MEMORY PROFILING ON SHARED-MEMORY MULTIPROCESSORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jeffrey S. Gibson
June 2004

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. John Hennessy
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Mark Horowitz

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Mendel Rosenblum

Approved for the University Committee on Graduate Studies:

Abstract

Tuning application memory performance can be difficult on any system but is particularly so on distributed shared-memory (DSM) multiprocessors. This is due to the implicit nature of communication, the unforeseen interactions among the processors, and the long remote memory latencies. Tools, called memory profilers, that allow the user to map memory behavior back to application data structures can be invaluable aids to the programmer. Unfortunately, memory profiling is difficult to implement efficiently since most systems lack the requisite hardware support. This dissertation introduces two techniques for efficient memory profiling, each requiring hardware support on either the processor or the system node controller.

The first technique, called TrapPoint, uses processor support for a trapping cache miss to point out memory bottlenecks. We construct a prototype on the versatile FLASH multiprocessor to study its feasibility. We show that modest processor support can be used to construct a useful memory profiler with acceptable overhead.

The FlashPoint memory profiler uses support on the system node controller to collect similar performance information. The FLASH multiprocessor was designed to allow for instrumentation of the node controller, enabling us to construct a prototype. Since profiling is done in the node controller, FlashPoint has access to more information about the memory traffic, such as cache-coherence events, than a processor-based monitor such as TrapPoint. It is therefore able to collect an extended memory profile.

Although FlashPoint requires more hardware support than TrapPoint, it overcomes many of TrapPoint's shortcomings. The required actions for memory profiling are quite similar to those required for cache coherence, so there are numerous synergies in implementing memory profiling on the same node controller that manages the cache-coherence protocol. Performing memory profiling in the node controller therefore allows a memory

profiler to collect more data with lower overhead and higher accuracy than is possible on the processor.

Since memory profiling data can be so valuable and it can be collected with relatively little hardware support, we argue that future DSM multiprocessors should be designed with support for memory profiling. This support is best done in the system node controller, but for implementations where this is infeasible, an acceptable monitor can implemented with processor support.

Acknowledgements

There are many people I'd like to thank for making this disseration possible. First, I'd like to thank my advisor, John Hennessy, for his guidance throughout my Stanford career. I would also like to thank my other committee members, Mark Horowitz and Mendel Rosenblum, for their advice and their help in shaping this thesis.

As FLASH is such a large, complex system, any FLASH work is necessarily a team effort, and this work would not have been possible without significant contributions from the entire FLASH hardware group: John Hennessy, Mark Horowitz, Mendel Rosenblum, Anoop Gupta, Dave Ofelt, Jeff Kuskin, Dave Nakahira, Mark Heinrich, Joel Baxter, Jules Bergmann, Hema Kapadia, Ravi Soundararajan, Jeff Solomon, Alan Swithenbank, Bob Kunz, and David Lie. I'd like to thank them all for their help, for their instruction, for their contributions to this thesis, and for generally making Stanford a pleasant place to spend nearly eight years.

The early implementation of FlashPoint was done on the SimOS simulator, and I'd like to thank the SimOS group, particularly Robert Bosch and Kinshuk Govil, for their help in all things SimOS. Similarly, I'd like to thank Kinshuk Govil for his help with the IRIX kernel.

I'd also like to thank everyone that made direct contributions to this thesis. Dave Ofelt, Mark Heinrich, and Margaret Martonosi did the early FlashPoint work that served as a starting point for this dissertation. Bob Kunz implemented a significant portion of the TrapPoint interrupt handler. David Lie added support to Extended FlashPoint and wrote a FLASH daemon to export FlashPoint data. Robert Bosch implemented Thor, the data visualization system for FlashPoint, and this was invaluable for analysis of application performance.

While working on this thesis, I was also an intern at Silicon Graphics, and I'd like to

thank them for their help and support.

Last, but not least, I'd like to thank my funding sources. This research was supported by Department of Energy ASCI contract LLL-B341491 and DARPA contract DABT63-94-C-0054.

Contents

Abstract **iv**

Acknowledgements **vi**

1 Introduction **1**

1.1 Distributed Shared-Memory Multiprocessors 3

1.2 Software Techniques 6

1.2.1 Tango Lite 6

1.2.2 Fast-Cache 7

1.2.3 ATOM 7

1.2.4 SimOS 8

1.2.5 Summary 9

1.3 Measuring Memory Performance in Hardware 9

1.3.1 What Information to Monitor 9

1.3.2 Hardware Implementations 10

1.4 Thesis Overview 13

2 Memory Profiling **14**

2.1 Description 14

2.1.1 Data Layout 15

2.1.2 Data placement 17

2.1.3 False Sharing 17

2.2 Examples 18

2.2.1 Blocking in FFT and Radix 18

2.2.2 Data Migration in IRIX 23

2.2.3 Microbenchmarks and Cache Replacement Policy 25
2.2.4 Cache Transfers in AMMP 26
2.2.5 Placement and False Sharing in VirtualMesh 30
2.3 Implementation Issues 35
2.4 Existing Implementations 37
2.4.1 Simulation: MemSpy 38
2.4.2 Software Shared Memory: Paradyn 39
2.4.3 Hardware Counters: `snperf` 41
2.4.4 Page Reference Counters 42
2.5 New Implementations 43

3 Processor-Based Memory Profiling **44**
3.1 FLASH, the prototype hardware 45
3.2 TrapPoint Implementation 47
3.2.1 Gross Miss Counts 47
3.2.2 Data Structures 48
3.2.3 Procedures 51
3.3 Methodology 52
3.3.1 SPLASH-2 Applications 52
3.3.2 SPEC OMP Applications 53
3.3.3 Research Application 54
3.4 Overhead 54
3.5 Sampled TrapPoint 56
3.5.1 Overhead 57
3.5.2 Accuracy 61
3.5.3 Evaluation 63

4 Node Controller-Based Memory Profiling **64**
4.1 FlashPoint Implementation 65
4.1.1 Protocol 65
4.1.2 Data Structures 68
4.1.3 Procedures 70
4.1.4 Remote Miss Example 72

4.1.5 TLB Misses . . . 75
4.2 Overhead . . . 76
4.2.1 Protocol Overhead . . . 78
4.2.2 Application Overhead . . . 79
4.2.3 TLB Overhead . . . 80
4.2.4 Memory Overhead . . . 80
4.3 Extended FlashPoint . . . 81
4.3.1 Protocol Extensions . . . 81
4.3.2 Overhead . . . 83

5 Conclusions **86**
5.1 Usefulness . . . 86
5.2 Implementation . . . 89
5.2.1 Processor Based . . . 89
5.2.2 Node Controller Based . . . 90
5.3 Conclusions and Future Work . . . 91

Bibliography **94**

List of Tables

2.1 Overhead of remote memory access on Blizzard and FLASH. 40

3.1 TrapPoint overheads. 55
3.2 Normalized execution time for Sampled TrapPoint at several average sampling periods. 57

List of Figures

1.1 Block diagram of a distributed shared-memory (DSM) multiprocessor. . . . 4

2.1 Naive matrix multiplication implementation 15
2.2 Transpose phase of a four-processor FFT blocked for first level data cache. . 20
2.3 Permutation phase of a four-processor Radix sort with a radix of 256. . . . 22
2.4 Extended memory profile of the main parallel section of AMMP. 27
2.5 Extended memory profile of the procedures in VirtualMesh on 8 processors. 31
2.6 Extended memory profile of the data structures in the `phcalc-fish` procedure of VirtualMesh. 32
2.7 Code excerpts from VirtualMesh 33
2.8 Extended memory profile of the procedures in VirtualMesh on 8 processors after a transpose phase has been added. 35

3.1 Block diagram of a FLASH node. 45
3.2 MAGIC block diagram. 46
3.3 Stats Record definition for TrapPoint 47
3.4 FLASH physical address layout. 50
3.5 Overheads for Sampled TrapPoint 58
3.6 Overheads for Sampled TrapPoint with high sampling periods. 60
3.7 Error fractions for Sampled TrapPoint at several sampling periods. 62

4.1 Stats Record definition for FlashPoint 66
4.2 The `PILocalGet` protocol handler. 67
4.3 Handlers and Messages invoked on a remote read miss. 72
4.4 The `NILocalGet` protocol handler. 74
4.5 The `NIRemotePut` protocol handler. 75

4.6 Overheads for FlashPoint 77
4.7 Stats Record definition for Extended FlashPoint 83
4.8 FlashPoint and Extended FlashPoint overheads. 84

5.1 Original FlashPoint procedure call macro. 87
5.2 Updated FlashPoint procedure call macro. 88

Chapter 1

Introduction

Memory behavior is often the primary performance bottleneck for many programs on a broad class of systems because of a large and ever-increasing performance gap between processors and memory [8]. Despite its importance, however, tuning application memory performance can be quite difficult. The performance cost of each memory operation is highly dependent on the performance of the memory hierarchy, which includes both the main memory and the processor caches.

Memory technology suffers from the trade-off that memories can be manufactured to be either large or fast, but not both simultaneously. Large main memories are necessary in modern machines to run large, data-intensive applications, so caching techniques are necessary to obtain high performance. A *cache* is a small, fast memory used to hold data that the processor is predicted to request soon. When the processor makes a memory request, it first checks to see if the required data resides in the first-level (L1) cache. This is the smallest, fastest memory in the machine, usually residing on the same chip as the processor core, and memory requests that can be serviced by the L1 cache (L1 *cache hits*) are the cheapest form of memory access, often taking only a single processor cycle. Requests that cannot be serviced by the L1 cache (L1 *cache misses*) are forwarded to the second-level (L2) cache, which is a larger, though somewhat slower, memory that may or may not be off-chip. L2 cache hits can be on the order of ten processor cycles. Some architectures even include a third-level cache that handles L2 cache misses. If a request cannot be serviced by any level of cache, then the processor has no choice but to issue a request to main memory, which can take on the order of one hundred processor cycles. The cost of a memory access

depends on the level of the hierarchy that services the access, with L1 cache hits being the fastest and main memory accesses being the slowest. The goal of memory hierarchy design is to present the illusion of a memory that is as large as main memory and almost as fast as the L1 cache. For the remainder of this thesis, we use the term *memory* to refer exclusively to main memory, not cache memory.

It is important to note that the contents of the caches are maintained by hardware, not the programmer. Caches exploit the properties of *spatial locality* and *temporal locality* in an attempt to capture the data that the processor will access soon. Spatial locality is the property of programs that if the processor accesses a particular memory address, it is likely to access nearby addresses soon. Temporal locality is the property that if the processor accesses a memory address, it will likely access the same address again soon. Spatial and temporal locality are properties that tend to be true of large sections of most programs. To the extent that a program exhibits spatial and temporal locality, the hardware will be more successful in maintaining the illusion that memory is as fast as the smallest levels of cache, and this will require no special action on the part of the programmer. When the caches do not perform well, however, the illusion breaks down as the processor must access main memory directly. In this case, the processor spends most of its time waiting for data from memory, rather than performing useful work.

It is therefore imperative that high-performance programs be written to maximize the cache performance. Unfortunately, it can be difficult to determine which memory accesses in a program will cause cache misses, since the programmer has no explicit control over the caches. Typically, the programmer will attempt to organize the program to maximize spatial and temporal locality and hope for the best. When this fails, the program will have poor performance, and it can be difficult to figure out why. Memory performance tools are required for determining which accesses cause poor cache behavior, thus giving the programmer an opportunity to fix the problem.

Multiprocessor environments introduce additional complexity to high-performance programming. It can be particularly difficult to write code that exhibits good cache behavior in an environment with multiple processors (each with its own caches) and multiple physical memory modules, because the programmer must worry about the physical distribution of memory and interactions among the processors and caches, in addition to the performance of each uniprocessor memory hierarchy. Multiprocessor performance issues are discussed

more thoroughly in the next section.

This thesis focuses on tools for collecting memory performance data on distributed shared-memory (DSM) multiprocessors. In this chapter, we first describe the DSM architecture and introduce some memory performance issues particular to this type of machine. We then describe what types of performance tools are desirable and what implementations of these tools currently exist. In the remainder of the thesis, we examine a particular type of performance monitor, a memory profiler, and introduce two efficient hardware implementations. We prototype both memory profilers on the FLASH [11] multiprocessor and contrast the two implementations.

1.1 Distributed Shared-Memory Multiprocessors

DSM machines are organized as a network of nodes, where each node contains a compute processor (possibly more than one), memory, I/O devices, and a system node controller to manage communication. A block diagram of such a machine is shown in Figure 1.1. The term *distributed* refers to the fact that portions of memory exist on the compute nodes rather than in a centralized location. Distributing memory is done in large multiprocessors (typically much larger than the four nodes shown in Figure 1.1) since contention for a centralized memory quickly becomes a performance bottleneck as machine size increases. The term *shared-memory* means that any compute processor can access any memory location. There is one logical address space across the physically distributed memory modules. The processors simply execute load and store instructions, and the system manages the communication if those addresses happen to be remote. The node controller orchestrates the communication: when the processor makes a memory request (i.e., after missing in its caches), it sends the request to its local node controller. The node controller either services the request from local memory or forwards the request across the interconnection network to a remote node controller. Examples of DSM machines include the Stanford DASH [15] and FLASH [11] multiprocessors and the Origin 2000 and Origin 3000 machines from Silicon Graphics, Incorporated (SGI). For the remainder of this thesis, we use the term shared-memory to refer to distributed shared-memory machines.

Since the latency of remote memory is even longer than that of local memory, caching is even more important in DSM machines than in uniprocessors. Having caches on multiple

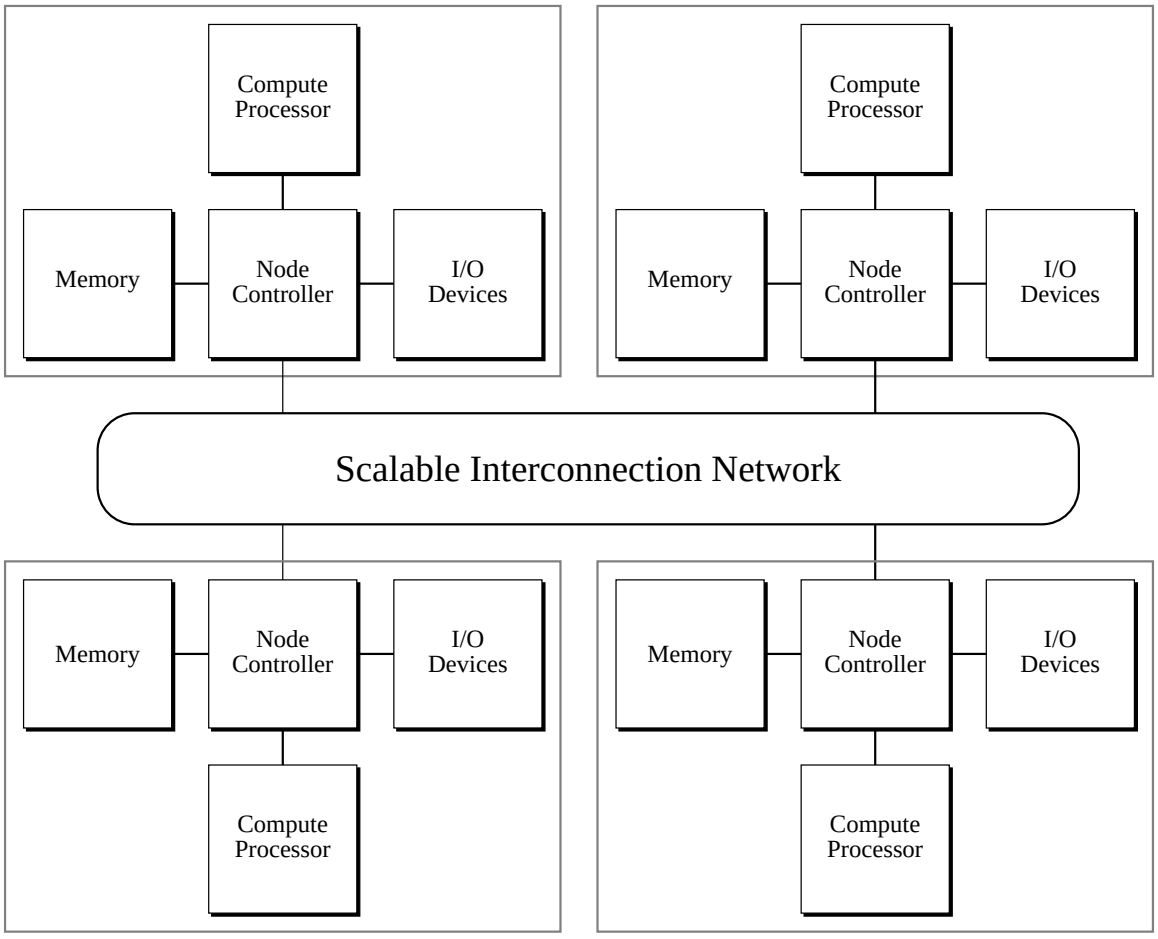

Figure 1.1: Block diagram of a distributed shared-memory (DSM) multiprocessor.

processors, however, introduces the problem of *cache coherence*. The contents of any particular memory location may exist in the caches of multiple processors. As long as a memory location is only read, there is no problem, but if any processor tries to write a shared value, the other processors' caches must be updated or invalidated. One function of the system node controller is therefore to manage the *cache-coherence protocol*, which is the set of messages and state transitions that ensures all processors see a consistent view of memory. Cache coherence is necessary if caching is to be invisible to the programmer, as it is with uniprocessors.

The alternative to DSM for large multiprocessors is *message passing*. The hardware

architecture is actually quite similar to DSM, but message passing machines support a different programming model. In this paradigm, processors can directly access only local memory, and communication is done via explicit send and receive primitives. Since memory is not cached by multiple processors, these machines do not have to maintain coherent caches.

The primary advantage of shared memory is that it presents a much more intuitive programming model to the user than does message passing. The user can allocate data structures that span the memory on multiple nodes and access them through normal memory operations. The system manages whatever communication is necessary. Having control over communication is a performance issue for DSM machines, but for message passing it is a matter of correctness.

Despite their ease of programming from a logical perspective, shared-memory multiprocessors can present significant challenges to programmers who are trying to enhance the performance of an application. Memory system performance can have a dramatic impact on application performance, with memory accesses costing from one to several hundred cycles. Viewed from any particular processor, memory nearby has lower latency then memory far away. The placement of physical memory therefore has performance implications. Because of the variations in memory latencies, distributed-memory machines are often referred to as NUMA (Non-Uniform Memory Access) and DSM machines as ccNUMA (cache-coherent NUMA). Support for cache coherence, while easing programming, complicates the process of understanding which references are expensive and which are not, since interactions among the processors can cause additional cache misses (called *coherence misses*). It can be difficult to determine which sections of application source code will cause remote memory traffic or will interfere with the caches of remote processors, since communication is handled by the hardware, not the programmer.

Tools are therefore required to study memory performance on DSM machines, and these tools can be implemented in either software or hardware. The next section describes software techniques for gathering memory performance data, and the subsequent section discusses hardware methods.

1.2 Software Techniques

One way of understanding memory behavior is to simulate the hardware. Simulation is the only available software technique because the programming model hides caching; there is simply no way to tell until run time where cache misses will occur. Simulators detect misses by instrumenting every memory operation and using the resulting miss trace to drive a model of the memory system.

Most simulation-based tools for studying memory behavior involve the use of simulators to detect cache misses and possibly attempt to determine their cause. We briefly describe several such tools.

1.2.1 Tango Lite

Tango Lite [9] is a simulation system used to evaluate memory performance. The application under study is instrumented by a pre-processing tool, `aug`, which adds calls to the simulator for each memory access. It also adds additional instrumentation such as procedure identification and timing estimation.

The goal of Tango Lite is efficient simulation of novel memory architectures. By simulating only the memory system and using the host machine to execute other application instructions, simulators such as Tango Lite can be made much faster than full machine simulators. There is a corresponding accuracy penalty as *only* the memory system is simulated in any detail. Rough estimates are used for all other timings.

Tango Lite was used in the design phase of the FLASH multiprocessor [11]. Linking Tango Lite with a detailed model of the FLASH memory system enabled the designers of the FLASH hardware to explore the design space. Similarly, early application performance studies of with FLASH were done well in advance of hardware [26], so Tango Lite was used to collect these results.

Tango Lite suffers from the same performance and accuracy problems of all architectural simulators. So much detail about the system is abstracted away for performance reasons that the results cannot be trusted completely. We show in later chapters how performance analysis done on the FLASH hardware differs with those performed by Tango Lite in [26]. The performance of Tango Lite also limits its applicability. Even with no memory system simulator, execution times of a simulated 24-processor system were measured to

be 10-40 times longer on Tango Lite than on the host machine. Overheads with a memory system simulator were well over 100 times the execution time. Tango Lite simulates multiprocessors, but the simulator itself is not parallel, so the simulation time is proportional to the size of the simulated machine. This is clearly a limitation in studying the behavior of large machines.

1.2.2 Fast-Cache

The Fast-Cache simulator [13] runs on a SuperSPARC microprocessor and is quite similar to Tango Lite. An instrumentation pass, called EEL [12], instruments each memory instruction in the application, and the instrumentation calls into a memory system simulator.

Like Tango Lite, Fast-Cache uses simplistic models of hardware to estimate the time not spent in the memory system, since it is mainly concerned with obtaining miss rates. The implementation goal of Fast-Cache is fast memory system simulation, and the instrumentation is optimized to be as lightweight as possible. The common case of a cache hit causes little overhead, and procedure calls into the simulator are also eliminated (via inlining) even for misses.

The result is that Fast-Cache is substantially faster than Tango Lite; its authors report slowdowns of 2-7 versus the performance on an uninstrumented program. Though certainly much faster than many other simulation techniques, even slowdowns of 2-7 make the cost of simulating many large applications prohibitive. The other drawback is that detail was sacrificed in pursuit of performance. Fast-Cache does not simulate multiple processors, thus severely limiting its applicability. Indeed, the overheads of Tango Lite would be much smaller (though perhaps not as small as Fast-Cache) if it were only simulating a single processor.

1.2.3 ATOM

The ATOM (Analysis Tools with OM) system [25] was developed by Digital Equipment Corporation (now Compaq) as a flexible interface for instrumenting executables with the goal of understanding their behavior.

ATOM is not a simulator in itself, but it can be used to generate simulators similar to

Fast-Cache and Tango Lite. Its authors demonstrate an ATOM-generated tool that instruments every memory access with calls into a very simple cache simulator (one level of direct-mapped 8 kB cache). It models only a uniprocessor, but the instrumented program still sees a slowdown of 12. ATOM could be used to build a more complex, useful simulator such as Tango Lite, but it would share all of Tango Lite's performance and accuracy concerns.

Of the three memory system simulators we have surveyed, only Tango Lite is capable of simulating multiprocessor memory systems, and it does so with a large (two orders of magnitude or more) slowdown. Both Fast-Cache and the ATOM-based simulator are faster, but if they were augmented to perform multiprocessor simulation, they would likely see performance similar to Tango Lite.

1.2.4 SimOS

The SimOS simulator [21] differs from the other simulators mentioned in that it simulates an entire machine. The somewhat inappropriately named tool does not simulate an operating system, but it models the machine in enough detail to boot and run an operating system on the simulated machine.

The advantage of complete machine simulation is detail. Virtually every aspect of the system is modeled, and being simulation, an arbitrary amount of instrumentation can be added to the system and not perturb the performance of the simulated program.

The disadvantage, of course, is performance. SimOS has a fast binary translation mode used to boot the operating system in a tolerable amount of time. This fast mode, called Embra, is not useful for collecting memory performance data, however, since it does not model caches. The more detailed modes that do model the memory system cause run times to be hundreds or thousands of times what they would be if the application were running on hardware. The detailed modes also share Tango Lite's limitation in that the simulator itself is not parallel, so simulation time is proportional to the number of processors.

The high overhead of complete machine simulation means that it is useful for studying small programs in great detail, but the simulation time for large applications is prohibitive. It is also worth noting that even simulators as detailed as SimOS make numerous abstractions for the sake of performance. Just because certain effects are accounted for in a simulator does not necessarily mean they are modeled correctly in terms of performance [6].

1.2.5 Summary

Simulation methods have the great advantage that any aspect of the processor or program that might affect performance can be instrumented. Simulation, however, suffers from two disadvantages. One is the question of simulation accuracy. As one recent paper [6] shows, simulators are often highly inaccurate because of omissions or lack of complete validation against real hardware. The more important disadvantage of simulation is that it is expensive, with run times of hundreds of times longer than the actual application run time. For this reason, simulation approaches have limited usefulness for large applications, especially on large multiprocessors.

1.3 Measuring Memory Performance in Hardware

Because of the extremely high cost of simulation, our focus in this thesis is in hardware techniques for measuring memory performance. We examine what sorts of information about memory system performance can be collected and what the costs of collecting such information are, assuming different levels of hardware support. We show that additional hardware support can be used to extract more detailed performance information while maintaining low overhead, little or no interference with the application, and a high degree of accuracy.

Before we examine methods to instrument running applications, it is useful to examine what types of information we might want to collect.

1.3.1 What Information to Monitor

Getting detailed information on memory references, including miss counts, requires the use of some sort of hardware based counters. With the ability to have detailed hardware instrumentation, four levels of performance data can be collected:

1. Gross event count information, such as the number of cache misses, for an entire program or for large program segments, such as procedures. We call this level *gross event counts*.

2. Event counts on a fine-grain basis, usually associated with a single line of code. This is sometimes called *instruction profile* information, since it can be used to compute a CPI approximation.
3. Detailed information about misses that allows a performance monitor to associate misses with the data structures that caused them. This is often called *memory profiling*.
4. A memory profile that includes cache-coherence protocol information such as invalidations and three-hop misses in addition to normal memory profile data. We call this *extended memory profiling.*

Each of these types of information can be used to explain program behavior at successively more detailed levels. Correspondingly, to obtain successively more detailed levels of information requires a more extensive instrumentation capability, if the information is to be gathered at low overhead.

1.3.2 Hardware Implementations

In this section, we discuss hardware implementation techniques and existing tools that collect the various types of performance data we describe above. We focus mainly on tools for MIPS processors, though similar tools exist for other commercial microprocessors.

Gross Miss Counts

Gross miss counting tools are implemented with the hardware performance counters that exist on many commercial microprocessors. We describe `perfex`, a performance tool available on SGI systems, as an example of a gross miss counting tool.

The MIPS R10000 processor contains two configurable performance counters, each of which can be made to count 16 distinct event types, for a total of 30 different event types (not 32 since two events can be counted by either counter) [28]. Examples of event types include graduated instructions, issued instructions, L1 data cache misses, L2 cache misses, and TLB misses. Though only two 32-bit counters exist, the operating system has sole access to them and is therefore able to provide the appearance of two 64-bit counters per process. The `perfex` tool was implemented using these performance counters. Since

cache misses are included in the set of countable events, `perfex` can be used to obtain a rudimentary view of application memory behavior.

In its simplest form, `perfex` takes arguments specifying the two events to count and a program to run. It reads the values of the performance counters, executes the program, then reads the counters again and reports the number of events counted during the course of the run. Multiple runs measuring different events can be used to study the application in more detail than two counters permit. The data returned by `perfex` is aggregated for the entire run of the program (actually, for each thread), so it provides no information as to what parts of the code or data are responsible for the counted events. There is also a library interface to the performance counters (called `libperfex`), and calls to this library can be inserted to read the counters during the execution of a program. This allows some coarse-level localization. Reading the performance counters requires an expensive system call, however, so `libperfex` can only be used to measure large program segments.

The operating system also supports a sampling mode that enables `perfex` to sample all events simultaneously, in a sense. On each scheduling interrupt, the OS switches the events being monitored, and `perfex` extrapolates values for all counters from the samples. Since the scheduling interrupt occurs every 10 ms and each counter multiplexes 16 events, this technique is only accurate for runs that last much longer (i.e., orders of magnitude longer) than 160 ms. Typically, the sampling mode should only be used in runs that last at least several minutes, but it provides, in a single run, counter values for all countable events.

We find that `perfex` is most useful for determining whether or not a program is dominated by memory effects. The primary disadvantage it shares with all other gross miss counting tools is that it provides no details as to what those memory effects are or what regions of code or data are responsible for them.

Instruction Profiling

To generate the information needed to create an instruction profile, we use sampling driven either by timer interrupts or by counter overflows. The key idea is that sampling can be used to interrupt the program and find the program counter (PC). This gives a distribution of where in the code the application spends most of its time. Early systems used timers to sample the execution, producing profiles of execution time distribution. More recent systems, such as SpeedShop [28] and DCPI [2], use counter overflow to generate sampling

interrupts. On a counter overflow, the counters are unloaded and reset. The location of the counter overflow is used to associate the events with specific locations in the code, since the probability of counter overflow is an approximation of the distribution of the events that cause the counter to be incremented. This is superior to sampling based on timers since using clock interrupts risks correlation with application behavior [28]. Sampling based on counter overflow can obtain a distribution of counter events that enables a performance tool to associate events such as cache misses or branch mispredictions with specific lines of code.

The SpeedShop suite of performance tools runs on SGI platforms and was augmented for the R10000 processor by adding support for instruction profiling [28]. Here we describe only the instruction profiling functionality of SpeedShop, which is based on the same hardware performance counters as `perfex`.

SpeedShop allows the user to specify an overflow value of the performance counters. For instance, the user can request an overflow interrupt on every hundredth L2 data cache miss. On each interrupt, the handler will record the value of the PC. The output data for SpeedShop is then a histogram of program counters that triggered event counter overflows.

This instruction profile allows the user to locate exactly where in the disassembled code certain events such as cache misses take place. Post-processing tools can often associate lines of source code with PC values, though compiler optimizations can make this somewhat ambiguous. If an application contains a few regions of code that dominate performance, then instruction profiling is the ideal tool for finding them. For instance, if long-running instructions are located in inner loops, an instruction profiler such as SpeedShop will identify this situation. It provides a very low-level view of application behavior, however, and instruction profile data is often not the correct tool for determining whether structural or algorithmic changes are necessary.

Another instruction profiler was implemented using DCPI (Digital Continuous Profiling Infrastructure) [2]. DCPI runs on Alpha processors, but the underlying data collection mechanism is quite similar to SpeedShop. DCPI was designed along with the Alpha processor, and this led to a particularly good implementation. The sampling period varies from 4096 to 65536 cycles, and the reported performance overhead is excellent — typically in the range of 1-3%. One consequence of the long sampling period, however, is that only long-running programs can be effectively profiled. Although cycle counts are used

for interrupt generation, on each interrupt the counters are initialized to a random value to eliminate correlation effects with the application.

The main difference between DCPI and SpeedShop (other than that DCPI has lower overhead and was designed to run for long periods of time), is that DCPI attempts to diagnose performance problems based on the result of the instruction profile. It does this in a post-processing stage by examining both the sample information and the application executable. It builds a control flow graph for each procedure and uses this in conjunction with the samples to estimate the frequency and CPI of each instruction. It then uses a model of the hardware to suggest possible reasons for stalls.

In short, DCPI runs with low overhead and produces an extremely detailed performance report. Even though much of the report is estimation and conjecture, it is generally accurate enough to enable performance tuning. It has the same basic advantages and disadvantages as SpeedShop: it is an excellent source of detailed information on specific performance hot spots, but it is less useful for pointing out high-level problems with application behavior.

We note that DCPI has recently been extended to perform value sampling [4], and this allows it to perform more detailed analysis of memory behavior. DCPI value sampling is discussed in more detail in Section 3.5.1.

Memory profiling

Previous to this work, there were *no* hardware implementations of memory profiling or extended memory profiling for cache-coherent DSM machines. This area will be covered in subsequent chapters.

1.4 Thesis Overview

Fast and efficient hardware performance monitors for gross miss counts and instruction profiling already exist, so this thesis focuses on memory profiling and extended memory profiling. Chapter 2 describes memory profiling in greater detail. It shows examples of how the data is used in performance tuning and describes implementation challenges. Chapters 3 and 4 introduce two new memory profilers, TrapPoint and FlashPoint, that are the basis of this work. We then conclude by comparing the two techniques and commenting on the feasibility of implementing them on future machines.

Chapter 2

Memory Profiling

Memory profiling is a useful way of studying program behavior, particularly on shared-memory multiprocessors. Due to implementation difficulties, however, memory profilers are not nearly as widely implemented nor as well-understood as gross miss counters or instruction profilers. This chapter describes how memory profiling can be used to diagnose performance problems and shows examples of using a memory profiler to tune applications. We then discuss implementation difficulties that limit the applicability of memory profilers, what implementations currently exist, and we introduce two new memory profiling techniques.

2.1 Description

A memory profiler is a tool that attributes cache misses to the data structures that caused them, is able to differentiate between local and remote misses, and performs some code localization, such as to the procedure in which the misses occur. Memory profiling is not intended to be a replacement for all other code-oriented performance monitors such as instruction profilers. Instruction profiling has shown itself to be a valuable technique for focusing a programmer's attention on the small regions of code that dominate performance. Both previous studies [16, 27] and our experience have shown, however, that having both data-oriented and code-oriented views of application performance can be useful in understanding performance. While localizing misses to an entire procedure is much

```
1:  for(i = 0; i<N ; i++) {
2:     for(j = 0; j<N; j++) {
3:        Z[i][j] = 0;
4:        for(k = 0; k<N; k++) {
5:           Z[i][j] += X[i][k] * Y[k][j];
6:        }
7:     }
8:  }
```

Figure 2.1: Naive matrix multiplication implementation

more coarse-grain than the per-line or even per-instruction categorization produced by instruction profilers, this higher level view can illuminate problems in applications that are difficult for instruction profilers to diagnose, as we discuss in this section.

2.1.1 Data Layout

Poor data layout can be a performance problem in a wide variety of programs. Programmers of DSM multiprocessors should exploit *physical locality* in addition to the the optimizations for uniprocessors. Physical locality is the property that data exists in memory near the compute processors that access it. Data structures should be organized in memory to maximize spatial, temporal, and physical locality. The absence of spatial locality leads to excessive cold misses, the absence of temporal locality leads to large numbers of capacity misses, and the absence of physical locality can lead to long remote miss times. Memory profilers are better-suited than other performance tools to identifying problems in data layout.

As a simple example, consider a naive implementation of a matrix multiplication, shown in Figure 2.1. An instruction profiler would easily determine that line 5 is responsible for majority of the execution time. Many such profilers would also be able to determine that cache misses were the dominant effect, as they surely would be with large matrices. It would be substantially more useful, however, if a performance tool could identify which matrix was causing the poor memory behavior and provide some details on the specific access patterns, thus informing the programmer of possible problems with how data is arranged in memory. This is the role of a memory profiler. For such a simple example as

Figure 2.1, an instruction profiler that reported data on machine instructions would actually be able to separate the behavior of these three matrices. If the code were parameterized to operate on many different matrices, however, data layout problems with an individual matrix (e.g., if one matrix were accidentally placed on a remote node) would be obscured. Instruction profiles can also be difficult to interpret because compiler optimizations can often move and transform code in such a way that it is difficult to map the results back to source code. Memory profilers, since they track references to data objects, do not suffer from this problem.

The best way to improve the performance of matrix multiplication is to *block* the computation for the memory hierarchy. Blocking is the operation of reorganizing a matrix computation to operate on small blocks instead of complete rows or columns with the intent of increasing temporal locality. Memory profilers are far better suited than instruction profilers for informing the programmer whether or not blocking is necessary or if it is already present, whether or not it is performing as intended.

Data layout problems can affect performance whenever certain data structures are accessed, which may or may not be localized to a few regions of code. Consider, for instance, a large two-dimensional matrix in C. Since C matrices are stored in row-major order, sequential access along a row exhibits good spatial locality, but access along a column does not. If a program, possibly ported from a column-major language such as FORTRAN, operated primarily on columns, it would experience a large number of cache misses. Whether or not an instruction profiler could identify the problem depends on the properties of the program: if matrix operations were localized to small regions of code, then the instruction profiler would be able to identify the regions, as the strength of instruction profilers is identifying small pieces of code that are the primary performance bottlenecks. A knowledgeable programmer might then intuit that the problem was the layout of the matrix. If there are many column-based operations spread throughout the program, however, then there are no specific regions of code that serve as bottlenecks. Rather, a data object is the bottleneck, as any column-based access to the object will suffer from poor performance. A memory profiler would be able to identify that accesses to the matrix exhibit a large number of cache misses wherever they occur. This would direct the programmer to either transpose the matrix in memory or adjust the algorithm to operate on rows.

2.1.2 Data placement

One common performance problem that we have observed with many parallel applications, including highly-optimized codes, is that *data placement* was done poorly. Data placement is the NUMA-specific operation of ensuring that data structures reside in memory near the processors that operate on it to exploit physical locality. Data placement is the domain of the operating system, since the OS controls the mapping from virtual to physical addresses. Nevertheless, NUMA systems generally provide the programmer with directives that can influence the manner in which the operating system places memory. To obtain high performance, it is *imperative* that programmers either use these directives with their critical data structures or write code that takes advantage of the operating system's specific placement policies.

If the placement is done poorly, then although accesses to a data structure might exhibit good spatial and temporal locality, misses will have high latency due to the long remote miss times and possibly memory or network contention. Memory profilers easily identify the problem, since they can show the fraction of remote misses for each data structure. The programmer can then examine how the data structure is allocated and placed and quickly fix the problem if the algorithm allows.

Data placement problems are far more difficult to diagnose with instruction profiling tools. If the tools sample on processor-based cache miss counters, they will have no information as to which misses are associated with local or remote addresses, and therefore they will not present any data on physical locality. If they instead sample based on cycle count (i.e., time), there will be a disproportional number of samples for accesses to remote addresses, since those addresses will take more time than local accesses. Still, determining that the reason the tool reports the misses it does are because of long remote miss times and not other effects is a very large leap for the programmer.

2.1.3 False Sharing

Another performance problem unique to shared-memory multiprocessors is that of *coherence misses*. These are cache misses that are caused not by capacity or conflict problems, but by actions of remote processors. For instance, when one processor writes to a shared line, it first must invalidate the shared copy in the caches of other processors. The next time

those other processors read that cache line, they will miss. The most pathological case of coherence misses is the phenomenon of *false sharing*, where coherence misses are caused by unrelated data structures sharing a cache line and not by any real communication of data.

Consider the worst-case example: if one variable is read-shared among many processors, then as long as it is read often, it will be cached in the shared state by all readers. The fact that it is remote to most of the processors will be of little relevance. If another variable is often written by a single processor, then it will likewise maintain an exclusive copy in its cache and cause no memory traffic. If these two variables happen to reside on the same cache line, however, there will be a substantial performance degradation. Each time the writer wishes to write the cache line, it must obtain an exclusive copy, which means sending invalidations to all sharers (and waiting for their acknowledgments, in the case of sequential consistency). Each reader will experience not only the overhead associated with the invalidations, but it will also take a cache miss on each access that follows a write. Combined, these two effects create a large amount of memory traffic. Worse yet, the traffic is completely unnecessary since there is no actual communication. An instruction profiler would show that the line which does the read (and perhaps also the write) often misses. A memory profiler would show write misses to a supposedly read-only variable — a hallmark of false sharing.

2.2 Examples

We have implemented two memory profilers, introduced in Section 2.5 and described in detail in subsequent chapters. In this section, we show examples of how our memory profilers were used to diagnose and fix performance problems in real applications. All runs were done on the Stanford FLASH machine [11], described in Section 3.1.

2.2.1 Blocking in FFT and Radix

We first investigate two programs from the SPLASH-2 benchmark suite [26], FFT and Radix. We originally profiled these codes only to verify the accuracy of our memory profilers, since we “knew” what the results would be. After all, these are benchmark codes

that have been thoroughly studied by many researchers. Surprisingly, we found that both of these programs exhibited rather severe and somewhat similar performance problems.

FFT

FFT is a C program that performs a Fast Fourier Transform on a matrix of complex doubles. The code is parallelized such that each processor is allocated a contiguous section of rows in the large matrix. The algorithm requires operations on both rows and columns, but column-wise access would be disastrous for performance since traversing a column lacks both spatial and physical locality. Therefore, FFT is structured such that it operates on local rows, then transposes the matrix to turn columns (most of which are remote) into rows so it can perform the column-wise operations in local memory. This transpose phase is responsible for *all* the communication and most of the cache misses in FFT.

To the first order, optimizing the performance of FFT is optimizing the performance of matrix transposition. Transposition consists of reading columns of a source matrix and writing rows of a destination matrix. Spatial locality is perfect for the writes: there will be a single miss when the first word of a cache line is written, but subsequent writes will fall in the same cache line and will therefore hit. Reading the columns, however, is more problematic. Since each value in the column occupies a separate cache line, there is no spatial locality reading the first column. When the *second* column is read, however, the reads could potentially hit in the cache, since data from the second column resides on the same cache lines as data from the first column. Unfortunately, if the size of a column is larger than the number of lines in the cache, as it would be for a large matrix, these lines will no longer be present in the cache when the second column is read.

The performance of the transposition can be improved substantially by blocking. Instead of reading an entire column and writing an entire row of an $N \times N$ matrix, we read columns and write rows of a $b \times b$ blocks, where b is chosen to be smaller than the number of lines in the cache. That the transpose phase in FFT should be blocked is well-known and well-documented [26]. It has also been shown that the block size should be chosen so that a block fits in the smallest level of the memory hierarchy, assumed to be the first level (L1) data cache. Our memory profilers have shown this assumption to be incorrect.

When FFT is blocked for the first level data cache, memory profiling produces results shown in Figure 2.2. The results shown are only for the procedure `Transpose`, since

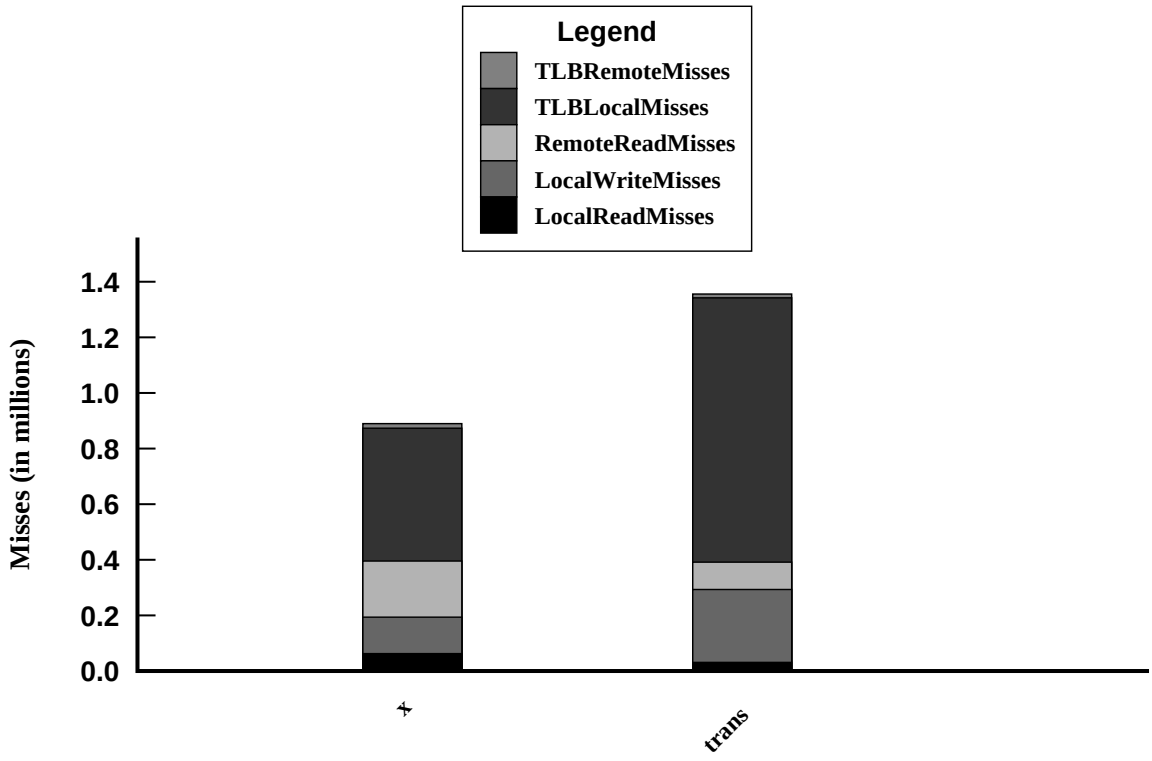

Figure 2.2: Transpose phase of a four-processor FFT blocked for first level data cache.

it is responsible for the most of the misses. FFT allocates two matrices, called `x` and `trans`. It calls `Transpose` three times, each time transposing one matrix into the other, and the results shown are aggregated across all three runs of `Transpose`. We see that, as expected, there are reads to both local and remote addresses, and writes are only to local addresses. There are, however, a large number of TLB misses. Unlike an instruction profiler, the memory profiler is able to indicate that these TLB misses occur to the matrices `x` and `trans` during the transpose phase, and this saves the programmer from having to investigate TLB misses coming from any other source, such as from system interference.

The *TLB*, or Translation Look-aside Buffer, is a small, fully-associative cache of the page table. It needs to be accessed on every user mode memory reference to translate the virtual address into a physical address. When the requisite translation does not exist in the TLB, the processor takes a TLB miss, an exception which loads the translation from the page table. This is a costly operation, taking on the order of 60 cycles on a MIPS

R10000, which is comparable in time to a local secondary cache miss. While not part of the cache hierarchy proper, the performance of the TLB can be quite significant, and TLB performance is dictated by the same factors that dictate cache performance: spatial and temporal locality. The TLB on the R10000 is 64 entries, with each entry mapping two consecutive 16 kB pages. This means that the miss rate of the TLB will be similar to that of a fully associative cache with 64 entries and a 32 kB line size, with the differences stemming from the fact the the kernel must occasionally flush TLB entries.

The version of FFT shown in Figure 2.2 has a block size, b, of 256 since this block size is the maximum allowable such that all cache lines from the first column will still be resident in the L1 cache when the second column is read. Note, however, that 256 is much larger than the number of TLB entries, so the TLB does not benefit from temporal locality. If the matrix is large (i.e., if the rows are larger than 32 kB), then there will be no spatial locality as well, so *every* read from an element in the source matrix will cause a TLB miss. This is the behavior shown in Figure 2.2.

The solution to this dilemma is to block FFT for the TLB, rather than the L1 cache. Using a block size smaller than 64 dramatically reduces the number of TLB misses. Reducing the block size nearly doubles the performance of the transposition and improves the overall performance of FFT by 14% on a uniprocessor and 16% on four processors with a million-point source matrix [6]. It surprised us that there was so much room for improvement on code as heavily studied as FFT. The reason that improper blocking parameters have been in use for *years* is that the original studies that recommended blocking FFT for the L1 cache [26] were done on a simulator that did not model the TLB [6]. The block size had not been examined since migrating the code to hardware. Memory profiling, however, was able to quickly identify the TLB as the primary performance bottleneck, and this enabled us to fix the blocking parameters.

Radix

Radix is an integer sorting program. It sorts by one "digit" at a time, with a digit size of $\log_2 r$ bits, where r is a parameter known as the *radix*. Each processor owns a section of the unsorted array. It begins by histogramming its entries by the digit currently being sorted, starting with the least significant. All processors then communicate to combine their local histograms into a global histogram. Once this global histogram is built, each processor

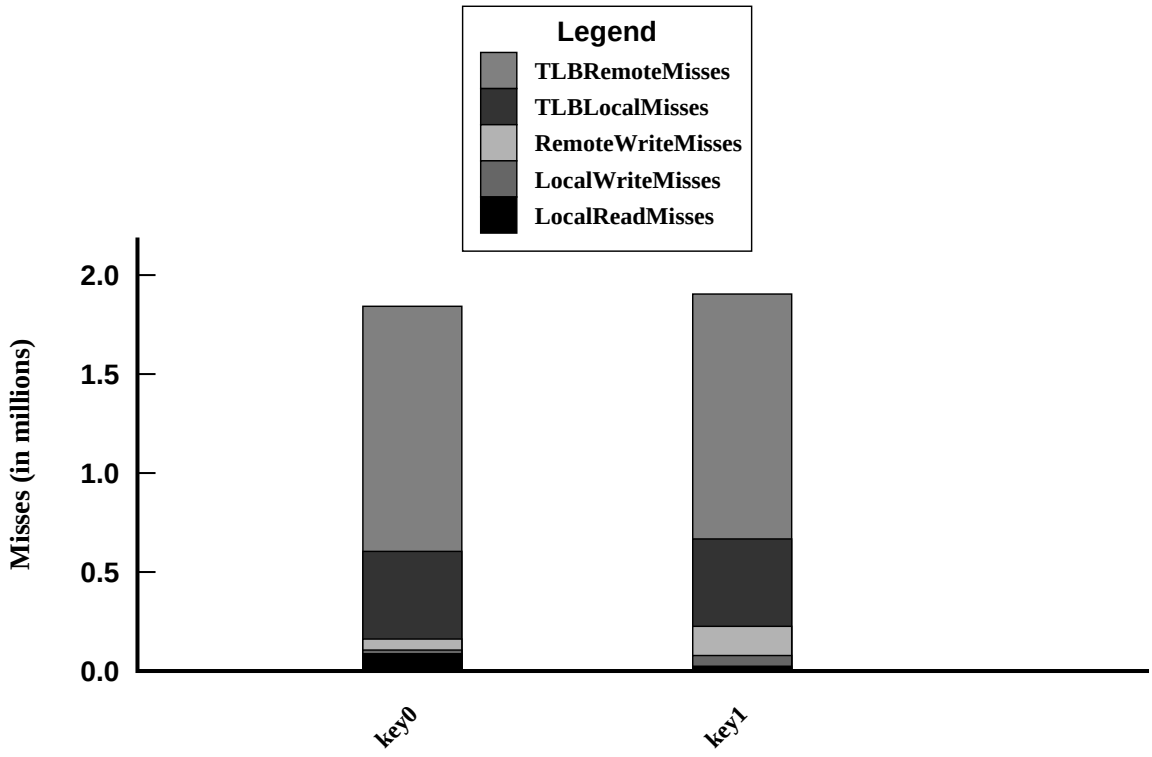

Figure 2.3: Permutation phase of a four-processor Radix sort with a radix of 256.

is able to calculate positions in the destination array for each value of the digit. In other words, the processor is able to compute: "I can write all of my 0's starting at index A, all of my 1's starting at index B, all of my 2's starting at index C, etc." In the subsequent permutation phase, each processor copies each of its entries into the destination array. The process is then repeated with the other digits until the entire array is sorted.

The vast majority of the memory traffic occurs during the permutation phase with writes to primarily remote addresses. A memory profile of this phase is shown in Figure 2.3. Radix holds its data in two arrays, named `key0` and `key1`. It starts with all the data in `key0`, sorts it by the first digit into `key1`, sorts by the next digit back into `key0`, and so on. Again, we see the dominance of TLB misses.

The size of the radix, 256, was chosen to be large to reduce overhead associated with forming histograms. This means, however, that during the permutation phase, each processor writes into 256 different locations of the destination. There is not sufficient temporal

locality to maintain acceptable TLB performance. If the destination array is large and these 256 locations are separated by more that 32 kB, there will not be any spatial locality, and nearly every write will cause a TLB miss, as shown in Figure 2.3. The solution is to use a radix smaller than the TLB, such as 32. This improves the performance of Radix by 31% on a uniprocessor and 34% on four processors with a two million point input array [6].

As with FFT, Radix has traditionally been run with incorrect parameters, at least for the R10000, because the original studies [26] that recommended parameter values were done with a simulator that does not model the TLB [6]. While this problem could have been found with existing tools, the simple fact is that it wasn't. A memory profiler made the pathological TLB behavior glaringly obvious, and once found, it was easily fixed.

2.2.2 Data Migration in IRIX

In addition to the problems mentioned above, our SPLASH-2 studies led to the discovery of another performance anomaly. While studying FFT, we displayed the graph from Figure 2.2 broken down by individual processor and saw that the total number of cache misses for CPU 0 was substantially less than the number of misses on the other processors. This is confusing because the transposition algorithm is completely symmetric across processors. Since `Transpose` is so regular, it is trivial to compute the expected number of cache misses. We found that CPU 0 had the expected number of misses, and the other processors had more misses than we could explain.

The memory profiler was able to point out the problem, and upon further investigation, we discovered that there was a bug in the operating system. FLASH runs a modified version of IRIX, and at the time we were running version 6.4. The way FFT initializes its data is that the master thread, which runs on CPU 0, allocates and initializes all shared data structures. IRIX has a first-touch placement policy, meaning that it attempts to allocate physical memory on the memory local to the cpu that touches it first. This means that *all* data is initially placed on node 0. The program then makes a call to the ANL macro `SYS_PLACE_RANGE` to move regions of data to other nodes. It subsequently creates its parallel threads and starts the computation.

The `SYS_PLACE_RANGE` macro expands into system calls that invoke an explicit data migration. This migration works in the sense that the data actually moves to the intended node. The bug, however, was related to *cache coloring*. The processors each have a 2 MB,

2-way set-associative second-level cache with a line size of 128 bytes. This means that there are 8192 cache lines in each way of the cache ($\frac{2\ MB}{2\ ways \times 128\ bytes} = 8192$), so the lowest 13 address bits are used to address the cache. The lowest 7 bits ($6:0$) are offsets within a cache line, so the 13 bits $19:7$ are used to index the cache. The default page size on IRIX is 16 kB, which implies that the lower 14 bits of the virtual address and physical address must be identical. The OS, however, has complete discretion in assigning bits $39:14$ of the physical address (these bits are known as the physical page number). Note that bits $19:14$ overlap in that they are part of both the physical page number and the cache index. These bits are known as the *cache color*. The cache is physically indexed, so the value of these cache color bits can affect the cache performance. For instance, if the user allocates a 1 MB array as a contiguous virtual address range, he may believe that there will be no internal cache conflicts. This may or may not be the case, however, as the OS could easily give multiple pages the same cache color, causing them to conflict with each other in the cache. The OS has several heuristics for cache coloring, and it is usually successful in avoiding unnecessary conflicts.

When data is migrated, the code checks the color bits in the source range and attempts to allocate a page on the destination node that has the same color. This is exactly what the programmer would want, assuming that the algorithms which originally colored the pages were working correctly. The bug was that when it checked the color bits on the source node, it mistakenly read the *level 1* cache color. The L1 cache is substantially smaller: 32 kB, 2-way set-associative, and there is *no* overlap between physical page number and level 1 cache index bits. The macro to read the cache color would therefore always return zero! The migration code would then request that all pages on the destination node have a level 2 cache color of zero. The data would migrate as intended, but all migrated ranges would have color bits of zero, so they would all conflict with each other in the level 2 cache. This is the cause of the additional cache misses on all non-zero nodes. Since CPU 0 originally had all data allocated to it, it did not need to migrate its own data and was therefore unaffected by the bug.

This bug was reported to SGI and fixed in IRIX 6.5.

2.2.3 Microbenchmarks and Cache Replacement Policy

When evaluating the performance of a machine's memory system, it is common practice to write suites of microbenchmarks to measure various memory latencies. An example of a microbenchmark suite designed to be portable is lmbench [18]. Since the timing granularity is often on the same order as a cache miss latency, cache misses can only be reliably timed by forcing the processor to take a large, *known* number of cache misses and dividing the total time by the number of misses. Unlike most programs, knowing the precise number of cache misses is essential to the proper operation of these microbenchmarks. This requires putting the cache in a known state before the test begins, typically by flushing a data array from the caches. To effect a cache flush, many processors, including the MIPS R10000, have privileged instructions for directly accessing the cache, and the operating system provides system calls that use them. These calls are quite system-specific, however, so programs that were designed to be portable, such as lmbench, do not use them. Rather, they assume that striding through a large array (one with more cache lines than the cache) will flush the cache.

The MIPS R10000 has a two-way set-associative L2 cache. This means that when a line is to be added to the cache, the hardware has a choice of two ways from which to evict a line. The R10000 is documented [19] to have a least recently used, or LRU, replacement policy so striding through an array the size of the cache *should* completely flush the cache. Memory profiling shows that it does not. It shows a lower than expected number of misses in tests run after the cache flush, indicating that some of the data that was present in the cache before the flush was still present afterwards. After noticing this anomalous behavior, we spoke to R10000 architects, who confirmed that the cache replacement policy on the R10000 is actually random, not LRU, in contradiction with its documentation.

While most programs are insensitive to the cache replacement policy, some microbenchmarks such as lmbench are adversely affected. With a random replacement policy, the cache cannot be reliably flushed by striding through a large array (though a flush can be asymptotically approached if the array is much larger than the cache). This is the primary reason that system-specific tools provide more accurate timings than lmbench on the R10000. Finding this problem requires a profiler that provides exact, not sampled, miss counts. Though this behavior could be observed with traditional miss counters, such counters always have some error introduced by the system calls needed to read the counters. The user can never

be sure that the counts are exact, and this makes it more difficult to make strong claims such as "some data *must* still be in the cache". However, a memory profiler can say with certainty which accesses were made to the various user data structures, allowing the user to put more faith in the exact numbers reported by the tool.

2.2.4 Cache Transfers in AMMP

To measure the performance of OpenMP on various hardware platforms the SPEC High-Performance Group published a suite of benchmarks called SPEC OMP 2001 [24]. One C program included in this suite is called AMMP (Another Molecular Mechanics Program). The SPEC benchmark version of AMMP is a simplified version of a real molecular mechanics program. Using extended memory profiling, we have identified its central bottlenecks.

AMMP begins execution by reading a list of atoms from a file. It constructs an `ATOM` structure for each atom. The main computation of the program iterates over the atoms. For each one, the program computes its effects on all other atoms. The parallelization has to be done carefully, since there is much more work for atoms early in the list than for atoms later in the list because when each atom is visited, interactions with earlier atoms have already been computed. Properly load-balancing the application therefore requires a dynamic scheduling algorithm (either *dynamic* or *guided* in OpenMP parlance).

Preliminary memory profiling results revealed that the program does all of its work after initialization in one parallel section of one procedure. All cache misses were to the `ATOM` data structures. Figure 2.4 shows the extended memory profiling results for the `ATOM` data structures in the main parallel section. This figure reveals several performance problems with AMMP.

Load Balancing

The most obvious problem from the graph is that CPU 0 takes nearly 80% more misses than the other processors. While a difference in the number of cache misses does not necessarily indicate a load balancing problem (execution time, not number of misses, is the true measure of load balancing), a disparity that large is nearly always a problem. In this case, the problem is most likely caused by the OpenMP scheduling algorithm that allocates iterations to threads. Assigning iterations to threads when the iterations have differing

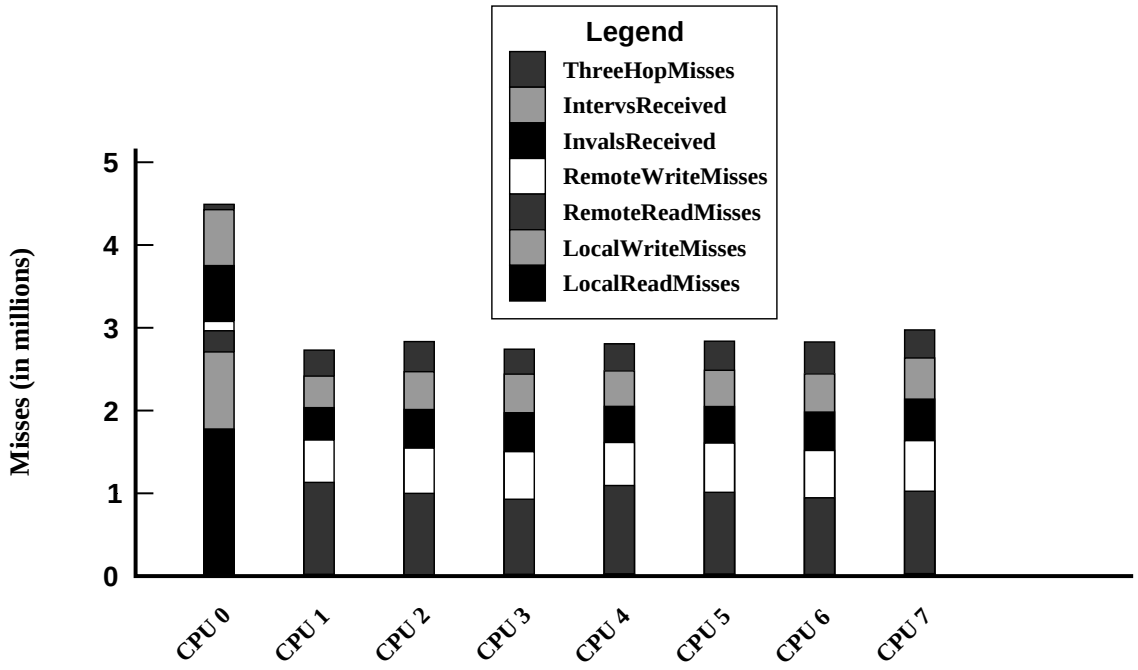

Figure 2.4: Extended memory profile of the main parallel section of AMMP.

amounts of work is a difficult problem in general, so it is not surprising that the OpenMP library does a suboptimal job.

Data Placement

Another problem clearly evident from Figure 2.4 is that CPU 0 takes mostly local cache misses while all other CPUs take *only* remote cache misses. This indicates that the `ATOM` structures were placed on CPU 0. This is clearly undesirable since it causes a memory bottleneck there. AMMP, in fact, runs faster with the OpenMP placement directive `ROUND_ROBIN`, which indicates that the operating system should try to spread data evenly across nodes and not try to allocate it on the node where it is first used as per the default OpenMP (and IRIX) placement directive of `FIRST_TOUCH`.

The reason that `FIRST_TOUCH` performs so badly for AMMP is that all atoms are read serially from a file by the master thread and allocated individually by a call to `malloc`. The fact it is done by a single thread means that data will tend to be placed on a single

node. Some attempts were made to mitigate this by placing a small parallel section in the code that allocates an `ATOM` in an attempt to "trick" the operating system into placing the `ATOM` on a different node. This technique is sometimes successful, but in this case it was not. OpenMP was not designed to give the user explicit control over data placement, and working around the OpenMP library is often not possible without resorting to OS-specific system calls, which are not allowed in portable benchmark codes such as AMMP.

In addition to the limitations of OpenMP, the placement suffers from a performance problem common to many applications we have examined. When trying to take advantage of the system's first-touch behavior, it is important to know exactly where a page of the virtual address space is touched *first*. Using `malloc` on small regions of memory, such as an `ATOM` (roughly 2 kB), the way AMMP does, gives the user poor control over the virtual address space and should be avoided. The data returned by `malloc` may or may not be on the same page that has been returned by a previous call to `malloc` and subsequently placed. If the page has already been placed, then any present action to manipulate its placement will not succeed.

Even more insidious is that if the program *ever* calls `free`, then it is entirely possible that the pointers returned by subsequent calls to `malloc` will have been used and placed before, and this will foil any attempt to take advantage of first-touch placement. We have found that there are two reliable ways under IRIX and OpenMP to place memory on desired nodes. The first is to explicitly use IRIX placement and perhaps even migration directives. The more portable (and slightly less reliable) method is to `malloc` a large range of page-aligned memory and immediately touch it (`bzero` does exactly the right thing) on the desired node. If there is a possibility that `free` has been called, then in System V variants such as IRIX, you can replace the call to `malloc` with an `mmap` of `/dev/zero`. It has much the same effect as `malloc`, but it will always return a new section of the virtual address space.

Sharing Patterns

The final problem we note in Figure 2.4 is the sharing pattern. The extended memory profile data shows numerous invalidations, interventions, and three-hop misses. These are not cache misses themselves, but they provide more detail about what sort traffic was caused by the cache misses. Since invalidations, interventions, and three-hop misses are coherence

protocol events, not simple miss counts, extended memory profiling techniques are required to collect this information. These events come about in AMMP because the program is not careful about managing its sharing.

For example, if CPU 1 wants to write an `ATOM`, it will take a remote write miss (remote since all `ATOM`s are allocated on node 0). It will then have an exclusive copy of the cache line in its cache, which it can write at will. The cache of CPU 1 will therefore have the only valid copy of that cache line in the system. If CPU 2 then decides to read that cache line, it sends a message to node 0 in attempt to find the line. Node 0 will redirect the request to node 1, causing a three-hop miss (so named since three nodes are involved in the critical path). The node controller on node 1 must retrieve the line from the processor's cache. The operation of the node controller forcibly taking a cache line away from a processor is called an *intervention*. A simple load on CPU 2 will have caused a remote read miss on CPU 2, an increment of the three-hop miss counter on CPU 2, and an intervention to be received on CPU 1.

Note that since the total of interventions, invalidations, and three-hop misses for processors 1 through 7 are nearly equal to the total of cache misses, we find that a large fraction of misses are serviced by remote *caches*, not remote memories. This leads to poor performance, as reading from a remote cache has roughly double the latency of reading from a remote memory. The intervention itself also interferes with computation on the target processor, thus causing more overhead. Cache-to-cache transfers can be so slow, in fact, that some programs might actually benefit from *smaller* caches!

The performance can be improved slightly by adding padding the `ATOM` such that the fields that are often written reside on different cache lines than those that are only read. Unfortunately, most of the reads occur to fields that are often written, so this improvement is small. The parallel section of AMMP would have to be radically reorganized to alleviate the sharing problem. We note, however, that Figure 2.4 was generated from a short run of AMMP. In a problem with more `ATOM`s, this effect is less pronounced since increasing the problem size has a similar effect to shrinking the caches.

It is important to realize that although a simple memory profiler could have identified the load balancing and data placement problems with AMMP, extended memory profiling was necessary to identify the problematic sharing behavior.

2.2.5 Placement and False Sharing in VirtualMesh

We used an extended memory profiler to improve the performance of VirtualMesh, a FORTRAN code that uses OpenMP compiler directives to express its parallelism. This is a scientific code that solves the Navier-Stokes equations, which describe the motion of a fluid. The solution is advanced in time by a low-storage, third-order Runge-Kutta method. The equations are solved by a fractional step method and an immersed-boundary method is used to handle complex geometries. The particular problem we used for tuning computes the air flow over a car. VirtualMesh is not a benchmark code. Rather, it is actively used for mechanical engineering research. The code is mature in that it has been used and studied for years, and it has no clear primary performance bottlenecks. Typical data runs last for weeks, so even small percentage performance improvements can save a substantial amount of time. For profiling runs, we ran the full problem but only for a small number of time steps, since our goal is simply to run long enough that the profiling run is representative of a full run.

The primary unknowns are three velocity components and pressure, represented as three-dimensional arrays. Each array contains several million elements. Preliminary memory profiling runs indicated that one performance problem was caused by a large number of remote misses. Profiling showed that the velocity and pressure arrays were allocated completely on node 0. The default placement policy for OpenMP is to place data on the node that first touches it. Many OpenMP programs share this placement problem because they often initialize their data in the master thread (for instance, by reading initial values from a file) and data is consequently placed on the node where the master thread runs. This issue was described in more detail in the previous section. For VirtualMesh, we were able to force proper placement by zeroing the arrays in a parallel section before they are touched by the master thread. Since FORTRAN 77 lacks dynamic memory allocation, the problems with `malloc` discussed in Section 2.2.4 do not apply. Placing data correctly improves the performance of VirtualMesh by roughly 15%, which corresponds to several days of run time with typical data runs.

After fixing placement problems, a profile of memory traffic by procedure was computed and shown in Figure 2.5. Note that there are many more procedures in the program, but none with as many misses as the ones shown — they were simply eliminated for the

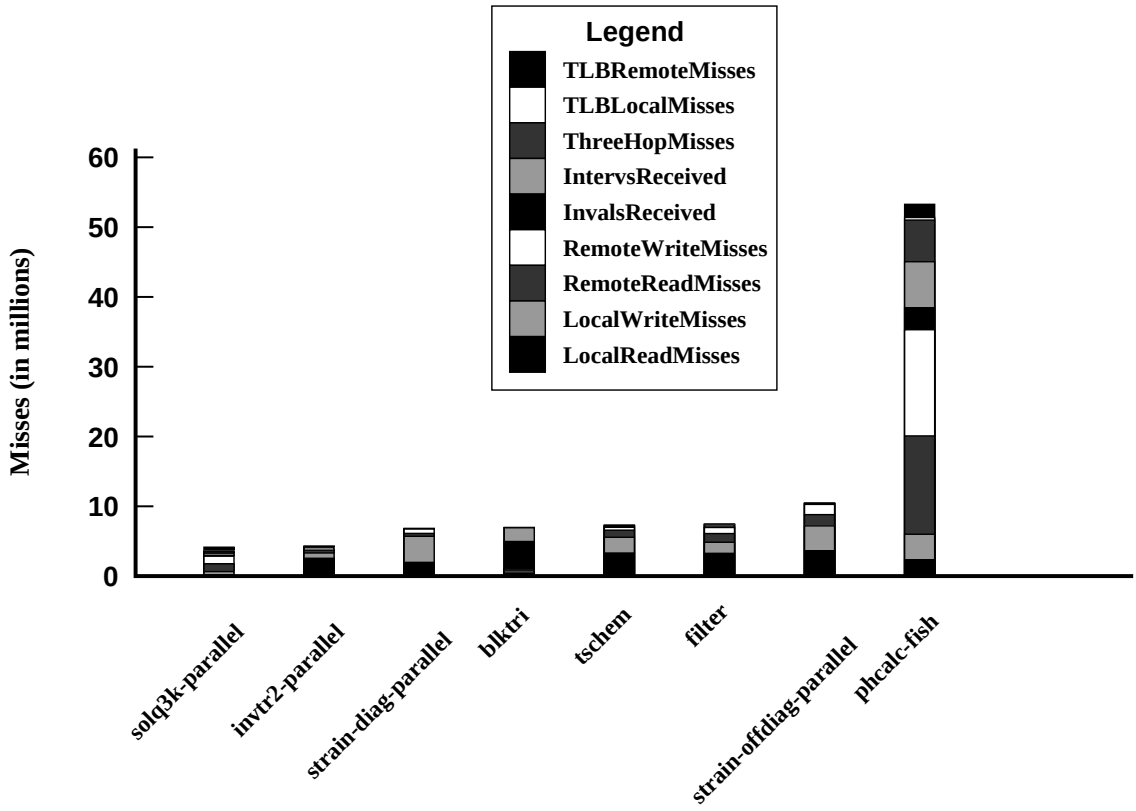

Figure 2.5: Extended memory profile of the procedures in VirtualMesh on 8 processors.

sake of clarity. Memory profiling clearly shows that the `phcalc-fish` procedure is responsible for a large amount of cache misses. The extended memory profile also shows that many of these misses cause interventions and three-hop misses. This behavior is indicative of multiple processors writing the same cache line, which is a performance problem and is best avoided if at all possible.

This procedure is instrumental in solving the Poisson equation, which is a critical part of the code in terms of performance. Solving the Poisson equation involves a one-dimensional FFT, solving equations in the remaining two dimensions, and transforming the results back. The `phcalc-fish` procedure is used to solve the two-dimensional problems in the transformed space. When we use the extended memory profile to examine the data structures in `phcalc-fish`, we see in Figure 2.6 that two data structures `qcap` and `dph` are responsible for the misses.

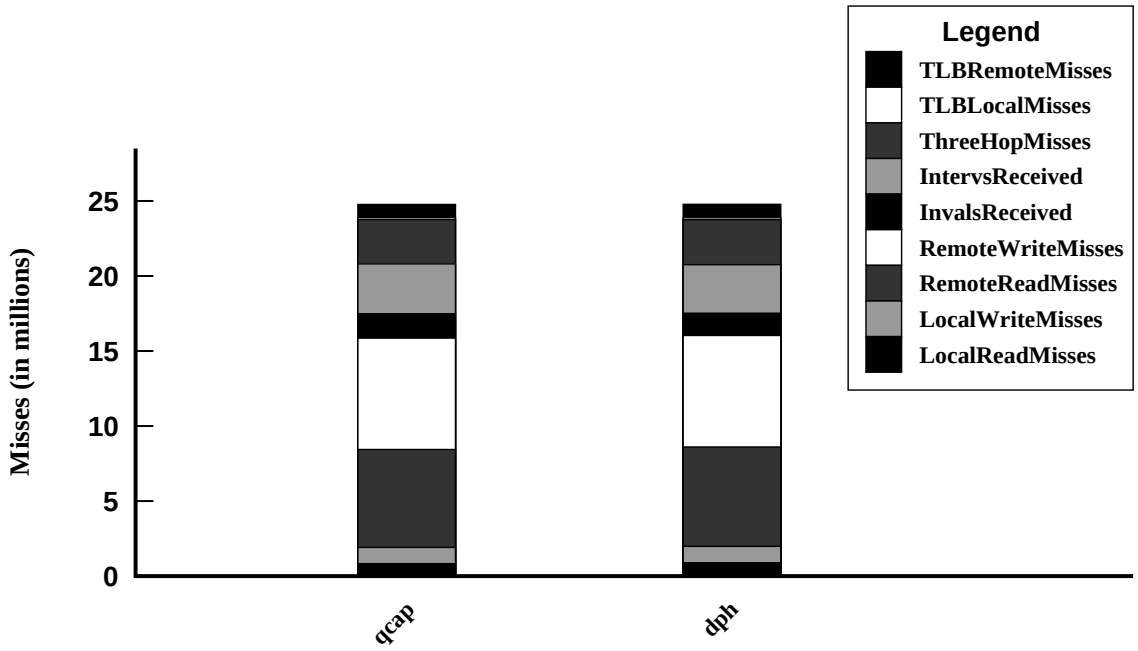

Figure 2.6: Extended memory profile of the data structures in the `phcalc-fish` procedure of VirtualMesh.

To understand the poor memory behavior, we examine excerpts from the code, shown in Figure 2.7. The OMP statement is a directive to the compiler to split iterations of the `do i=1,n1mh` loop across the threads. The parallelization is done this way to achieve acceptable performance of the FFT. Successive values of `qcap` and `dph` in the `i` dimension are in consecutive locations of memory, so an FFT in the `i` direction will see good spatial locality. The matrices are rather small in the `i` direction (i.e., `n1mh` is 64 with this particular problem), but each iteration can be lengthy, so this is an effective way to parallelize the code.

From Figures 2.5 and 2.6, we saw behavior that indicated several processors were writing the same cache line in the `qcap` and `dph` arrays. The elements in these arrays are 8-byte double-precision floating point numbers, and since the hardware cache line size is 128 bytes, there are 16 elements on a cache line. FORTRAN stores multi-dimensional arrays in column-major format, so for a given value of `j` and `k`, there are 16 consecutive values of `i` that reside on the same cache line. Iterating over `i`, `j`, then `k` results in strided

```
FFT in i direction

!$OMP PARALLEL DO
do i=1,n1mh


----> phcalc-fish begins here

code which reads qcap, solves problem in temp array yfisl

  do j=1,n2m
    do k=1,n3m
      qcap(i,j,k) = yfisl(k,j)
    end do
  end do

code which reads dph, solves problem in temp array yfisl

  do j=1,n2m
    do k=1,n3m
      dph(i,j,k) = yfisl(k,j)
    end do
  end do

---->phcalc-fish ends here

end do
!$OMP PARALLEL END DO

Inverse FFT in i direction
```

Figure 2.7: Code excerpts from VirtualMesh

memory access, exhibits no spatial locality, and is best avoided. Indeed, this is the only region of VirtualMesh that accesses the arrays this way, but it is necessary for acceptable performance of the FFT computation, as mentioned above.

Worse even than the complete lack of spatial locality is the fact that the parallelization is done across iterations of `i`, so different processors can be writing parts of the same cache line! This is the cause of the three-hop misses and interventions shown in Figures 2.5 and 2.6. This problem could be solved by requiring that OpenMP place a consecutive block of 16 iterations (or a multiple of 16) on a single thread. Unfortunately, since there are only 64 iterations of `i`, this only leaves enough work for 4 processors. The small number of iterations is *not* a consequence of scaling the problem down for profiling. It has to do with the geometry being modelled. The only modification we made to the program for profiling was reducing the number of time steps, which is independent of the number of points in the `i` direction. Scaling the problem past 4 processors (the profiled runs were done on 8 processors) requires a different solution.

We modified the code so that instead of writing to `qcap` and `dph` in `phcalc-fish`, we write into new matrices, which are transposed versions of `qcap` and `dph`. Writing into the transposed arrays exhibits perfect spatial locality and sees no interference among the processors. Of course, this introduces the additional step of having to transpose data back into `qcap` and `dph` after `phcalc-fish` has completed. This transposition is blocked for the caches and TLB, just as in the FFT code in Section 2.2.1, so it also sees spatial and temporal locality. In other words, writing the transposed arrays should see one miss per cache line, and the transposition itself should see one cache miss per cache line. Even though there is overhead in doing an additional transpose step, this should increase performance over the base case, which has one cache miss for each write (due to poor locality) and where many of those misses are slow, three-hop intervention misses (due to false sharing).

Using transposed arrays eliminates the write misses to `qcap` and `dph`, but there are still a substantial number of read misses to these structures. We made the small optimization of adding some explicit prefetches, and this slightly improved the performance of the code.

After adding the transposition and prefetch code, the per-procedure extended profile is shown in Figure 2.8. There are clearly far fewer misses in `phcalc-fish` than were shown in Figure 2.5, and although the newly-added procedure `newtranspose` contains

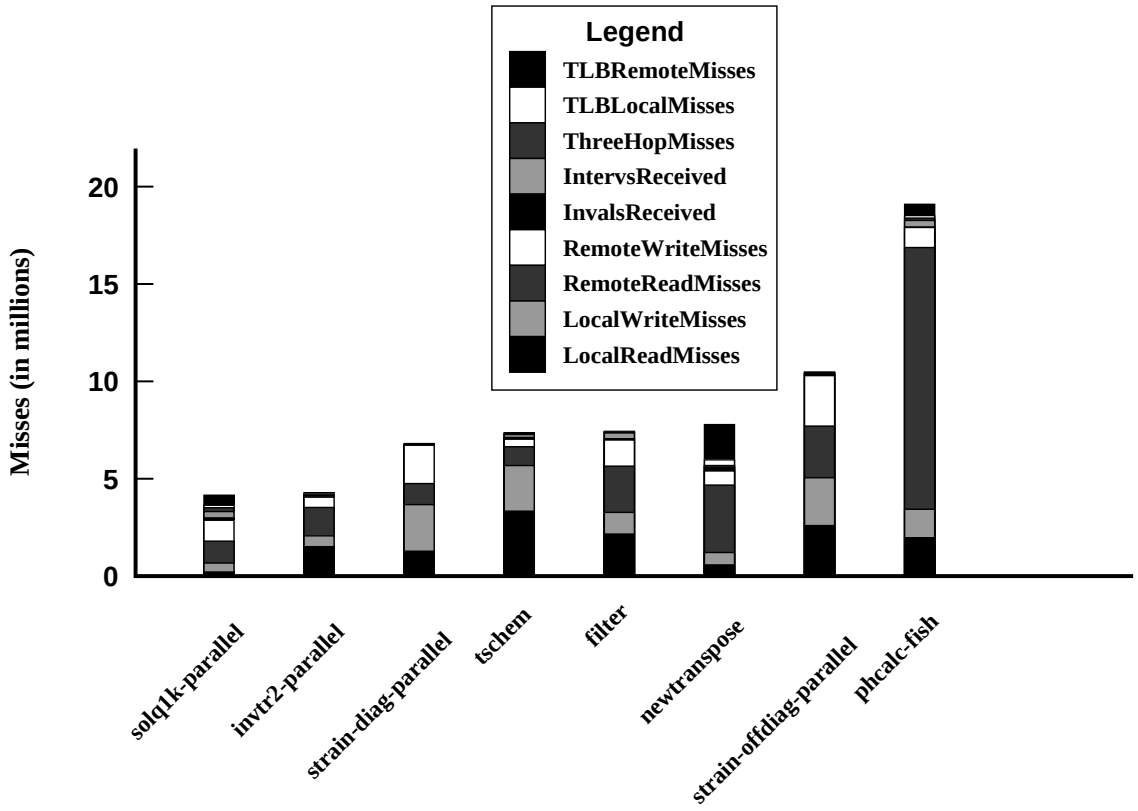

Figure 2.8: Extended memory profile of the procedures in VirtualMesh on 8 processors after a transpose phase has been added.

misses, overall there are 11.7 million fewer misses than before. This represents a 9% reduction in the total number of misses in VirtualMesh! In addition, there is no false sharing, so there are only a negligible number of interventions and three-hop misses. This version of code runs 15% faster than the version without the extra transposition. The combined performance gain from improving data placement and eliminating false sharing is 28%, which translates into roughly a week of run time for typical data runs.

2.3 Implementation Issues

Since both we and previous researchers [16, 27] have found memory profiling to be so useful, it may seem surprising that memory profilers are not more common. The reason is that

it is difficult to perform memory profiling efficiently. We view overheads less than 20% of the execution time as desirable for a performance monitor. Tools with this magnitude of overhead can be used even on applications with long run times. For shorter programs, 20% may not even be noticeable. While we have no doubt that heavier weight performance monitors have their uses, they suffer from several drawbacks: they are typically unable to handle large programs in a reasonable (to the programmer) amount of time, the high overhead prevents them from being used in all but the most dire performance debugging projects, and the high-overhead performance monitors may seriously distort the performance they are trying to monitor. Currently, no memory profiler for hardware cache coherent distributed shared memory multiprocessors exists that even approaches this level of performance. In this section, we discuss the implementation issues that have prevented memory profiling from being implemented with the requisite efficiency.

One straightforward method of memory profiling would be to instrument every cache miss in software. Unfortunately, this cannot be done since the programming model hides caching. Determining which memory accesses will miss in the cache cannot be done until run time, so any software instrumentation technique must instrument *every* memory access. The instrumentation would make calls into a memory system simulator, which would determine which accesses miss, and profile them accordingly. This is the technique used by MemSpy [16], which is discussed in the next section. Neither MemSpy nor any other software instrumentation technique, however, is capable of achieving the desired level of performance. Memory accesses are so common that instrumenting each access by even a single instruction would produce a system with borderline overhead. Of course, memory system simulators will take far more than a single cycle per access, so overheads of 50 times the execution time or more for this sort of technique are not uncommon.

Thus, high performance memory profiling requires that some hardware support exist to trigger monitoring only on cache misses. A sampling approach using counter overflow works well for instruction profiling. Such an approach cannot, however, be used to perform accurate memory profiling. Instead there are two possible techniques for obtaining the data needed to perform memory profiling. The first technique is to trap into a performance monitor on every cache miss or on a fraction of misses. Since the miss address and the breakdown of application data structures is not generally available to hardware performance monitors, obtaining the miss address requires generating an interrupt on each cache miss.

We examine a memory profiler that generates interrupts on cache misses in the next chapter.

Another approach requires the ability to monitor every cache miss on the fly and collect whatever information is required to construct a memory profile. This approach is the other method we examine for generating measurements for memory profiling. In particular, we make use of the programmable nature of MAGIC, the memory and coherence controller in the FLASH multiprocessor [11], to monitor every second-level (L2) cache miss generated by the processor. We restrict our attention to L2 cache misses because the cost of an L2 miss is substantially larger than an first-level (L1) cache miss. Many modern processors can often effectively hide the latency of an L1 miss, but this can rarely be done with L2 misses due to their high latency. Also, in DSM machines, coherence is maintained among the L2 caches (because of inclusion, this also maintains coherence in the L1 caches), and this makes the L2 cache a more useful level at which to monitor the system.

Performing extended memory profiling requires that we be able to instrument a multiprocessor at several points. In particular, capturing information such as the frequency of three-hop coherence transactions, requires that we instrument a memory reference at multiple points: when it is generated, when the home directory for the request is accessed, and when the result is finally returned. It is not possible to perform this type of performance monitoring with either sampling or interrupt-driven approaches since these only capture the initiation of a memory reference. The only way to perform extended memory profiling is by using built-in support in the memory controller. DASH [15], for instance, uses trace buffers in the memory controller, while an extended memory profiler that runs on the FLASH system (see Chapter 4) relies on the programmability of MAGIC to instrument a memory reference at multiple points. The primary disadvantage of the DASH approach is that the limited space for storing reference information means that the processor must periodically interrupt its execution and unload the trace buffers. In contrast, in the case of FLASH, the trace information is stored in memory, reducing the overhead of collecting the information.

2.4 Existing Implementations

In this section, we examine the state of the art for memory profiling. We discuss two memory profilers, MemSpy and Paradyn, that meet all of our memory profiling criteria (see

Sections 2.1 and 2.3) except for performance. We also examine two higher-performance techniques that fall short of actually providing a memory profile.

2.4.1 Simulation: MemSpy

The MemSpy memory profiler [16] is a simulation-based tool that computes a complete memory profile for an instrumented program. The source code of an application under study is instrumented for MemSpy, as described below. The compiled binary is then rewritten so that each load or store instruction in the program is instrumented to make a call into the Tango memory system simulator [5], the precursor to Tango Lite. Likewise, procedure entry and return is instrumented for use by Tango. Tango simulates the memory hierarchy so it can determine which memory accesses cause misses. Since MemSpy runs as part of Tango, it can record counts as misses occur.

To categorize misses by application data structure, MemSpy instruments the application source code to assign *bin numbers* to each application data structure. These bin numbers are simply tags for the data structures. The simulator maintains an address-to-bin number translation table so it can create a histogram of miss statistics based on the bin number of the data structure that caused the miss.

As mentioned in Section 2.1, memory profilers should provide a code-oriented, as well as a data-oriented, view of application behavior. MemSpy does this by categorizing misses by application procedure, as well as by data structure. When a program is instrumented for use with Tango, instructions are added to log procedure entry and exit. MemSpy is able to use this information to accurately determine the current procedure. It is therefore able to categorize misses by both data structure and procedure, providing a complete memory profile.

MemSpy produces exactly the sort of memory profile data we desire. In fact, our work in subsequent chapters is an extension of MemSpy. Our work addresses MemSpy's two important limitations: performance and accuracy. While the marginal performance cost of adding MemSpy to Tango is rather small, the user must endure the hefty simulation overhead of Tango. Uniprocessor applications run 22-58 times slower with MemSpy-Tango than when running natively on the processor [16]. The overhead of simulating a multiprocessor application is far worse, since Tango interleaves the execution of multiple application threads into a single simulator thread. This means that an application with N threads

will see at least a factor of N performance degradation over the already high MemSpy-Tango overhead. Such overheads can easily be a factor of several hundred — hardly the 20% overhead we desire. While simulation technology has improved since MemSpy and Tango were developed, we contend that no simulation technique for memory profiling is or will be capable of running with 20% overhead. Simulators must detect, as well as instrument cache misses, and they will always have unacceptable overhead since they must instrument every memory access to do this detection.

The accuracy problems associated with MemSpy come from the limitations of Tango. The extensive instrumentation of the program hopelessly alters the instruction cache behavior to such a degree that Tango does not even bother simulating instruction cache behavior. There is also no kernel running on the simulated machine so there are no kernel effects such as data placement, cache coloring, or TLB behavior. While Tango does have directives for placing data, they are implemented by the simulator. Typically, users want to know about the effectiveness of the data placement on their hardware with their operating system, not on an arbitrary simulation platform. Tango, like all simulators, also suffers from not modeling the system in perfect detail. Perhaps a given run of the simulator models all important effects, and perhaps not. There is simply no way to tell. Blind faith in simulation can lead to some unpleasant surprises and incorrect conclusions [6].

2.4.2 Software Shared Memory: Paradyn

The Paradyn system [27] from the University of Wisconsin implements an enhanced variant of memory profiling they term “shared-memory performance profiling.” In addition to producing a memory profile as we define it, Paradyn also performs pattern detection in the memory reference stream to allow the tool to determine sharing patterns in user data structures. Their reported overheads are 2-10%, so at first glance, high-performance memory profiling seems to be a solved problem. The reason it is not has to do with their hardware platform.

Paradyn runs on a cluster of workstations (40 Sun SPARCStation 20 workstations containing dual 66 MHz processors, connected by Myricom Myrinet and 100MB Ethernet) running a fine-grain distributed shared memory system known as Blizzard [22]. Cache coherence is maintained by Blizzard, which runs the cache-coherence protocol as a process on each of the compute processors. The Blizzard processes communicate with each other to

Machine	Processor Clock	Remote Access (cycles)	Remote Access (μs)
FLASH	225 MHz	256	1.1
Blizzard/CM-5	33 MHz	6000	182

Table 2.1: Overhead of remote memory access on Blizzard and FLASH.

present the shared-memory abstraction to other user processes. Since the shared-memory abstraction is maintained by software on the compute processors, platforms that work in this manner are called *software shared-memory* machines. These machines are popular in the research community for studying shared-memory because large shared-memory machines can be built from inexpensive compute nodes and commodity interconnection technologies. The obvious disadvantage of software shared-memory is performance. Communication runs at network speeds, which can be orders of magnitude slower than memory speeds. The cache coherence process also runs on the compute processor, so it interferes with the performance of the user application, thus causing additional overhead.

Paradyn collects its data by instrumenting the Blizzard cache-coherence protocol. As the authors report, overhead due to the Paradyn instrumentation is quite small. The cost of remote memory access on Blizzard, however, is enormous compared to a hardware cache-coherent machine, as shown in Table 2.1. Blizzard is a software system that exports a shared memory interface on hardware platforms that do not support it. Therefore, it has the burden of detecting remote accesses as well as handling them. Performing this detection in software requires simulation techniques, and Blizzard is in fact based on the Fast-Cache simulator [13] discussed in Section 1.2.2! Each store in the program is instrumented with 50 instructions that perform a table lookup and decide whether a memory access is remote or requires some other coherence transaction. For loads, Blizzard is able to exploit the memory's error-correcting codes (ECC) as hardware support to dramatically reduce the cost of detecting remote accesses. When remote access does occur, however, the system must context switch from the application to and from a Blizzard process, and these context switches are costly operations.

Blizzard was originally implemented on a Thinking Machines CM-5 [14] that had a network latency of 3-7 microseconds, depending on topology. The round-trip cost of remote memory on Blizzard, was estimated by the authors to be 6000 processor cycles [22]. With 33 MHz processors, this translates to 182 microseconds — far higher than the underlying machine network access time. This is in stark contrast with the hardware cache-coherent

FLASH machine, where a remote memory access to the nearest remote node has a round-trip latency of 1.1 microseconds, only 2.6 times the latency of a local memory access.

The hardware platform used to report Paradyn results, as stated above, is a cluster of workstations, not a CM-5, and the authors do not report Blizzard performance data on that platform. Nevertheless, Blizzard must perform the same actions as on the CM-5 and will therefore see similar overhead. In fact, nodes on a cluster of workstations are likely to be less tightly coupled than the nodes in a CM-5, and this could cause the remote memory latencies to be even higher, particularly when expressed in processor cycles.

Because Blizzard has such long communication latencies, Paradyn is able to perform an almost arbitrary amount of computation and still have a negligible effect on application performance. Paradyn performs both memory profiling and pattern detection *in software* on a stream of memory references, while the memory references are occurring, with less than 10% overhead! It is important to remember that although Paradyn adds only a small percentage overhead, a small percentage of a huge remote memory latency is still a substantial amount of time.

We agree that it is both possible and desirable to perform memory profiling by instrumenting a cache-coherence protocol. In fact, we examine this technique ourselves in Chapter 4. In hardware cache-coherent machines, the class of machines of interest in this work, considerable expense and engineering effort was made to allow the cost of remote memory access to be small. This makes memory profiling far more challenging, since hardware cache-coherent machines present a tight set of performance constraints. Monitors such as Paradyn are too expensive to run on such machines. For instance, if we make a conservative estimate that Paradyn adds 10% to the remote memory latency (conservative since it assumes the reported 10% overhead was for a completely remote-memory-bound application), that would imply 600 processor cycles of Paradyn instrumentation per remote memory access. On the FLASH machine, 600 cycles of overhead would more than triple the cost of remote memory access!

2.4.3 Hardware Counters: `snperf`

The `snperf` suite of performance tools [20] uses hardware performance counters on the Origin 2000 to profile application memory behavior. Most platforms offer processor-based performance counters which can count, among other metrics, the number of cache misses

taken by a program. The Origin 2000, however, also implements hardware performance counters in the system node controller (called HUB) and in the routers. The goal of the `snperf` tools is to use sampling of the HUB and router performance counters to provide the same sorts of insights into the memory behavior of a program that SpeedShop provides on the processor behavior.

The data collected by `snperf` is similar to extended memory profile data in that it counts events invisible to the processor such as local vs. remote cache misses, the number of cache interventions and invalidations, and the utilization of various system resources. It suffers from the same drawback of all performance counter-based tools, however, in that there is no mechanism to tie events back to the application data structures that cause them since only aggregate counts are kept. Also, there is no hardware or system software support for interrupting the processor on an overflow of a HUB or router counter, so instruction profiling is not possible with `snperf`, though one could imagine a different hardware architecture where this was possible. This means that `snperf` is essentially a gross event counting tool — just one that can count a large variety of quite low-level events. As a result, `snperf` is most useful for understanding the detailed memory behavior of small sections of code, where the user already knows the location of the central bottleneck and is trying to make improvements. Unlike a memory profiler, `snperf` is not able to focus a user's attention on the code and data structures that limit performance.

2.4.4 Page Reference Counters

We note that an approximation of memory profiling can be done with per-page reference counters, such as those in the Origin 2000. Such counters typically exist to provide the operating system with data on when page migration and/or replication might be worthwhile. This approximation is crude, however, as counts are only obtainable on a per-page basis. Also, operating system intervention is required to handle the physical-to-virtual translation necessary to make the miss counts meaningful to the user. This intervention can either be done during the execution of the monitored program or by post-processing. The former causes high overheads, as the entire virtual address space of the application must be examined, and the latter is inexact, as virtual-to-physical mappings can change over the course of the run.

2.5 New Implementations

In this work, we introduce two high performance memory profilers, TrapPoint and FlashPoint. Both rely on hardware support that is not standard in current machines to achieve the requisite performance, but we make use of the flexibility of the FLASH multiprocessor [11] to prototype both systems.

TrapPoint, described fully in Chapter 3, is a processor-based technique that works by trapping into the performance monitor on cache misses. The hardware support required is a trapping cache miss, a processor innovation which has been proposed [10], but not implemented on any commercial microprocessor. TrapPoint has the distinguishing feature that it allows high-performance memory profiling with only a modest amount of additional processor hardware. This additional processor hardware does not exist on FLASH, but the system's flexibility makes prototyping TrapPoint possible.

The other memory profiler we have implemented, FlashPoint (discussed in Chapter 4), relies on instrumentation in the system node controller. Flexible instrumentation of the system node controller is a unique aspect of the FLASH machine [11]. This allows FlashPoint to perform extended memory profiling, and it addresses several of TrapPoint's shortcomings.

The remainder of this dissertation describes the implementation of both TrapPoint and FlashPoint. We study the performance and accuracy of each memory profiler, and draw general conclusions on the feasibility of high performance memory profiling.

Chapter 3

Processor-Based Memory Profiling

Using the processor for high performance memory profiling requires hardware support to trigger the performance monitor on the processor when cache misses occur. This triggering could be done either by the processor itself when it detects a miss or by the system when it receives a memory request. The most straightforward triggering method would be a mechanism for trapping on a cache miss. Several methods of implementing miss traps have been proposed by Horowitz et. al. [10] Since memory profilers categorize as well as count cache misses, the miss trapping mechanism must possess two characteristics. The first is that the trap handler must have access to the address that caused this miss. This is necessary because the performance monitor cannot determine the data structure that caused the miss unless it has access to the miss address. The second restriction is that the trap be precise. This is required so that the monitor can collect temporal statistics, such as which procedure caused the cache miss. We note that if the interrupt is precise, it should be possible to determine the miss address from the program counter and the register state, and this relaxes the first constraint somewhat.

Many commercial processors implement performance counters that can count cache misses. Some even support interrupts that are triggered on overflows of these counters. Unfortunately, none of them fully account for our two restrictions, probably because they were not designed with memory profiling in mind.

This chapter describes TrapPoint, a performance monitor that uses miss traps to enable memory profiling. We first discuss the Stanford FLASH Multiprocessor, the hardware platform used for the TrapPoint prototype. We then describe the TrapPoint implementation

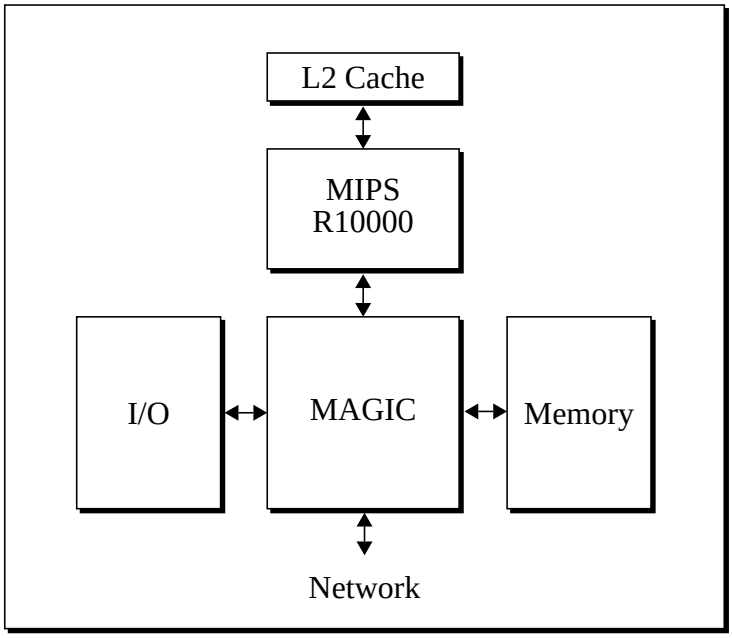

Figure 3.1: Block diagram of a FLASH node.

and evaluate it for performance and accuracy.

3.1 FLASH, the prototype hardware

We prototype our memory profilers on the Stanford FLASH (FLexible Architecture for SHared memory) multiprocessor [11], so we briefly describe the FLASH architecture.

FLASH is a cache-coherent distributed shared-memory multiprocessor. The machine is organized as a network of nodes. Figure 3.1 shows a diagram of a node in the FLASH machine. Each node contains a MIPS R10000 compute processor, a portion of globally-accessible memory, and a node controller. The node controller, called MAGIC (Memory And General Interconnect Controller), manages all communication among the compute processor, the memory, the I/O subsystem, and interconnection network.

The principal contribution of the FLASH machine is that the MAGIC chip contains an embedded processor and is programmable. Most machines implement their node controller, and thus their cache-coherence protocol, as a set of hardware state machines. In FLASH,

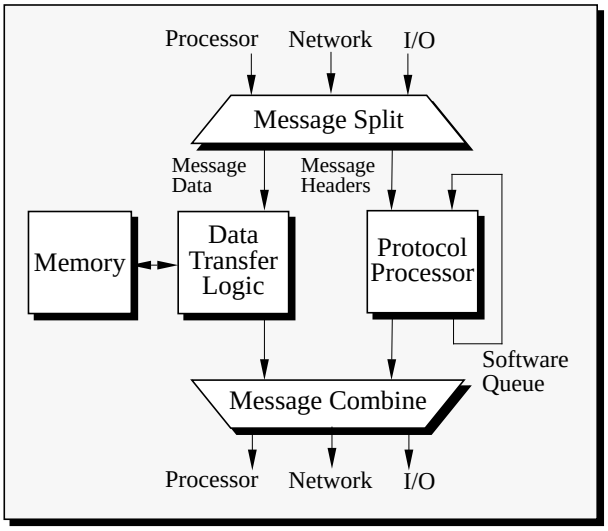

Figure 3.2: MAGIC block diagram.

however, the cache-coherence protocol is implemented as a set of MAGIC functions, called handlers. Whenever the compute processor experiences a miss in its secondary cache, it sends a request to MAGIC. The source and type of the incoming message are used to schedule the appropriate handler on MAGIC's embedded protocol processor. The handler code is responsible for sending whatever messages are necessary to memory, I/O, or even remote MAGIC chips (which will then invoke their own handlers). MAGIC also manages cache coherence and performs cache interventions and invalidations as necessary.

Though MAGIC contains a programmable processor, it was designed to give performance comparable to a pure hardware implementation [11]. Figure 3.2 shows the MAGIC architecture. Incoming messages are split into header information and data. The embedded protocol processor examines the header information and coherence state and constructs outgoing messages. Data transfers are handled by specialized hardware so as not to burden the protocol processor.

For this study, we ran MAGIC at 75 MHz. Its embedded processor is a two-issue VLIW modified MIPS core. The compute processor is a 225 MHz MIPS R10000. As in the SGI Origin 2000, the nodes are connected via CrayLink and SPIDER routers. FLASH runs a slightly modified version of IRIX 6.5. Some boot code and device drivers specific to the SGI Origin 2000 were altered to boot IRIX on FLASH.

```
struct StatsRecord {
   uint64 LocalReadMissCount;
   uint64 LocalWriteMissCount;
   uint64 RemoteReadMissCount;
   uint64 RemoteWriteMissCount;
}
```

Figure 3.3: Stats Record definition for TrapPoint

3.2 TrapPoint Implementation

We wish to prototype TrapPoint on FLASH, but as mentioned previously, a trapping cache miss mechanism is not supported by commercial processors, including the MIPS R10000. Fortunately, we can make use of FLASH's flexibility for our prototype. We program MAGIC to send the processor an interrupt whenever it services a cache miss. Note that while we are using MAGIC support for our prototype, we are not making use of any information not known to the processor. Thus, if the R10000 implemented trapping cache misses, no MAGIC support would be necessary.

This section describes the TrapPoint implementation in detail. We start by describing a simplified version which only collects gross miss counts on each node. We then show how this initial design was augmented until it was capable of memory profiling.

3.2.1 Gross Miss Counts

We begin by defining a data structure in kernel memory on each node called the *Stats Record*, as shown in Figure 3.3. Misses are categorized by type (read vs. write) and by location (local vs. remote).

The main work of TrapPoint is done in the interrupt handler, which runs on the processor after being invoked by MAGIC's cache miss interrupt. The interrupt handler polls a MAGIC register that contains the physical address which caused the miss and a load/store bit. It uses the load/store bit to determine whether the miss was caused by a read or write. The home node of the address is encoded in the address itself, and the interrupt handler uses this to determine whether the access was local or remote. Once the handler has categorized

the type of miss, it simply increments the appropriate field of the Stats Record. This technique can count all secondary cache misses, save for those that occur while interrupts are disabled.

This method gives gross miss counts for each node, categorized by type and location. This simplified version of TrapPoint is similar in functionality to existing tools based on performance counters.

3.2.2 Data Structures

The goal of memory profiling is to categorize the misses by the data structures that caused them. To extend the simplified TrapPoint monitor described above into a memory profiler, we must add instrumentation to both the application and the kernel.

Application Instrumentation

The user applications to be profiled are instrumented to tag each of their data structures with a *bin number*. These bin numbers are then used to histogram misses by data structure. The assignment of bin numbers to virtual addresses is arbitrary and can be done any way the application programmer sees fit. The application is responsible for informing the kernel which bins correspond to which ranges of virtual addresses. It does this with a system call giving start address, length, and bin number.

Though any application can be instrumented by hand, we use a heuristic that allows many applications to be instrumented automatically. Each static instance of `malloc` becomes its own bin. A program called `fp_instrum`, a special pass of the SUIF [1] compiler, replaces calls to `malloc` with calls to an instrumented version. This proves to be a useful mapping for many programs, as users are typically interested in the memory behavior of entire arrays much more than the behavior of individual indices. Statically-allocated variables are handled by a script called `staticBins` that examines the compiled application and assigns bin numbers to the data symbols. It dumps this information into a file. When the application starts, it reads this file and makes the requisite system calls to inform the kernel of its bins.

The granularity of binning is a secondary cache line, 128 bytes on FLASH, so any two addresses on the same cache line will necessarily belong to the same bin. Since 128

bytes is small relative to most data structures, we have not observed this to be an important limitation. Having some minimum granularity is necessary to limit the size of kernel data structures, though making it the size of a cache line is arbitrary for TrapPoint. However, we wish to compare TrapPoint to another memory profiler, FlashPoint, described in the next chapter. As we will discuss, FlashPoint must have a cache line granularity, and since there is no reason to use a different size for TrapPoint, we choose to use 128 bytes for TrapPoint as well. Data structures smaller than 128 bytes can still be instrumented. They can either be padded out to 128-byte boundaries, or multiple data structures can be aggregated into the same bin.

Kernel Instrumentation

Once the application has partitioned its data structures into bins, the remainder of the instrumentation is done by the kernel. Kernel involvement is necessary since the application assigns bin numbers to *virtual* addresses, while the interrupt handler will see only *physical* addresses. The kernel manages the page table, so only it can do the requisite address translation.

Region bin array. Under IRIX, an application's virtual address space consists of contiguous segments called *regions*. Examples are program text, the heap, and memory-mapped files. TrapPoint extends the region data structure to contain an array of bin numbers, called the *region bin array*, with one element for each cache line in the region. This data structure allows the kernel to look up a bin number, given a virtual address. It simply finds the region data structure associated with the address, and then uses the cache line offset from the beginning of the region to index the array.

When new regions are created, all cache lines have a bin number of zero assigned to them. This makes bin zero an "other" bin, in that all memory accesses to uninstrumented data structures are attributed to bin zero. The region bin array is not allocated for a region until the first non-zero bin number is assigned. This improves the performance of non-instrumented applications.

The region bin array is written when the application makes a system call to map bin numbers to virtual addresses. Other than this initial allocation, it only needs to be updated when the region data structure itself is copied, split, or destroyed.

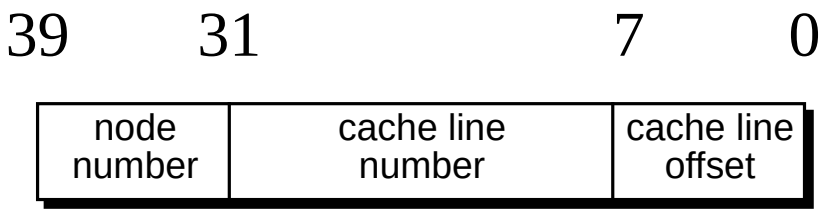

Figure 3.4: FLASH physical address layout.

Bin map. The most important TrapPoint data structure in the kernel is called the *bin map*. The bin map is an array of bin numbers accessed by physical address. Since the format of the 40-bit physical address has implications in how the bin map is constructed, we briefly describe the physical address format of FLASH here. The physical address space of FLASH is discontiguous; each node contains a contiguous range of physical addresses, but they are not adjacent to each other. The layout of a physical address on FLASH is shown in Figure 3.4. The home node for each physical address is contained in the highest eight bits of the address. Within each node, the node offsets (bits $31:0$) range from zero through the size of memory on the node.

Each node contains a portion of the bin map for its local addresses, and there is one bin map entry for each cache line on the node. Thus, the bin map is implemented as an array of bin numbers, indexed by bits $31:8$ of the physical address. The bin map gives the kernel a mechanism for obtaining a bin number, given a physical address.

When a physical page is allocated by the kernel and mapped to a virtual address, the kernel uses the region bin array to look up bin numbers for each cache line on the page. It then writes them into the appropriate entries in the bin map for the physical page being allocated. It does the same operation when an application does a system call to assign bins to a range where physical memory has already been allocated. Similarly, when physical memory is deallocated, the bin map entries are cleared to zero. Updates to the bin map are infrequent; they are only done during bin allocation and during paging.

Interrupt handler. Once the kernel instrumentation is in place to build and maintain the bin map, extending the interrupt handler to categorize misses by data structure is straightforward. The Stats Record is extended to be an array of structures on each node, called the

Stats Record Array. The interrupt handler uses the bin map to look up a bin number for the address that caused the miss. It then uses the bin number as an index into the Stats Record Array.

The described instrumentation allows TrapPoint to collect memory profile data. All misses are categorized by the data structures that caused them. Misses to uninstrumented data structures are aggregated together in bin zero. The cost of running an interrupt handler can be high, but as will be shown later in this chapter, the marginal cost of adding memory profiling to such a handler is not. TrapPoint also makes efficient use of statistics memory since it bins misses on a granularity that is meaningful to the application programmer: data structures. This avoids overheads of storing data of an arbitrary granularity, such as pages.

3.2.3 Procedures

A memory profile is a high-level view of application behavior. It does not, as instruction profiles do, attempt to attribute memory behavior to specific instructions. It instead provides a broad mechanism for studying how an application uses memory. Nevertheless, a gross instruction-based breakdown, such as at the procedure level, can add useful information to the memory profile.

TrapPoint accomplishes this by having applications instrument their procedures. Each procedure is numbered, and macros are added to the entry and exit of each procedure to maintain a stack of procedure numbers. This instrumentation can be done either by hand or by the special pass of the SUIF [1] compiler, `fp_instrum`. The current procedure number for each processor is maintained on a page of memory that is special in that the kernel has pinned it in physical memory so that it is never paged out or moved. The kernel can access this page to find the current procedure number of the thread running on any of its processors.

Since the kernel maintains one procedure number for each processor, this technique requires that instrumented jobs be pinned to a particular processor for their entire run and that only one instrumented job run on a processor at a time. Instrumenting the context-switching code would lift these constraints, but we did not find that necessary for this implementation.

We extend the array of Stats Record Array on each node into a second dimension. The interrupt handler simply looks up the current procedure number and uses it as an index into

the second dimension.

Note that the current "procedure number" need not actually number procedures, as TrapPoint does not actually interpret this number. It is simply a way of having a temporal index into the Stats Record Array. The index is temporal in the sense that once it is set, it doesn't change until set again, so it can be used to categorize misses that happen before or after certain events. It could just as easily number loop iterations or even the hour of the day. We simply find procedures to be the most generally useful. It is important to realize that because of the way procedure numbers are implemented, when misses occur in a procedure that has not been instrumented, those misses will be attributed to the procedure's caller.

With the data structure and procedure extensions, the TrapPoint implementation is now complete. It categorizes misses by the data structures and procedures that caused them. We have also implemented a visualization tool [3] that allows for exploration of this data.

3.3 Methodology

To evaluate the performance and accuracy of TrapPoint, we use a collection of applications, described in this section. We evaluate performance by comparing the run times of TrapPoint-instrumented programs on a TrapPoint-enabled system to the times of uninstrumented runs of the same applications on an uninstrumented system. We evaluate the accuracy of TrapPoint by comparing the memory profile collected by TrapPoint with the profile collected by FlashPoint. FlashPoint is described in Chapter 4, and for reasons described there, it collects much more accurate memory profiles than TrapPoint. All data was collected for 16 processor runs on a FLASH machine with more than 16 processors to minimize operating system interference.

3.3.1 SPLASH-2 Applications

We select four commonly-used benchmark programs from the SPLASH-2 suite [26]: FFT, Radix, LU and Ocean. Problem sizes were chosen to be quite large, so they are representative of real applications. The problems are sized to give execution times on an uninstrumented system of roughly one minute on a sixteen-processor machine. These problems are therefore substantially larger than those typically run in simulation. The SPLASH-2 programs used here are all written in C, they express their parallelism through calls to `sproc`

(an IRIX system call, similar to `fork`, in which the child process runs in the same virtual address space as the parent), and they were instrumented for TrapPoint automatically by `fp_instrum`. More detail on these programs can be found in [26].

FFT

FFT is a Fast Fourier Transform program described in detail in Section 2.2.1. The input matrix is a sixteen-million point matrix of complex doubles (256 MB), and a block size of 32 is used for optimal TLB performance.

Radix

The Radix application sorts large arrays of integers and is described in Section 2.2.1. The source array is 256 million integers (1 GB), and we use a radix of 32 for good TLB behavior.

LU

LU does a dense-matrix LU factorization. We use a source matrix of sixteen-million doubles (128 MB). It is blocked into 16×16 blocks for good L1 cache behavior. We have made one major modification to the code from the version described in [26]: instead of allocating and placing each block separately, we first allocate a large range of data on each of the nodes and allocate blocks from those ranges. This substantially improves both the initialization time and the cache conflict behavior of LU.

Ocean

Ocean simulates large-scale ocean movements. A 2050×2050 grid was used as the problem size.

3.3.2 SPEC OMP Applications

OpenMP is an industry-standard method of expressing parallelism. It consists of programmer-inserted compiler directives to identify parallel regions and a runtime library. The SPEC OMP suite of benchmarks was released in June 2001 with the goal of providing a platform-independent benchmark suite of shared-memory applications [24].

These applications were instrumented for TrapPoint by hand since the SUIF compiler does not correctly interpret the OpenMP directives. Unlike those handled by `fp_instrum`, programs that were instrumented by hand do not have every procedure and every data structure instrumented, since the time involved with exhaustively instrumenting large codes can be prohibitive. The typical use of TrapPoint for large applications is to instrument the main routines and the larger data structures. We then successively add instrumentation only in those locations where many misses occur.

Swim

Swim is a widely-used benchmark application. It is a FORTRAN program that models shallow-water dynamics. The problem size is a scaled down version of the training input set. It uses 80 iterations instead of 800 to achieve tolerable execution times.

AMMP

The AMMP (Another Molecular Mechanics Program) benchmark code is a scaled-down version of a popular molecular dynamics engine. It is written in C. The application is described in detail in Section 2.2.4. The test input data files were used.

3.3.3 Research Application

One program used for our evaluation, VirtualMesh, is not part of any benchmark suite. This FORTRAN program, described in Section 2.2.5, is a research code from the Center for Turbulence Research at Stanford. It solves the Navier-Stokes equations for modelling fluid flow — in this case, air flow over a car. Typical runs of this application last for weeks. For profiling, we use the full problem but only compute the first four time steps, which gives us uninstrumented execution times of roughly one minute.

3.4 Overhead

The drawback of TrapPoint is its intrusiveness. The processor takes an interrupt on every cache miss, and while the interrupt handler is running, the application is not progressing.

Application	Normalized Execution Time
FFT	15.89
Radix	28.33
LU	10.56
Ocean	25.84
Swim	12.01
AMMP	3.65
VirtualMesh	8.76

Table 3.1: TrapPoint overheads.

This not only causes large increases in run time, but also can cause significant performance distortions.

The overheads of TrapPoint on the applications are shown in Table 3.1. The runtimes are normalized to those of uninstrumented applications on an uninstrumented system. In other words, all times would be 1.0 if TrapPoint had no overhead. The overheads shown, 3x-28x, are so large as to be intolerable. In fact, these speeds are comparable to simulation.

There are two operative effects. The first and most obvious is that the cache miss time is greatly increased. The other is a side-effect of TrapPoint running on the compute processor. Although misses caused by the interrupt handler itself are not counted, they pollute the processor caches. This cache pollution can evict user data, which will cause user misses that would not have occurred had the instrumentation not been present. Indeed, most programs under study exhibit pathological cache behavior caused by TrapPoint. The large data sets present in many of these applications, particularly those from the SPLASH-2 suite, are substantially larger than the caches, so the TrapPoint interrupt handler causes a large number of both conflict and capacity misses. For instance, Radix, which has the largest data set and not coincidentally, the highest overhead, experiences 1.9 times as many cache misses with TrapPoint than without. This does not even even take into account the unreported misses taken by the TrapPoint interrupt handler! The facts that programs take more misses with TrapPoint and that these miss times are quite long are the reasons that all applications see high overhead.

We find that the overheads depend strongly on the problem size used for the applications. Preliminary TrapPoint studies with the SPLASH-2 programs on much smaller data sets gave us much smaller, though still considerable, overheads of 3-7x. The discrepancy occurred because these small data sets could exist in the cache along with the TrapPoint

data structures. Conflict misses, then, were the only problem. We view the large program results as more relevant, however, since none of the smaller applications ran longer than five seconds — hardly the sort of application it is important to profile.

TrapPoint causes so many additional cache misses that its memory profiling results are of no value. The cache misses caused by profiling generally outnumber the misses caused by program behavior, so the results say more about TrapPoint than about the application under study. We therefore conclude that TrapPoint, as currently described, is not an acceptable memory profiling technique.

3.5 Sampled TrapPoint

Both the overhead and distortions of TrapPoint can be mitigated by sampling. For the Sampled TrapPoint implementation, we add a small amount of instrumentation to the MAGIC cache-coherence protocol. Instead of sending the processor an interrupt on every cache miss, we now send interrupts only for selected misses. We choose not to sample at regular intervals (i.e., every 256th miss for a period of 256), since there is a possibility of performance artifacts due to correlation between the samples and the cache miss behavior. We have not observed this behavior, but we recognize the possibility and wish to avoid it. The correlation issue is solved by sampling at random intervals. The only problem is that since MAGIC software controls the sampling, MAGIC needs to be able to generate a random number. The protocol processor on MAGIC was optimized for common cache coherence transactions, not for complex math. The processor, for instance, lacks both a multiplier and a divider, which makes random number generation difficult. Though MAGIC does contain a cycle counter which could potentially be used for random numbers, a cycle counter might also suffer from correlation behavior. Therefore, we implement a 63-bit LFSR and XOR its value with the cycle counter to obtain a random number. With random sampling, a period of, say 256, means that on average one miss in 256 generates an interrupt. This is not an optimal source of random numbers, but more sophisticated techniques would require hardware support that MAGIC lacks.

Application	Period 8	Period 64	Period 128	Period 256	Period 4k	Period 64k
FFT	6.39	1.64	1.38	1.28	1.17	1.15
Radix	4.95	1.69	1.55	1.45	1.41	1.37
LU	3.16	1.25	1.18	1.15	1.12	1.11
Ocean	12.90	2.04	1.47	1.28	1.16	1.13
Swim	5.88	1.67	1.34	1.21	1.09	1.08
AMMP	2.40	1.73	1.67	1.61	1.56	1.58
VirtMesh	4.63	1.75	1.60	1.44	1.42	1.38

Table 3.2: Normalized execution time for Sampled TrapPoint at several average sampling periods.

3.5.1 Overhead

Table 3.2 shows that the overheads are dramatically reduced by interrupting the processor on only a fraction of the cache misses. With an average sampling period of 65536 (64k), all overheads are less than 58%. In fact, save for AMMP, which has the worst overhead with a period of 65536, the applications run in the range of 11-38% overhead. This exceeds our ideal target performance of 20% (see Section 2.3), but it is close. The poor behavior of AMMP is explained later in this section. While sampling improves the conflict behavior substantially, pathological cases are still possible. Most programs, however, see little conflict behavior with the higher sampling periods.

The observed performance of sampled TrapPoint is similar to (actually, slightly worse than) recent work with DCPI value sampling [4]. DCPI uses performance counters to interrupt the processor. These interrupts cause the program to drop into a simulation mode. Since the simulator has access to the operands of each instruction, it is able to sample the values of these operands. DCPI achieves high performance by using hardware to send interrupts only in "interesting" sections of code, so the simulator does not need to instrument every memory access to find misses. It should be possible to collect memory profile data with the DCPI infrastructure, since miss addresses are simply operands to memory operations that cause cache misses.

We examine the overhead breakdown in Figure 3.5. Each bar is broken down into various components of execution time as explained below:

1. **Base.** This represents the execution time of an uninstrumented application on an uninstrumented system. Since all execution times are normalized to the base case,

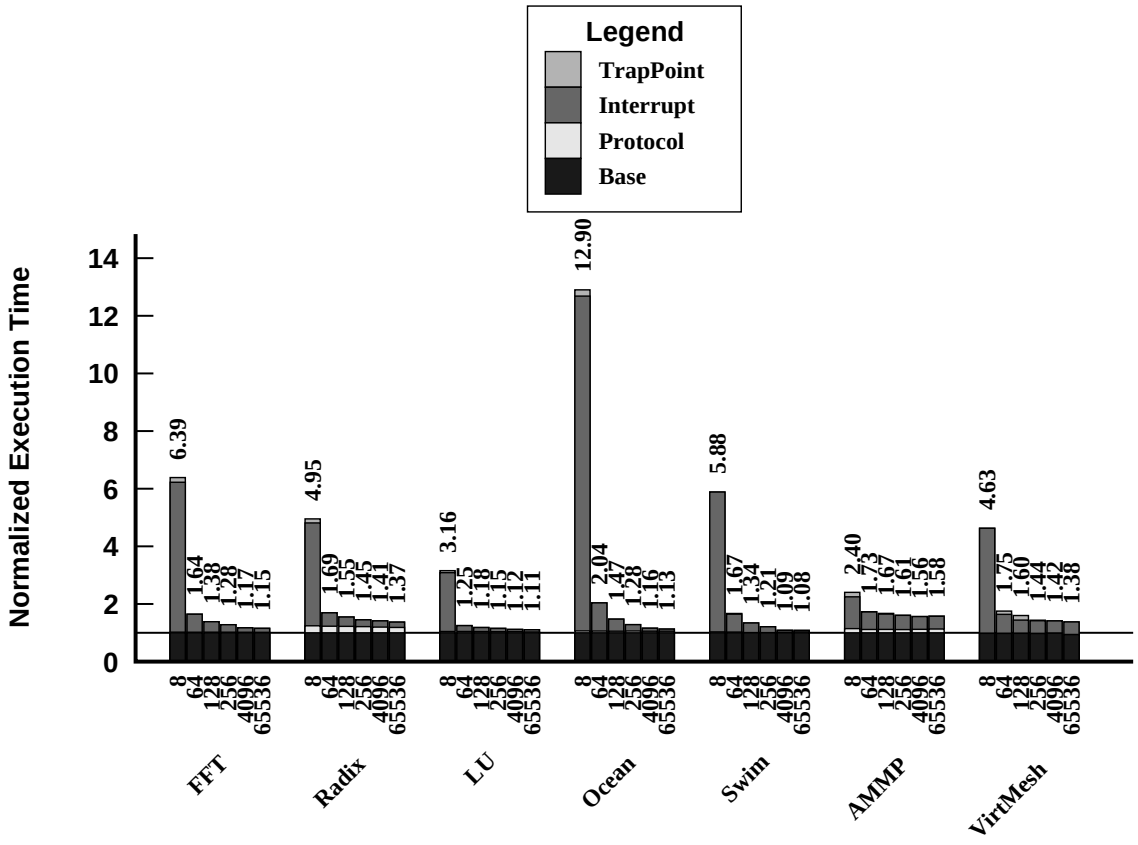

Figure 3.5: Overhead breakdown for Sampled TrapPoint at various sampling periods.

this component is shown as 1.0.

2. **Protocol.** The protocol overhead is the increase in execution time due solely to the cache-coherence protocol instrumentation. This was measured by a run that had the protocol instrumentation present, except for the code that actually sends the interrupt. There is still significant protocol code instrumentation, even without the interrupt, because the sampling mechanism is implemented in MAGIC software. Much of protocol instrumentation is, in fact, the sampling mechanism, so protocol overhead decreases only slightly as the sampling period increases. A hardware implementation would not see any such overhead.

3. **Interrupt.** The interrupt overhead is the overhead due to the interrupt mechanism.

This was measured with a fully instrumented TrapPoint run except that the TrapPoint interrupt handler simply returns (technically, it drains a MAGIC buffer to ensure forward progress, then returns). This is the overhead of running any sort of interrupt. Bear in mind that this is the overhead from adding the interrupt handler, not the time actually spent taking the interrupt. Any extra cache misses in user code caused by the behavior of the interrupt mechanism will be counted as interrupt overhead. Since this overhead is proportional to the number of interrupts taken, we expect it to decrease as the sampling period increases.

4. **TrapPoint.** The TrapPoint component is the additional overhead of actually running the full TrapPoint interrupt handler, which involves looking up a value in the bin map, finding the current procedure number, and updating the Stats Record Array. Again, any interference misses will contribute to this overhead. As with the interrupt overhead, we expect the TrapPoint overhead to decrease as the sampling period increases.

Our results show the protocol overhead to be insignificant at low sampling periods. For most runs, the overhead of taking an interrupt is the major performance factor, and it far exceeds the overhead of the interrupt handler itself. This shows that an alternative mechanism for sampling, namely sending all interrupts and having most of them return prematurely, would suffer from much worse performance than the current scheme. Taking an interrupt is an expensive operation since it involves spilling the application's registers to memory, context switching into the kernel, going through the kernel's interrupt dispatch code, and context switching back to user mode. Processor support for a fast miss trap [10] could somewhat reduce this overhead, but this will likely be the major source of overhead in any trap-based monitor.

The large overheads of the low sampling periods are prohibitive not only because of the increase in execution time, but also because their intrusiveness substantially alters the behavior of the application, as will be shown in the next section. This indicates that slower sampling (i.e., longer sampling period) leads to better memory profilers. Therefore, we now restrict our attention to sampling periods of 128 and higher, as shown in Figure 3.6.

For the applications where protocol overhead is noticeable, it is largely insensitive to the sampling period, as expected. The counterintuitive result shown in Figure 3.6 is that the interrupt overhead does *not* decrease as the sampling period increases for AMMP and

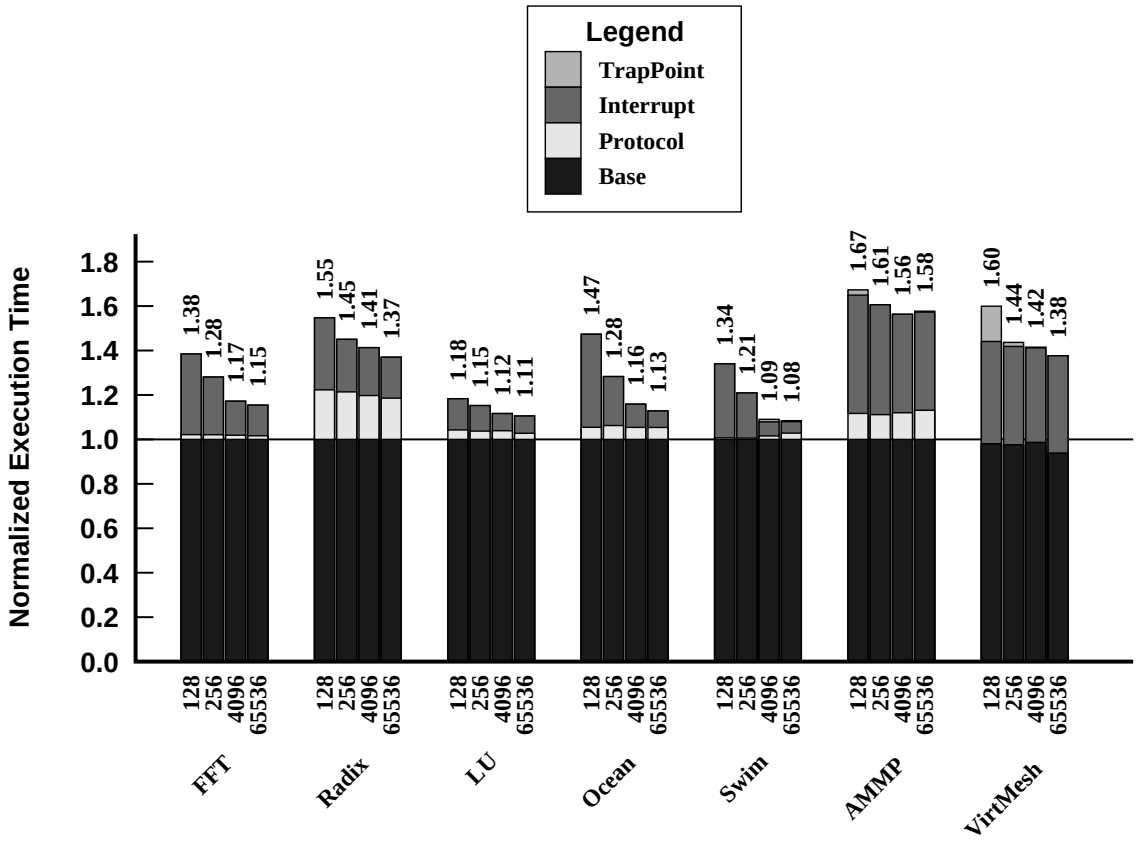

Figure 3.6: Overhead breakdown for Sampled TrapPoint at high sampling periods.

VirtualMesh. This odd effect occurs because the embedded MAGIC protocol processor takes more instruction cache misses when it runs the version of the protocol that sends the interrupts. MAGIC has an undersized direct-mapped instruction cache, so instruction cache conflicts are always a concern. This is discussed in more detail in Section 4.2.1. The result is that overhead that is actually related to the cache-coherence protocol is mistakenly reported as interrupt overhead. This means that Radix, AMMP, and VirtMesh all experience significant protocol overheads, and the fact that this is not mitigated by sampling is the reason that these applications see the highest overhead with long sampling periods. A related effect is that the reported protocol overhead for VirtualMesh is actually *negative*. Negative protocol overhead is shown on the graph by subtracting from the height of the base bar. The version of the protocol used to measure protocol overhead, which does sampling

but does not send interrupts, actually has slightly fewer instruction cache misses than the base protocol for VirtualMesh.

The existence of substantial protocol overhead for Radix, AMMP, and VirtualMesh is not an important result since it is simply an implementation artifact of MAGIC. A hardware trapping cache miss would not see this overhead. For applications without significant protocol overhead, TrapPoint overhead is quite tolerable (28% or less) for sampling periods of 256 and higher. In fact, at sampling periods of 4096 and above, these applications meet the design goal of 20% overhead. Sampled TrapPoint is therefore a viable memory profiling technique, provided that the profiles it produces are accurate.

3.5.2 Accuracy

To investigate the accuracy of Sampled TrapPoint, we compare its reported memory profiles to the true memory profiles of these applications. The true profiles were obtained by a more exact tool, FlashPoint, which is described in the next chapter.

For each benchmark at each sampling period, we compute the *error fraction*. For each cpu/procedure/bin combination, we compute the absolute value of the difference between the number of misses reported by TrapPoint (number of samples times the sampling period) and the number of misses reported by FlashPoint. This represents the number of erroneously classified misses. The sum of these bad counts across all cpu/procedure/bin combinations is the total number of misses in error. The reported error fraction is the number of misses in error divided by the total number of misses reported by FlashPoint.

We compute error in this manner because it measures both effects that we are attempting to capture. The first is that TrapPoint, in general, reports more misses than the program would take if TrapPoint were not present. This is because the TrapPoint instrumentation pollutes the processor caches, causing more misses. We define the term *interference error* to mean error caused by TrapPoint changing the behavior of the monitored program, such as extra misses caused by TrapPoint cache pollution. This type of error tends to decrease as the sampling period increases. The other effect is that if the sampling period is long, then it is possible that samples do not occur frequently enough to get an accurate picture of program behavior. This is called *sampling error* and it increases as the sampling period increases.

A graph of error fractions at various sampling periods is shown in Figure 3.7. At low

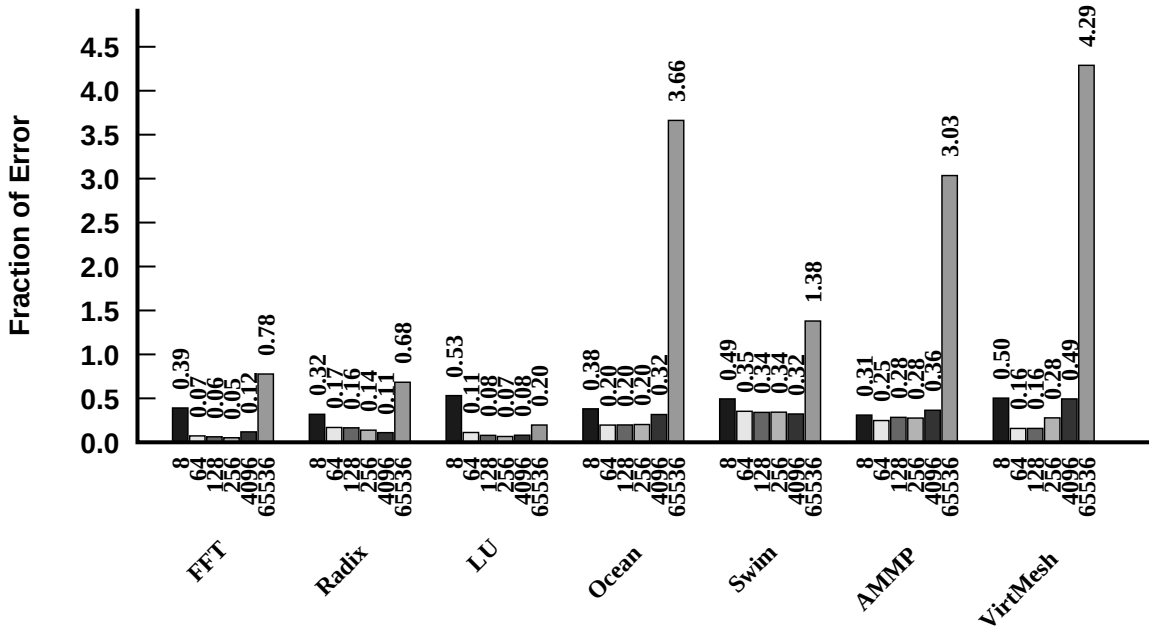

Figure 3.7: Error fractions for Sampled TrapPoint at several sampling periods.

sampling periods, interference error is the dominant effect, and error is reduced by increasing the sampling period. If the period is increased too much, however, then sampling error dominates, so the mid-range sampling periods of 128, 256, and 4096 perform the best across the applications.

Our results show that a sampling period of 65536 is inadequate for capturing the behavior of these programs, particularly Ocean, AMMP, and VirtualMesh. There are simply not enough samples for each cpu/procedure/bin combination to get an accurate performance estimate. Longer running applications would tend to improve the accuracy of the longer sampling period, but a performance monitor that was only accurate for very long runs would be difficult to use in many performance debugging situations.

We believe that in general, the best sampling periods are 256 and 4096. A period of 256 produces the most accurate memory profiles, while a period of 4096 has slightly more error but also slightly lower overhead. Since Sampled TrapPoint is a performance monitor, some error is tolerable. The real concern is that it reports memory profiles that are “good enough” to find performance problems in applications. With sampling periods of 256 or

4096 , it generally does. All important real effects show up in the data, though occasionally some anomalous results may cause the user to search for problems that do not exist.

3.5.3 Evaluation

Though none our measured results with Sampled TrapPoint met our goal of 20% overhead, when we disregard protocol overhead (as it would not be present with real hardware miss traps), a sampling period of 256 yields overheads of 15-28% and a period of 4096 yields of 9-17%. These are acceptable overheads and would not preclude TrapPoint's use in many situations. This overhead could be reduced further if the miss traps were implemented in hardware in such a way as to not require context switches. A fundamental problem, however, would remain: TrapPoint runs monitor code on the processor being monitored, and it therefore distorts its behavior. Multiprocessor applications, particularly when they experience poor behavior such as cache-to-cache transfers or hotspotting, can be quite timing dependent in that their performance depends on the specific interleaving of events, and TrapPoint alters these.

TrapPoint with a period of 256 or 4096, despite its limitations, is an acceptable memory profiler. The memory profiles produced by a period of 256 or 4096 are accurate enough to use, provided that the user keep in mind that some results may just be artifacts of the monitoring. It is clearly superior to any existing implementation. Its main distinguishing feature is that the only required hardware support is the addition of a trapping cache miss to the processor. The next chapter describes FlashPoint, a memory profiler that requires more hardware support but addresses many of TrapPoint's shortcomings.

Chapter 4

Node Controller-Based Memory Profiling

As an alternative to processor monitoring, we can perform memory profiling on the system node controller. To understand why this is both possible and desirable, we examine the required support for hardware memory profiling.

1. A triggering mechanism is needed to activate the profiler when cache misses occur.
2. Handler code needs to run when triggered.
3. The handler needs access to state storage to record the events it monitors.

For instance, in TrapPoint, the cache miss traps provide the triggering mechanism, the interrupt service routine serves as the handler, and normal memory provides the state storage. We notice that the required hardware support already exists in the node controller of a cache-coherent shared-memory multiprocessor since *these same criteria are required to implement cache coherence!* [17]

When a processor in a shared-memory multiprocessor misses in its caches, it sends a request to the system node controller (MAGIC, in the case of FLASH). Thus, the arrival of a memory request at the node controller serves as a triggering mechanism for cache misses. In the FLASH multiprocessor, MAGIC runs a section of code, called a handler, in response to a cache miss, which satisfies criterion 2. In full hardware implementations, such as the SGI Origin 2000, the handler is actually a hardware state machine, but its required actions

are the same. The handler, either hardware or software, needs access to the coherence state of the cache line and to memory.

The system node controller seems ideally suited to memory profiling. By using the node controller as a coprocessor, we can remove the burden of memory profiling from the compute processor. This offers advantages in both performance and accuracy. There is considerable performance gain since the processor is never interrupted — it can continue to execute application instructions while profiling is occurring. Accuracy also improves since the profiler is not running on the compute processor, polluting its caches. Memory profilers that run on the node controller are also able to collect extended memory profile information, which is impossible for processor-based tools such as TrapPoint.

This chapter discusses FlashPoint, a performance monitor that uses FLASH to point out program memory behavior. FlashPoint is implemented by modifying the cache-coherence handlers that run on MAGIC. On systems without a flexible node controller, memory profiling would require additional hardware. We believe this additional hardware support, however, would be minimal since the actions required by FlashPoint are quite similar to those the node controller already performs.

4.1 FlashPoint Implementation

FlashPoint consists of instrumentation at three levels of the system: the cache-coherence protocol, the kernel, and the application.

4.1.1 Protocol

We redefine the Stats Record structure as shown in Figure 4.1. This represents aggregate counts of common protocol events and is very similar to the structure described for TrapPoint, shown in Figure 3.3. In addition to the fields collected by TrapPoint, FlashPoint also collects information on TLB misses to local and remote addresses. The mechanics of how we collect these statistics is described below. Just as in TrapPoint, we allocate a two-dimensional Stats Record Array on each node, indexed by procedure number and bin number. The memory footprint for the Stats Record Array is twice as large for FlashPoint as for TrapPoint because of the extra TLB fields (for fast indexing, the Stats Record should be a power of two, which requires some padding). The main difference is that FlashPoint

```
struct StatsRecord {
   uint64 LocalReadMissCount;
   uint64 LocalWriteMissCount;
   uint64 RemoteReadMissCount;
   uint64 RemoteWriteMissCount;
   uint64 TLBLocalMisses;
   uint64 TLBRemoteMisses
   uint64 Padding[2];
}
```

Figure 4.1: Stats Record definition for FlashPoint

allocates the Stats Record Array in memory that is reserved for the cache-coherence protocol, while TrapPoint allocates it in kernel memory.

The cache-coherence protocol is instrumented to count each type of cache miss as it is serviced. A code example of how this instrumentation works is in Figure 4.2. When a load instruction causes the processor to miss in its cache, it sends a `GET` request to the MAGIC on the local node. MAGIC receives a `GET` message through its processor interface (called the PI). It then must schedule an appropriate handler to run on the embedded protocol processor. The hardware dispatch mechanism does this by examining the message type (`GET`), the source interface (the PI), and the address to determine whether it is local or remote. In the event that MAGIC receives a `GET` request on the PI to a local address, it will schedule the handler `PILocalGet`, some of which is shown in Figure 4.2.

Before it can take any action, `PILocalGet` must first determine the current coherence state of the cache line. The protocol maintains a data structure, the *directory*, that contains the coherence state of each local cache line. That is to say that each cache line in the system has one directory entry (also called its head link), and it resides on the home node for that cache line. Since `PILocalGet` runs only for local addresses, the protocol can simply read the head link from the local memory, which is accomplished by the `READ_HEADLINK` call. The simplest case for `PILocalGet` is the case where local memory contains the latest copy of the data: no invalidations are pending for the line and no processor's cache contains a dirty copy of the line. This is the case shown in Figure 4.2. For this case, the protocol should simply return the data to the processor and update the head link to show that the processor cache on the local node contains a copy of this cache line.

```
PILocalGet(addr) {
  // Read directory entry
  headlink = READ_HEADLINK(addr);
  if(!headlink.pending && !headlink.dirty) {

    // Common protocol case - latest copy resides in memory
    // Simply return the data to the processor

    PI_SEND(data);

    // Data transfer to processor now in progress
    // Following code is overlapped with transfer

    // Mark local node as a sharer
    headlink.local = 1;

    // Added FlashPoint instrumentation
    StatsRecord[procedure_num][headlink.bin].
                LocalReadMissCount++;

  } else {
    ... more complex protocol cases ...
  }
}
```

Figure 4.2: The `PILocalGet` protocol handler runs in response to a processor's `GET` request for local memory.

MAGIC was designed for high-performance, and even this short handler exhibits substantial concurrency. Before the `PILocalGet` code is even scheduled to run, MAGIC speculatively initiates a data transfer from the local memory into one of its internal data buffers. The call to `PI_SEND` will begin a data transfer from this internal data buffer, through the processor interface (the PI), to the processor. Protocol code that is executed after the `PI_SEND` will therefore not affect the latency of the memory request *at all*, though the handler must run to completion before MAGIC will be able to schedule another handler on the protocol processor.

Updating the head link to mark the local node as a sharer does not need to happen before the `PI_SEND`, and placing this code after the `PI_SEND` takes it off the latency path. Similarly, the code added for FlashPoint instrumentation is not on the latency path of the handler. The entire instrumentation involves only an array access and an increment operation, and it does not affect the memory latency. The procedure number and bin number used to index the Stats Record have the same meaning as in TrapPoint — exactly how they are determined is described later in this section. Note that no computation is necessary to determine the type of miss (Local Read), since `PILocalGet` is only executed in response to local read misses. Similar handler instrumentations are used to count remote read misses, local write misses, and remote write misses.

4.1.2 Data Structures

Just as in TrapPoint, FlashPoint needs a mechanism to associate physical address with application data structures. In fact, FlashPoint uses the identical application instrumentation; the application associates bin numbers with its data structures and communicates these mappings to the kernel via system calls. However, the kernel instrumentation in FlashPoint is different.

The principal difference is that the bin map (Section 3.2.2), which associates bins with physical addresses, needs to be accessible by the cache-coherence protocol, rather than by a kernel trap handler. Recall that maintaining the bin map is a two-step process. First, the kernel must associate bin numbers with application virtual addresses, since the application specifies virtual address ranges for each bin. This mapping is done by the region bin array, which is identical in FlashPoint and TrapPoint. The second step is to associate bin numbers with physical addresses (i.e., update the bin map) when virtual-to-physical mappings are

created or destroyed. In TrapPoint, the bin map is maintained as an array of bin numbers on each node, indexed by bits $31:8$ (see Figure 3.4) of the physical address. While the arrangement would be possible with FlashPoint, it would suffer from poor performance.

The layout of the bin map is nearly identical to the layout of the directory, and FlashPoint makes use of this similarity. When a MAGIC handler runs on the home node, it must retrieve the directory entry for that cache line. The directory is implemented as an array of entries (each entry is 8 bytes) in a region of memory reserved for the coherence protocol. The protocol uses bits 31:8 of the physical address as an index into the array of directory entries. This is exactly how the `READ_HEADLINK` macro, shown in Figure 4.2, is implemented. This macro is always called by handlers that run on the home node, since the directory state is needed to determine which coherence action to take. Since the bin map lookup mechanism is identical to the directory entry lookup, we combine the two into a single operation. The cache-coherence protocol on FLASH has a directory entry format that contains ten unused bits. FlashPoint simply uses these bits to store a bin number. This implies that FlashPoint does not need to perform any explicit bin map lookup — the bin number is contained in the directory entry which is already loaded into a MAGIC register when the handler starts.

Since the directory stores information on a per-cache line basis, it is the natural granularity for binning. This is the reason, alluded to in Section 3.2.2, that each cache line has its own bin number. Having a different granularity than the directory would require FlashPoint to maintain an explicit bin map, and the additional lookups would adversely affect performance. There are other performance implications of storing bin numbers in memory reserved for the cache-coherence protocol. Although the protocol is capable of reading and modifying the contents of normal (i.e., processor-accessible) memory, it is much more efficient for it to use memory that is inaccessible to the processor. One reason is that the memory reserved for use by the protocol is not cache-coherent, so using it saves the overhead of running coherence handlers. Also, MAGIC has a data cache that can be used to reduce the observed memory access time. Cached memory accesses, however, can only be used by MAGIC when accessing its reserved section of memory since MAGIC's data cache is not coherent with the processor caches.

The differences between the kernel instrumentation in FlashPoint and TrapPoint are that the FlashPoint kernel does not allocate space for a bin map and that any place the

TrapPoint kernel would update the bin map, FlashPoint sends a directive to MAGIC. These directives take the form of uncached stores to special addresses that MAGIC interprets as commands. They invoke specialized handlers on MAGIC that write bin numbers into the directory entries. These handlers are expensive in terms of performance since they can occupy the MAGIC protocol processor for an extended period of time while they write bin numbers to multiple directory entries. Fortunately, this operation is rare as it only occurs during initialization and paging.

The FlashPoint method of storing bin numbers in the directory entry is attractive since it allows misses to local addresses to be categorized by data structures without any impact on the local memory latency. It is also illustrative of the similarity between the primitives required for cache coherence and those required for memory profiling. Figure 4.2, for instance, shows how the bin number is available without any lookups or computation. There is some latency penalty, however, for remote accesses because the requester always updates the Stats Record when it sends data to the processor and only the home node has access to directory state. A handler that runs on the home node must include the bin number in any network messages it sends so that the requester will have access to it when it updates the Stats Record. Forming the network messages is on the critical path of a remote cache miss, so there is a non-zero FlashPoint overhead for such misses. A protocol example of how remote misses are implemented is shown later in this section.

4.1.3 Procedures

FlashPoint uses the same procedure numbering scheme as TrapPoint. The only difference is that with FlashPoint, the current procedure number is stored in a protocol variable, rather than a kernel variable. This means that the application macros that set the current procedure number now contain uncached store instructions. These uncached stores are interpreted by MAGIC as commands to set the node's current procedure number.

The code in Figure 4.2 is actually slightly inaccurate. The code that updates the FlashPoint statistics is shown as a two-dimensional array access.

```
StatsRecord[procedure_num][headlink.bin].
   LocalReadMissCount++;
```

Note that the index of the first dimension changes only when the current procedure number

changes, so the offset in the first dimension does not need to be computed in each handler. In the actual implementation, the protocol stores the pointer value of the first index, rather than `procedure_num` directly, to speed the indexing. The application macros that set the current procedure number actually send MAGIC a delta between the current procedure number and the old procedure number. MAGIC then increments its pointer by

```
delta * MAX_BINS * sizeof(struct StatsRecord)
```

Since the latter two terms are not only constants, but also powers of two, the pointer increment translates into only two MAGIC instructions: a shift and an add. This keeps the MAGIC overhead for procedure changes to a minimum. Since the real protocol code to access the correct Stats Record is much less readable than the code in Figure 4.2 but has the same effect, we use the two-dimensional array access in sample protocol code for the sake of clarity.

Even with the aforementioned performance optimizations, under certain circumstances there can be noticeable overhead due to procedure instrumentation. The application macros which set the procedure number contain 14 assembler instructions that maintain a stack of procedure numbers and do an uncached store of the procedure number delta. Each procedure contains two such macros (call and return), so 28 instructions are added to each procedure. For many procedures, 28 instructions is insignificant, but it can be noticeable overhead for very short procedures.

Procedure calls can be frequent, so performance dictates that we make one more kernel modification. The kernel normally does not allow user programs to access hardware via uncached stores for obvious reasons. In this case, however, we need to send an uncached store at the beginning and end of every procedure. The normal mechanism for user code of making a system call, thus allowing the kernel to make the restricted accesses, is inadequate since transitioning into and out of kernel mode is a very expensive operation that could take thousands of processor cycles. The resultant overhead would make procedure instrumentation impractical. Instead, we modify the kernel to allow a user application to map MAGIC registers uncached directly into its virtual address space. This gives us the necessary performance, but it unfortunately has security implications since any user program can now crash the machine by hostile writes to MAGIC register state. Even if the FlashPoint registers were segregated on “safe,” mappable pages, the user could still wreak havoc, for instance, by storing an out-of-bounds value for the procedure number.

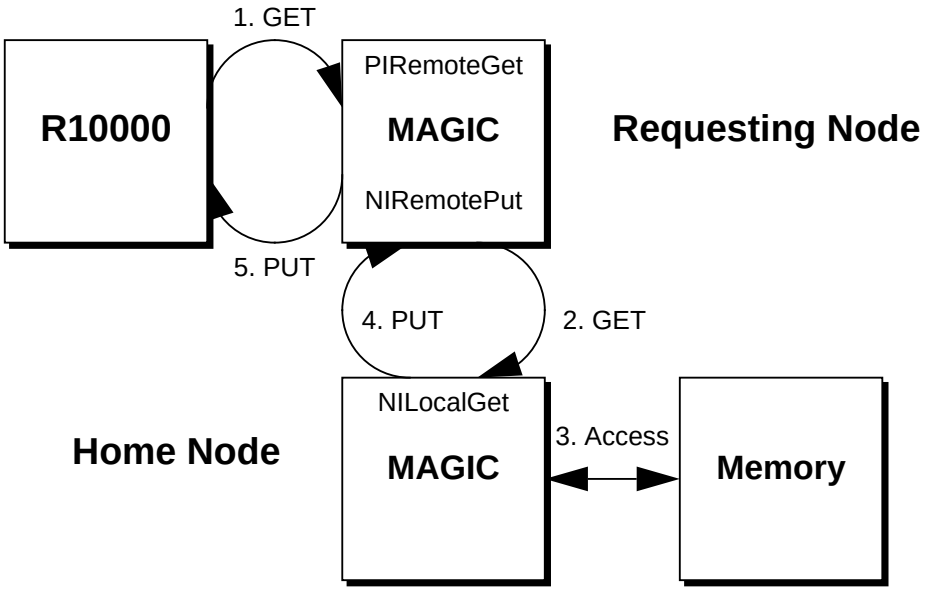

Figure 4.3: Handlers and Messages invoked on a remote read miss.

If FLASH were a commercial machine, writes to MAGIC registers would have to be error-checked. A unique fragment of MAGIC handler code runs for each register access, and this code could easily be augmented to perform the necessary checks. Error checking, however, substantially increases the overhead of procedure instrumentation, and since we do not wish to pay the performance penalty, we accept this small amount of vulnerability. After all, if a user is intent on crashing a machine, he can usually find a way to do it, even on a commercial system. We do not view this as a limitation of our technique, however, as a full hardware implementation of FlashPoint would be able to perform the error check with no performance overhead.

4.1.4 Remote Miss Example

The protocol code example shown in Figure 4.2 is the simplest protocol case since no remote nodes were involved. The protocol instrumentation is more complex for remote transactions because FlashPoint statistics are counted on the requesting node, but the bin number information is stored with the directory at the home node. This section describes an example of how FlashPoint counts a remote cache miss.

Figure 4.3 shows the messages and handlers that run in response to a read miss to

remote memory. The following is a description of how the system responds to a remote read miss, even without FlashPoint.

1. The requesting processor misses in its cache and sends a `GET` request to its local MAGIC. In response, MAGIC schedules the `PIRemoteGet` handler, so named since the request entered MAGIC through the PI (the processor interface), it was to a remote address, and it was a `GET` request. The handler realizes that it cannot service this request, so it forwards the `GET` request to the home node.

2. The `GET` request arrives at the home node's MAGIC chip, which schedules the `NILocalGet` handler since the request arrived through the network interface, the NI, and was a `GET` request to a local (to this node) address.

3. The simplest case for `NILocalGet` is when a valid copy of the requested address exists in memory. In this case, it retrieves the cache line from the local memory. As the data returns from memory it formats a `PUT` reply with the data and sends it back to the requester. Before the handler exits, it must add information to the directory that marks the requesting node as a sharer of the cache line.

4. The `PUT` reply arrives at MAGIC on the requesting node. It schedules the handler `NIRemotePut`, which simply forwards the reply back to the processor.

5. The processor receives the `PUT` reply and the memory request is satisfied.

The FlashPoint instrumentation is constrained by the fact that *only* the `NILocalGet` handler can look up the bin number. The bin numbers are stored in the directory, the directory resides on the home node, and `NILocalGet` is the only handler that runs there. Likewise, FlashPoint statistics are always updated as the reply is being sent to the processor, so this update must occur in the `NIRemotePut` handler.

The FlashPoint instrumentation code for a remote read miss is shown in Figures 4.4 and 4.5. The code for `PIRemoteGet` is not shown since it contains no FlashPoint instrumentation. The `NILocalGet` handler, shown in Figure 4.4, is the handler that runs on the home node. Its FlashPoint instrumentation extracts the bin number from the directory entry (the head link) and adds it to the reply message header. This only adds a small amount of overhead, but it is on the latency path, since it occurs before the `NI_SEND`. This means

```
NILocalGet(addr) {
  headlink = READ_HEADLINK(addr);   // read directory entry
  msg.len = LEN_CACHELINE;          // reply will have data
  msg.msgType = MSG_PUT;            // reply will be a PUT

  if(!headlink.pending && !headlink.dirty) {
    if(headlink.numsharers == 0) {
      // no previous sharers

      // Added FlashPoint instrumentation
      msg.bin = headlink.bin;

      // send the reply
      NI_SEND(msg, data);

      // not on latency path
      // mark the requesting node as a sharer
      headlink.ptr = msg.source;
      headlink.headPtr = 1;
    } else {
      ... more complex protocol case ...
    }
  } else {
    ... more complex protocol cases ...
  }
}
```

Figure 4.4: The `NILocalGet` protocol handler runs on the home node to service remote memory requests.

```
NIRemotePut(addr, data, msg) {
  PI_SEND(data);

  // Added FlashPoint instrumentation
  StatsRecord[procedure_num][msg.bin].
                          RemoteReadMissCount++;

}
```

Figure 4.5: The `NIRemotePut` protocol handler runs on the requesting node in response to data reply.

that FlashPoint does add a small amount of latency to remote memory accesses. When the `PUT` message is received by the home node, it runs `NIRemotePut` shown in Figure 4.5. The code is very simple, since it only needs to forward the reply to the processor. The FlashPoint instrumentation increments the appropriate entry in the Stats Record. The only difference between this update and the update for the local case, shown earlier in Figure 4.2, is that the bin number is extracted from the network message, rather than from the directory entry.

4.1.5 TLB Misses

Poor memory locality causes TLB misses in addition to cache misses. TLB misses can be an important effect, yet many programmers often overlook TLB performance. We found it useful to enable FlashPoint to track TLB misses as well as cache misses.

Counting TLB misses is not as natural an application of MAGIC as cache-miss counting because TLB misses are processor events — MAGIC is not usually aware of their occurrence. Since TLB misses on the R10000 are handled in software, we can instrument the TLB miss handler to inform MAGIC. We simply do an uncached store of the page table entry (which is already in a processor register). The page table entry contains a physical address, which MAGIC can use to extract a bin number. Unfortunately, the page table's physical address is always page-aligned, so MAGIC is forced to use the bin number of the first cache line in the page. Though this entails some loss of accuracy, we have yet to find a real application where this is problematic.

The TLB miss handler is quite performance critical, so even the small amount of instrumentation can cause noticeable overhead. It is therefore an optional part of FlashPoint. The kernel is compiled with two copies of the TLB miss handler, one of which is instrumented. The machine boots with the standard (i.e., uninstrumented) handler. When a FlashPoint-instrumented application starts, it checks the environment to see if the user desires TLB instrumentation, and if so, it makes a system call that rewrites the interrupt vector to use the instrumented TLB miss handler. This method ensures that the overheads of TLB miss instrumentation are only seen when the instrumentation is desired.

Though it would have been possible to instrument the TLB miss handler in TrapPoint, this was not done since it could not have been done efficiently. The miss handler is a very short piece of code that can be called often, so adding even a few instructions causes substantial overhead. The handler only has two available processor registers, since the authors of the operating system recognized that spilling user registers to memory would cause unacceptable performance. The fact that the code is extremely register-limited is of no consequence to FlashPoint. One of the registers already contains the page table entry. FlashPoint temporarily uses the other to hold the address of the uncached store. The more complex operations of looking up the bin number, reading the procedure number, and updating the statistics take place on MAGIC, so they occur in parallel with the miss handler and do not use processor registers. TrapPoint, however, runs on the processor, so TrapPoint TLB miss instrumentation could not run in parallel with the TLB miss handler, and it would have to spill user registers to memory to do its computation. The overhead of collecting TLB miss data on TrapPoint is therefore prohibitive.

The technique of using uncached stores to inform MAGIC of processor events is a general technique that can be used to count a wide variety of events. Adding uncached stores to the appropriate kernel code could, for instance, be used to count page faults or context switches.

4.2 Overhead

FlashPoint is designed to have as little performance impact as possible. Nevertheless, there are some overheads. These are caused by instrumentation of the cache-coherence protocol, instrumentation of the application, and instrumentation of the TLB miss handler. These

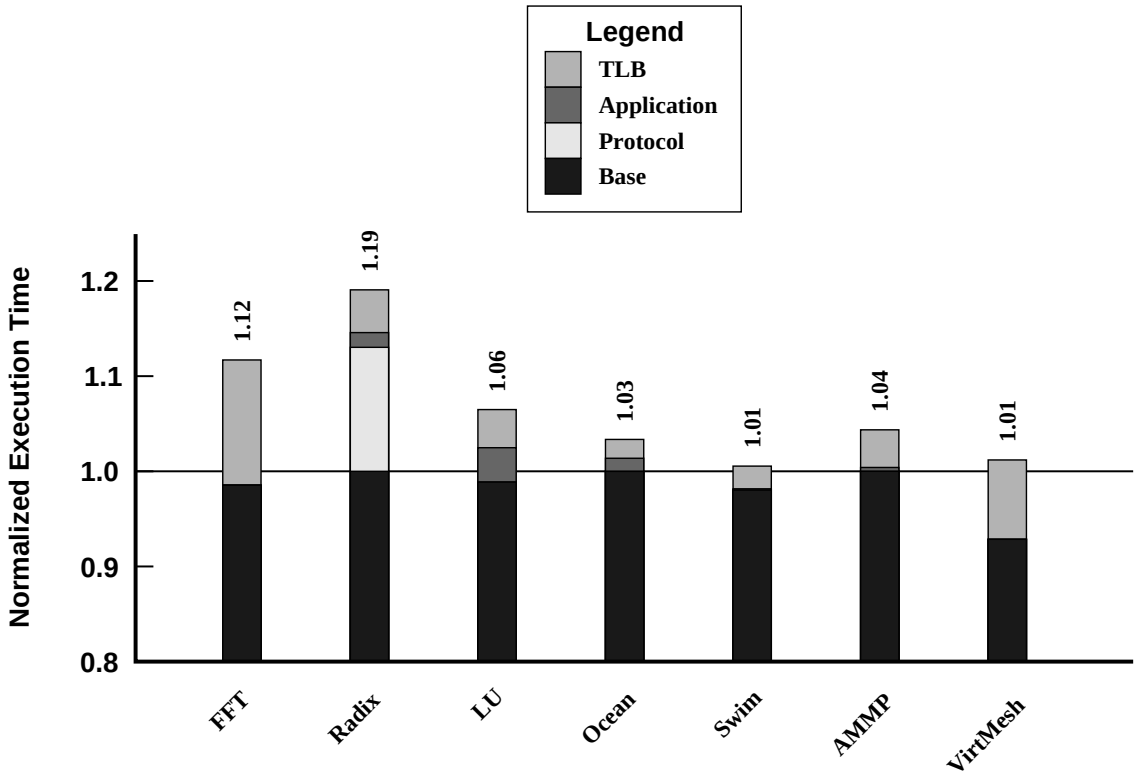

Figure 4.6: Overheads for FlashPoint. Execution times normalized to an uninstrumented run on an uninstrumented system. Note that the graph does not start at zero to make the overhead breakdown more visible.

overheads are shown in Figure 4.6. The base component is the execution time of an uninstrumented program on an uninstrumented system. The protocol, application, and TLB components are the overheads due to cache-coherence protocol instrumentation, application instrumentation, and TLB miss instrumentation, respectively.

The bars in Figure 4.6 are execution times normalized to an uninstrumented run on an uninstrumented system, so if FlashPoint had no overhead, all bars would have a height of 1.00. Note that to make the overhead breakdown more visible, the axis does not begin at zero, and this has the effect of making the overheads appear larger than they are. The data were collected by first running a base case (i.e., no instrumentation), successively adding more instrumentation, and subtracting the execution times. The anomalous result of the base being less than 1.00 for many of the applications is due to negative protocol overhead,

which is explained later in the section. Negative protocol overhead is shown on the graph by reducing the height of the base component.

Figure 4.6 shows that the FlashPoint overhead is quite small, with the highest reported overhead of 19%, including TLB instrumentation. Without the TLB instrumentation, the highest overhead is only 15%. Unlike Sampled TrapPoint, FlashPoint is able to count *every* cache miss with only a small cost in performance.

4.2.1 Protocol Overhead

There is a small performance degradation due to the cache-coherence protocol instrumentation. As mentioned previously, there is zero overhead for a local access that can be serviced from local memory because the added instrumentation is not on the latency path of the handler. Remote accesses generally have a 5-7% overhead. Although FlashPoint adds little to the latency of a memory access in isolation, FlashPoint instrumentation does occupy MAGIC's embedded protocol processor for several cycles. This has no effect on accesses that occur in isolation because the time is overlapped with data transfer. The increase in occupancy, however, will delay an incoming request that has already arrived and is waiting to run a handler, since a new handler cannot be scheduled until the current handler relinquishes the protocol processor. For this reason, the 5-7% remote memory latency increase is not an upper bound on protocol overhead. In practice, however, MAGIC occupancy caused by FlashPoint has not been a serious problem.

Figure 4.6 shows that the the protocol overhead is at most 13% for the applications under study. Radix, which exhibits this high overhead, has mostly bursty, remote write traffic, so this high overhead is a combination of the increased remote miss latency and some modest protocol processor occupancy effects.

The protocol overhead in many of the applications is actually *negative*. This result is counterintuitive since FlashPoint instrumentation should only make the protocol slower. The cause of this behavior has to do with the MAGIC hardware. MAGIC has a small (16 kB), direct-mapped instruction cache for protocol code. The line size is 128 bytes, which only leaves ($\frac{16\ kB}{128B} = 128$) 128 cache lines, so instruction cache conflicts are a serious concern. The protocol processor running the base protocol normally spends about 3% of its non-idle time servicing instruction cache misses. Different applications exercise the protocol in different ways, however, so higher overheads are possible. Some applications

run faster with a FlashPoint protocol because with their access patterns, the FlashPoint protocol has fewer instruction cache conflicts than the base protocol. This is simply a MAGIC implementation artifact, however, as all applications would have positive protocol overhead if only MAGIC had a larger instruction cache.

4.2.2 Application Overhead

There is negligible overhead for instrumenting data structures for the vast majority of applications, since the overhead of allocating bins is done only when data structures are first instrumented and when they are paged in. Most applications will instrument the important data structures during initialization, so the cost is only incurred once. Although writing bin numbers into MAGIC directory entries is a time-consuming operation, after initialization it only occurs during paging. Paging is both rare and expensive, so this overhead is also insignificant.

The real cause of the overhead due to application instrumentation is the procedure call and return macros. These macros translate into 14 assembly instructions, including an uncached store. For large procedures, this overhead is negligible, but for very short procedures, it can be considerable. The overhead due to application instrumentation is proportional to the dynamic frequency of instrumented procedure calls, and this can be controlled by the user by selectively not instrumenting certain procedures.

We have found it advantageous not to instrument very short procedures. This not only reduces overhead, but it also improves the quality of results from the point of view of understanding program behavior. Not instrumenting these procedures has the effect of attributing misses to their callers. If every procedure in LU is instrumented, for instance, the results show that nearly all misses occur in the procedure `daxpy`. That function is a simple vector operation, and since it is used to do nearly all of the computation, this result is not helpful. It is much more important to determine which calls to `daxpy` are problematic. By not instrumenting `daxpy`, we not only reduce the overhead, but we also obtain a more high-level view of where in the application misses occur. The results in Figure 4.6 for LU are for the version with no instrumentation in `daxpy`.

The results in Figure 4.6 indicate that the impact of application instrumentation varies across the applications. FFT, Swim, and VirtualMesh, for instance, have a low dynamic frequency of procedure calls, so the cost of application instrumentation is near zero. Radix

and LU have the highest amount of per-procedure overhead (3.6% for LU and 1.5% for Radix), because they only operate on small amounts of data a time. LU uses small blocks to optimize L1 cache performance, and Radix uses a relatively small radix of 32, necessary for good TLB performance (see Section 2.2.1). These optimizations, however, cause the program to do only a small amount of work in each phase, thus increasing the frequency of procedure calls. The other applications have negligible application overheads. While pathological cases for procedure overhead are possible, they can *always* be resolved by removing instrumentation from troublesome procedures, thus attributing the misses to the procedures' callers.

4.2.3 TLB Overhead

Instrumentation of the TLB miss handler significantly increases the cost of a TLB miss, so the overhead seen by this instrumentation is proportional to the number of TLB misses. Figure 4.6 shows that FFT has by far the worst TLB miss overhead (13%). This is because the Transpose phase in FFT, even with the blocking fixes described in Section 2.2.1, has a substantial number of TLB misses with large matrices. The other applications show much smaller overheads due to the TLB.

4.2.4 Memory Overhead

Though FlashPoint comes at a small performance cost, it has an associated memory overhead. The overhead is in three data structures: the directory in the protocol, the protocol statistics, and the kernel's region data structure.

There is one directory entry for every cache line in the system, and FlashPoint maintains a ten bit bin number in each one. Since there are 128 bytes per cache line in FLASH, there is a memory overhead of 1% of the total machine memory ($\frac{10\ bits}{128\ bytes \times 8 \frac{bits}{byte}}$). On FLASH, a section of main memory is reserved for protocol directory and protocol data, so this memory overhead comes out of normal DRAM. On other systems, such as the Origin 2000, where directory memory is in fast SRAM, this cost would be more substantial.

In its current configuration, FlashPoint is configured to have room to store 128 bins and 64 procedures on each node. This can be modified by recompiling the cache-coherence protocol. The current size of the Stats Record Array is 64 bytes times the number of bins

times the number of procedures. This leads to a footprint of 1 MB per node. This is a part of main memory that is reserved by the protocol and is unavailable for other uses. Still, this is small compared to the total amount of memory in the system (256 MB per node) and can be reduced by recompiling the protocol should the need arise.

The other memory overhead is the kernel data structure that stores bin numbers for each region. This is an array of 2 byte bin numbers, one for each cache line in the region. This leads to an memory overhead of 1.5% ($\frac{2}{128}$). This overhead exists for any region that has non-zero bins. In the case of a region with no instrumented data structures, this array is never allocated. Also note parallel threads that share an address space (such as those created by `sproc`) also share regions. This means the memory overhead is not proportional to the number of threads.

4.3 Extended FlashPoint

In addition to the memory profiling statistics described above, an enhanced version of FlashPoint is able to record some transient protocol state with essentially the same overhead as base FlashPoint. The result is a low-overhead extended memory profiler.

4.3.1 Protocol Extensions

The Extended FlashPoint protocol collects several statistics in addition to the normal FlashPoint memory profile.

Invalidations. In a multiprocessor environment, cache misses can be caused by actions on remote processors. These are known as coherence misses. For instance, if a processor reads a shared variable often, this variable will be stored in its cache. If a different processor, however, writes to this variable (or any part of the cache line containing the variable), it will send an *invalidation* message to all sharers. The reader will take a cache miss on its next access, even though it may have had no problems with capacity or conflicts.

When MAGIC receives an invalidation message, it runs a specific handler to service it. We can therefore count invalidations in this handler. The home node always sends the invalidations, and it adds the bin number to the network message header, so that nodes receiving invalidations are able to find the correct bin number.

Interventions. Another multi-processor event of interest occurs when a processor makes a request for a cache line and the only valid copy of the cache line is in the cache of another processor. In this case, MAGIC must retrieve the cache line from the processor's cache. This is called an *intervention* and can be a slow operation. For instance, on FLASH the observed round-trip time for a read request to remote memory is 75% longer when the data is dirty in the home node's cache than when the data can be read directly from the home node's memory (the time is longer still if the data is dirty in a third processor's cache).

Since interventions are slow, they can be of interest in performance tuning. They are counted by instrumenting the protocol at every point it sends an intervention request to the processor. As with invalidations, the bin lookup is done at the home, and the bin number is added to the network message header if the intervention is to take place on a different node.

Three-hop Misses. The other protocol case that we found interesting was for the case of a *three-hop miss*. This occurs when a processor does a request to remote memory. The home node's MAGIC, however, finds that the only valid copy of the cache line exists in a third node's cache. It sends a message to the third node, instructing it to intervene the line out of its cache and forward the result to the requesting node (there are other messages as well, but this is the critical path). This is the slowest protocol case, since its critical path includes handlers on three different nodes. Three-hop misses are a special case of interventions where the intervention does not take place at the home node. We find it useful to count three-hop misses in addition to interventions because the extra network messages involved in the extra hop add significantly to the cost of a cache miss.

The slow, three-hop case can be counted easily by using a bit in the network message header to signify the three-hop miss. This bit is set by the home node, and when the reply finally makes it back to the requester, it checks the bit to see whether or not to increment the three-hop counter.

Summary. These three protocol enhancements taken together: invalidations, interventions, and three-hop misses, are what we refer to as Extended FlashPoint. Extended FlashPoint collects protocol statistics over and above the straight categorization of cache misses. The full Stats Record data structure for Extended FlashPoint is shown in Figure 4.7.

```
struct StatsRecord {
   uint64 LocalReadMissCount;
   uint64 LocalWriteMissCount;
   uint64 RemoteReadMissCount;
   uint64 RemoteWriteMissCount;
   uint64 TLBLocalMisses;
   uint64 TLBRemoteMisses;
   uint64 Invalidations;
   uint64 Interventions;
   uint64 ThreeHopMisses;
   uint64 Padding[7];
}
```

Figure 4.7: Stats Record definition for Extended FlashPoint

Since indexing the arrays of Stats Record is performance-critical, the size of the Stats Record needs to be padded out to a power of two. This enables indexing using shifts instead of multiplications.

Extended FlashPoint offers substantially more detail about memory performance than FlashPoint. Cache-to-cache transfers, which cause interventions instead of memory accesses are a serious performance concern, as shown in Sections 2.2.4 and 2.2.5. The information on interventions and three-hop misses collected by Extended FlashPoint was essential in diagnosing the pathological memory behavior in AMMP and VirtualMesh. Note that extended memory profile data cannot be collected by a processor-based technique such as TrapPoint.

4.3.2 Overhead

Figure 4.8 shows overheads of both FlashPoint and Extended FlashPoint on the benchmarks. Again, all execution times are normalized to runs with uninstrumented applications on an uninstrumented protocol. The FlashPoint data is exactly the same as Figure 4.6 and is included here for comparison.

The extra code associated with Extended FlashPoint causes little extra latency. While Extended FlashPoint does occupy MAGIC for a few more cycles in some protocol handlers, none of our benchmarks ran significantly slower with Extended FlashPoint than with base

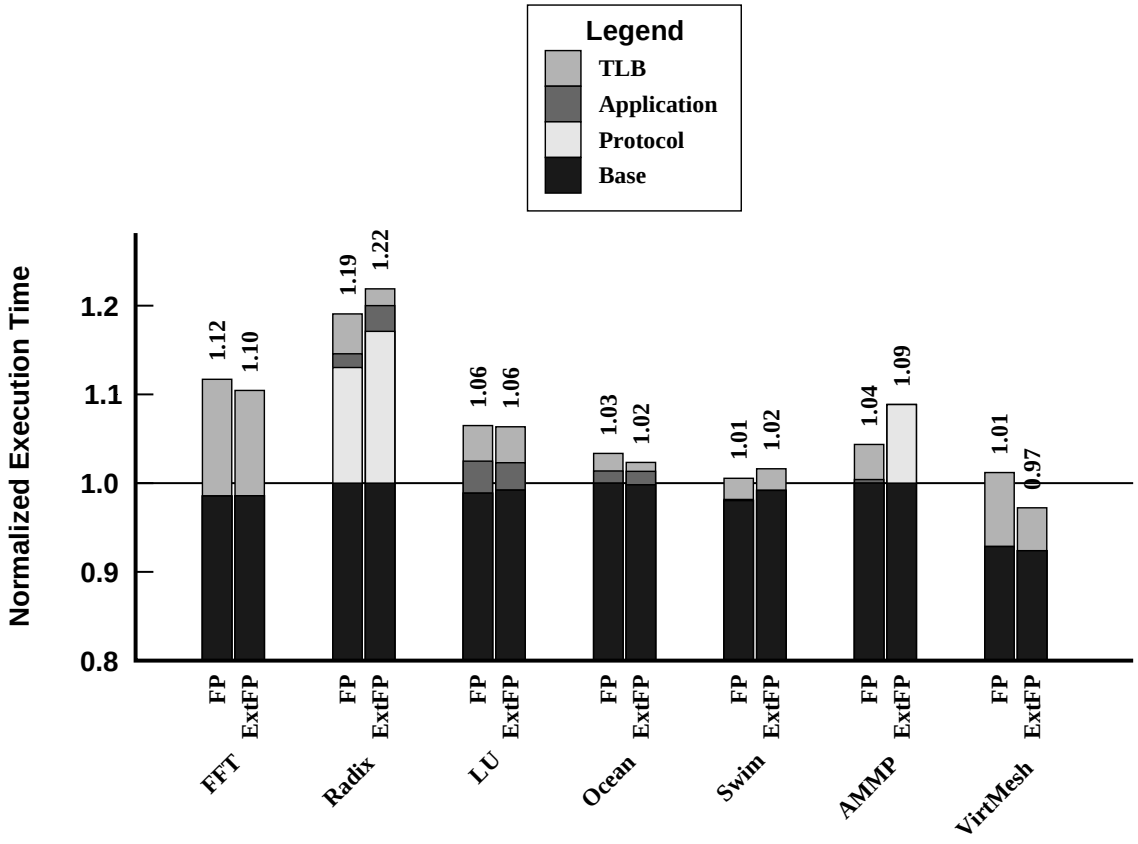

Figure 4.8: FlashPoint and Extended FlashPoint overheads.

FlashPoint. The expected result is that Extended FlashPoint should have slightly larger overhead due to the protocol, but that other overheads should be close to the same. This is the observed behavior for Radix and AMMP. VirtualMesh actually has slightly *less* overhead with Extended FlashPoint due to reduced MAGIC instruction cache conflicts.

Some of the applications show differences in the application instrumentation overhead and the TLB instrumentation overhead between FlashPoint and Extended FlashPoint. This is somewhat counterintuitive, since these components are unaffected by the extensions. These differences account for only a small fraction of total execution time, however. The extra protocol instrumentation simply causes a slightly different interleaving of events on the MAGICs, and this causes small performance variations.

Our results show that extended memory profile information can be collected quite efficiently, as the highest overhead seen by Extended FlashPoint was 22%. Extended FlashPoint does incur additional memory overhead: the size of the Stats Record array is twice that of basic FlashPoint.

Chapter 5

Conclusions

Memory profiling on hardware cache coherent machines has not been widely applied because no efficient implementation has existed. This thesis demonstrates prototypes of two efficient memory profilers. We show not only that an efficient implementation is possible but that the insights it provides into memory system behavior are extremely useful for identifying and fixing performance problems.

5.1 Usefulness

Since our memory profilers have been developed, we have found them to be incredibly useful in performance debugging. Extended FlashPoint, in particular, has been used both to study mature codes and to aid in writing new, efficient parallel programs. Several specific examples were discussed in Section 2.2.

FlashPoint has been used to get performance improvements even out of highly-optimized mature codes. For instance, it was used to obtain large performance gains in three SPLASH-2 applications: FFT, LU, and Radix. This was a quite surprising result since the behavior of these applications has been extensively studied. FlashPoint was also used to study applications for the SPEC OMP 2001 benchmark suite [24]. Silicon Graphics (SGI) is one of the major vendors interested in SPEC OMP, and since FLASH is binary compatible with their machines, we used FlashPoint to study applications that experience poor speedup on the SGI systems before the benchmark suite was released. FlashPoint was able to easily diagnose the problems in several of the supposedly well-tuned applications, and improved

```
#define FLASHPOINT_CALL_PROC(_name, _num) {    \
  FP_ProcedureNames[_num] = _name;             \
  ... push _num onto procedure stack ...       \
  ... send _num to MAGIC ...                   \
}
```

Figure 5.1: Original FlashPoint procedure call macro.

versions were sent back to SGI. In fact, these optimizations were used by SGI to make recommendations to SPEC on how the benchmarks could be modified to better utilize shared memory. The current SPEC OMP benchmarks reflect some of these changes.

As another example, VirtualMesh, a research code used by the Center for Turbulence Research at Stanford, has been in use for several years and has been highly tuned. Nevertheless, FlashPoint provided enough data to allow optimizations that give a 28% performance improvement on eight processors. This can shorten the execution of a typical run by as much as a week! Our experience in studying mature codes is that with few exceptions, FlashPoint can point out enough memory performance problems to allow at least 10-15% performance gains and sometimes much more.

With less optimized codes, the FlashPoint results are much more dramatic. FlashPoint enabling speedup of over an order of magnitude is not uncommon. When writing programs that access large amounts of data, it is quite easy to create large miss rates. With shared memory programs, the additional complexities of sharing patterns and data placement increase the chances of accidentally creating performance bottlenecks. Often small changes in code or data layout can lead to huge performance improvements as pathological behavior is eliminated.

FlashPoint was even used to fix a sharing problem in the FlashPoint instrumentation itself! The original implementation of the macro that sets the current procedure number is shown in part in Figure 5.1. The FlashPoint application library maintains a list of procedure names so that it can associate meaningful names with the procedure numbers used for data collection. Procedure names do not change over the course of a run and since the assignment of the string `_name` is simply a pointer assignment, the rationale was that it would be cheaper to assign to `FP_ProcedureNames` on every procedure call rather than to check to see if it was already assigned. This performance “optimization,” however, was disastrous

```
#define FLASHPOINT_CALL_PROC(_name, _num) {    \
  if(!FP_ProcedureNames[_num]) {               \
    FP_ProcedureNames[_num] = _name;           \
  }                                            \
  ... push _num onto procedure stack ...       \
  ... send _num to MAGIC ...                   \
}
```

Figure 5.2: Updated FlashPoint procedure call macro.

as multiple threads calling procedures simultaneously caused the cache lines containing `FP_ProcedureNames` to be moved around the system in a large number of cache-to-cache transfers. Extended FlashPoint immediately pointed out the problem by showing numerous three-hop, remote write misses to `FP_ProcedureNames`. This problem was easily fixed by not writing to `FP_ProcedureNames` if it has already been written. The updated version is shown in Figure 5.2. There is an extra branch, but it results in a large overall performance gain.

Our experience has been that if FlashPoint is being used on an application during its development, then large performance problems can be found and fixed soon after they are created. This sort of tool is invaluable since it works to mitigate a major disadvantage of shared memory. The chief advantage of shared memory over alternatives such as message passing is that shared memory makes having tight control over communication a performance issue, while for message passing, it is necessary for correctness. Shared memory thus provides a much more user-friendly programming paradigm. The argument in favor of shared memory is that the user only needs to carefully manage communication in those regions of code and on those data structures where it significantly affects performance. Unfortunately, finding such places is exceedingly difficult, making shared memory machines easy to program, but difficult to program efficiently. Memory profilers such as FlashPoint attack the core of the problem by making memory behavior easy to examine, diagnose, and fix.

Many of the codes mentioned in this work (and several others) were brought to our attention by users of SGI Origin systems who were trying to improve their program's performance. FlashPoint is ideally suited to this since the system architectures of FLASH and Origin are quite similar and the two platforms are binary compatible. *Every* one of these

programmers who saw FlashPoint and its associated visualization tool, Thor [3], was impressed by the fact that FlashPoint could find in minutes performance problems that had eluded them for weeks, months, or even years. They were quite disappointed, however, to learn that FlashPoint does not run on an Origin. It could run on future versions, though, if the requisite hardware support were designed into the system node controller.

5.2 Implementation

Efficient memory profiling requires hardware support because software is not able to identify cache misses without resorting to simulation. With the hardware support, however, we show that memory profiling can be done with little overhead. The hardware must be able to trigger the performance monitor on cache misses and allow the monitor access to the address that caused this miss. This support can be built into either the processor or the system node controller.

5.2.1 Processor Based

In Chapter 3, we discuss TrapPoint, a performance monitor that uses processor support for memory profiling. Sampling techniques must be used to achieve tolerable performance and accuracy, but we show that with a carefully-chosen sampling period (one every 256 or 4096 misses), this technique yields a useful memory profiler.

The advantage of this technique is that the required hardware support is rather modest, namely a cache miss trap. Such miss traps have already been proposed [10] for use in simpler performance monitors. If some sampling mechanism were added, then sufficient hardware support would exist for an efficient memory profiler.

Our work shows that the TrapPoint overheads are somewhat higher than our design goal of 20%, but when we factor out protocol overhead as an implementation artifact, we meet the design goal with a period of 4096 and come close with a period of 256. With hardware support absent in our prototype, this overhead could undoubtedly be reduced somewhat, but some amount of perturbation will always exist in any performance monitor that runs code on the monitored processor. The perturbation causes both overhead and accuracy concerns for TrapPoint arising from several sources:

1. The monitor pollutes the processor caches.
2. The monitor spills user data to memory to free registers for itself.
3. Instrumentation can significantly disturb the timing of multiprocessor events.
4. Monitor interference can cause the operating system to make different resource allocation and scheduling decisions.

In fact, much of the error reported in Section 3.5.2 is due to interference error. This is error not caused by TrapPoint incorrectly reporting on the behavior of the system but by the way TrapPoint has changed the system's behavior. In other words, TrapPoint correctly reports a memory profile, but it has modified the profile in the process. The magnitude of this problem is related to the quality of our implementation (which is necessarily limited by the lack of a hardware miss trap), but the existence of the problem is endemic to this sort of performance monitor. Sampling reduces, but does not eliminate, this effect.

It is impossible to collect extended memory profile information on the compute processor, since cache-coherence protocol transactions are not visible to the processor. We have found extended memory profile information to be valuable in the diagnosis of sharing problems that cause cache-to-cache transfers, such as false sharing. The inability to perform extended memory profiling is therefore a significant limitation of processor-based memory profiling techniques.

5.2.2 Node Controller Based

Our preferred method of memory profiling is to use the system node controller. FlashPoint, described in Chapter 4, works in this manner. It addresses many of the shortcomings of TrapPoint, but it requires more sophisticated hardware support.

The two main advantages of using the node controller for memory profiling are that removing monitoring code from the processor dramatically reduces the interference error and that the cache coherence transactions are visible to the monitor, thus permitting extended memory profiling.

FlashPoint and Extended FlashPoint only experience protocol overhead in two places: there is a small latency penalty for remote accesses, and contention for the MAGIC protocol processor can cause occupancy effects under high load. Even these small overheads

(at most 13% for the applications under study) can be reduced or even eliminated by a full hardware implementation since the requisite actions (adding the bin number to the network message and updating the Stats Record) could be done in parallel with the necessary computation. The other source of overhead, procedure instrumentation, is usually small, and it can always be reduced by the user by selectively removing instrumentation.

The low overhead of FlashPoint makes sampling unnecessary, so there is no sampling error. FlashPoint could possibly introduce a small amount of interference error because of protocol and procedure instrumentation. This interference, however, is quite small and does not in general pollute the caches because the procedure instrumentation almost never causes cache misses and the cache-coherence protocol does not even access the processor caches. Unfortunately, what little interference error may exist is impossible to quantify since the "correct" memory profile cannot be measured without using FlashPoint. The overheads of FlashPoint are well within the known errors of our architectural simulators [6], making any sort of simulation comparison meaningless. In several years of using FlashPoint to tune applications, we have *never* found an interference problem caused by FlashPoint, so we do not view this to be a relevant effect.

Extended memory profiling on the system node controller is both possible and quite easy, as shown by Extended FlashPoint. The additional information has proven useful in performance debugging when the application experiences many cache-to-cache transfers, perhaps caused by false sharing or excessive synchronization. Since these are warning flags of pathological memory behavior, we believe this data is extremely important. The overheads are so low that our default cache-coherence protocol for the FLASH machine has Extended FlashPoint instrumentation. Users not using Extended FlashPoint experience only the protocol overhead, which is small enough that they do not even notice it is present!

5.3 Conclusions and Future Work

Since memory profilers are so useful and our work shows that they can be implemented efficiently, we believe that future shared memory multiprocessors should be designed with hardware support for memory profiling. Our preferred implementation is an extended memory profiler with support in the system node controller, though a processor-based tool could

provide a reasonable approximation if node controller support were infeasible for a particular implementation.

Extended memory profiling can be implemented on a flexible machine such as FLASH with no additional hardware. Machines with a pure hardware node controller, such as the SGI Origin 2000 and Origin 3000, would require hardware changes, but we believe these changes to be minimal since the work required for memory profiling is so similar to the work these machines already perform for cache coherence. The main cost on such machines would be the memory required for storing both the bin numbers and the statistics. As shown in Section 4.2.4, the memory required for FlashPoint is only a small fraction of the total RAM, but the memory used for profiling must be easily accessible to the node controller. FLASH uses normal DRAM with caches for the directory and other protocol data, so the cost of the additional memory is of little concern. The SGI machines, however, use special SRAM for this purpose. An efficient profiler would need to store bin numbers into the directory and store its Stats Record in the SRAM, and since SRAM is substantially more expensive than DRAM, this could be an issue. Nevertheless, we believe the usefulness of the data that memory profilers can produce far outweighs the cost.

Though a flexible node controller is not required for memory profiling, our work has shown the flexibility of FLASH to be extremely useful. The design of FlashPoint and Extended FlashPoint evolved over a period of years. Some features, such as TLB miss counting, were added after we discovered a need (i.e., after finding several applications that were limited by TLB performance). Other features that were present in early iterations of the tools, such as Extended FlashPoint counting cold misses, were phased out when they were shown not to be useful (what exactly constitutes a “cold” miss makes much more sense in simulation than it does on hardware). This could not have been done if the cache-coherence protocol had been frozen in place when the machine was manufactured. One useful property of Extended FlashPoint, for instance, is that it can be easily extended. Granted, extending it requires that a user be fluent in the cache-coherence protocol, but if such a user suspects that some previously unmeasured protocol transaction is limiting performance, he can easily add code to the protocol to measure these transactions. This is impossible on other shared memory machines that lack this flexibility.

This thesis describes using FlashPoint to understand application behavior. It is equally

useful, however, for understanding cache-coherence protocol behavior. As more cache-coherence protocols are ported to the FLASH hardware, we expect that FlashPoint instrumentation will be added to them. For instance, even with simple cache-coherence protocols, designers can be faced with a situation such as to whether to fully handle a particularly difficult transient condition at substantial hardware cost or negatively acknowledge (NAK) such requests at perhaps considerable performance cost. In practice, simulation is used to provide the necessary data (i.e., the frequency of this condition) needed to make this trade-off. Simulation is only feasible for small data sets, so information gathered in this manner may or may not be representative of what will be run on the production machine. FlashPoint, however, could be used to collect this sort of data on hardware with full size data sets.

More complex cache-coherence protocols exhibit more complex performance issues, and we expect that much of the future work with FlashPoint will be to study them. For instance, a Remote Access Cache (RAC) [23] protocol can be used where a portion of local memory is reserved as a cache for remote addresses. Remote accesses first access this "cache", and if it hits, the remote access is avoided. Once this protocol is ported to hardware, FlashPoint could be used to study how different organizations of the RAC affect the miss rate. Similarly, MAGIC could be programmed such that part of memory is non-coherent, and block transfers could be efficiently implemented in this space [7]. When this is ported to hardware, FlashPoint could be used to study its performance. Such techniques have thus far only been studied in simulation, and the flexibility of FLASH and the availability of FlashPoint are the necessary components for studying complex protocols running on hardware with large data sets.

In summary, we have shown that efficient, accurate extended memory profiling is possible on distributed shared memory machines and that it requires only modest hardware support. Such profilers have proven invaluable for performance debugging on these machines. We have also shown the usefulness of flexibility in the FLASH machine for our prototypes, and we believe that our memory profilers will be useful in the future cache-coherence protocol research for which FLASH was designed.

Bibliography

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. An overview of the SUIF compiler for scalable parallel machines. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[2] J. Anderson, L. Berc J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, October 1997.

[3] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: A flexible computer systems visualization environment. *Computer Graphics*, 34(1), February 2000.

[4] M. Burrows, U. Erlingson, S-T.A. Leung, M.T.Vandervoords, C.A. Waldspurger, K. Walker, and W.E. Weihl. Efficient and flexible value sampling. In *Proceedings of the 9th Architectural Support for Programming Languages and Operating Systems*, pages 160–163, November 2000.

[5] H. Davis, S. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using tango. In *Proceedings of the International Conference on Parallel Processing*, pages 99–107, 1991.

[6] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich. FLASH vs. (Simulated) FLASH: Closing the simulation loop. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[7] J. Heinlein, K. Gharachorloo, R. Bosch, Jr., M. Rosenblum, and A. Gupta. Coherent block data transfer in the FLASH multiprocessor. In *Proceedings of the International Parallel Processing Symposium*, pages 18–27, 1997.

[8] J. Hennessy and D. Patterson. *Computer Architecture - A Qunatitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[9] S. Herrod. Tango lite: A multiprocessor simulation environment. Technical report, Department of Computer Science, Stanford University, 1993.

[10] M. Horowitz, M. Martonosi, T. Mowry, and M. Smith. Informing memory operations: providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 260–270, 1996.

[11] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[12] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[13] A. Lebeck and D. Wood. Active memory : A new abstraction for memory-system simulation. In *Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 220–230, 1995.

[14] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. Hillis, B. Kuszmaul, M. St Pierre, D. Wells, M. Wong-Chan, S. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[15] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–103, 1992.

[16] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[17] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating performance monitoring and communication in parallel computers. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 138–147, May 1996.

[18] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.

[19] MIPS Technologies, Inc. MIPS R10000 microprocessor user's manual, 1996.

[20] U. Prestor. Evaulating the performance of a ccNUMA system. Master's thesis, University of Utah, 2001.

[21] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[22] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus, and D. Wood. Fine-grain access control for distributed shared memory. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 297–306, San Jose, California, 1994.

[23] V. Soundararajan. *Flexbile Use of Memory for Replication/Migraion in Cache-Coherent DSM Multiprocessors*. PhD thesis, Stanford University, Mar 1999.

[24] SPEC OMP2001, 2001. http://www.spec.org/hpg/omp2001.

[25] A. Srivastava and A. Eustace. ATOM: A system for building customize program analysis tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, June 1994.

[26] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.

[27] Z. Xu, J. Larus, and B. Miller. Shared-memory performance profiling. In *SIGPLAN*, 1997.

[28] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings Supercomputing'96*, November 1996.