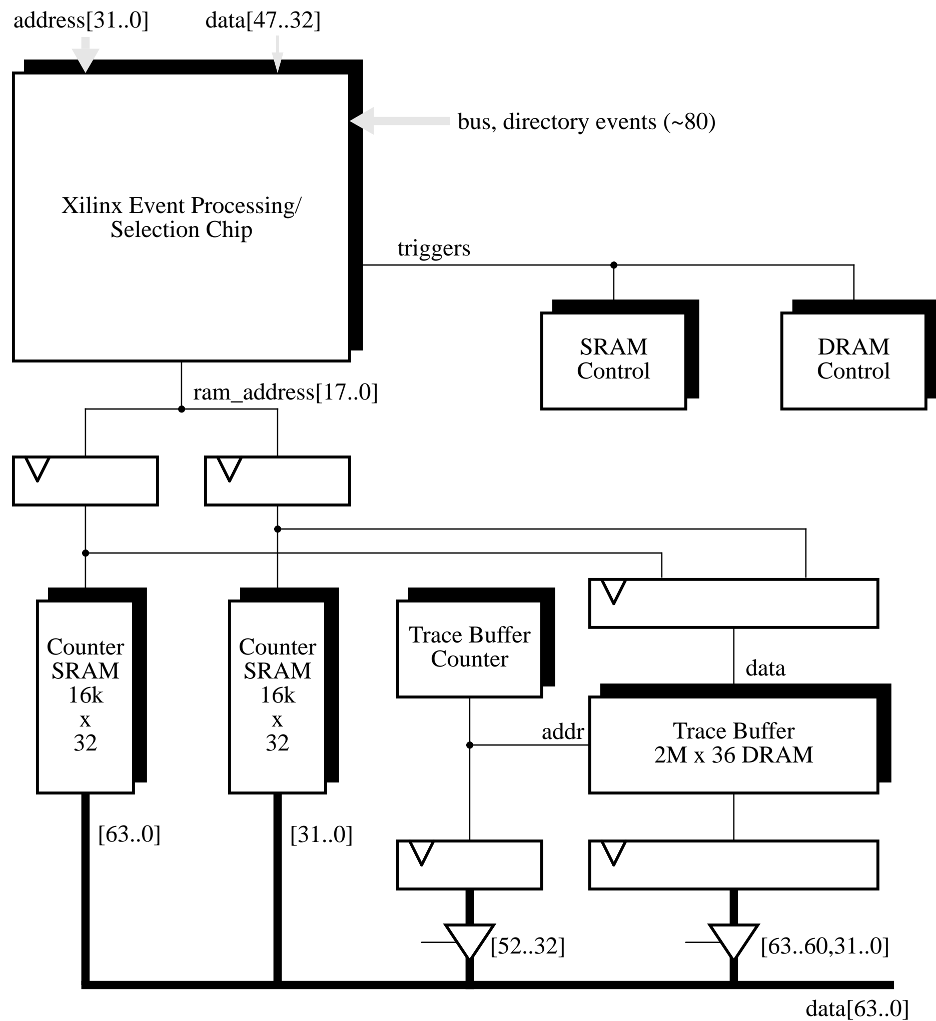

FIGURE 2.2. DASH performance-monitoring hardware.

operation before overflowing. The counters can be used to count events, where the address is formed by concatenating different event bits, or as a histogram array where the address comes from a counter in the programmable controller.

In addition to the counters, a trace buffer provides a detailed history of bus activity. The trace buffer DRAM, organized as 2M by 36 bits, is deep enough to capture information for over one-half a second; if longer traces are desired, the operating system must suspend all process activity, dump the buffer to disk, and resume the processes. A common use of the trace buffer is to capture read and write requests, including the address and processor number. Another use adds the directory controller's PROM address and the time since the last request to each trace entry, thus halving the effective buffer size.

The programmable controller is an SRAM-based FPGA, the Xilinx XC3090. To change the monitoring mode or triggering conditions, the FPGA is reconfigured with a different circuit. For example, the SRAM can count 14 independent events by directly forming the address with the 14 signals. Alternatively, SRAM can be used as a histogram array by implementing counters in the FPGA and using them to form the address. Although this

is a flexible approach, there are two drawbacks. First, reconfiguration time is on the order of 100 ms, so a program cannot frequently change the monitoring mode. Second, a software programmer must design a new hardware circuit if one doesn't already exist.

2.3.3 MIT M-Machine

M-Machine [Dally94] is a multiprocessor currently being designed at MIT to support massive parallelism. Up to 64K compute nodes are connected in a three-dimensional mesh. Each compute node consists of a custom processor, a multi-ALU processor or *MAP*, and 8 MB of memory. The MAP consists of four execution *clusters* and a switch matrix connecting them to four cache banks. Each cluster contains an integer, memory and floating-point unit as well as integer and floating-point register files. Within each cluster, an instruction can dispatch up to three operations. Also, each cluster can quickly switch between four user threads, an exception handling thread, and a system thread. The machine effectively supports a message-passing model, but it can also simulate a cache-coherent shared-memory model by using a small amount of support hardware along with the fast exception support and special system software.

The MAP also contains counters for performance evaluation. System software must be called to write or configure the counters, but user-level software can quickly read them. There are two 64-bit counters: a processor-cycle counter and an event counter. Using numerous mask fields, the event counter can be configured to count one or a combination of events. The mask fields select:

1. which operation unit and which thread slot (counts *operations*)
2. which thread slot (counts *instructions*)
3. which cache bank and which access type (counts cache *accesses, reads, writes, hits, and/or misses*)
4. which network event type (message send, flit send, message receive, flit receive, buffer allocates, event queue entries, overflows), which network priority level, and which thread slot (counts *network events*)
5. local TLB or global TLB hits (counts *TLB hits* — misses are counted by software)

2.3.4 Cray Research Inc.

Cray Research produces two types of parallel computers: parallel vector machines and parallel scalar machines. The parallel scalar machines, the Cray T3D and T3E, use the DEC Alpha microprocessors and do not have any hardware monitoring features (even the Alpha's internal counters are unused) so all performance debugging is done with software tools. The vector machines contain custom features and are described below.

The classic line of Cray supercomputers, from the CRAY X-MP up to the CRAY Y-MP C90, have all had performance monitoring capabilities [Cray92]. These computers are shared-memory vector processors. Rather than cache memory, they have instruction

buffers and vector registers; no coherence is maintained. The main memory is highly interleaved and tuned to deliver large blocks of data at very high bandwidths to quickly fill the instruction buffer and vector registers. Instructions are obtained from the instruction buffer and can operate either on normal scalar data or on vectors up to 64 elements long[4]. If an instruction demands use of a particular vector register or functional unit, for instance, it is delayed until the resources are free; this is called *holding issue*, and is typical because vector operations can take many cycles to complete.

The performance-monitoring counters are 48-bits wide. Initially, Cray divided up the counters into four groups of eight, and only one group could be counted at a time. The CRAY Y-MP C90 removes this restriction and provides 32 dedicated counters, shown in Table 2.7. Although the grouping of the events has changed from machine to machine, there has been little change to the event list itself.

2.3.5 Convex Computer Corporation

The Exemplar Scalable Parallel Processing (SPP) systems by Convex [Convex94] are distributed shared-memory cache-coherent machines. At the lowest level, up to eight processors and caches are connected to memory using a high-speed 5 x 5 crossbar forming a *hypernode*. Currently, there are two types of hypernodes available which can be freely mixed in a system, but they differ in the processor used and performance monitoring capabilities. Up to 16 hypernodes can be tightly coupled using four SCI rings to provide a single shared-memory system.

The performance monitoring features of the two Exemplar hypernodes are described in [Convex95]. For timing, both systems contain a high-precision per-processor timer and a separate per-hypernode timer. Additional performance-monitoring capabilities are specific to each system, so they are described separately below.

Exemplar SPP-1000

The SPP-1000 hypernode, based on the Hewlett-Packard PA-7100 processor, has two performance-monitoring registers. One register may count cache misses, including any combination of local (same hypernode), remote (other hypernode), read or write misses. Alternatively, it can also count the number of hardware messages sent or the number of coherence requests sent and/or received. The other register can measure total cache miss latency, but only when the first is counting cache misses.

Exemplar SPP-1200

The SPP-1200 hypernode uses the more recent Hewlett-Packard PA-7200 processors and has more performance monitoring features. A total of four registers can count from a variety of events. One of these registers counts any combination of local, remote, read, and write cache misses (this is similar to one of the Exemplar counters). The other three

4. The Cray Y-MP C90 can operate on vectors containing up to 128 elements.

Counter Group	Counter	Event
number of:	0	clock cycles
	1	instructions issued
	2	clock cycles holding issue
	3	instruction buffer fetches
	4	CPU port memory references
	5	CPU port memory conflicts
	6	I/O port memory references
	7	I/O port memory conflicts
number of cycles holding issue for:	8	A registers
	9	S registers
	A	V registers
	B	B/T registers
	C	functional units
	D	shared registers
	E	memory ports
	F	miscellaneous
number of instructions:	10	000-004 (special)
	11	branches
	12	A-register instructions
	13	B/T memory instructions
	14	S-register instructions
	15	scalar integer instructions
	16	scalar floating-point instructions
	17	S/A register memory instructions
number of operations:	18	vector logical
	19	vector shift/pop/LZ
	1A	vector integer adds
	1B	vector floating multiplies
	1C	vector floating adds
	1D	vector floating reciprocals
	1E	vector memory reads
	1F	vector memory writes

TABLE 2.7. CRAY Y-MP C90 performance-monitoring counters.

counters select from the seven events in Table 2.8. Additionally, one of these three registers can also count instruction or data cache miss latencies.

Index	Event
0	data cache accesses
1	data cache misses
2	instruction cache misses
3	data TLB misses
4	instruction TLB misses
5	pipeline advances (slightly overcounts instructions issued — it likely includes pipeline slips)
6	processor cycles

TABLE 2.8. Convex Exemplar SPP-1200 performance-monitoring counter inputs.

2.3.6 Kendall Square Research

The KSR1 [KSR92a] and KSR2 machines by Kendall Square Research are interesting multiprocessors because they implement a cache-only memory architecture, or *COMA*. To support COMA, KSR designed a custom processor. The processors are connected in a hierarchical ring structure: up to 32 processors are connected together on the first-level ring and up to 34 rings can be connected by a second-level ring. Before describing the performance monitoring features on the machine, it is first necessary to describe how COMA works.

The *local cache* coupled to each processor, akin to main memory, is 32 MB of DRAM. This should not be confused with the processor's *data subcache*, which is essentially a normal data cache. When a processor accesses memory and it misses in the local cache, a page-sized chunk of 16 KB is allocated in the local cache. Cache-coherence hardware searches for the address throughout the system, first on the local ring and then on remote rings. If found, a copy of the 128 byte *subpage*, similar to a cache line, is copied or moved to the local cache. For details concerning actions taken when the address is not found, or what is done to the displaced page, the reader is referred to [KSR92a].

Performance Monitoring

The KSR processor, which is identical in KSR1 and KSR2 except for clock speed, is rich in performance-monitoring counters [KSR92b]. Fourteen 64-bit counters are implemented in hardware and four measurements are taken in software for every thread. The hardware counters are described in Table 2.9. Additionally, the operating system counts the number of page faults and processor migrations experienced by every thread. Also, the number of page hits is computed in software by adding subpage hits and subpage misses. Finally, the compiler inserts code to measure the number of instructions executed, but this information is not normally accessible and must be extracted by inserting special assembly code [Manjikian95].

Index	Event
1	user clock cycles / 8
2	wall clock cycles / 8
4	stalled cycles due to instruction fetch miss
5	cycles lost to instructions ‘inserted’ for I/O and timer interrupt processing
6	cycles lost to instructions ‘inserted’ for cache coherence processing
8	subpage hits
9	page misses
A	subpage misses
B	subpage miss time
C	data subcache subblock miss (*i.e.*, data cache misses)
D	subpage misses which are satisfied through the second-level ring
E	issued prefetches
F	issued prefetches that miss in the local cache (*i.e.*, a potentially useful prefetch)
10	issued prefetches to an address that are currently outstanding (*i.e.*, useless prefetches)

TABLE 2.9. KSR performance-monitoring counters.

2.4 Summary

A number of software and hardware performance tools have been described in this chapter. On the software side, a program may be profiled by counting basic blocks and using program sampling or high-resolution timers with an overhead within 10%, but detailed information about cache activity and multiprocessor data sharing is not available. The more detailed software tools are extremely useful for this because they can correlate cache misses to exactly *where* the misses are in the program code and data, and also indicate *why* the cache misses occur. However, their main drawback is that they are roughly two orders of magnitude slower. Additionally, software techniques often ignore details such as network or memory contention and the highly-variable memory latency of a cache-coherent NUMA multiprocessor. Adding these features would likely reduce speed by another order of magnitude. The extremely slow nature of these tools hinders their usability.

With respect to processors, performance monitoring features are becoming quite common. The latest processors include performance counters as well as a high-resolution timestamp counter. However, some designs have hindered the usability of these counters by requiring privileged access to them. Furthermore, the majority of processors contain only two counters. Also, most counters are set up to have asymmetrical sets of input events so they cannot measure an arbitrary pair of performance events. Since obtaining good utilization of resources is key to performance, more area should be devoted to performance-observing hardware. A greater number of counters and improved performance-monitoring features would make it much easier to tune a program. If the monitoring features are not easy to use, many programs will never be tuned.

Multiprocessor systems also include performance monitoring features. Research systems have included tracing ability along with SRAM counters and histograms, but commercial systems focus specifically on counting the type of instructions issued and cache misses. However, few systems measure network or memory contention; the closest measurements are Cray memory conflicts and the KSR or Convex miss latencies.

Chapter 3

Multiprocessor Hardware Performance Monitoring

In the previous chapter, a number of software- and hardware-based performance tuning tools were presented. This chapter builds upon these tools by taking many key ideas and merging them together. The result is a description of hardware support that would be most useful for performance monitoring in cache-coherent, shared-memory multiprocessors; many of these features can easily be placed directly on a processor.

First, the different sources of performance loss from the software and hardware viewpoints are analyzed. This is followed by a comprehensive description of memory stalls. Then, hardware features that can assist performance-tuning software tools are described. At the end of each of these sections, a summary states the recommended features that should be included in a hardware-based performance monitor. Many of these features will be implemented in the NUMAchine performance monitor to be described in the next chapter.

3.1 Performance Losses: Software View

This section focuses on performance losses from a software perspective. This strict software view disregards the operation of the hardware and explains performance loss in terms of inefficient algorithms, extra parallel work, synchronization, load imbalance, inefficient system calls, and Amdahl's Law. In this section, a brief description of these components is presented. Together, they share the common quality that they can all be measured by instrumenting a program with timers. This simple instrumentation provides insight into *where* software performance is being hindered, but details about the hardware are often required to fully understand the performance loss.

3.1.1 Inefficient Algorithms

It is well known that choosing the right algorithms and data structures is imperative for fast programs. For example, using a hash table can reduce lookup times from O(n) to O(1). However, hash tables can have poor spatial locality, so even a hash table may exhibit poor performance. Obviously, the choice of algorithm can greatly influence performance and the programmer should be sensitive to this. In this thesis, it shall be assumed that a reasonable algorithm has already been developed and coarsely tuned using standard analytical techniques such as time-complexity analysis and software profiling.

In addition to the standard algorithms used to improve performance, the programmer should be aware of many software transformations that can accelerate performance in

cached-memory systems. Techniques such as loop interchanging, loop fusion, strip mining, blocking, merging arrays, and packing or padding structures all can improve locality and accelerate performance [Bacon94]. However, gaining insight into whether these options are useful requires insight into the hardware operation.

3.1.2 Extra Parallel Work

By its nature, a parallel program must do more work than a sequential program to manage its activity. For example, threads must be spawned, locks must be acquired and released, and the task needs to be divided among the threads. Because it is unnecessary in a sequential program, this activity constitutes overhead which reduces performance of a parallel program.

3.1.3 Synchronization

The time spent by one processor waiting for another to complete a task is time wasted due to synchronization. Although time used to acquire a lock is considered to be extra parallel work, the time spinning while the lock is busy is lost to synchronization. Also, waiting for a semaphore to be signaled is considered to be synchronization overhead.

3.1.4 Load Imbalance

A parallel program often divides a problem into many parts that are run in parallel. Ideally, the work is divided such that each processor takes exactly the same length of time to complete. In reality, however, there exists indeterminism that causes the work to be divided in an unequal manner, so that some processors complete before others and end up waiting idle. The indeterminism can be caused by software, where it is unknown whether some portions of code will be executed due to conditional statements, or by the hardware, where, for example, caches can cause unpredictable memory access times.

The time spent idle waiting for other processors to complete their portion of work is referred to as load imbalance. Although it is sometimes considered to be a part of synchronization overhead, load imbalance is usually more coarse-grained. Specifically, time spent waiting at a barrier is typically due to load imbalance.

3.1.5 Inefficient System and Library Calls

Because their operation is usually hidden from the user, system calls and library calls are a source of unknown behaviour and can impact performance. First, these routines are often general in nature and may not be optimized for the required use. Second, these routines may have side effects, such as explicitly flushing the data cache or replacing critical portions of the instruction cache. Third, routines that involve input or output can cause performance loss. Finally, performance tuning tools usually cannot profile operating system activity, so it can be difficult to tune this aspect of a program. As a result, system and library calls may cause significant performance degradation.

3.1.6 Amdahl's Law

The final source of parallel performance loss is a result of Amdahl's Law [Amdahl67], which states that the speedup resulting from a specific performance optimization is limited by the fraction of time in which the optimization applies. Although Amdahl's Law is not a cause of lost performance per se, it explains why a parallel program does not track the ideal linear speedup curve. For example, suppose a parallel program is run on two processors in five seconds, of which one second is spent in unparallelizable sequential code. When run on 16 processors, one might expect the program to run eight times faster, *i.e.,* run in 0.625 seconds, but Amdahl's Law dictates that the best running time is 1.5 seconds. To further illustrate this, the speedup curve for this example is shown in Figure 3.1. By timing the sequential and parallel portions of a program, the amount of speedup lost due to the sequential portion of a program becomes obvious.

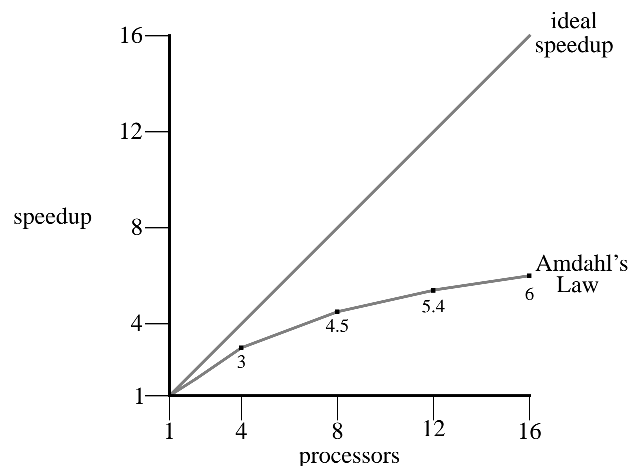

FIGURE 3.1. An example of Amdahl's Law limitations on speedup.

3.1.7 Summary

From a software perspective, parallel performance is affected by the choice of algorithm, the amount of extra parallel work, waiting for synchronization, improperly balanced workloads, and inefficient system software and libraries. Additionally, parallel speedups are governed by the fraction of parallel versus sequential work that exists in a problem. The time to execute these components can easily be measured with software, but without an in-depth knowledge of the hardware operation, it is difficult to understand *why* it is slow. The next section addresses this issue by providing a hardware-oriented view of performance.

3.2 Performance Losses: Hardware View

In the previous section, the inefficiencies of a program are described from the viewpoint of a programmer. That perspective helps address *where* the program is running slowly, but it does not readily explain *why* the operations themselves are slow. To understand this, it is necessary to measure the performance of the underlying hardware. In this section, the hardware-oriented factors which influence the speed of a program are presented. Note that factors such as processor cycle time and instruction set architecture are considered to be fixed characteristics which cannot be changed.

3.2.1 Instructions

The first natural source of performance loss relates to how many instructions are needed to accomplish the task. Modern compilers perform a number of optimization steps, such as common subexpression elimination, which directly minimize the number of instructions required. In particular, since it is more important to minimize the *dynamic instruction count*, or the number of executed instructions, than to minimize the static instruction count, compilers spend more effort optimizing code inside loops.

Although it is not generally possible to predict the minimum number of instructions possible, it is feasible to measure improvements in dynamic instruction count. Software tools such as Mtool are quite good at accumulating dynamic basic block counts, hence, instruction counts. Also, a number of processors and systems provide direct hardware support for counting instructions: Pentium, Alpha 21064, MIPS R10000, SuperSPARC II, PowerPC 604 and POWER2, MIT M-Machine, and Cray vector computers can all do this. Furthermore, both hardware and software systems can break down the instruction counts into groups of instructions, with software having the advantage of being able to form any number of arbitrary groups. On the other hand, hardware has the advantage of being completely transparent to the user. This is especially powerful in a machine like the Cray Y-MP C90, where each counter monitors only one event, because one can profile the machine use for workload characterization and benchmark synthesis with no impact on users [Gao95].

3.2.2 Instruction Scheduling and CPI

After instructions have been generated, compilers often rearrange their order to improve performance while still preserving semantics. Additionally, out-of-order execution techniques employed by processors such as the MIPS R10000 or Pentium Pro can further rearrange the order of instruction execution. In these cases, the instructions are *scheduled* to meet the availability and speed of resources such as a floating-point division unit. The average number of cycles required to execute an instruction is a common measurement of efficiency called *CPI*, or cycles per instruction.

Performance is lost, *i.e.*, CPI increases, when instructions are scheduled in a poor order. This can be observed in one of two ways: by the number of executed NOP instructions and by the number of pipeline slips (also known as interlocks or bubbles). Software

like Mtool can easily determine the number of NOPs from basic block profiles, but slips are transparent.

During a *slip*, some portion of the instructions in progress are still executing while others are halted. Slips occur because there is a dependency between an instruction currently in execution and one that was just issued, namely they both access a common resource or the second instruction uses the result of the first. Good instruction scheduling can reduce CPI by putting more independent work between two dependent instructions. Of course, there are limits to the amount of independent work that can be found and this has been well documented in literature [Hennessy96]. As a result, it is desirable to reduce compute cycles lost to interlocks as much as possible, so some means of measuring interlocked cycles should exist.

Currently, programmers rely upon processor pipeline simulators, such as *pixie* [Smith91], to measure these lost cycles. However, these tools are primarily aimed at uniprocessor applications and haven't yet been extended to handle parallel programs.

Surprisingly, few systems are capable of counting interlocked cycles in hardware. In particular, only POWER2 and Cray systems can directly measure interlocks separately from memory stalls. Also, the MIPS R4400 and SuperSPARC processors have observable pins that indicate this type of pipeline activity, but external counters must be constructed. Processors should measure the performance lost due to interlocks to motivate and measure improved scheduling algorithms.

3.2.3 Memory Stalls

Although loads and stores must be scheduled to reduce their likelihood of causing stalls, it is useful to measure the different types of such stalls separately. The most significant type of load or store stall is a cache miss stall. The performance impact of cache misses is significant and complex, so Section 3.3 will be devoted fully to this issue.

3.2.4 Other Stalls

Stalls which are caused by loads and stores but do not involve cache misses are considered in this subsection. These include stalls due to write buffer overflows or attempting to access a busy cache. The cache may be busy satisfying a previous request by the processor, or it may be processing coherence events, such as snoops or invalidates, from outside the processor.

First, consider measuring these stalls in software. Because they are transparent to the program, they can only be measured by simulation. The pixie simulator, for example, measures write buffer overflows but it cannot model coherence events. Since coherence events are relatively frequent and very sensitive to the order in which instructions on different processors are run, they are very difficult to simulate. Consequently, it is best to count them in hardware.

Existing processors count these events with varying usefulness. For example, the Pentium counts pipeline write buffer stalls, snoops and snoop hits. Similarly, MIPS R10000 and PowerPC can count snoops and snoop hits. Additionally, the SuperSPARC outputs a signal while the write buffer is full and the MIPS R4400 outputs a signal for coherence 'multiprocessor stalls' and another for 'other stalls', presumably due to write buffer overflows and uncached-memory operations. The MIPS R4400 provides the most useful information, but the counter hardware must be built externally. The Pentium provides the next most useful information by counting write buffer stalls. Many of the processors count cache snoops and snoop hits, but this information is not useful unless these events directly affect performance by causing the cache to be busy when it is needed by the processor. As will be shown later, these counts are also misleading when measuring performance loss due to the memory system.

3.2.5 Exceptions and Interrupts

Because they temporarily usurp a currently running process, processor exceptions and interrupts can significantly impact performance. Examples of common exceptions are page faults, TLB misses (faults), or floating-point underflow.

By their nature, most exceptions and interrupts must be serviced by a software routine which can be instrumented to measure the amount of time executing outside of the usurped process. However, two exception-related conditions cannot be measured by software: hardware-serviced TLB misses and pipeline flushing. First, some processors contain sophisticated TLB-fault handlers in hardware, so an exception is never raised on a miss. The PowerPC is an example of such a processor and, fortunately, its counters can monitor TLB misses. Second, the pipeline must be flushed and restarted when an exception is raised and completed, respectively. The performance penalty for doing this is not easily accounted for by software because the operations take a variable length of time. Consequently, hardware should measure the performance impact of pipeline flushes and restarts.

Limited hardware support for this is provided on the Pentium, Alpha and PowerPC, where pipeline flushes can be counted, and on the MIPS R4400 where an output indicates pipeline cycles killed for an exception. However, none of these processors can measure the cost of restarting the pipeline; future processors should monitor this.

3.2.6 Branch Mispredictions

All new processors have some type of branch prediction scheme to indicate which way conditional branches are likely to flow. Prediction schemes can be static, such as those used by the KSR, or dynamic as in the Pentium. The dynamic branch prediction schemes are generally more effective because they can handle cases where the branches alternate between taken and not-taken, for example. A mispredicted branch can result in many lost compute cycles, so it is important that branches are well predicted.

Interestingly, all processors that employ *dynamic* branch prediction can also count the number of mispredictions with their monitoring hardware. Unfortunately, they do not

account for the cost of mispredictions, which can vary depending upon instructions in-issued and to-be-issued. The performance lost due to mispredictions should be measured to motivate better prediction schemes and improvements in the recovery time of a misprediction. However, this is very difficult to measure in hardware because it implies a non-causal view of program execution.

Even assuming that such measurements can be made, it is difficult to improve the number of mispredictions or the associated penalty by improving a compiler. However, the recent advocacy of conditional move instructions may allow new freedom for compilers to exploit the trade-offs between using branches, which may mispredict, or using conditional moves, which may be translated into NOPs. Tools should be provided to explore such trade-offs.

3.2.7 Special Cases

Due to architectural or implementation differences in processors, there will always be special cases for a given processor which will affect performance. For example, the KSR inserts instructions into the instruction stream to handle some types of coherence requests, and the MIPS R10000 uses a novel prediction table to provide a two-way set-associative second-level cache. In each of these special cases, the effectiveness of the special feature can be measured with the performance-monitoring hardware designed in the processor.

In addition, new processor implementations are including special features that, when utilized correctly, accelerate performance. Examples are data prefetching, fast context switching, and more aggressive schemes for extracting instruction-level parallelism. Given that these new developments continue, it is important to provide monitoring hardware to evaluate their performance. In particular, it is desirable to measure the amount of use of a specific feature, the number of times it helped improve performance, and the amount of performance that was lost if it involved some type of misprediction. These measurements will help determine the value of these new concepts and allow them to be adopted more quickly in the general marketplace.

3.2.8 Summary

As shown, there are many aspects of performance loss that are invisible to a programmer of parallel applications. Software tools are powerful and can calculate or estimate some of the sources of such loss, but hardware is generally better suited to the task. This section has considered a number of measurements that can be made in hardware:

1. **Count the number of dynamic instructions**, preferably with the ability to organize instructions into groups.
2. **Count the number of cycles lost due to NOPs and pipeline slips**.
3. **Count the number of cycles lost due to stalls.** Stalls caused by the memory subsystem should be reported separately and are detailed in the next section. Stalls arising from write buffer overflows or busy caches should be mea-

sured. These stalls can sometimes be reduced by scheduling and also by improving interprocessor coherence activity, which will also be described in the next section.

4. **Processing opportunity lost due to exceptions should be measured**. In particular, pipeline flushing and restarting cycles should be counted by hardware since it is unobservable by software.
5. If **TLB faults** are serviced in hardware, they should be counted. Additionally, the total service time is useful.
6. The number of **mispredicted branches** and, if possible, performance loss due to branch misprediction should be quantified.
7. If a system includes **special hardware features**, it is important to measure their effectiveness.

These measurements can be used to improve compiler development, characterize workloads, and tune the operating system behaviour. In addition, they can point a programmer towards selecting the proper compiler optimization flags or suggest code transformations to improve performance.

3.3 Processor View of Memory Stalls

In the previous section, numerous sources of performance loss were discussed. However, the single most significant source of performance loss is a processor stalled while waiting for the memory system to resolve a cache miss. Consequently, cache misses are a significant source of performance degradation for parallel programs and often account for 50% of performance loss [Hennessy96].

To regain this performance, it is necessary to understand the various types of misses that occur and how they impact a program's performance. This section describes the different types of misses and the components of miss latency. Additionally, it outlines hardware to measure these losses and how to trace them back to the program. In this way, it is easy to detect *when* and *where* a program is suffering from performance loss. The focus is primarily on a sequentially-consistent memory system with either an update or invalidate coherence protocol, but some attention is given to weaker memory consistency models.

Tools such as CProf and MemSpy have shown how memory tuning is an effective technique for gaining performance. However, these tools are slow and cumbersome because they are simulation-based. Additionally, they do not model contention and make assumptions about uniform memory speeds that are unrealistic in a NUMA machine. The hardware described in this section is intended to be more realistic about measurements, cost-effective and significantly faster than these tools. It should also provide significantly better insight into multiprocessor data sharing patterns.

This section is divided into three primary subsections. The first presents how cache misses can be counted and correlated with the running program, the second suggests a tax-

onomy of misses and how to detect them, and the third illustrates the different components of miss latency and how they can be measured.

3.3.1 Counting Cache Misses

It is a trivial matter for hardware to count the number of cache misses, either directly on the processor or with external hardware. However, it is more difficult to count misses separately for different regions of code or data or for different stages of a program's execution state. Two methods capable of providing this distinction will be presented below.

Counting Misses: Informing Memory Operations

A good way of counting cache misses incurred by each memory instruction is described in [Horowitz95]. The authors recommend a mechanism to invoke a low-overhead miss handler routine whenever a data miss occurs. They refer to this mechanism as an informing memory operation. The miss handler, which is generated by the programmer or is part of a library, can record the cache miss by incrementing a counter assigned to the particular memory instruction that missed. Thus, every memory instruction has a record of the number of cache misses it caused. Because every instruction can be traced back to a particular line of code or a certain data object, the programmer can use this information to locate potential bottlenecks.

To be useful, the informing memory operation must be fast, especially in the common case of a cache hit. Simulation results in [Horowitz95] indicate that a 10-cycle miss handler increases run time by no more than 30% for most applications. This slowdown is what motivates a fast mechanism for invoking the miss handler.

In [Horowitz95], three fast implementation techniques for informing memory operations are suggested, but they all require existing processors to be modified. A fourth technique, which they briefly hint at, sets up the memory system to return bad ECC data and causes an ECC-based trap to masquerade as a miss trap. However, the overhead of invoking such a trap is prohibitively expensive. Although informing memory operations are very attractive from a performance monitoring standpoint, they are fairly intrusive and cannot be implemented with current processors.

Counting Misses: Informing Hardware

Informing memory operations rely on hardware to give feedback to software to detect a cache miss. The opposite viewpoint would have software inform the hardware of the current software state or *phase*. This involves a program informing the performance-monitoring hardware of details such as which line of source code, basic block, loop iteration, or procedure is currently being executed.

A novel way to inform hardware of phase changes is to have software explicitly modify the contents of a special-purpose external hardware register, called a *Phase Identifier Register* or *PhaseID*. The contents of the PhaseID are under software control, so a pro-

gram can indicate whether information should be collected at a fine or coarse granularity, or both. When PhaseID changes, a different counter is selected to count cache misses. In this way, the cache misses for each phase are collected separately. Of course, the use of PhaseID need not be limited to counting cache misses; it can be used as a general mechanism for dividing-up performance data.

Numerous counters which are individually selected based upon the contents of PhaseID can be cost-effectively stored in SRAM. Connected to this, a single high-speed loadable incrementer can do the actual measurement. The cost of SRAM is modest and fast counters are simple to implement with inexpensive PALs or CPLDs [Zilic95], so it is reasonable to implement a 16-bit PhaseID with a 32-bit counter, for example.

To change the contents of an off-processor PhaseID, three different implementation methods can be used. In the first method, the data portion of an uncached-write is used to update PhaseID. Later in this chapter, this store will be used for a dual purpose in which the data portion will contain timing information. For this reason, the second method encodes the new PhaseID value into the address component of the write[1]. The third method of changing PhaseID is via a read, where the new contents must be encoded into the address and the response is discarded[2]. These different methods have different strengths and weaknesses, as discussed below.

Using a write to change the PhaseID has the least impact on performance, but some caching policies allow reads to bypass writes (which are held in a buffer). Consequently, read misses may be incorrectly counted in an earlier phase. The read method solves this problem, except that it may have a greater impact on performance[3] and improper counting of write-throughs or write-backs may occur. Other similar problems exist, especially for recent superscalar out-of-order processors with multiple outstanding reads. These problems can be solved by serializing reads and writes, but imposing an order on these instructions imposes unreasonable performance loss.

To avoid the serialization penalty in future processors, it is best to add PhaseID hardware directly to the processor; a user-level register-to-register move instruction can change PhaseID in a sequentially-consistent fashion. Rather than add the expense of moving the numerous counters on-chip as well, the PhaseID contents could appear in the address during every off-processor memory access. This can appear as part of unimplemented upper address bits or during an extra cycle after the address is emitted.

Informing hardware of the current software state can be used in a complementary fashion with informing memory operations. For example, software-controlled prefetching, an application suggested by [Horowitz95], would benefit from the lower overhead of recording discrete miss counts in the hardware.

---

1. There is an implicit assumption here that the hardware is memory-mapped and that there is sufficient room in the address space. These assumptions are true in NUMAchine.

2. An interesting use of this read is to return the previous PhaseID or some performance-counter result, but such options have not been explored in this thesis.

3. The read can be acknowledged immediately, so the performance impact is not too large.

3.3.2 Classifying Cache Misses

Once the number of cache misses is known, it is useful to further divide each group of misses to help identify the causes. The common nomenclature of compulsory, capacity, and conflict misses describes all forms of conflicts in a uniprocessor. In a shared-memory multiprocessor, additional misses occur because of the need to keep caches coherent. Such misses are often overlooked and have been poorly defined in the literature; the discussion below proposes definitions and less-ambiguous names for these misses.

Classifying Misses: Compulsory, Capacity, and Conflict Misses

Classifying cache misses as compulsory, capacity or conflict is difficult for hardware to do. First, to recognize compulsory misses it must know that the memory location was never referenced before, so additional memory state must be kept. In a large-scale multiprocessor, this represents a significant overhead of one bit per processor per memory block. Second, to detect capacity or conflict misses the last ***n*** most recently used memory blocks must be remembered and accessed each time the cache is consulted, where ***n*** is the number of cache lines. This is equivalent to simulating a fully-associative cache in hardware. Clearly, a large amount of state must be kept and complex processing must be performed to detect these different types of misses. While such processing is possible, it would significantly slow down the system and cause intrusion. Instead, compulsory misses will be ignored and a method to estimate conflict misses will be developed.

Compulsory misses are ignored because they are hard to measure. However, the significance of compulsory misses can be mitigated by two arguments: they occur infrequently and they are hard to reduce. First, compulsory misses are constant regardless of cache size because they occur when a memory block was never before referenced within the lifetime of the process. For this reason, the number of compulsory misses is constant. In a long-running program which touches memory many times and suffers an increasing number of misses, compulsory misses become more and more infrequent. Second, compulsory misses can be reduced by only a few methods: increasing the cache line size, compacting data structures, and trading off computation for memory accesses (by replacing a table lookup with a computation, for example). Unfortunately, the programmer seldom has the freedom to use these techniques because most machines have a fixed cache line size, data structures are usually as compact as possible, and adding more computation does not always work.

On the other hand, misses due to capacity or conflict are important but they are also very difficult to distinguish. To detect capacity misses, a fully-associative cache structure with least-recently used (LRU) replacement must be emulated in hardware — obviously this cannot be done economically at full speed otherwise the processor cache would employ this scheme. Likewise, some conflict misses can be measured by simulating a cache that is more associative cache than the processor's, but this is not feasible for the same reason. Another way of detecting conflict misses is to remember the last few misses and count every time a new miss maps to the same location. Unfortunately, this technique can only capture a few of the conflict misses.

A different approach to detect conflict misses is to profile the cache activity. Specifically, each cache line has a separate counter to tally its misses. By sampling this profile over the life of a program, an interesting history of cache use results. A completely uniform profile primarily indicates capacity misses, but it can also indicate conflict misses that cover the entire cache. To distinguish these, the cache profile may need to be sampled more frequently or the programmer may have to guess. However, sharp spikes in the profile indicate excessive conflicts at a specific cache line address. A more restrictive technique than this is used with some success in the SPARCcenter 2000 to identify conflict misses [Singhal94]. In their implementation, two separate counts for misses to even and odd cache lines are used to indicate potential conflict misses. By profiling every cache line, however, more conflicts can be determined and, more importantly, it is easier to trace the conflicts back to the program.

Conflicts can be traced back to specific code or data in the following way. Software should be able to match the cache location containing excessive conflicts to all static objects in a program and produce a candidate list of conflicting variables. To pinpoint dynamically allocated objects, a second run of the program with hardware watchpoints set on the cache line in question can stop a program when and where the conflict occurs. Unfortunately, processors aren't usually set up to have watchpoints on a cache line; instead, the performance-monitoring hardware should have this ability and generate an interrupt when a match is detected.

A variation of the watchpoint scheme can help identify conflicts in only one pass. Suppose the hardware monitor creates a cache profile and, at the same time, records the average number of misses per cache line; this is easily done by counting all misses and shifting the count by $log_2(\boldsymbol{n})$ bits, where $\boldsymbol{n}$ is the number of cache lines. While profiling, a watchdog inspects the difference between the average miss per line and the miss count of the line that just missed. When this difference exceeds some threshold, it can be assumed that the miss was caused by excessive conflicts at that location. As before, an interrupt can be generated to stop the program and pinpoint the source of the problem[4].

The hardware cache profiling described above does not distinguish compulsory, capacity, or conflict misses directly. Rather, it helps show active regions of the cache which may be suffering from excessive conflict misses. Although it is possible that invalidation or ownership misses (described below) can also create active regions, they can be directly detected by hardware so it is easy to exclude them from the profile. A drawback of this approach is that conflict misses which uniformly cover the entire cache, such as those arising from some types of array operations, may be unobservable from the profile. In these cases, software tools can be used instead.

Classifying Misses: Invalidation Misses

The first type of multiprocessor cache miss is presented here. When an invalidate-based coherence protocol is employed, data in the cache is sometimes changed by another pro-

---

4. A simpler, but less accurate, variation of this method generates an interrupt as soon as a cache line exceeds a certain number of misses. This variation is implemented in NUMAchine.

cessor, which also has a cached copy, and the stale copy needs to be eliminated or *invalidated*. These eliminations which originate from outside processors are called *external invalidation hits*. If the data is invalidated but subsequently required by the processor, a miss is incurred; the data was removed because of the invalidation. In previous literature, these misses have been called coherence misses [Jouppi90], but such terminology is misleading because it is not clear whether another type of coherence miss, which will be defined shortly, is included. In this thesis, the term used in [Martonosi95], *invalidation misses*, is recommended, but it should be defined differently:

Definition ***Invalidation miss*** is a miss that is incurred because a reference to data which otherwise would have been present in the cache was previously invalidated in order to keep memory coherent. Invalidation misses only occur in invalidation-based coherence protocols.

It is important to note that this definition does not double-count certain misses. Specifically, accessing data in a cache line that was marked invalid but then replaced *before reuse* is not considered an invalidation miss; it is a capacity or conflict miss. This distinction is missing from the Martonosi definition.

There are always more external invalidation hits than invalidation misses. This follows because cache lines marked invalid by an external invalidation hit may never be reused or the invalid line may be replaced before it is reused. This difference is important because external invalidation hits don't cause memory stalls; only invalidation misses result in memory stalls. However, current processors such as MIPS R10000 count the external hits instead of the misses. Unfortunately, the number of invalidation hits will probably be mistaken by many as a measure of processor performance loss[5] or as an estimate of invalidation misses.

Measuring invalidation misses is easy to do in the processor, but difficult to do externally. A processor simply counts the number of loads in which the tag comparison matches but the line is marked invalid. To make this measurement off-chip, external hardware much capture the cache line state, tag address, and the read address sent to main memory. If the tag and read addresses match and the state was invalid, an invalidation miss occurred.

If invalidation misses are being measured, it is important to perform cache flushes properly. When a line is to be flushed, it is possible for the processor to merely mark the line invalid without changing the tag. However, this will cause a subsequent miss to be improperly counted as an invalidation miss even though no invalidate was received. To fix the problem, either an additional state must be used to indicate the line is completely empty, or the tag address must be cleared when flushing the line.

An alternative method of counting invalidation misses was used in [Singhal94], but it involves running the application twice: once using an invalidate-based protocol and once

---

5. Recall that the overhead of processing the invalidate transaction may cause the processor to stall, but this was already quantified in Section 3.2.4 on page 31 and is not considered here as a memory stall.

using an update-based protocol. Since there are no invalidation misses with an update protocol, the difference in the number of misses is used as an estimate. The obvious problems with this method are that two runs are required and the number of misses can vary depending upon the exact sequencing of the processors.

Performance suffers when many invalidation misses occur. They indicate false sharing of a memory block or that data is actively shared by multiple-readers and multiple-writers (a.k.a. *true sharing*). False sharing can be reduced by placing independent variables in different memory blocks. Some types of true sharing can be made faster if an update-based coherence protocol is used instead; alternate strategies include marking the data as uncached or altering the way the program uses the data.

A caveat arises here when the external invalidation hits measured by a processor are mistaken for invalidation misses. When there are many more hits than misses, the data is likely to be migratory in nature. However, a naive user could interpret the large number of invalidation hits to mean that data is shared by multiple-readers and multiple-writers and switch to an update-based protocol instead. Unfortunately, migratory data is best matched with an invalidate-based protocol because it behaves as single-reader, single-writer data when examined in smaller time quanta. Consequently, an update-based protocol will likely perform worse because of the numerous ownership misses, which shall be described below.

Classifying Misses: Ownership Misses

The second type of multiprocessor miss is unusual because it is only caused by a write; all other misses presented thus far are caused by reads or writes. This particular write miss occurs when the data *is* present in the cache, but the proper ownership has not yet been established at the time of the write. In this case, it is said that the cache contents hit but the ownership misses.

This type of miss appears to have been overlooked in literature despite its common occurrence as a distinct event in parallel programs. The term coherence miss is sometimes used in previous literature, but the definition given or implied is often inconsistent or ambiguous. Other terms, such as upgrade miss or initial-write miss, are more specifically oriented at invalidate-based coherence protocols only. For this reason, it is proposed that an ownership miss be given a separate distinction and defined as follows:

Definition ***Ownership miss*** is a write miss that occurs when data to be overwritten is present in the cache but proper ownership must be established, such as obtaining exclusive access to the block, before the write can proceed. Ownership misses can occur in update- or invalidate-based coherence protocols.

As mentioned in the definition, ownership misses apply to both invalidate- and update-based coherence protocols. They occur in invalidate-based protocols when the data is initially in a shared state. Upon a write, an invalidate is broadcast to all potential sharers to ensure the writer obtains an exclusive copy. In a strongly-ordered memory consistency

model, the write is stalled until an acknowledgement is received to guarantee exclusivity. In comparison, update-based protocols have an ownership miss *on every write* because the data is always assumed to be shared; the home memory, not a processor, is typically considered to be the owner. The write-update transaction is broadcast to all potential sharers to update their copy. Again, strong consistency models demand the write must stall until an acknowledgement is received that all updates were successful. In both of these cases, the latency of obtaining proper ownership permission involves considerable network communication and is comparable to reading a cache line and other such transactions.

If an attempt to obtain proper ownership is not acknowledged, it is deemed to be *cancelled* by a competing transaction. In the invalidation case, an external invalidate removes the processor's copy of the data and it must be re-read. In the update case, the processor accepts an update from another processor then reissues its own update. These compound transactions take longer to complete but are expected to be rare cases. Also, it is unclear whether these compound transactions should be counted separately, counted once for each of the component transactions, or counted by one component transaction. Yet, because of the expected rarity of these events, it shouldn't matter how they are counted.

Like invalidation misses, ownership misses are more easily measured by the processor than off-chip. External hardware must observe the cache line state, tag address, and ownership request address sent to main memory, just as the invalidation-miss hardware does. If the tag and request addresses match, and the state is originally valid and shared, an ownership miss is counted.

In some cases, a simpler way of measuring ownership misses exists. If the protocol is strictly invalidate-based, the number of (successfully) issued invalidates equals the number of ownership misses. Similarly for an update protocol, the number of updates is the same as the number of ownership misses. Additionally, some processors already count ownership misses: MIPS R10000, for example, can count writes to shared secondary cache lines, *i.e.*, ownership misses, as well as writes to clean exclusive lines. This latter measurement is useful because it helps quantify the effectiveness of loading the line in the proper state beforehand, which is one of the many optimizations which will be discussed below.

The performance impact of ownership misses can be reduced in many ways. First, switching to a weaker consistency model will eliminate the need to stall by providing an immediate acknowledgement for most writes. (Interestingly, the total ownership miss latency is a good estimate of the maximum performance advantage of a weak consistency model.) Second, switching to an invalidate-based protocol will reduce some ownership misses, but only at the expense of introducing invalidate misses; this trade-off will be discussed below. Third, the effect of the remaining ownership misses (in invalidate-based protocols) can be reduced by providing hints to the memory system that a line is likely to be modified and should be loaded in exclusive mode. For example, the KSR load instruction contains a hint whether a store is likely and the PowerPC and MIPS R10000 can try to prefetch a line into exclusive state. Performance increases because these hints start the invalidation process sooner than if the processor waited for the write to occur. Fourth, a processor may delay the stall, potentially even avoiding it altogether, by continuing execu-

tion until a second write occurs. At this point, sequential consistency disallows the second write to pass the first and the processor may have to stall to avoid this. For example, this is done by the Pentium processor and it is likely counted by the ‘pipeline stalled by write to exclusive or modified line’ event [Glew95]. By applying these techniques, performance loss from ownership misses can be reduced.

Finally, there is an implied relationship between ownership misses and invalidation misses in an invalidate-based coherence protocol. An ownership miss in one processor generates invalidates which are broadcast to the sharing processors. If these processors re-reference that cache line after the invalidate, they will suffer from an invalidation miss (provided all the previously discussed conditions still hold). If subsequent writes are done by the original processor before the data is shared again, there are no more ownership misses incurred. However, if an update-based strategy was being used the sharing processors would all receive the write update and keep a copy of the data in their cache. Subsequent writes will always incur an ownership miss and contention may increase, but the sharers never suffer from an invalidate miss.

Classifying Misses: Summary

It is hard to distinguish between compulsory, capacity, and conflict misses in hardware, so simulation should be used when these details are needed. Hardware can construct a cache miss profile for each line over time, and this can help identify conflict misses when they are centralized about a particular memory address. Additionally, invalidation misses should be counted in hardware because they directly impact performance and characterize multiprocessor data sharing. External invalidation hits originate from outside the processor and do not directly reflect performance loss, but are useful to help identify migratory data sharing patterns. Finally, ownership misses have not been properly identified in cache literature, yet they can result in performance loss as well. They should also be measured by performance-monitoring hardware.

It is worthwhile to point out that the PhaseID register defined in the previous subsection is a very useful concept that helps pinpoint misses to the suffering code or data structures. Consequently, the classified misses should be counted separately for different PhaseID values. In this way, every phase in the program will have a separate count for invalidation, ownership, and other (compulsory, capacity, and conflict) misses.

3.3.3 Measuring Cache Miss Latency

So far, cache misses have been described in terms of counting how many of what type occurred where. This data is useful, but when contention exists or when memory access time is nonuniform, some cache misses are much more significant because they take a very long time to resolve. In a NUMA system, a cache miss has a highly variable service latency. Clearly, it is equally important to measure miss latencies along with the number of misses and the different types.

The *basic* service latency of a cache miss, defined on an unloaded system, is only a portion of the actual service time experienced in a loaded system. Increased delays come

from memory and network contention and from additional coherence traffic required to maintain consistency.

Miss Latency: Basic Service Latency

Basic service latency can normally be characterized in advance, so performance-monitoring hardware need not measure it. To illustrate this, consider two cases: one, where the basic service latency is constant and two, where the service latency is variable. Of course, when measuring basic service latency the system must be completely idle except for the single transaction in question.

In the absence of contention, most systems have a constant base communication time so there is no need to explicitly measure this component of latency. Instead, it can be determined exactly by analyzing the hardware delays encountered while a miss is in transit. However, some part of the system may contain a certain degree of uncertainty, and even a small amount of this can cause variable delays.

Examples of uncertainty in a system are DRAM busy due to refresh, mismatched clocks or clocks without phase locking, and strict round-robin resource arbitration. If small, the uncertainty can usually be ignored; for example, DRAM refresh is often overlooked. But even when the uncertainty is large, system designers can analytically or empirically derive an estimate of the average basic service latency. This statistical estimate is often sufficient because memory transactions are very frequent events, so the number of transactions is almost always large. Consequently, basic service latency can be characterized in advance, so it is not necessary to measure it with performance monitoring hardware.

Miss Latency: Memory Contention

Memory contention is a source of performance loss that is difficult to observe using software tools. It is typically non-existent in uniprocessors because there are so few masters attempting to access memory; normally it is just the processor and one or two DMA devices. But in shared-memory multiprocessors, every processor is a master which demands access to the memory.

The classic memory contention problem is when a program develops memory ‘hot spots’ and all processors attempt to access the *same physical memory location* at the same time. This can be quite common, especially just after synchronization points. Another contention problem is best illustrated when there is only one memory module or resource. Here, simultaneous accesses to *different memory locations* also result in hotspots because the single resource can only service one transaction at a time. In this case, performance is clearly limited by the service rate of the memory module. These two types of hotspots form the core of memory contention and tend to create long queues.

When the memory is busy, requests must queue up and wait until they are served. Thus, the performance of a memory resource can be measured by classical queueing theory metrics: time-average queue length, maximum queue length, average/maximum/total

length of time in queue, and average/maximum/total service time. These metrics are easy to measure in hardware when the queue is centralized, as in a single FIFO buffer, and the buffer is large enough to avoid overflow. If a hardware buffer does overflow, the data is often refused and the 'problem' is pushed back to the network or the processor. This complicates some memory queue measurements such as maximum queue length, but the total miss latency information is still retained.

For performance tuning, the proportion of cache miss latency that comes from memory contention should be explicitly measured. To do this for a memory resource, it is sufficient to add the number of transactions in the queue into an accumulator on every cycle. The cumulative value represents cycles which were spent waiting for previous transactions to complete.

Of course, it also helps to know what part of a program is causing the memory contention. The two mechanisms described earlier, informing memory operations and PhaseID, are both applicable here, but some enhancements are necessary. First, informing memory operations are extended to be reactive to more than just cache misses. Specifically, the informing operation should test the status of a special *TRIGGER* pin, which can be asserted by off-chip hardware. This allows the program to respond to all cache misses (when it is continuously asserted) or only to those misses that also satisfy some external condition (such as 'memory queue is full'). Second, the presence of PhaseID is extended by tagging it to all memory transactions. Consequently, a memory module always has information about the current phase of the program from each processor. Obviously, these two techniques can be used to trace back memory contention events to particular regions of a program.

To relieve memory contention in a program, both hardware and software techniques can be used. If the hardware can efficiently broadcast or multicast data, it can be used as a more efficient means to distribute data. It works because contention that is caused by multiple requesters pulling the same data is reduced to a single push with a broadcast. Hardware can also provide more memory modules so that simultaneous requests to different memory locations can be serviced in parallel. This parallelism can be better exploited when the software explicitly takes advantage of it by intelligently placing data across the memory resources. Also, contention at memory modules holding shared data can be reduced by converting some data to private or by replicating read-only portions. By effectively using hardware broadcast and multiple memory modules, software can be modified to reduce contention at memory.

Miss Latency: Network Contention

Analogous to memory, *network contention* arises from simultaneous requests for access to the interconnect. The network itself can be considered to be a network of queues and resources which must be acquired and released. For example, a processor that issues a bus request and waits for a bus grant is considered to be waiting in a queue.

The same metrics and methods that are used to measure memory contention also apply to the network. Also, the extensions of triggered informing memory operations and PhaseID are useful for pinpointing losses.

To relieve network contention, traffic which is waiting for a resource must be diverted. If the network supports dynamic routing, it may reroute the traffic so that contention is reduced. However, there are also software-based methods of reducing traffic. First, the techniques used to reduce memory contention may also cause a decrease in network queueing. This is true for two reasons: 1) multicasts merge multiple transactions, and 2) some memory transactions may be spread out in space and use different network resources. Second, data placement plays a critical role in network use. Private data should always be placed as *close* as possible to the processor so it uses the fewest network resources. In some architectures, this may not be directly compatible with reducing memory contention because it may recommend numerous processors contend for the same memory over different network links. Nevertheless, such trade-offs are common in optimization. Although network contention can be reduced, it may be at the expense of some other performance metric.

Since data placement is important, a hardware monitor should provide a way of measuring its effectiveness. This can be done by counting processor accesses to local and remote memory separately. Also, Appendix A describes work in progress that can be used at the memory to measure locality.

Miss Latency: Coherence Traffic

Often, a processor read request cannot be satisfied by memory immediately because the block is not in the proper state. As a result, additional coherence transactions are generated to bring about the desired consistency. This can turn into a complex chain of events as the state and memory block are traced throughout the system. This type of detail is too complex for programmers to unravel, so a less-detailed summary of coherence transactions is preferred.

The chain of events starts when a transaction reaches memory in a state that requires coherence transactions. The contents of this state, called a *memory state indicator* (MSI), should be returned to the processor along with the final response. In this way, a processor can count the number of times it hit memory in an optimal state or in one which may have required much coherence activity.

To show how to reduce the miss latency using the coherence activity information, consider the following example. Suppose an invalidate-based protocol is used and a processor wishes to perform a write, but misses. An attempt is made to fetch the cache line from memory, but memory does not have a valid copy because it is dirty in another processor's cache. After some transactions, the requesting processor receives the data and an MSI that indicates the memory block was invalid but dirty elsewhere. A programmer may notice that the program suffers a significant number of MSIs that all indicate this same state on a write. The cause is probably false sharing because it is unlikely for one processor to write to the same memory block as another processor without a read in-between. If, on the other

hand, the requesting processor had missed frequently because of a read, it is more likely that true sharing or migratory data is present.

From this example, it can be seen that a memory state indicator can give useful information about data sharing patterns. In addition to this, it can also be used to estimate the amount of coherence activity that was required to maintain consistency.

Miss Latency: Other Effects

The miss penalty measurements reflect the performance of the memory subsystem from the viewpoint of a single processor memory request. Although the total miss penalty does impact performance, cycles spent servicing a miss are not necessarily wasted processor cycles. This is because counting cache misses or miss latency doesn't encapsulate other activities that go on in parallel.

A common metric used by memory system designers is miss (or memory) cycles per instruction, or *MCPI*. This is defined as the total latency of misses divided by the number of instructions executed; the goal is to reduce MCPI by reducing the latency of misses. This metric is useful for gauging memory system performance, but it falls short of accounting for program performance because of latency-hiding mechanisms used in the latest generation of processors.

Today's processors employ numerous micro-architectural features to tolerate long memory latencies. Among the many features used, the most common are: non-blocking loads and stores, lockup-free caches, dynamic scheduling, speculative execution, and prefetching. These features are all based upon extracting instruction-level parallelism and overlapping execution of multiple events.

And yet, because of these advancements, the *apparent* miss penalty of a cache miss is reduced. That is, while the latency of a miss may remain the same or become worse (servicing multiple transactions tends to increase waiting in the network and at memory), the program usually *gets faster* because the processor is still busy finding and completing other useful work. The result is an increase in utilization of processor resources and fewer unused cycles.

To a programmer trying to extract the most performance, it is crucial to measure these unused cycles; MCPI is not sufficient. The causes are numerous, including new stalls due to lack of resources (*e.g.*, all outstanding loads are still in-flight, no free reservation stations, or no free renaming registers) and time wasted executing speculative instructions which are eventually discarded. In measuring these unused cycles, it is useful to construct a new metric, apparent MCPI or AMCPI, which is defined as follows:

Definition **AMCPI or Apparent Miss Cycles Per Instruction** is the total wasted processor cycles divided by the number of (useful) instructions executed. The wasted cycles accrue due to waiting on a result from memory or instructions cancelled due to speculative memory operations.

This metric captures the positive effect that latency-hiding mechanisms have on performance. Furthermore, reducing AMCPI is more likely to improve performance, so it is more useful as an optimization target than MCPI.

A problem with AMCPI is that it does not capture all of the subtle effects of the latest processors. For example, suppose an instruction cache miss occurs and fewer instructions are available for dynamic scheduling. Assume that because of the miss, some functional units are left idle. AMCPI counts these cycles as lost opportunity, so it increases. However, suppose the instruction cache miss had not occurred but dependency restrictions still left the same functional units idle. Here, AMCPI is lower than the cache miss case, but performance has not improved. Numerous other subtle effects exist and these make AMCPI a difficult metric to measure.

The end result is that new processors are quite successful at hiding the latency of memory on the given benchmarks. To capture this, a metric similar to AMCPI would be useful. However, the complexity of the latest processors makes it very difficult to measure AMCPI. In addition to AMCPI, the older MCPI is still a useful memory system metric. Programs with limited instruction-level parallelism suffer more because of MCPI, so it cannot be ignored when designing a memory system.

Miss Latency: Summary

The complete latency of a miss can be divided into basic service time, waiting due to memory and network contention, and delays caused by coherence activity. While the basic service time can likely be determined in advance, the memory and network contention latency components can be measured by monitoring the status of their service queues. The delay and causes of additional coherence activity are not convenient to measure explicitly, so a memory state indicator (MSI) should be included with data responses. The MSI provides insight into the processor's view of the type of coherence transactions that may accompany the misses, as well as an indication of the data sharing pattern.

The role of PhaseID has been extended so that it tags memory references throughout the system. Additionally, informing memory operations have been made reactive to a TRIGGER pin. These changes help to pinpoint *when* and *where* a program is inefficient.

Finally, it is noted that the *apparent* latency to the program, AMCPI, can be much less than the actual latency because of advanced micro-architectural features which hide latency by extracting parallelism. Although processors employing these features exhibit excellent benchmark performance, they can still experience a significant slowdown if memory latencies are not kept in check or if limited parallelism exists. Consequently, these new processors should measure both types of memory latencies: MCPI and AMCPI. Together, they provide an optimization target and a measure of the effectiveness of the latency-hiding mechanisms. Unfortunately, AMCPI is difficult to quantify because of the many subtle aspects of processor operation.

3.3.4 Summary: Processor View of Memory Stalls

In this section, memory stalls have been characterized from the perspective of a processor executing a parallel program. Where possible, general hardware schemes have been shown to recognize different types of cache misses (especially those arising from multiprocessor activity), measure latency and performance loss, and correlate misses with program activity. The primary goal of this hardware has been to give fine-grained insight into program memory behavior so that performance bottlenecks can be understood and removed without becoming intrusive or necessitating excessive cost. However, the information also enables software to be reactive to prevailing conditions and adjust its behaviour accordingly.

A summary of the hardware recommendations to measure memory stall activity is as follows:

8. **Provide informing memory operations from [Horowitz95], with external TRIGGER pin support.** If the pin is active during a response, the appropriate miss handler is invoked. This feature enables sophisticated software-collected performance-monitoring and adaptive program behaviour.

9. **Create a user-level PhaseID register.** Changes to the register should be fast and appear to be done in-order. The contents of the register should always accompany all memory transactions throughout the system and are used to separate performance data. This register need not be large; it can be 4 to 16 bits wide.

10. **Profile cache activity** by counting misses to each line separately. This is useful for identifying some cache conflict misses. Periodic sampling of a profile can help show some miss trends.

11. **Support cache-profile watchpoints and/or thresholds which generate interrupts.** These are necessary to trace back misses (observed during profiling) from dynamically-allocated data. By specifying a threshold, the watch is delayed until a real problem occurs. This can help software be reactive to cache misses.

12. **Count invalidation misses.** These are very important multiprocessor-specific misses present in invalidation-based coherence protocols. To implement this, cache line tags must be cleared properly during cache flushes or an additional empty state is required.

13. **Count external invalidation hits.** This metric is not as important as invalidation misses, but it can help identify migratory data.

14. **Count ownership misses.** In update-based coherence protocols, they indicate performance that can be gained through relaxed consistency models. In invalidate-based protocols, they indicate the need for load-with-intent-to-store hints.

15. **Measure queue performance** of memory modules and the network. This is important for understanding components of miss latency, which should be reduced.
16. **Measure locality of memory references.** By separating local and remote misses, the effectiveness of locality enhancements can be measured.
17. **Return the memory block state with every response.** Processor performance monitoring hardware can count the number of hits to each state. This helps convey information about data sharing patterns.
18. **Measure total miss cycles** to establish MCPI.
19. **Measure apparent miss cycles** to establish AMCPI. This can only be measured by the processor because it knows what instructions are issued, retired, stalled or cancelled. It measures the effectiveness of latency-hiding techniques and can be used as a performance optimization target.

3.4 Hardware Support for Software Tools

Although CProf and MemSpy are more powerful than gprof or Mtool, they run too slowly to be usable. In previous sections, it has been shown that similar information about memory system performance can be collected at full speed by using hardware. In this section, additional hardware mechanisms to reduce the intrusiveness of profiling tools like Mtool are introduced.

3.4.1 Timer Support

As recommended by [Goldberg93] and [Hollingsworth93], processors should provide a high-resolution cycle, or *timestamp*, counter for accurate timing. The counter should be readable in one or two cycles and run continuously, no matter if an interrupt occurs or an outstanding memory reference is being satisfied, for example.

In addition, another timer local to each process should also be available. By having the operating system save and restore this *process timer* on a context switch, the process is able to exclude time spent in other processes from its measurements. This simple concept was used in [Shand92] to measure and improve the performance of interrupt handlers.

3.4.2 Basic Block Profiling

A common component of CProf, Mtool and MemSpy is that they all do some form of program profiling. The most useful level of granularity is at the basic block level, because it can be guaranteed that all instructions in a basic block are executed the same number of times. To improve the efficiency of profiling at this level, hardware support is considered for two types of profiling, counting and timing. The hardware is used to reduce the overhead and to improve the timing resolution of these software monitoring tools. Since these applications can reuse previously suggested hardware, they are very cost-effective.

Basic Blocks: Counting Profile

The number of times a basic block is executed is useful for generating dynamic instruction counts of a program, estimating ideal run time and overhead, and dividing up execution time among program components. Mtool relies upon powerful analytic techniques to reduce the intrusion level of basic block counting to 5%, which is quite acceptable for most applications. However, it is possible to virtually eliminate this overhead using the previously-suggested cache profiling hardware to count basic blocks instead.

An analysis of the PERFECT Club benchmarks [Berry89] using pixie reveals that the largest program, SPICE, contains approximately 15,000 basic blocks in 18,000 lines of code. To profile all of these blocks with a 32-bit counter, about 60,000 bytes of storage is required, a reasonable amount that is comparable in size to the cache profile SRAM. By placing a single store instruction in each basic block, a control unit could be signalled to increment an appropriate counter in the SRAM. This store is called a *profiling store*. The target address of the store can indicate which basic block is being profiled, hence which counter to access in the SRAM. Since this is easy to hard-code into the instruction itself as a fixed offset, no additional instructions are needed. However, a processor register is still needed to hold the base address of the control unit.

One advantage of this design is that many basic blocks already contain a NOP in the delay slots of loads, stores or branches into which the additional store can be scheduled. As long as the write buffers don't overflow, there should be minimal impact on runtime. Also, the difficult control-flow analysis used by software is only necessary for extremely large programs that have too many basic blocks for the hardware to manage.

Another advantage of the profiling store is that it essentially accomplishes the same purpose as updating an off-processor PhaseID register. In this sense, the purpose of the profiling store is to signal a change of program state to the hardware, notably that a different basic block is being executed.

Basic Blocks: Timing Profile

Timing basic blocks can be considerably more complex than counting them. Most tools use periodic program counter sampling, but this statistical estimation can be intrusive and error-prone. Instead, block-level timing can be done using the hardware already recommended for cache profiling and basic block counting. Basic block execution times are accumulated in SRAM by placing the timestamp *in the data portion* of the profiling store used for basic block counting. The exact procedure for timing is described below.

Immediately upon entry to a basic block, the current timestamp should be loaded into a general purpose register. Then, the time difference since the previous basic block timestamp can be computed, or it can be left for off-chip hardware to calculate. The result is the time to execute the *previous* basic block. Next, a profiling store can be executed at any point within the block to write the timestamp difference to the performance monitor. Previously, when counting basic blocks, the profiling store contained the hardwired basic block identifier, but this no longer works because the time is for a previous basic block of

unknown origin. In this case, the profiling store must use an address from a processor register which was loaded in the previous basic block. After the store, the monitor SRAM accumulates the execution time for that previous basic block. Finally, the address identifying the current basic block must be loaded into a register for use by the next basic block. Thus, the time spent in each basic block can be accurately measured.

Generic assembler code for this process is given in Figure 3.2, and code for machines with an automatic-clear-on-read timestamp counter or with a hardware monitor that will automatically compute time differences is shown in Figure 3.3. The overhead ranges from three to five instructions per basic block. Furthermore, between two and three processor registers must be dedicated to monitoring functions and one or two registers is needed for temporary computation.

```
% Conventions:
% r1 and r3 are temporary-lifetime registers
% r2 contains the timestamp at the entry of the previous basic block (BB)
% r4 contains the address to be used for the profiling store
% r5 contains the base address of the monitor hardware
% Basic block code may be scheduled anywhere after the first instruction.
mov    r1, timestamp          % first basic block instruction
sub    r3, r2, r1             % store execution time of previous BB in r3
mov    r2, r1                 % save timestamp in r2
sw     r3, 0(r4)              % profiling store
addi   r4, r5, bb_identifier  % create new address for next basic block
```

FIGURE 3.2. Time-based basic block profiling without special hardware support.

The same SRAM which is used to accumulate basic block counts can also be used to accumulate timing information. This has the advantage of saving cost, but it requires two runs of the same program to collect both counting data and timing data because the information is often used together. As Goldberg [Goldberg93] points out, this is unacceptable for programs whose compute time can vary widely with only small changes in runtime conditions, but such programs are ill-behaved and uncommon. Alternatively, basic block counting could be done in software and timing could be remain in hardware.

```
% Conventions:
% r3 is a temporary-lifetime register
% r4 contains the address to be used for the profiling store
% r5 contains the base address of the monitor hardware
% Basic block code may be scheduled anywhere after the first instruction.
mov    r3, timestamp          % first basic block instruction
sw     r3, 0(r4)              % profiling store
addi   r4, r5, bb_identifier  % create new address for next basic block
```

FIGURE 3.3. Time-based basic block profiling with special hardware support.

A disadvantage of this hardware timing solution is the need for two or three registers to be reserved exclusively for profiling. It is hoped that this does not impose a significant increase in register spill code or execution time. Also, the write buffer overflow problem still exists, but possible solutions include not profiling very small basic blocks. Finally, the instruction overhead of three to five instructions is not negligible, but this is no worse than what current basic block profiling code must add, and the benefits of timing information outweigh this small additional cost.

3.4.3 Pointing Losses Back to Code and Data

Software tools that do profiling can benefit from understanding how to use the profiling store address as the PhaseID register. By selecting a different base address (`r5` in the examples), the changing program state is reflected in the otherwise static basic block. In this way, performance data specific to each phase can automatically be separated as a program passes through different phases of execution. This is helpful when the same procedure is used to perform different types of computation, depending upon the value of a parameter passed to it. Such code is very common in object-oriented or structured code where one routine may operate on the same data structure in different ways. Thus, using this variant of PhaseID can help overcome one of gprof's poor assumptions that every procedure invocation takes the same length of time.

3.4.4 Summary: Hardware Support for Software Tools

Software performance-monitoring tools range from mildly intrusive to highly intrusive. Various hardware structures have been shown which can make these tools less intrusive and more accurate. The specific hardware features that have been recommended are:

20. **Timestamp counter** (a.k.a. cycle counter). Key features are: high-precision, low-latency user-level read access. A 64-bit counter should be sufficient. A variant of the counter should automatically reset itself after a read.
21. **Local process counter.** Save and restored with context switches, this permits a process to exclude other system activity from measurements. Also, the difference between the timestamp and local process counter shows how much system activity took place. A variant of the counter should automatically reset itself after a read.
22. **Basic block *counting* SRAM.** The SRAM needn't be large: for example, 64 KB is enough to profile *all* of the basic blocks in SPICE, an 18,000-line program.
23. **Basic block *timing* SRAM.** This may be merged with the counting SRAM, but programs must then be run twice to collect both pieces of information.
24. Implementation note: the off-processor PhaseID register should be used to address the counting and timing SRAM.

Of these features, the first two are the most important and should be included on all future processors. Although all of the latest processors have timestamp counters, they do not all provide user-level access to it.

3.5 Summary

In the software view of performance, speed is compromised when algorithms are inefficient, extra parallel work is present, synchronization and load imbalance cause waiting delays, system and library calls are inefficient or have side effects, and when limited paral-

lelism exists. Measurements of these performance areas are readily done with software-based instrumentation techniques. The measurements provide indicators of *where* a program is slow, but they do not sufficiently explain *why* or *when* performance suffers.

The *why* question is better answered by a hardware view of performance. In this domain, there are many sources of performance loss in a computer system: too many instructions, limited instruction-level parallelism from scheduling, memory and other stalls, exceptions or interrupts, mispredicted branches, and other special cases. Although software can simulate some of these effects, accurate results require very slow and detailed simulations. In contrast, these performance impediments are generally easy to measure with hardware.

Of the different sources of hardware performance loss, memory stalls are the most significant in shared-memory cache-coherent multiprocessors. A thorough view of memory stalls was presented in this chapter, including the identification of two types of multiprocessor data-sharing cache misses: invalidation misses and ownership misses. These misses are easily measured by hardware, and such measurements can be valuable for characterizing data sharing patterns and providing insight to speed up a program. This is contrasted with external invalidation hits, a measurement made by some processors, which has less influence on performance but can also help identify data sharing patterns. In addition to counting multiprocessor misses, a hardware technique for identifying some forms of conflict (and invalidation) misses and tracing these misses back to a program was presented. The technique involves interrupting the processor when a particular cache line incurs too many misses.

The *when* question can be answered by a combination of hardware and software. Informing memory operations allow loads or stores to invoke a miss handler when a cache miss occurs. By adding an external hardware TRIGGER pin to the informing memory operation, the miss handler can be bypassed until an important event occurs, such as too many misses to a certain cache line. Thus, the miss handler is invoked only when needed. An additional scheme permits the software to inform the hardware when an important change of software state has occurred by updating the PhaseID hardware register. This register is used to separate performance data collected for different states, and in this way the performance-costly state can be identified. A PhaseID register can be implemented on- or off-processor. In the former case, load and store serialization problems with respect to PhaseID changes are solved internally and changes to the state can occur more quickly. Unfortunately, the PhaseID contents must be brought off the processor somehow, either on dedicated or multiplexed pins, so that external performance-monitoring SRAM can be used to track the changes; currently, such SRAM is too costly to implement directly on-chip. In the latter case, the PhaseID register can also be used to enable basic block counting and time profiling to quickly collect accurate data for software tools.

Many of the hardware monitoring features described here will be present in the NUMAchine hardware performance monitor, described in the next chapter.

Chapter 4
Hardware Implementation

In the previous chapter, numerous hardware features that are useful for performance monitoring of a multiprocessor were given. As an example of how the features can be implemented, this chapter will describe the hardware monitoring system being built into the NUMAchine processor card. The implementation uses FPGAs and CPLDs to keep prototyping costs low, but the reprogrammable nature of these devices is also advantageous for changing or improving the way data is collected.

Beginning with an introduction of the NUMAchine architecture, the context of the monitored environment is constructed. Following this, a description of the processor card and its monitoring subsystem are given. Then, the programmable configuration modes of the monitor are described. Finally, the implementation is evaluated based on cost and how well it realizes the recommended features. These results are summarized in tables at the end of the chapter.

4.1 Introduction to NUMAchine

NUMAchine is a distributed shared-memory multiprocessor which supports a sequentially-consistent memory model. The smallest NUMAchine unit is a station composed of four 150 MHz MIPS R4400 processors, memory, and I/O on a split-transaction bus. An example station is illustrated in Figure 4.1. Larger systems are built by adding a network cache and ring interface to each station and connecting a number of stations together in a slotted ring network. The network cache is necessary to provide an intermediate node in the coherence protocol, which will be discussed shortly, but it also provides data caching and reduces network traffic by combining remote requests. Still larger systems can be built by connecting multiple rings together with a larger ring and inter-ring switches; such a system is depicted in Figure 4.2. The NUMAchine prototype being constructed is limited to 64 processors, but the architecture is designed to be scalable to a few hundred.

NUMAchine has a two-level cache coherence protocol: at the *network level*, between network caches and memory, and the *station level*, between processor caches and the network cache. Both levels employ a write-back, invalidate-based strategy, but flexibility in the protocol and hardware has been reserved for simultaneously supporting an update-based scheme. This flexibility allows a program to choose the best strategy for its different types of data. Additionally, the R4400 processors force primary caches to maintain a strict subset of secondary cache contents, a property called *inclusion*, and follow a strict allocate-on-write policy while handling a write miss. In contrast, the network cache does not guarantee this inclusion (with respect to secondary caches) and it does not allocate a line when writing back a block if it misses. The two-level coherence scheme used by NUMAchine is structured to match the hierarchy of the interconnect, so good performance is expected.

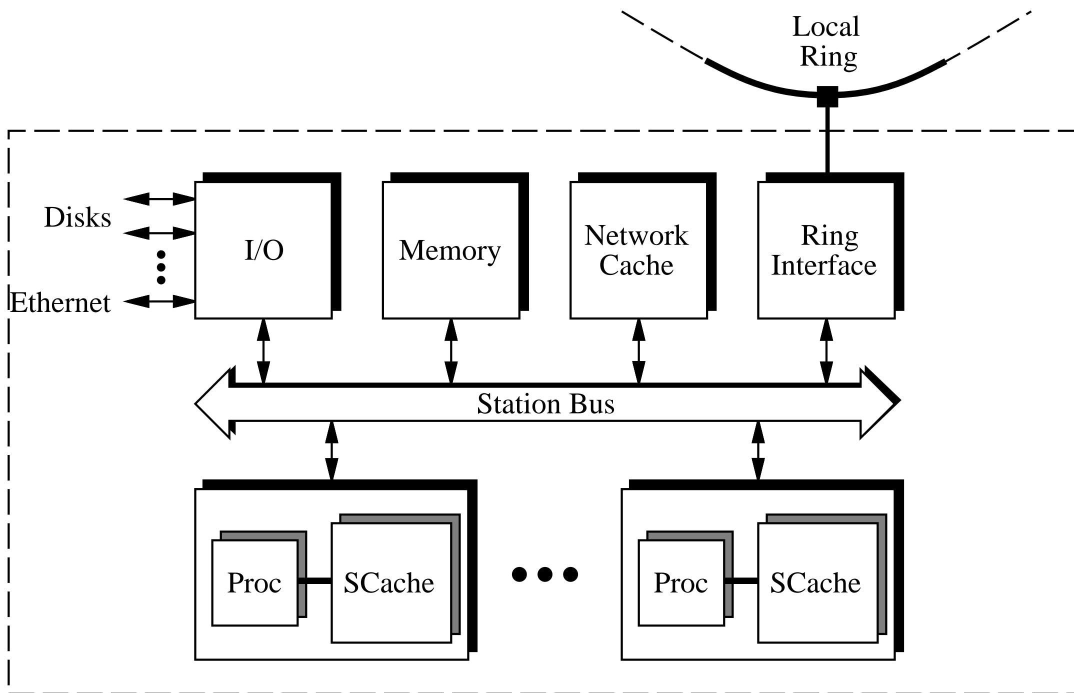

FIGURE 4.1. The NUMAchine station.

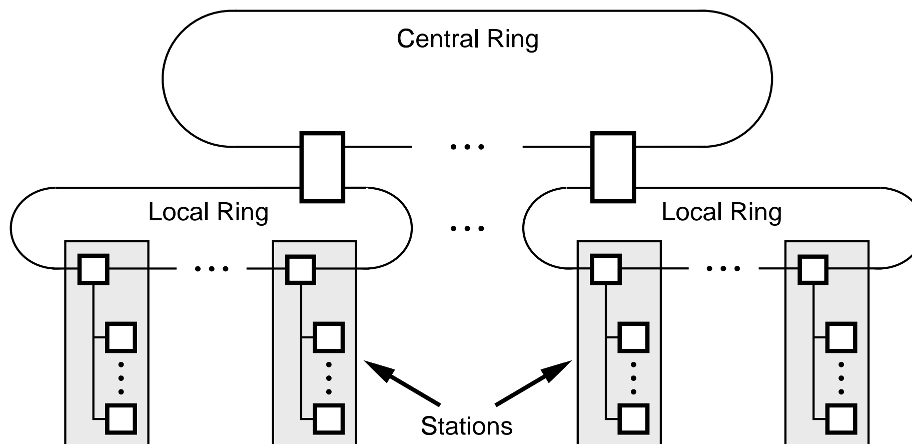

FIGURE 4.2. The NUMAchine hierarchy.

At the station level, processors keep cache lines in a variant of the well-known MESI (modified, exclusive, shared, or invalid) states. NUMAchine drops the use of the exclusive state so that memory is properly informed of a transition to the dirty (modified) state; normally, a processor simply makes the transition silently. If a processor wishes to obtain data in exclusive state, it is likely to modify the data. For this reason, NUMAchine places exclusively-read data into the dirty state immediately.

At the network level, the network caches and memory maintain state information about a memory block. Specifically, four primary states are defined: *local valid* (LV), *local invalid* (LI), *global valid* (GV), and *global invalid* (GI). These states indicate whether a local copy of the data exists on the station or if it exists remotely, and whether the current copy in the network cache or memory is outdated because a (local or remote) processor has a dirty copy.

Beyond the states already mentioned, the network and processor caches both have a logical *not in* (N) state to indicate that the memory block is not present. This is different from an invalid state because the latter implies a tag match. For further information about the states and requests causing transitions, the reader is referred to Figure 4.3. As well, a more detailed discussion about cache coherence policies can be found in [Vranesic95].

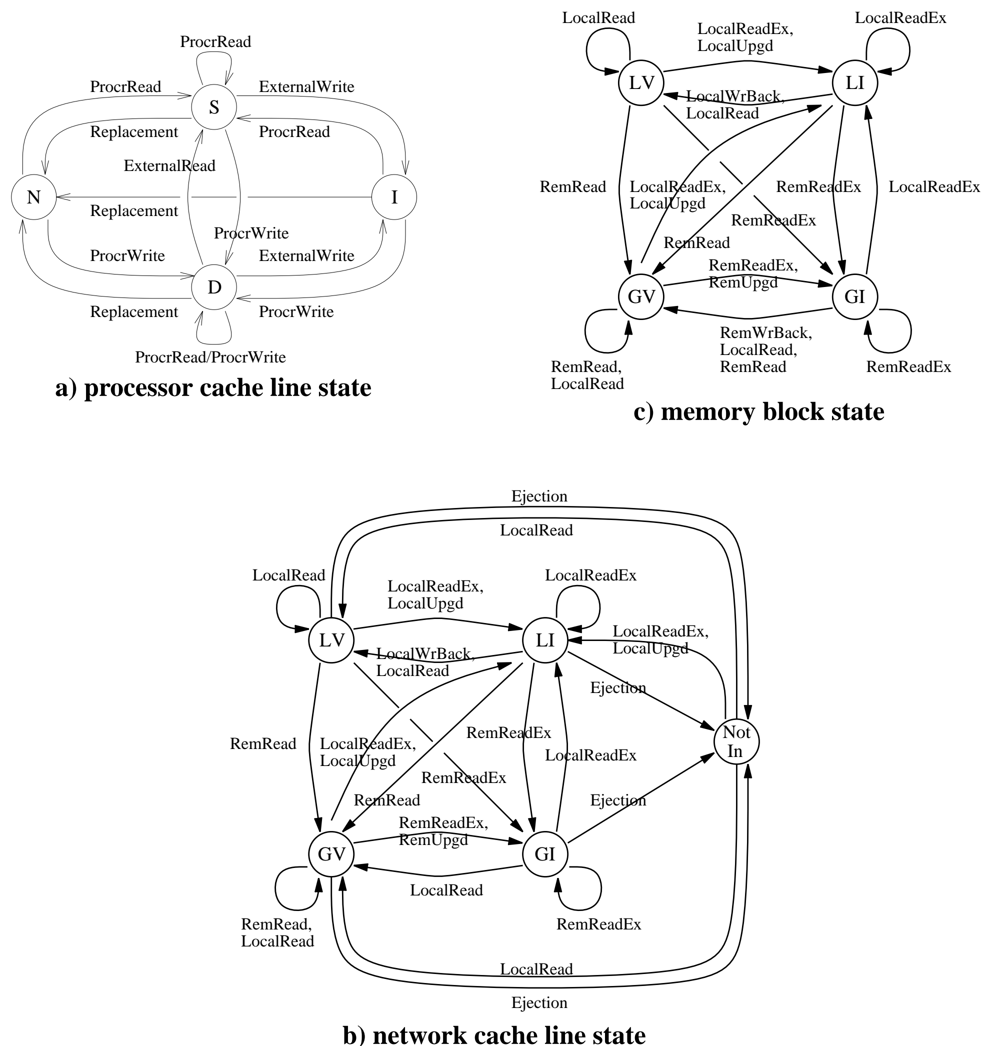

FIGURE 4.3. Memory consistency state transition diagrams.

4.2 Processor Card Organization

Each NUMAchine station contains up to four processor cards on the NUMAchine bus. The processor card datapath is organized according to Figure 4.4. The *Bus Interface* provides access to the station bus where the main memory, network cache, ring interface, primary I/O (disks, ethernet), and other processors reside. Data is passed into and out of the processor card through *FIFOs* to smooth flow control and decouple the processor card and bus clocks so they can run at independent speeds, if necessary. The *External Agent* performs two functions: 1) it acts as a bridge between the R4400 and the NUMAchine bus, and 2) it controls data flow between the R4400, Monitor and FIFOs. The *Monitor* is situated to observe traffic on the external agent bus as well as accept uncached reads and writes from the processor or uncached writes from the system. The latter allows for simpler initialization, synchronization, and reconfiguration of the monitor by using broadcast writes throughout the system. For convenience, the monitoring interface serves an additional role of providing *Local I/O* for the processor bootstrap code and a UART for debugging. Finally, the processor has a dedicated interface to the *Secondary Cache*, which can be split in half between instructions and data or provide a single unified cache. Typical operation will be done with a unified cache, but the split organization can be used in conjunction with some types of monitoring in which accesses to data and instructions should be isolated; this feature is recommended in [Singhal94].

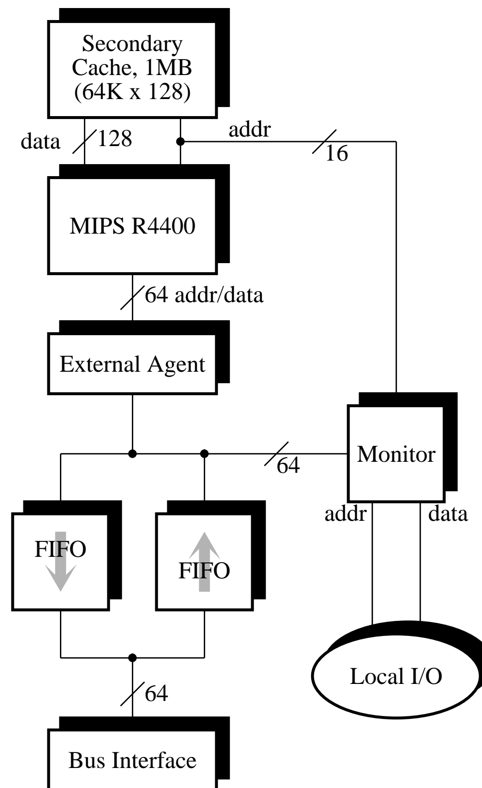

FIGURE 4.4. NUMAchine processor card datapath organization.

The NUMAchine bus and all of the logic on the processor card are designed to run at 50 MHz. Additionally, a small portion of the monitoring circuits connected directly to the processor must run from a 75 MHz clock which is locked to one-half the processor speed. Such aggressive system speeds are difficult to achieve in the implementation technology (FPGAs) in a cost-effective manner and, consequently, have influenced many of the design choices.

4.3 Monitor Organization

The processor card performance monitor is positioned so that it can monitor traffic to and from the processor in a non-intrusive fashion. To do this, the monitor is situated on the external agent bus, between the FIFOs and External Agent. When a program chooses to consult with the monitor to read performance data, it uses normal load and store instructions to an uncached, but TLB-mapped, region of memory. By mapping the region, the operating system can protect the monitor from unprivileged user processes. For convenience, both word (32-bit) and doubleword (64-bit) accesses are supported. As previously mentioned, the External Agent also allows writes to the monitor from other processors.

The internal organization of the monitor is illustrated in Figure 4.5, but control signals are omitted from the figure for clarity. As shown, the external agent bus is split into the monitor bus and local I/O bus by the *Local Bus Controller*. From the monitor bus, transactions and configuration information travel to the *Configuration Controller*. Additionally, the monitor bus allows data in the *SRAM* or *Counters & Interrupts* circuits to be read or written. The *Latency Timer*, *Count & Increment*, *Pipeline Status*, SRAM and Counters & Interrupts circuits all operate with the cooperation of the Configuration Controller to provide the monitoring functions. The monitor bus, however, is under the control of the Local Bus Controller.

4.3.1 Local Bus Controller

To allow nonintrusive monitoring of processor activity, the Local Bus Controller (LBC) passively observes external agent bus transactions. It also gives processors direct access to the monitor bus and local I/O bus. The former is used so that a program can quickly obtain performance feedback, while the latter is incidental to NUMAchine test and development.

During the normal LBC operation, transactions on the external agent bus are captured in latches and placed onto the monitor bus. Conveniently, the monitor exploits two facts: 1) only the 64-bit address is needed from the transaction for monitoring, and 2) the external agent cannot process more than one transaction every two cycles. These facts are used to reduce the width of the monitor bus to only 32 bits by holding the upper-half address in a register and sending it on the monitor bus after the lower half is sent. This does not present a problem for doubleword writes to the monitor because the upper address bits can be safely discarded and overwritten as the upper data word is written during the second clock cycle. The upper address bits for doubleword writes to the monitor are not important because it only contains network transaction information, as shown in Table 4.1.

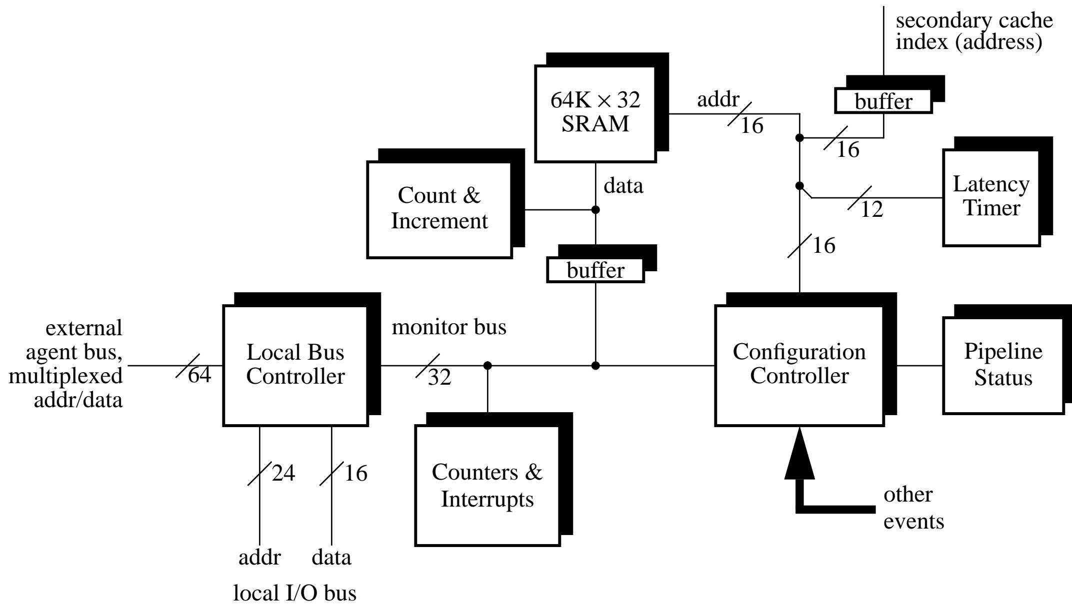

FIGURE 4.5. Processor card performance-monitoring subsystem.

Address Bits	Name	Description
63..56	SRC Routing Mask	identifies source processor station of remote transactions
55..52	PID	processor identifier: which processor in station ‘SRC’ issued transaction
51..48	PhaseID	indicates phase of remote or local processor, but monitor has own copy
47..40	DST Routing Mask	identifies all destinations that were to receive this transaction
39..36	Station Address	indicates packed-encoding of DST when there is only one destination
35..32	Magic Bits	indicates special functions; the monitor has no special functions
31..28	Reserved	future physical address extension
27..0	Physical Address	normal memory address

TABLE 4.1. Breakdown of 64-bit address space in NUMAchine.

The LBC’s second function is to allow reading and writing of the performance data and configuration of the monitor. While the processor or system is reading or writing the monitor, the External Agent is held busy to ensure that no traffic to be monitored will be lost. This should not be considered intrusive because accessing the monitor is inherently an explicit, intrusive decision on the part of the program. Additionally, writes may be used to reconfigure the monitor in two ways: either 1) a *hard configuration*, or 2) a *soft configuration*. The former involves changing the FPGA circuitry in the Configuration Controller (CC) and Counters & Interrupts circuits and the latter simply writes a new value into the configuration registers implemented in the CC.

A third function of the Local Bus Controller is to give the R4400 access to boot ROM and a debugging UART via a 68000-style local I/O bus. During testing, a Motorola 68000-based computer is situated on this local I/O bus to provide scratch memory and Ethernet to the processor card. The LBC is designed so that a microcontroller situated on this bus can also configure the monitor or access the performance data. Beyond its usefulness for testing, this interface can be used by a remote workstation to continuously and dynamically instrument, analyze, and display performance data without intruding upon a running program.

4.3.2 Pipeline Status

In Section 3.2, the importance of processor efficiency was described. Although most recent processors already have two or more performance counters for this task, the MIPS R4400 does not. Instead, it outputs pipeline status bits so that external hardware can measure this performance. The Pipeline Status (PS) circuit is used to help count the MIPS R4400 pipeline states listed in Table 2.3.

The 150 MHz processor produces two sets of status signals once every 75 MHz cycle to indicate the pipeline events in the last two processor cycles. Consequently, the PS circuit must operate at 75 MHz. Ideally, the PS would contain 15 counters to measure each possible event separately, but this is not economical because FPGAs combining high-speed and high-density are very costly. Instead, the PS circuit is used to decode and select the events of interest so they may be counted by four slower (50 MHz) general-purpose counters in the Counters & Interrupts circuit.

To account for the speed difference between the pipeline status changes and the counters some special preprocessing is needed. The PS produces two bits every cycle to indicate whether zero, one, or two events appeared in the last two processor cycles. These two bits are then added into a 2-bit accumulator that is hidden from the user. Every time the 2-bit accumulator overflows, it triggers circuitry that synchronizes the overflow to a 50 MHz clock and then informs the counter of the event. As a result of this action, the general-purpose counters are only incremented after every fourth event. This loss of accuracy is minimal and should be acceptable for almost all applications.

To specify which pipeline events to count, it would be most convenient to use a 15-bit control word to enable individual events from Table 2.3. However, this is not economical because the resulting circuit is large and would require a fast, expensive FPGA. On the other hand, using a 4-bit control word to count only one event would require four runs of a program to count all 15 events in the four counters. This is not reasonable in the cases where full detail is not required. To permit greater flexibility while counting, but still keep the circuit small, a 7-bit control word is used instead. The resulting circuit is more flexible, yet it is small enough to fit in two inexpensive Altera MAX7064 CPLDs which easily meet the timing requirement. These seven bits, named PS[6..0], are realized in the Configuration Controller as a portion of the control word, *GPCMx*, for each general-purpose counter. The function of these bits will be described below.

The PS[6..0] bits are divided into two mode bits in the upper portion, and five selection bits in the lower portion. The two most-significant bits indicate one of the three counting modes listed in Table 4.2. The single event mode monitors one specific pipeline state and the other two modes allow certain states to be counted together, or *merged*, in one counter.

PS[6]	PS[5]	Monitoring Mode
0	0 or 1	counts *single event* specified by PS[3..0]
1	0	counts *running cycles*, multiple events are selected by setting the appropriate bits in PS[2..0]
1	1	counts *idle cycles*, multiple events are selected by setting the appropriate bits in PS[4..0]

TABLE 4.2. Pipeline Status bits PS[6..5] specify one of three operating modes.

The lower five bits select which specific event or events are of interest according to Table 4.3. In the single event mode, PS[3..0] specifies one pipeline state using the same encoding shown in Table 2.3. The other two modes use the lower bits to merge multiple pipeline states together by setting one or more bits in PS[4..0]. The specific events merged are listed in Table 4.3.

	Single Events	Running Cycles	Idle Cycles
PS[4]	reserved	reserved	integer + floating-point pipeline slips
PS[3]	see Table 2.3	reserved	instructions killed due to exception + branches
PS[2]	see Table 2.3	other integer + other floating-point instructions	multiprocessing + other stalls
PS[1]	see Table 2.3	taken + not-taken branch instructions	secondary cache stalls
PS[0]	see Table 2.3	load + store instructions	primary instruction + data cache stalls

TABLE 4.3. Pipeline Status bits PS[4..0] select which events to monitor.

It should be noted that the Pipeline Status circuit uses a fairly complex method to keep FPGA costs low but still maintain reasonable flexibility. However, since full-custom VLSI can easily implement fast, area-efficient counters [Vuillemin91], it would be better to include a large number of counters directly on a processor.

4.3.3 Counters & Interrupts

The Counters & Interrupts (C&Int) circuit is constructed to hold up to four general-purpose 32-bit counters and, for convenience, a barrier register and two interrupt registers. The interrupt registers are necessary for NUMAchine interrupt processing and the barrier register is an experimental hardware synchronization mechanism used to accelerate performance, but neither are essential for monitor operation. The counters, however, are essential for monitoring high-speed or overlapping events such as the pipeline states. The selection of which event to count is governed by the Configuration Controller.

The counters have been designed to produce a maskable interrupt on overflow and to automatically reset if read from a specific address. By preloading a negative number, a program can wait for a precise number of events. This allows software to be reactive to an excessive number of cache misses, for example. Also, overflow interrupts allow system software to create the illusion of a larger counter, if desired[1]. Additionally, the automatic reset-on-read gives software a useful atomic fetch-and-clear option. For more information about the implementation of the counters, see [Zilic95].

The C&Int components are interconnected so that they may be read or written from the monitor bus, as depicted in Figure 4.6. The read path is obvious but the write path is unusual so it is highlighted by the shaded arrow. This strange path is an example of how the FPGA architecture has influenced the design: rather than using separate read and write paths, they are merged so that fewer FPGA logic blocks are required. In the merged path, the *data hold* multiplexer serves two purposes. First, during a counter read it captures the count in one cycle and holds it for as long as is necessary for the weak FPGA pins to drive the monitor bus. Second, during a counter write it holds the new counter value for multiple cycles while the counter's complex carry chain stabilizes. Without this organization, a larger and more expensive FPGA would be necessary.

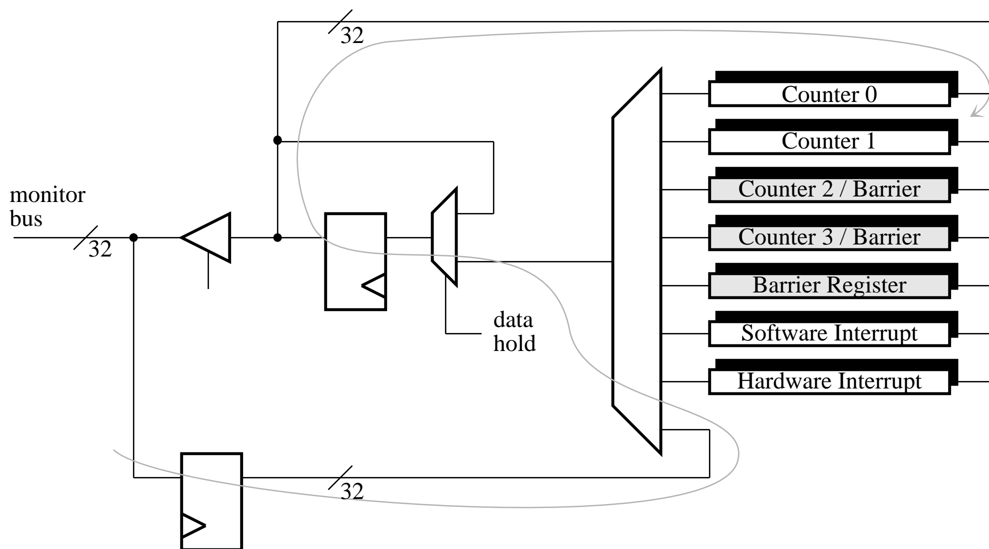

FIGURE 4.6. Counters & Interrupts datapath, with write path highlighted.

Economics also play a role in the number of counters and barrier registers. The design in the figure does not currently fit into the target FPGA, an Altera FLEX8636, because of routing constraints. Consequently, a simple design which does not include the shaded registers, hence is easier to route, is currently implemented. Designs with an additional counter or barrier register have also been realized, and these may be programmed into the FPGA via a hard reconfiguration of the C&Int device. Also, future experimentation may

1. The 32-bit width is sufficient to limit the counter overflow interval to approximately 1.4 minutes. This is considered infrequent enough to be nonintrusive.

eventually realize a design with all the features because there is considerable flexibility in the counter design to trade-off logic cell use for routability.

Because of these routing constraints, it was decided to fix part of the C&Int architecture (the unshaded registers) and allow the implementation of the shaded portions to float. This allows a program to reconfigure the C&Int device with more barrier registers or more counters, depending upon its requirements and the latest developments in fitting the circuit in the FPGA.

4.3.4 SRAM Memory

The processor card includes fast SRAM that can be used for a variety of functions. The primary use of the SRAM is to store the state for a large number of infrequently-used counters. For this use, the depth of the SRAM (64k) is chosen so that it can be used to profile secondary cache accesses as suggested in Section 3.3; this is the purpose of the 'secondary cache index' buffer in Figure 4.5. Like the general-purpose counters, the SRAM is wide enough (32 bits) to limit overflow frequency.

The exact function of the SRAM is governed by the Count & Increment, Latency Timer, and Configuration Controller devices which will each be described below. However, the SRAM can also be used as scratch memory by the processor for any purpose. One such use is for storing the FPGA configuration data before it reprograms them.

4.3.5 Count & Increment

The Count & Increment (C&Inc) circuit operates on data from the SRAM. Its main operation is to fetch a word from SRAM, add one to it, and write it back. Each operation requires one clock cycle. Alternatively, the write back cycle may be extended for multiple cycles. During this time, the counter is incremented on each cycle if a count-enable signal is asserted, and the latest count is continuously written back to the SRAM. An example of this use is to separately count stalled cycles for each basic block. The final use of C&Inc is as an accumulator for timing basic blocks. In this mode, the C&Inc is initialized by a word write from the processor and then the contents of the SRAM are added to it before being written back. However, this mode is not yet implemented in the current design.

The C&Inc circuit is simple enough to be implemented in the smallest Altera CPLD, a MAX7032.

4.3.6 Latency Timer

Although cache misses are important to count, Section 3.3 motivated the significance of memory latency. To time the memory response to a processor read, the Latency Timer (LT) is used. This timer is reset as the first part of the transaction is written from the external agent into the outgoing FIFO and it increments every cycle until the response or a BUS_ERROR is returned. The BUS_ERROR occurs after a time-out of 2^{12} (4096) cycles; this determines the size of the LT counter. The timer value is used to produce a histogram

of memory access times in the SRAM. Additionally, it will be shown later that the histograms can be separated by a 4-bit PhaseID, for example.

In addition to the normal 12-bit time representation, the LT can compact the time value into a 7-bit floating-point representation. The seven bits are divided into a 3-bit exponent (high bits) and 4-bit mantissa (lower bits). The significand contains an implied leading '1' unless the exponent is zero, in which case a denormal representation is used. The exact time implied by this is best explained using the following pseudocode:

```
if( exponent == 0 ) time = 0.mantissa * 2^5;
else if( exponent > 0 ) time = 1.mantissa * 2^5 * 2^(exponent-1);
```

The advantage of this format is it uses fewer bits to represent the latency by grouping longer latency measurements together into larger histogram buckets. The idea is not new; it was also used by SUPERMON to compact addresses. The bits saved will be used to separate the histograms more; for example, PhaseID can be extended to 9 bits.

Implementing the compaction requires a barrel shifter, an exponent generator, and a state bit for denormal support. Despite the seemingly complex behaviour, this circuit can also be implemented in the smallest Altera CPLD, a MAX7032.

4.3.7 Configuration Controller

The most complex portion of the monitor is the Configuration Controller (CC). It is the command centre of the monitor that controls what is to be monitored and, in some cases, determines whether an event being monitored has just occurred. It also controls whether an interrupt should occur when a counter overflows. Due to the detail and complexity involved, the next subsection will describe the CC in greater detail.

4.4 Programmable Configuration

The Configuration Controller (CC) is implemented in a reprogrammable FPGA so that its function may be changed to collect new data or change the conditions of collection. However, for typical uses a Master CC (MCC) circuit was designed to provide most of the flexibility a user will require. This is done by *soft configuration* of the circuit with simple writes to the user configuration registers listed in Table 4.4. The operation of the registers will be explained below.

4.4.1 PhaseID Register and PhaseID Watch

The PhaseID Register (PR) is a 16-bit implementation of the PhaseID recommended in Chapter 3 to easily partition the collected performance data. By writing a new PR value[2],

2. In Chapter 3, two other methods of changing PhaseID were suggested which involved encoding the new value into the address during a read or a write. These alternative methods have not yet been implemented, but they are easy to add.

Address	Acronym	Name	Bits	Notes
0	PR	PhaseID Register	15..0	PhaseID Register
1	PW	PhaseID Watch	15..0	compared against PR
2	CW	Command Watch	12..0	compared against filtered command
3	CF	Command Filter	12..0	removes unwanted command bits
4	AWhi	Address Watch High	7..0	forms AW bits 39..32
5	AWlo	Address Watch Low	31..0	forms AW bits 31..0
6	AFhi	Address Filter High	7..0	forms AF bits 39..32
7	AFlo	Address Filter Low	31..0	forms AF bits 31..0
8	GPCM0	General-Purpose Counter 0 Mode	23	counter enable
			22..16	pipeline status configuration bits PS[6..0]
			15	enable interrupt on overflow
			14..11	event select (count one of 16 events)
			10	count cycles high or low-to-high transitions
			9..8	enable masks below .. invert mask sense
			7..6	enable PW compare .. invert compare sense
			5..4	enable AW compare .. invert compare sense
			3..2	enable CW compare .. invert compare sense
			1	enable sending mask, SM
			0	enable receiving mask, RM
9, A, B	GPCM1, GPCM2, GPCM3	General-Purpose Counter 1, 2, 3 Mode	23..0	same as GPCM0
C	SCM	SRAM Counter Mode	22	enable interrupt on overflow
			21..20	counter mode
			19	muxMISS — uses miss type when set
			18	muxRSR — uses RSR when set
			17	muxPRhi — uses upper 5 bits of PR when set
			16	muxPRmid — uses middle 7 bits of PR when set
			15	muxPRlo — uses lower 4 bits of PR when set
			14..11	event select (count one of 16 events)
			10..0	same as GPCM0 bits 10..0
D	reserved	reserved	n/a	reserved
E	GPCE	Master General-Purpose Counter Enable	0	master enable for all four general-purpose counters
F	SRAMCE	Master SRAM Counter Enable	0	master enable for SRAM counters

TABLE 4.4. Master Configuration Controller user configuration registers.

the address to the SRAM is changed and a different counter is selected. As will be shown later, some SRAM counting modes may use only a portion of PhaseID. In these cases, a new PR value will select a different bank of counters.

The lower 4 bits of the PR have an additional purpose. They are brought outside of the processor card monitor and attached to some bits in the outgoing FIFO so that they are attached to all memory transactions initiated by the processor. Performance monitoring hardware in the memory card and network can use these 4 bits to demarcate transactions originating from different phases of a program.

The PR can also selectively enable the SRAM or C&Int counters. A constant, called PhaseID Watch (PW), is compared against the PR. The result of this comparison drives a counter-enable circuit which will be described later. The primary use of this feature is to enable a C&Int counter during one specific phase.

4.4.2 Command Watch and Command Filter

The Command Watch (CW) and Command Filter (CF) registers are used to restrict counting to only certain types of transactions. Every NUMAchine transaction is composed of multiple network *packets*, each containing a 13-bit Command identifier to indicate the transaction type. For example, it can identify whether the packet is a cache line read, a request or response, or whether it contains an address or data[3].

The CF and CW registers are used in conjunction with the Command Register (CR). The CR is automatically updated with every packet's Command as it is passed to and from the External Agent. The Command Filter (CF) register is used as a bitmask to remove uninteresting bits, and the result is compared against the Command Watch (CW) register[4]. Again, the comparison result is used to drive a counter-enable circuit.

By carefully selecting proper CW and CF values, it is possible to specify multiple events to be monitored at once. For example, cache line read and read-exclusive responses can be counted together.

4.4.3 Address Watch and Address Filter

Similar to CW and CF, the Address Watch (AW) and Address Filter (AF) registers can be used to watch accesses to a region of memory. Because of the 32-bit width of the monitor bus, the upper and lower portions of the AW and AF registers must be written separately. This can be done by software with two word writes or with a single doubleword write in big-endian data-word order.

---

3. Although packets forming a transaction are always placed contiguously on a NUMAchine bus, they may be separated and interleaved with other transaction packets on a ring.

4. Software must ensure that a new CW value is compatible with CF by masking CW first, or the comparison may never match.

The AW and AF registers are used in the following manner. First, an Address Register (AR) is automatically updated with the most recent physical address passed to or from the External Agent. Then, AF is applied to AR as a bitmask and the result is compared to AW. With this setup, a contiguous region of memory which is aligned and sized to a power-of-2 can be monitored. However, for this to work properly the operating system must allow a program to allocate a contiguous portion of memory. Additionally, the operating system should lock these memory pages down so they don't migrate and aren't demand-paged to disk.

4.4.4 General-Purpose Counter Modes

The general-purpose counters in the Counters & Interrupts circuit can count from a number of events and have a number of different operating modes. The operation of these counters is governed by the four GPCMx registers; each register is identical except that it governs a different counter. The role of the various bits in these registers is described below.

The lower 10 GPCMx bits are invert and enable bits for the count-enable circuit shown in Figure 4.7. To limit the circuit size but maintain some flexibility, the counters share comparators that feed individual count-enable circuits. One way this can be used, for example, is to enable one counter on address matches while another is enabled for all single-word reads. Another use is to AND together multiple comparators by enabling them simultaneously to form a compound condition for enabling a counter. Alternatively, the invert bits can be used to reverse the sense of the comparison (*i.e.*, not-equals) or to merge multiple comparisons in an OR fashion. Finally, the count-enable circuits can also be enabled while transactions are sent or received (or both) by the external agent.

Bits 11 through 14 control a multiplexer that selects which event count according to the list in Table 4.5. When bit 10 is clear, the counters will count time by incrementing for every cycle the event is asserted. However, it makes sense to set bit 10 on events 3, 4, 5, 6, 7, 8, A, D and F so that only low-to-high transitions of the event, *i.e. event occurrences*, are counted instead of latency; these event occurrences are indicated with {braces} in the table. Additionally, when the counter is configured for pipeline status events, the PS configuration bits, GPCMx[22..16], are used to specify the proper pipeline event. The function of these bits was already discussed in Section 4.3.2. The counters have a master enable, bit 23, and can be configured to trigger an interrupt when they overflow by setting bit 15.

General-Purpose Counter Event Details

A number of events in Table 4.5 require some additional explanation. First, the latency or number of *successful* ownership misses are measured with event 7; an unsuccessful miss is caused by a time-out, so a bus error exception is taken and may be counted by software. This must be contrasted to cancelled ownership misses, which occur because a competing processor 'won' the ownership first, that are counted by event 8. The total number of ownership misses is the sum of the successful, unsuccessful, and cancelled ownership misses.

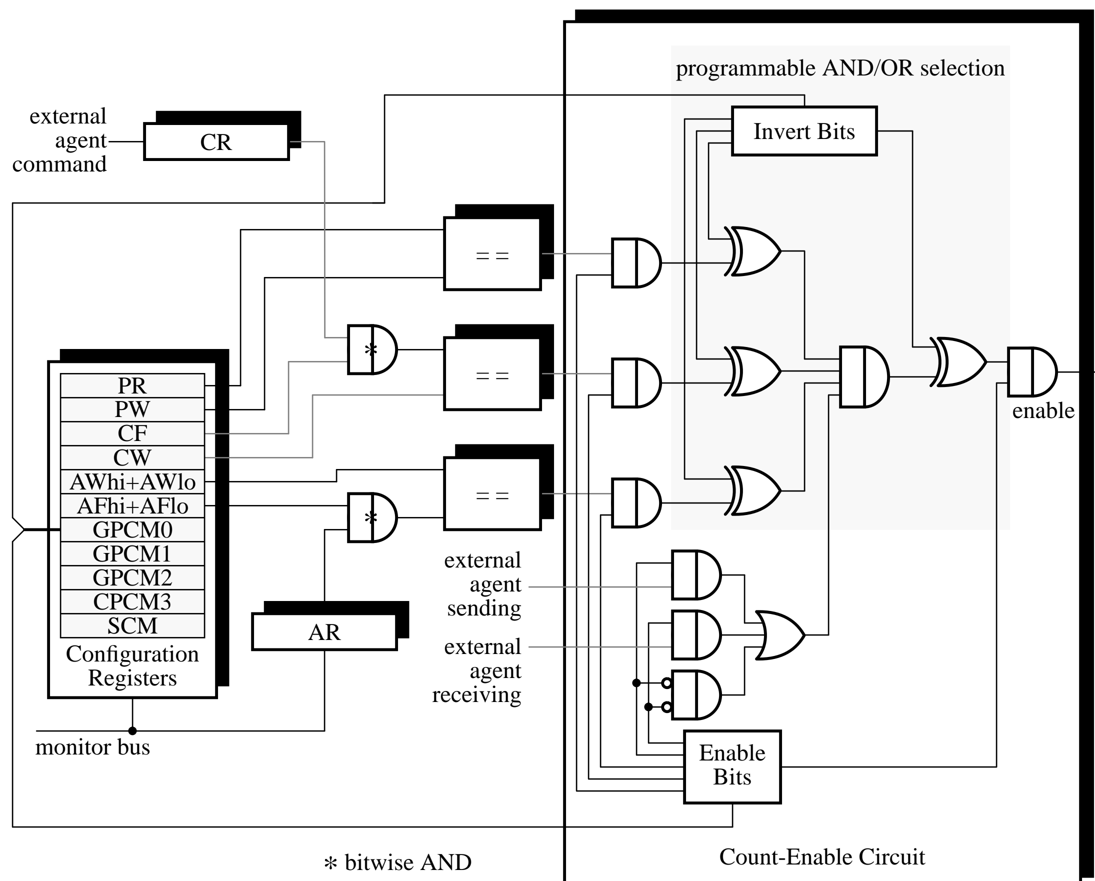

FIGURE 4.7. MCC registers with one Count-Enable Circuit.

Second, some transactions such as a read request may be negatively acknowledged (NACKed) and returned by a device when it cannot service the transaction, so the External Agent must retry these requests. This may happen if memory temporarily has a block locked, for example, and it may be NACKed several times before a response arrives; this is measured by event 9. Also, the total cycles spent from the first retry to the final response is represented in event A; counting the low-to-high transition of this event counts the number of requests that received one or more NACKs.

Third, the total latency of the most recent transaction is measured by event B. Here, the counter is reset every time a request is issued and stopped whenever a response is returned. Fourth, every cycle the external agent bus is used will be counted with event C; measuring transitions of this corresponds to counting the total number of transactions. Fifth, event D measures the response time of the bus, which may be slow due to contention. Sixth, invalidations that originate from outside the processor and hit in the secondary cache are measured with event E and the number of invalidation misses are reflected by event F.

To measure external invalidation hits, the state machine in Figure 4.8a is used. The invalidate starts a cache-watching state which waits for a secondary cache probe (from the processor) that maps to the same address as the invalidate; a successful mapping is called

GPCMx[14..11]	Event
0	count nothing (disable)
1	count always (high-resolution cycle counter)
2	pipeline status events
3	interrupt request latency {interrupt requests}
4	cache line read latency {cache line reads}
5	cache line read-exclusive latency {cache line read-exclusives}
6	cache line read latency + read-exclusive latency {cache line reads + cache read-exclusives}
7	successful ownership miss latency (upgrades + updates) {successful ownership misses}
8	cancelled ownership miss latency (upgrades + updates) {cancelled ownership misses}
9	negative acknowledgement retries (may be >1 per request)
A	negative acknowledgement cycles {transactions with one or more negative acknowledgements}
B	latency of most recent (cached or uncached) request, eg: read, read exclusive, upgrade
C	external agent bus activity (*i.e.*, utilization)
D	bus request latency (until a bus grant is given) {bus requests}
E	external invalidations that hit in the cache (estimate)
F	invalidation miss latency (estimate) {invalidation misses}

TABLE 4.5. General-purpose counter events.

an *index match*. If the state of this line is valid and subsequently changed to invalid before a different cache line is accessed, the state machine passes through the shaded state in the figure and the event is counted. Otherwise, the state machine returns to the idle state. This algorithm is only approximate because the MCC does not do the same tag comparison the processor does; there are not enough pins left on the FPGA to monitor the tag SRAM. Despite this, it should still be accurate. The state machine will only over-count if the processor intentionally invalidates a block which maps to the same line within an approximately 34-cycle time window (the external invalidate is guaranteed service within this time by the processor design). This is unlikely to occur because the processor seldom intentionally invalidates a line (only explicit cache flushes will do this). Also, it only under-counts if the processor has 2 consecutive primary cache misses within this window and the first maps to the same line as the invalidate. This, too, is unlikely because of high primary cache hit rates and the improbable index match. Finally, ownership misses which are cancelled will be included in the external hit count, but they can also be measured by a

performance counter and subtracted out, if desired. Thus, the approximation should be sufficient for most uses of the data.

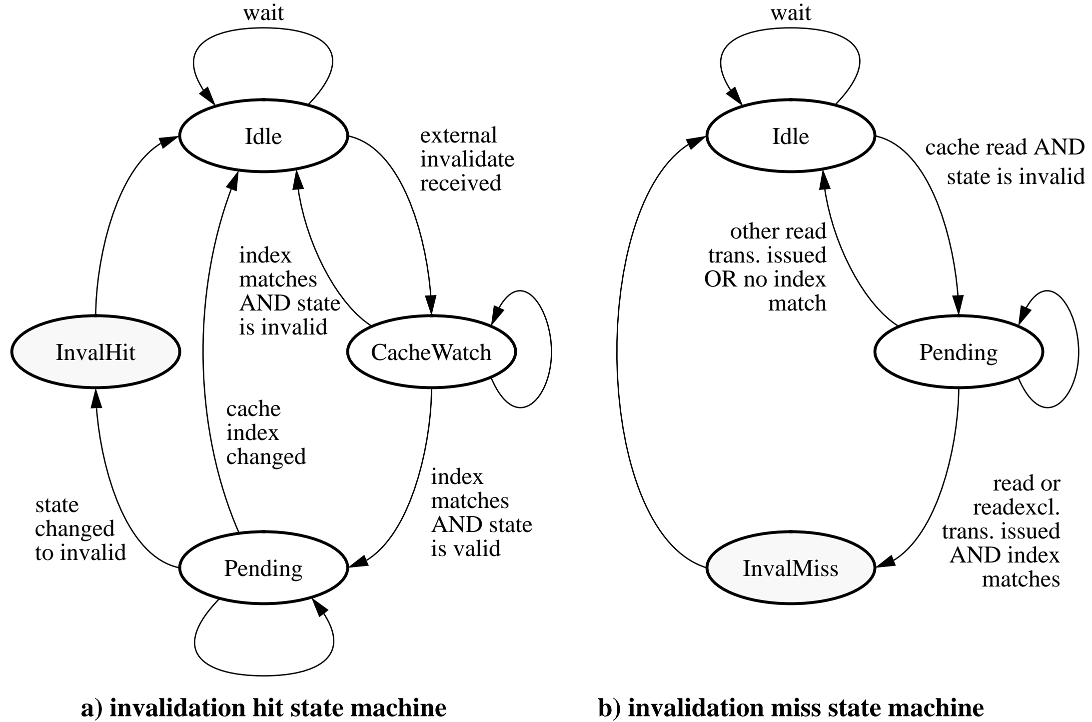

FIGURE 4.8. State machines to detect invalidation hits and misses.

Similarly, invalidation misses are detected by the state machine in Figure 4.8b. It waits for an access which reads an invalid state from the secondary cache. If the next read or read-exclusive transaction emitted by the External Agent maps to the same line, the event is counted. If the index doesn't match, or an uncached read is performed, the state machine resets. However, if a write or writeback transaction is encountered instead, it must still wait for an External Agent read transaction because the write may have been generated earlier than the invalidation miss. Again, this circuit only forms an estimate of invalidation misses because the cache tag is unknown and unchecked. Hence, it may incorrectly count some other types of misses as invalidation misses, but the error is expected to be small.

4.4.5 SRAM Counter Mode

The purpose of the SRAM Control Mode (SCM) register is to: 1) control the count-enable signal for the C&Inc device, 2) select the events to count, and 3) choose the address supplied to the SRAM. The count-enable circuit, which is identical to that used for the general-purpose counters, is controlled by the lowest 11 bits. The event to count is selected using the next 4 bits, SCM[14..11], as shown in Table 4.6. Not all possible events have been defined in the table, so room is left for future expansion. However, the few events that are present can be combined with the count-enable circuit to collect a wide variety of data. This will become apparent below.

SCM[14..11]	Appropriate Mode	Event
0	any	count nothing (disable)
1	any	count always
2	0	count secondary cache accesses
3	0	count secondary cache reads
4	0	count secondary cache writes
5	0	count secondary cache misses
6	1, 2, or 3	cache line reads + read exclusives + upgrades
7 to F	any	reserved

TABLE 4.6. SRAM counter events.

The address supplied to the SRAM counters is determined by the next group of seven SCM bits. The lower five of these, *i.e.* bits 15 through 19, control a collection of multiplexers in the MCC that form the SRAM address. This circuit is shown in Figure 4.9; refer to Table 4.4 for definitions of the various signals in the figure. The two upper bits control the tristate-enables in the figure and are encoded so that software cannot mistakenly enable competing drivers; the encoding is shown in Table 4.7. Furthermore, the multiplexer and tristate controls can be overridden by the muxAR control which is generated automatically when the R4400 attempts to read or write the SRAM counters.

SCM[21..20]	Mode	SRAM Address Source
0	Secondary Cache Mode	secondary cache index
1	Latency Timer Mode	latency timer (all 12 bits) plus muxPRlo data
2	MCC Mode	all MCC sources (muxPRlo, muxPRmid, and muxPRhi)
3	Latency Timer Compact Mode	latency timer (lower 7 bits, compressed format) plus muxPRlo and muxPRhi data

TABLE 4.7. SRAM counter modes.

The Secondary Cache Mode allows the SRAM counters to count secondary cache accesses, reads, or writes. Because all secondary cache accesses are caused by a primary cache miss, the number of primary cache misses can be monitored. By counting the number of loads and stores with the general-purpose counters, primary cache miss rates can easily be determined. Similarly, secondary cache miss rates can be measured. Note that the miss counts are per cache line, so small regions that suffer from conflict or invalidation misses can be determined. By initializing all of the SRAM with a suitable initial count and enabling interrupts on overflow, software can be informed when a cache line receives too many misses. Additionally, the R4400 can optionally split the secondary cache by placing instructions in the upper half (index bit 17 is set) and data in the lower half. By using the split mode, separate statistics on instructions and data can be collected.

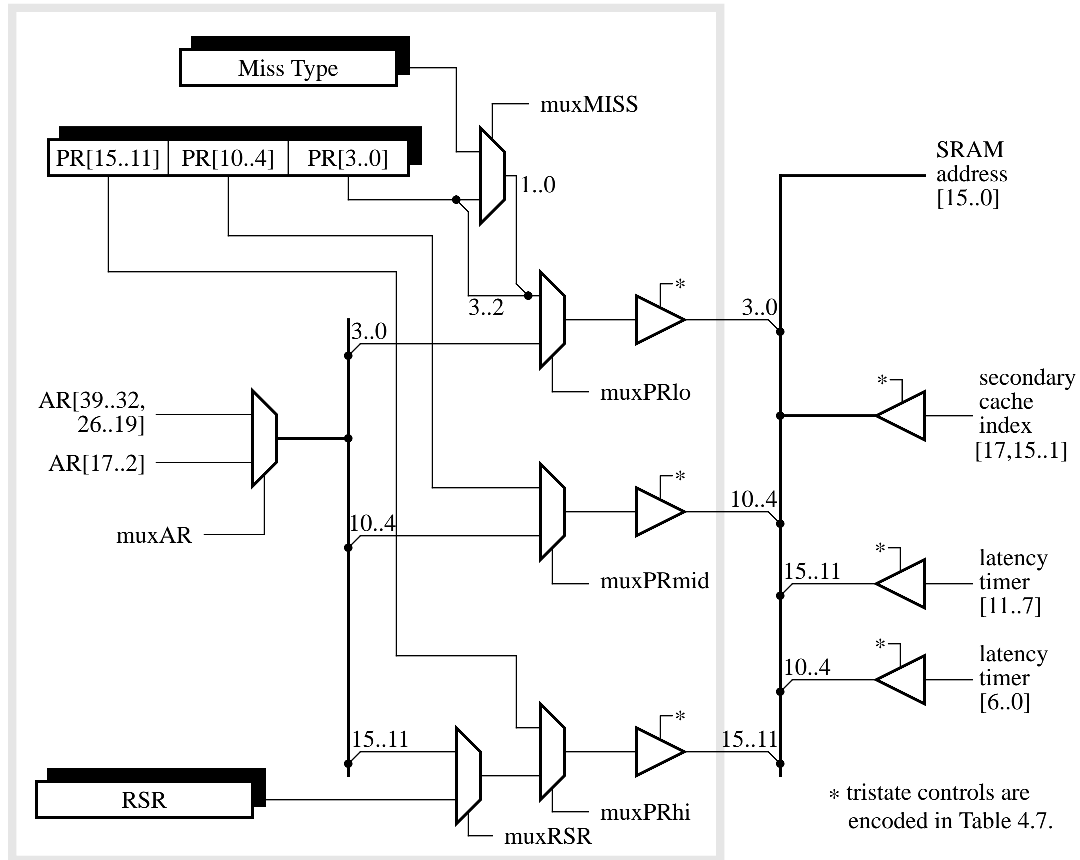

FIGURE 4.9. SRAM address generation. The outlined portion is inside the MCC.

The two Latency Timer Modes construct a histogram of memory access latencies. The histogram from the ‘full width’ timer shows exactly how many read requests took 0, 1, 2, ..., 4095 cycles to obtain a response. The compact timer divides these times into increasingly-sized bucket intervals as follows:

```
0-1, 2-3, ..., 62-63,
64-67, 68-71, ..., 124-127,
128-135, 137-143, ..., 248-255,
256-271, 272-287, ..., 494-511,
512-543, 544-575, ..., 992-1023,
1024-1087, 1088-1151, ..., 1984-2047,
2048-2175, 2176-2303, ..., 3968-4095.
```

From either of these histograms, a memory latency distribution, including characteristics such as average and variance, can be constructed.

The statistics generated with the Latency Timer Modes can be also be separated by the PhaseID Register, a *Miss Type Register (MTR)*, and a 5-bit *Response State Register (RSR)*. The PhaseID has been described previously, but MTR and RSR are new registers that describe a cache miss. The MTR is a 2-bit quantity that describes the type of cache miss according to Table 4.8. In contrast, the RSR contains the 3-bit memory state indicator (MSI) in the upper half and two bits in the lower half to indicate the request type.

Value	Description
0	successful ownership misses
1	cancelled ownership misses
2	invalidation misses
3	other misses

TABLE 4.8. Miss types encoded in the Miss Type Register.

The MSI bits returned with every NUMAchine memory response indicate whether data was returned from the network cache (a network cache hit) or memory and one of the four memory states: LV, GV, LI, and GI. In the case of a network cache hit, the state of the network cache line is returned; in all other cases the state at the home memory module is returned. The additional two bits in the RSR indicate whether the access was to local or remote memory and whether the request was a read or a read exclusive/upgrade. Together, these five bits give details about how far the request had to travel, the effectiveness of the network cache, and the amount of coherence traffic that was required to respond. *When using the RSR, a user must be careful to use the receiving (incoming) mask lest the SRAM counters be incorrectly updated before the RSR arrives.* The encoding of the RSR is shown in Table 4.9.

Value	RSR[4]	RSR[3]	RSR[2]	RSR[1]	RSR[0]
0	Network Cache Miss	Local State	Invalid	Local Address	Read
1	Network Cache Hit	Global State	Valid	Remote Address	Read Exclusive or Upgrade

TABLE 4.9. Response State Register encoding.

The final SRAM counter mode places the SRAM address completely under MCC control. This means the entire SRAM can be addressed by PhaseID register, part of it can come from the RSR or MTR, or portions can be constructed from the Address Register. The flexibility comes from the ability to specify the individual multiplexer control signals. One particular configuration, when these controls are all set to zero, uses the AR[39..32,26..19] bits to provide the SRAM address. In this 'Address Mode', all references to secondary-cache sized blocks (1 MB) are counted. This is an experimental feature to count how widespread the cache-miss access patterns of the application are. In particular, remote memory is counted distinctly from local memory, so a measure of locality can be constructed. By counting access latency, this mode can also help identify whether a particular memory module or network connection is more congested than others.

4.4.6 Master Counter Enables

Two master counter enables are provided so that monitoring can be easily stopped or started without affecting the configuration of the counters. Separate enables are provided for the SRAM and the general-purpose counters, but both may be enabled (or disabled)

simultaneously with one doubleword write. Furthermore, multiple processor card monitors can be enabled using a NUMAchine multicast write.

4.5 Summary

The NUMAchine monitor described above is capable of measuring many events, but it may not clear to the reader whether it satisfies the measurements proposed in Chapter 3. A summary of the 23 specific features recommended to monitor and how the implementation meets these requirements is shown in Table 4.10. From this table, it can be seen that only five items cannot be monitored, of which three are not applicable to the R4400. The other two items, memory and network queue measurements and basic block timing support are left as future implementation items. Of these, the memory and network queues should be given higher priority because they are relevant for evaluating NUMAchine architecture performance.

One of the primary objectives of this thesis is to maintain a cost-effective focus. In this regard, several design decisions for the NUMAchine processor card monitor, such as what size FPGA to use, were influenced by minimizing the cost of the hardware. To illustrate the costs, Table 4.11 shows the cost of the major performance monitor components described in this chapter. The total price of $345 is slightly inflated because two of the more expensive components, namely the Local Bus Controller and Counters & Interrupts circuits, contain not only monitoring circuits, but also other non-monitoring functions that are required by NUMAchine.

Section		Recommended Item	Implemented?	Notes
Section 3.2	1	dynamic instruction count	y	pipeline counters form groups of instruction counts
	2	cycles lost due to NOPs and pipeline slips	y	pipeline counters can measure slips NOPs can be measured via basic block counts
	3	cycles lost due to stalls	y	pipeline counters allow flexible grouping of different stalls
	4	pipeline flushing and restarting	partial	pipeline counters can measure pipeline flushes, processor support required for counting restarts
	5	TLB faults	y	can be monitored in software
	6	branch prediction	n	n/a — processor does not use branch prediction
	7	special hardware features	y	network cache hits
Section 3.3	8	informing memory operations	n	n/a — processor must have special support for this
	9	phaseID register	y	varies between 4-bits and 16-bits
	10	profile cache activity	y	SRAM can count primary cache misses, secondary cache misses, etc.
	11	cache-profile watchpoints or thresholds	y	can preload SRAM with negative-threshold and interrupt on overflow
	12	invalidation miss count	y	estimate only, need more FPGA pins to form an accurate count
	13	external invalidation hits	y	estimate only, need more FPGA pins to form an accurate count
	14	ownership misses	y	estimate only, need more FPGA pins to form an accurate count
	15	memory and network queue performance	n	future monitoring implementation
	16	measure locality of references	y	the RSR indicates a local/remote request; address-region monitoring can watch memory that is local/remote/both
	17	memory state indicator	y	result returned from memory indicates network cache hits and local/global or valid/invalid states
	18	measure total miss cycles	y	pipeline counters can be merged to count all cache miss stall cycles
	19	measure apparent miss cycles	n	n/a — R4400 has a simple pipeline with no latency-hiding mechanisms
Section 3.4	20	timestamp counter	y	provided by R4400
	21	local process counter	y	a general-purpose counter can be used for this
	22	basic block counting SRAM	y	supported by using entire 16-bit PhaseID
	23	basic block timing SRAM	n	accumulate function not implemented, possible future extension; SRAM must be shared with basic block counting item

TABLE 4.10. Comparison of recommended versus implemented features.

Circuit	Device	Quantity	Approximate Price ($CAD)
Local Bus Controller	FLEX8452A-5, 160-pin QFP	2	$90
Pipeline Status	MAX7064-10, 44-pin PLCC	2	$35
Counters & Interrupts	FLEX8636A-5, 84-pin PLCC	1	$70
SRAM	64k x 16	2	$30
Count & Increment	MAX7064-15, 84-pin PLCC	1	$15
Latency Timer	(shared with C&Incr)		
Configuration Controller	FLEX8636A-5, 160-pin PLCC	1	$75
buffers	latching, tristate	3	$15
other control	MAX7064-15, 84-pin PLCC	1	$15
		TOTAL	$345

TABLE 4.11. Approximate cost of monitoring components.

Chapter 5
Conclusions

Motivated by improving software performance in multiprocessors, this thesis has identified a number of performance measurements that can be made in hardware. These measurements were coalesced into a hardware performance monitor for NUMAchine which implements 18 of the 23 recommended features.

In retrospect, the MIPS R4400 processor used in NUMAchine was an excellent choice from a performance monitoring standpoint. The observable pipeline status and the single in-order pipeline were especially helpful at keeping the design simple yet flexible.

5.1 Contributions

In addition to developing hardware performance monitoring circuitry for NUMAchine, this thesis has made the following contributions:

Comprehensive List of Performance Measurements for Multiprocessors

A thorough examination of factors affecting performance in cache-coherent, shared-memory multiprocessors resulted in a list of 23 important types of performance measurements. Many of the measurements can be implemented directly by the processor, but some of them require specialized off-processor hardware. The cost of the external hardware is modest, so it should be feasible to add to all but the most cost-sensitive computers. In particular, the hardware suggested would add significant value to application developers and many scientific and engineering users.

Informing Memory Operations, TRIGGER Support

The idea of informing memory operations, where a cache miss stimulates the invocation of a software miss handler, was proposed in [Horowitz95]. To enhance performance and permit greater selectivity, this thesis has proposed that an external TRIGGER pin on the processor enable or disable the handler's invocation.

PhaseID Register

To separate performance statistics at a fine or coarse granularity, this thesis has recommended a PhaseID register. To guarantee proper consistency under all circumstances, the register should be integrated into the processor. A novel aspect of PhaseID use is attaching it to all network and memory transactions so that those subsystems can also be sensitive to changes in program state. Additionally, the problem of maintaining consistency of PhaseID contents to memory operations was briefly described.

Recent independent work by Martonosi [Martonosi96] has proposed a similar *category* register for separating performance data. However, in their use the category does not travel through the network with memory transactions as done with PhaseID.

Memory State Indicator

In addition to transporting the PhaseID value with transactions from the processor, this thesis has suggested attaching a memory state indicator (MSI) to responses. In the NUMAchine implementation, the MSI describes the state of the memory block as found at the home memory module or the network cache. It is used to help identify data sharing patterns and estimate the amount of coherence overhead necessary to construct the response.

To date, the only feature (of which we are aware) that is similar to this is the ability of some processors to count hits to cache lines found in a particular state.

Invalidation and Ownership Misses

Recent work on multiprocessor misses from invalidate protocols [Dubois93] concentrates on the precise identification of invalidation misses and obtaining ownership for writes. In particular, the penalty of obtaining ownership is downplayed because of relaxed consistency models. Instead, invalidation misses are categorized as true and false sharing misses by simulating updates within a write-invalidate protocol. Unfortunately, this scheme is not practical to implement in hardware because the intrusive updates involved could saturate an interconnection network.

This thesis has described obtaining ownership as an ownership miss because of the penalty associated in sequentially-consistent multiprocessors. Also, practical hardware mechanisms for detecting invalidation and ownership misses were outlined. Additionally, although invalidation misses are not categorized distinctly as true or false sharing misses, it was noted that some false sharing patterns can be detected using the external invalidation hit counts and separating misses based on the MSI and the processor request type.

Other Contributions

A reactive cache conflict detection scheme was developed by separately counting accesses to each cache line. A similar, but more limited, scheme was used in [Singhal94] which involved counting accesses to even and odd lines separately. Also, by raising an interrupt when one line suffers from excessive misses, our approach can be used to help identify the data objects that conflict in the cache.

Using the same base hardware, a method was developed for performing fine-grained timing that is comparable in overhead to basic block counting. This timing information is an alternative to program counter sampling, the current widely-used technique.

Due to current latency-hiding mechanisms found in recent processors, miss cycles per instruction (MCPI) no longer gauges processor performance well. Instead, this thesis pro-

posed a new metric, apparent MCPI, to help quantify the positive effect these mechanisms have on performance by measuring lost processing opportunity. In practise, it may be difficult to precisely quantify this opportunity cost; this could be the subject of future investigation.

5.2 Future Work

As testified by new processors, the realm of hardware-based performance monitoring is just beginning to take shape. Future work in the monitoring area involves better integration of monitoring functions in processors and new software tools to exploit the monitoring features. Also, improvements in computer architecture will demand new measurements be made for effective deployment of resources.

With respect to NUMAchine, future work involves the construction of hardware to monitor queue lengths and memory state transitions. A brief introduction to this work in progress is presented in Appendix A. Additionally, other work will involve providing an off-processor functional equivalent to the informing memory operations, exploring a use for the response to the read that changes the PhaseID register, creating a Configuration Controller circuit to use the SRAM as a small trace buffer, and the development of a bus-transaction tracing board.

Appendix A
Memory Card Monitoring

This appendix describes some of the monitoring features planned for the NUMAchine memory card performance monitor.

The memory card is organized as shown in Figure A.1. The monitor receives memory block addresses, the type of transaction (Command), FIFO depth changes, and information about the state of the memory block from other components of the memory card. The monitor contains reprogrammable FPGAs and counter SRAM so that complex histograms and event counting can be done.

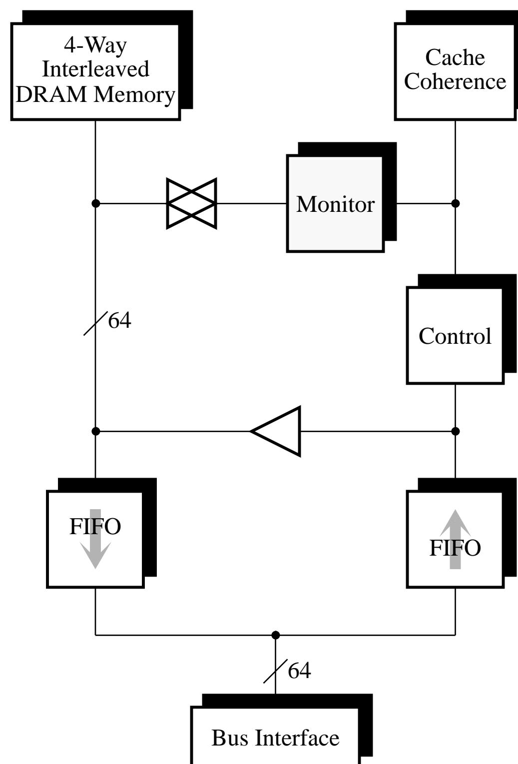

FIGURE A.1. Memory card organization.

FIFO Depths

Dedicated hardware counters are used to maintain the current depth of the incoming and outgoing FIFOs. Two types of depths are recorded: number of packets and number of transactions. Additionally, the maximum FIFO depth is recorded in a special register.

The current queue depth is added in an accumulator on every cycle. Dividing this by the number of cycles yields the time-average queue length. If the queue depth is only accumulated after every change in queue length, a population average is obtained instead.

Histogram Statistics

An SRAM memory and controller are used to produce different types of histograms. In particular, the most interesting types of histograms are a memory state hit table and page use statistics.

Data sharing patterns and performance prediction can make use of a memory state hit table. This table counts the number of request types that hit a particular memory block state. An example of this is shown in Table A.1.

	Request Type			
State	**Read**	**Read Exclusive**	**Update**	**Writeback**
LV	#	#	#	#
LI	#	#	#	#
GV	#	#	#	#
GI	#	#	#	#

TABLE A.1. Memory state hit table.

When collecting page use statistics (a.k.a. frequent flyer miles) every station's access to a page is recorded in the SRAM memory. If one page is accessed by a particular station significantly more often than the others, that page should be migrated to that station. The term 'frequent flyer miles' refers to the fact that frequent users of the page are logged, and eventually the station may collect enough points to get a free 'trip' to a new home. The best way of detecting when and how to migrate the page has not yet been decided.

Bibliography

[Amdahl67] G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proceedings of the AFIPS Spring Joint Computer Conference*, April 1967, pp. 483-485.

[Bacon94] D.F. Bacon, S.L. Graham, O.J. Sharp, "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, Vol. 26, December 1994, pp. 345-420.

[Bacque91] J.B. Bacque, "SUPERMON: Flexible Hardware for Performance Monitoring," M.A.Sc. Thesis, University of Toronto, 1991.

[Ball94] T. Ball, J.R. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4), July 1994, pp. 1319-1360.

[Berry89] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, J. Martin, "The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers," Technical Report CSRD-827, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1989.

[Collins95] R. Collins, "int$_e$l s$_e$crets," `http://x86.metronet.com`, December 1995.

[Convex94] Convex Computer, *SPP1000 Systems Overview*, Convex Computer Corporation, 1994.

[Convex95] Convex Computer, pmon man page, Convex Exemplar SPP-UX 3.1.134, Convex Computer Corporation, 1995.

[Cray92] Cray Research, *UNICOS Performance Utilities Reference Manual*, Cray Research Inc., May 1992.

[Dally94] W.J. Dally, S.W. Keckler, N. Carter, A. Chang, M. Fillo, W.S. Lee, "M-Machine Architecture v1.0," MIT Concurrent VLSI Architecture Memo 58, MIT Artificial Intelligence Laboratory, MIT, August 1994.

[Digital92] Digital Equipment Corporation, "DECchip 21064-AA Microprocessor Hardware Reference Manual," Digital Equipment Corporation, October 1992.

[Dubois93] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, P. Stenström, "The Detection and Elimination of Useless Misses in Multiprocessors," *Pro-*

ceedings of the 20th Annual International Symposium on Computer Architecture, San Diego CA, May 1993.

[Gao95] H. Gao, J.L. Larson, "A Year's Profile of Academic Supercomputer Users Using the CRAY Hardware Performance Monitor," Technical Report 1403, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, February 1995.

[Glew95] A. Glew, Intel, personal communication, December 1995.

[Goldberg93] A.J. Goldberg, J.L. Hennessy, "Mtool: An Integrated System for Performance Debugging Shared Memopry Multiprocessor Applications," *IEEE Transactions on Parallel and Distributed Systems*, 4(1), January 1993, pp. 28-40.

[Graham82] S.L. Graham, P.B. Kessler, P.B. McKusick, "gprof: A Call Graph Execution Profiler," Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, *SIGPLAN Notices*, 17(6), June 1982, pp. 120-126.

[Graham83] S.L. Graham, P.B. Kessler, P.B. McKusick, "An Execution Profiler for Modular Programs," *Software — Practice and Experience*, vol. 13, 1983, pp. 671-685.

[Heinrich94] J. Heinrich, *R4000 Microprocessor User's Manual*, Second Edition, MIPS Technologies Inc., Mountain View CA, April 1994.

[Hennessy96] J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, San Franscisco, CA, 1996.

[Hill88] M.D. Hill, "A Case for Direct Mapped Caches," *Computer*, 21(12), December 1988, pp. 25-40.

[Hollingsworth93]
J.K. Hollingsworth, B.P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," *Proceedings of 7th International Conference on Supercomputing*, July 1993, pp. 185-194.

[Hollingsworth94]
J.K. Holllingsworth, B.P. Miller, "Dynamic Program Instrumentation for Scalable Performance Tools," *Proceedings of the 1994 Scalable High-Performance Computing Conference*, Knoxville TN, May 1994.

[Horowitz95] M. Horowitz, M. Martonosi, T.C. Mowry, M.D. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors," Technical Report CSL-TR-95-673, Computer Systems Laboratory, Standford University, Stanford, CA, July 1995.

[HP94] Hewlett-Packard, "PA-RISC 1.1 Architecture and Instruction Set Reference Manual," Third Edition, HP Part Number 09740-90039, Hewlett-Packard Company, February 1994.

[Jouppi90] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th International Symposium on Computer Architecture*, 1990, pp. 364-373.

[KSR92a] Kendall Square Research, *KSR 1 Principles of Operation Manual*, Kendall Square Research Corporation, Boston, MA, 1992.

[KSR92b] Kendall Square Research, pmon man page, KSR OS R1.2.2, Kendall Square Research Corporation, August 1994.

[Larus93] J.R. Larus, "Efficient Program Tracing," *Computer*, 26(5), May 1993, pp. 52-61.

[Lebeck94] A.R. Lebeck, D.A. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study," *Computer*, 27(10), October 1994, pp. 15-26.

[Lenoski92] Daniel E. Lenoski, "The Design and Analysis of DASH: A Scalable Directory-based Multiprocessor," Technical Report CSL-TR-92-507, Computer Systems Laboratory, Standford University, Stanford, CA, February 1992.

[Ludloff94] C. Ludloff, "4P: Programmer's Processor Power Package," `http://www.x86.org/4p`, Version 3.14, January 1995.

[Manjikian95] N. Manjikian, personal communication, November 1995.

[Martonosi95] M. Martonosi, A. Gupta, T.E. Anderson, "Tuning Memory Performance of Sequential and Parallel Programs," *Computer*, 28(4), April 1995, pp. 32-40.

[Martonosi96] M. Martonosi, D.W. Clark, M. Mesarina, "The SHRIMP Performance Monitor: Design and Applications," *First SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.

[Mathisen94] T. Mathisen, "Pentium Secrets," *Byte*, 19(7), July 1994, pp. 191-192.

[Miller95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *Computer*, 28(11), November 1995.

[MIPS95] MIPS Technologies, *R10000 Microprocessor User's Manual — Version 1.0*, MIPS Technologies Inc., Mountain View, CA, June 1995.

[Motorola94a] Motorola, *PowerPC Microprocessor Family: The Programming Environments*, Motorola Inc., 1994.

[Motorola94b] Motorola, *PowerPC 604 Risc Microprocessor User's Manual*, Motorola Inc., 1994.

[Shand92] M. Shand, "Measuring System Performance with Reprogrammable Hardware," PRL Research Report 19, Digital Equipment Corporation, Paris Research Laboratory, Rueil-Malmaison, France, August 1992.

[Singhal94] A. Singhal, A.J. Goldberg, "Architectural Support for Performance Tuning: A Case Study on the SPARCcenter 2000," *Proceedings of The 21st Annual International Symposium on Computer Architecture*, Chicago, IL, April 1994, pp. 48-59.

[Smith91] M.D. Smith, "Tracing with pixie," Technical Report CSL-TR-91-497, Computer Systems Laboratory, Standford University, Stanford, CA, November 1991.

[Stumm93] M. Stumm, Z. Vranesic, R. White, R. Unrau, K. Farkas, "Experiences with the Hextor Multiprocessor," CSRI Technical Report CSRI-276, Computer Systems Research Institute, University of Toronto, Toronto, 1993. Extended version of paper with same title in Proc. Intl. Parallel Processing Symposium Parallel Systems Fair, 1993, pp. 9-16.

[Sun95] Sun, *SuperSPARC II Data Sheet*, Sun, 1995.

[Torrellas95] J. Torrellas, C. Xia, R. Daigle, "Optimizing Instruction Cache Performance for Operating System Intensive Workloads," to appear in *IEEE Transactions on Computers*, 1995.

[Torrie95] E. Torrie, C-W. Tseng, M. Martonosi, M.W. Hall, "Evaluating the Impact of Advanced Memory Systems on Compiler-Parallelized Codes," *International Conference on Parallel Architectures and Compilation Techniques*, June 1995.

[Varley93] D.A. Varley, "Practical Experience of the Limitations of gprof," *Software — Practice and Experience*, 23(4), April 1993, pp. 461-463.

[Vranesic91] Z.G. Vranesic, M. Stumm, D.M. Lewis, R. White, "Hector — A Hierarchically Structured Shared-Memory Multiprocessor," *Computer*, 24(1), January 1991, pp. 72-80.

[Vranesic95] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, S. Srbljic, "The NUMAchine Multiprocessor," CSRI Technical Report CSRI-324,

Computer Systems Research Institute, University of Toronto, Toronto, June 1995.

[Vuillemin91] J.E. Vuillemin, “Constant Time Arbitrary Length Synchronous Binary Counters,” *1991 IEEE 10th Symposium on Computer Arithmetic*, Grenoble, France, June 1991.

[Welbon94] E.H. Welbon, C.C. Chan-Nui, D.J. Shippy, D.A. Hicks, “POWER2 Performance Monitor,” PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000, IBM Corporation, SA23-2737, 1994, pp. 55-63.

[Xia96] C. Xia, J. Torrellas, “Improving the Data Cache Performance of Multiprocessor Operating Systems,” to appear in *2nd International Symposium on High-Performance Computer Architecture*, 1996.

[Zilic95] Z. Zilic, G. Lemieux, K. Loveless, S. Brown, Z. Vranesic, “Designing for High Speed-Performance in CPLDs and FPGAs,” *Proceedings of The 3rd Canadian Workshop on Field-Programmable Devices (FPD’95)*, Montreal, Canada, June 1995, pp. 108 - 113.