

Chapter 3

Multiprocessor Hardware Performance Monitoring

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

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

3.1 Performance Losses: Software View

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

3.1.1 Inefficient Algorithms

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

In addition to the standard algorithms used to improve performance, the programmer should be aware of many software transformations that can accelerate performance in cached-memory systems. Techniques such as loop interchanging, loop fusion, strip mining, blocking, merging arrays, and packing or padding structures all can improve locality and accelerate performance [Bacon94]. However, gaining insight into whether these options are useful requires insight into the hardware operation.

3.1.2 Extra Parallel Work

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

3.1.3 Synchronization

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

3.1.4 Load Imbalance

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

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

3.1.5 Inefficient System and Library Calls

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

3.1.6 Amdahl's Law

The final source of parallel performance loss is a result of Amdahl's Law [Amdahl67], which states that the speedup resulting from a specific performance optimization is limited by the fraction of time in which the optimization applies. Although Amdahl's Law is not a cause of lost performance per se, it explains why a parallel program does not track the ideal linear speedup curve. For example, suppose a parallel program is run on two proces-

sors in five seconds, of which one second is spent in unparallelizable sequential code. When run on 16 processors, one might expect the program to run eight times faster, *i.e.*, run in 0.625 seconds, but Amdahl's Law dictates that the best running time is 1.5 seconds. To further illustrate this, the speedup curve for this example is shown in Figure 3.1. By timing the sequential and parallel portions of a program, the amount of speedup lost due to the sequential portion of a program becomes obvious.

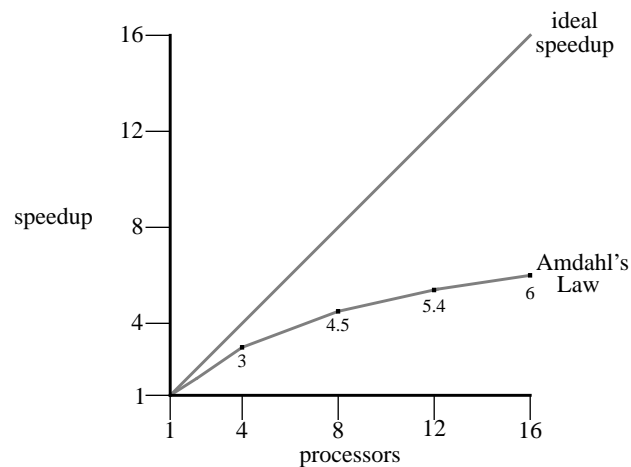


FIGURE 3.1. An example of Amdahl's Law limitations on speedup.

3.1.7 Summary

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

3.2 Performance Losses: Hardware View

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

3.2.1 Instructions

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

Although it is not generally possible to predict the minimum number of instructions possible, it is feasible to measure improvements in dynamic instruction count. Software tools such as Mtool are quite good at accumulating dynamic basic block counts, hence, instruction counts. Also, a number of processors and systems provide direct hardware support for counting instructions: Pentium, Alpha 21064, MIPS R10000, SuperSPARC II, PowerPC 604 and POWER2, MIT M-Machine, and Cray vector computers can all do this. Furthermore, both hardware and software systems can break down the instruction counts into groups of instructions, with software having the advantage of being able to form any number of arbitrary groups. On the other hand, hardware has the advantage of being completely transparent to the user. This is especially powerful in a machine like the Cray Y-MP C90, where each counter monitors only one event, because one can profile the

machine use for workload characterization and benchmark synthesis with no impact on users [Gao95].

3.2.2 Instruction Scheduling and CPI

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

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

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

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

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

3.2.3 Memory Stalls

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

3.2.4 Other Stalls

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

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

Existing processors count these events with varying usefulness. For example, the Pentium counts pipeline write buffer stalls, snoops and snoop hits. Similarly, MIPS R10000 and PowerPC can count snoops and snoop hits. Additionally, the SuperSPARC outputs a signal while the write buffer is full and the MIPS R4400 outputs a signal for coherence

‘multiprocessor stalls’ and another for ‘other stalls’, presumably due to write buffer overflows and uncached-memory operations. The MIPS R4400 provides the most useful information, but the counter hardware must be built externally. The Pentium provides the next most useful information by counting write buffer stalls. Many of the processors count cache snoops and snoop hits, but this information is not useful unless these events directly affect performance by causing the cache to be busy when it is needed by the processor. As will be shown later, these counts are also misleading when measuring performance loss due to the memory system.

3.2.5 Exceptions and Interrupts

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

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

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

3.2.6 Branch Mispredictions

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

Interestingly, all processors that employ *dynamic* branch prediction can also count the number of mispredictions with their monitoring hardware. Unfortunately, they do not account for the cost of mispredictions, which can vary depending upon instructions issued and to-be-issued. The performance lost due to mispredictions should be measured to motivate better prediction schemes and improvements in the recovery time of a misprediction. However, this is very difficult to measure in hardware because it implies a non-causal view of program execution.

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

3.2.7 Special Cases

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

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

3.2.8 Summary

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

1. **Count the number of dynamic instructions**, preferably with the ability to organize instructions into groups.
2. **Count the number of cycles lost due to NOPs and pipeline slips.**
3. **Count the number of cycles lost due to stalls.** Stalls caused by the memory subsystem should be reported separately and are detailed in the next section. Stalls arising from write buffer overflows or busy caches should be measured. These stalls can sometimes be reduced by scheduling and also by improving interprocessor coherence activity, which will also be described in the next section.
4. **Processing opportunity lost due to exceptions should be measured.** In particular, pipeline flushing and restarting cycles should be counted by hardware since it is unobservable by software.
5. If **TLB faults** are serviced in hardware, they should be counted. Additionally, the total service time is useful.
6. The number of **mispredicted branches** and, if possible, performance loss due to branch misprediction should be quantified.
7. If a system includes **special hardware features**, it is important to measure their effectiveness.

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

3.3 Processor View of Memory Stalls

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

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

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

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

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

3.3.1 Counting Cache Misses

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

Counting Misses: Informing Memory Operations

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

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

In [Horowitz95], three fast implementation techniques for informing memory operations are suggested, but they all require existing processors to be modified. A fourth technique, which they briefly hint at, sets up the memory system to return bad ECC data and causes an ECC-based trap to masquerade as a miss trap. However, the overhead of invoking such a trap is prohibitively expensive. Although informing memory operations are

very attractive from a performance monitoring standpoint, they are fairly intrusive and cannot be implemented with current processors.

Counting Misses: Informing Hardware

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

A novel way to inform hardware of phase changes is to have software explicitly modify the contents of a special-purpose external hardware register, called a *Phase Identifier Register* or *PhaseID*. The contents of the PhaseID are under software control, so a program can indicate whether information should be collected at a fine or coarse granularity, or both. When PhaseID changes, a different counter is selected to count cache misses. In this way, the cache misses for each phase are collected separately. Of course, the use of PhaseID need not be limited to counting cache misses; it can be used as a general mechanism for dividing-up performance data.

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

To change the contents of an off-processor PhaseID, three different implementation methods can be used. In the first method, the data portion of an uncached-write is used to update PhaseID. Later in this chapter, this store will be used for a dual purpose in which the data portion will contain timing information. For this reason, the second method encodes the new PhaseID value into the address component of the write¹. The third method of changing PhaseID is via a read, where the new contents must be encoded into

the address and the response is discarded². These different methods have different strengths and weaknesses, as discussed below.

Using a write to change the PhaseID has the least impact on performance, but some caching policies allow reads to bypass writes (which are held in a buffer). Consequently, read misses may be incorrectly counted in an earlier phase. The read method solves this problem, except that it may have a greater impact on performance³ and improper counting of write-throughs or write-backs may occur. Other similar problems exist, especially for recent superscalar out-of-order processors with multiple outstanding reads. These problems can be solved by serializing reads and writes, but imposing an order on these instructions imposes unreasonable performance loss.

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

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

3.3.2 Classifying Cache Misses

Once the number of cache misses is known, it is useful to further divide each group of misses to help identify the causes. The common nomenclature of compulsory, capacity,

-
1. There is an implicit assumption here that the hardware is memory-mapped and that there is sufficient room in the address space. These assumptions are true in NUMAchine.
 2. An interesting use of this read is to return the previous PhaseID or some performance-counter result, but such options have not been explored in this thesis.
 3. The read can be acknowledged immediately, so the performance impact is not too large.

and conflict misses describes all forms of conflicts in a uniprocessor. In a shared-memory multiprocessor, additional misses occur because of the need to keep caches coherent. Such misses are often overlooked and have been poorly defined in the literature; the discussion below proposes definitions and less-ambiguous names for these misses.

Classifying Misses: Compulsory, Capacity, and Conflict Misses

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

Compulsory misses are ignored because they are hard to measure. However, the significance of compulsory misses can be mitigated by two arguments: they occur infrequently and they are hard to reduce. First, compulsory misses are constant regardless of cache size because they occur when a memory block was never before referenced within the lifetime of the process. For this reason, the number of compulsory misses is constant. In a long-running program which touches memory many times and suffers an increasing number of misses, compulsory misses become more and more infrequent. Second, compulsory misses can be reduced by only a few methods: increasing the cache line size, compacting data structures, and trading off computation for memory accesses (by replacing a table lookup with a computation, for example). Unfortunately, the programmer seldom has the freedom to use these techniques because most machines have a fixed cache line size,

data structures are usually as compact as possible, and adding more computation does not always work.

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

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

Conflicts can be traced back to specific code or data in the following way. Software should be able to match the cache location containing excessive conflicts to all static objects in a program and produce a candidate list of conflicting variables. To pinpoint dynamically allocated objects, a second run of the program with hardware watchpoints set on the cache line in question can stop a program when and where the conflict occurs.

Unfortunately, processors aren't usually set up to have watchpoints on a cache line; instead, the performance-monitoring hardware should have this ability and generate an interrupt when a match is detected.

A variation of the watchpoint scheme can help identify conflicts in only one pass. Suppose the hardware monitor creates a cache profile and, at the same time, records the average number of misses per cache line; this is easily done by counting all misses and shifting the count by $\log_2(n)$ bits, where n is the number of cache lines. While profiling, a watchdog inspects the difference between the average miss per line and the miss count of the line that just missed. When this difference exceeds some threshold, it can be assumed that the miss was caused by excessive conflicts at that location. As before, an interrupt can be generated to stop the program and pinpoint the source of the problem⁴.

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

Classifying Misses: Invalidation Misses

The first type of multiprocessor cache miss is presented here. When an invalidate-based coherence protocol is employed, data in the cache is sometimes changed by another processor, which also has a cached copy, and the stale copy needs to be eliminated or *invalidated*. These eliminations which originate from outside processors are called *external invalidation hits*. If the data is invalidated but subsequently required by the processor, a miss is incurred; the data was removed because of the invalidation. In previous literature,

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

these misses have been called coherence misses [Jouppi90], but such terminology is misleading because it is not clear whether another type of coherence miss, which will be defined shortly, is included. In this thesis, the term used in [Martonosi95], *invalidation misses*, is recommended, but it should be defined differently:

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

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

There are always more external invalidation hits than invalidation misses. This follows because cache lines marked invalid by an external invalidation hit may never be reused or the invalid line may be replaced before it is reused. This difference is important because external invalidation hits don't cause memory stalls; only invalidation misses result in memory stalls. However, current processors such as MIPS R10000 count the external hits instead of the misses. Unfortunately, the number of invalidation hits will probably be mistaken by many as a measure of processor performance loss⁵ or as an estimate of invalidation misses.

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

5. Recall that the overhead of processing the invalidate transaction may cause the processor to stall, but this was already quantified in Section 3.2.4 on page 38 and is not considered here as a memory stall.

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

An alternative method of counting invalidation misses was used in [Singhal94], but it involves running the application twice: once using an invalidate-based protocol and once using an update-based protocol. Since there are no invalidation misses with an update protocol, the difference in the number of misses is used as an estimate. The obvious problems with this method are that two runs are required and the number of misses can vary depending upon the exact sequencing of the processors.

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

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

Classifying Misses: Ownership Misses

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

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

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

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

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

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

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

The performance impact of ownership misses can be reduced in many ways. First, switching to a weaker consistency model will eliminate the need to stall by providing an immediate acknowledgement for most writes. (Interestingly, the total ownership miss latency is a good estimate of the maximum performance advantage of a weak consistency model.) Second, switching to an invalidate-based protocol will reduce some ownership misses, but only at the expense of introducing invalidate misses; this trade-off will be dis-

cussed below. Third, the effect of the remaining ownership misses (in invalidate-based protocols) can be reduced by providing hints to the memory system that a line is likely to be modified and should be loaded in exclusive mode. For example, the KSR load instruction contains a hint whether a store is likely and the PowerPC and MIPS R10000 can try to prefetch a line into exclusive state. Performance increases because these hints start the invalidation process sooner than if the processor waited for the write to occur. Fourth, a processor may delay the stall, potentially even avoiding it altogether, by continuing execution until a second write occurs. At this point, sequential consistency disallows the second write to pass the first and the processor may have to stall to avoid this. For example, this is done by the Pentium processor and it is likely counted by the ‘pipeline stalled by write to exclusive or modified line’ event [Glew95]. By applying these techniques, performance loss from ownership misses can be reduced.

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

Classifying Misses: Summary

It is hard to distinguish between compulsory, capacity, and conflict misses in hardware, so simulation should be used when these details are needed. Hardware can construct a cache miss profile for each line over time, and this can help identify conflict misses when they are centralized about a particular memory address. Additionally, invalidation misses should be counted in hardware because they directly impact performance and characterize

multiprocessor data sharing. External invalidation hits originate from outside the processor and do not directly reflect performance loss, but are useful to help identify migratory data sharing patterns. Finally, ownership misses have not been properly identified in cache literature, yet they can result in performance loss as well. They should also be measured by performance-monitoring hardware.

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

3.3.3 Measuring Cache Miss Latency

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

The *basic* service latency of a cache miss, defined on an unloaded system, is only a portion of the actual service time experienced in a loaded system. Increased delays come from memory and network contention and from additional coherence traffic required to maintain consistency.

Miss Latency: Basic Service Latency

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

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

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

Miss Latency: Memory Contention

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

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

When the memory is busy, requests must queue up and wait until they are served. Thus, the performance of a memory resource can be measured by classical queueing theory metrics: time-average queue length, maximum queue length, average/maximum/total length of time in queue, and average/maximum/total service time. These metrics are easy to measure in hardware when the queue is centralized, as in a single FIFO buffer, and the buffer is large enough to avoid overflow. If a hardware buffer does overflow, the data is often refused and the ‘problem’ is pushed back to the network or the processor. This complicates some memory queue measurements such as maximum queue length, but the total miss latency information is still retained.

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

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

To relieve memory contention in a program, both hardware and software techniques can be used. If the hardware can efficiently broadcast or multicast data, it can be used as a

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

Miss Latency: Network Contention

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

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

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

mization. Although network contention can be reduced, it may be at the expense of some other performance metric.

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

Miss Latency: Coherence Traffic

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

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

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

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

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

Miss Latency: Other Effects

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

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

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

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

other useful work. The result is an increase in utilization of processor resources and fewer unused cycles.

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

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

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

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

The end result is that new processors are quite successful at hiding the latency of memory on the given benchmarks. To capture this, a metric similar to AMCPI would be useful. However, the complexity of the latest processors makes it very difficult to measure AMCPI. In addition to AMCPI, the older MCPI is still a useful memory system metric.

Programs with limited instruction-level parallelism suffer more because of MCPI, so it cannot be ignored when designing a memory system.

Miss Latency: Summary

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

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

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

3.3.4 Summary: Processor View of Memory Stalls

In this section, memory stalls have been characterized from the perspective of a processor executing a parallel program. Where possible, general hardware schemes have been shown to recognize different types of cache misses (especially those arising from multi-

processor activity), measure latency and performance loss, and correlate misses with program activity. The primary goal of this hardware has been to give fine-grained insight into program memory behavior so that performance bottlenecks can be understood and removed without becoming intrusive or necessitating excessive cost. However, the information also enables software to be reactive to prevailing conditions and adjust its behaviour accordingly.

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

8. **Provide informing memory operations from [Horowitz95], with external TRIGGER pin support.** If the pin is active during a response, the appropriate miss handler is invoked. This feature enables sophisticated software-collected performance-monitoring and adaptive program behaviour.
9. **Create a user-level PhaseID register.** Changes to the register should be fast and appear to be done in-order. The contents of the register should always accompany all memory transactions throughout the system and are used to separate performance data. This register need not be large; it can be 4 to 16 bits wide.
10. **Profile cache activity** by counting misses to each line separately. This is useful for identifying some cache conflict misses. Periodic sampling of a profile can help show some miss trends.
11. **Support cache-profile watchpoints and/or thresholds which generate interrupts.** These are necessary to trace back misses (observed during profiling) from dynamically-allocated data. By specifying a threshold, the watch is delayed until a real problem occurs. This can help software be reactive to cache misses.
12. **Count invalidation misses.** These are very important multiprocessor-specific misses present in invalidation-based coherence protocols. To implement this, cache line tags must be cleared properly during cache flushes or an additional empty state is required.
13. **Count external invalidation hits.** This metric is not as important as invalidation misses, but it can help identify migratory data.
14. **Count ownership misses.** In update-based coherence protocols, they indicate performance that can be gained through relaxed consistency models. In invalidate-based protocols, they indicate the need for load-with-intent-to-store hints.
15. **Measure queue performance** of memory modules and the network. This is important for understanding components of miss latency, which should be reduced.

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

3.4 Hardware Support for Software Tools

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

3.4.1 Timer Support

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

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

3.4.2 Basic Block Profiling

A common component of CProf, Mtool and MemSpy is that they all do some form of program profiling. The most useful level of granularity is at the basic block level, because it

can be guaranteed that all instructions in a basic block are executed the same number of times. To improve the efficiency of profiling at this level, hardware support is considered for two types of profiling, counting and timing. The hardware is used to reduce the overhead and to improve the timing resolution of these software monitoring tools. Since these applications can reuse previously suggested hardware, they are very cost-effective.

Basic Blocks: Counting Profile

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

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

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

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

Basic Blocks: Timing Profile

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

Immediately upon entry to a basic block, the current timestamp should be loaded into a general purpose register. Then, the time difference since the previous basic block timestamp can be computed, or it can be left for off-chip hardware to calculate. The result is the time to execute the *previous* basic block. Next, a profiling store can be executed at any point within the block to write the timestamp difference to the performance monitor. Previously, when counting basic blocks, the profiling store contained the hardwired basic block identifier, but this no longer works because the time is for a previous basic block of unknown origin. In this case, the profiling store must use an address from a processor register which was loaded in the previous basic block. After the store, the monitor SRAM accumulates the execution time for that previous basic block. Finally, the address identifying the current basic block must be loaded into a register for use by the next basic block. Thus, the time spent in each basic block can be accurately measured.

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

registers must be dedicated to monitoring functions and one or two registers is needed for temporary computation.

```

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

```

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

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

```

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

```

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

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

3.4.3 Pointing Losses Back to Code and Data

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

3.4.4 Summary: Hardware Support for Software Tools

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

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

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

3.5 Summary

In the software view of performance, speed is compromised when algorithms are inefficient, extra parallel work is present, synchronization and load imbalance cause waiting delays, system and library calls are inefficient or have side effects, and when limited parallelism exists. Measurements of these performance areas are readily done with software-based instrumentation techniques. The measurements provide indicators of *where* a program is slow, but they do not sufficiently explain *why* or *when* performance suffers.

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

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

The technique involves interrupting the processor when a particular cache line incurs too many misses.

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

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