

Chapter 2

Background

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

2.1 Monitoring in Software

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

2.1.1 gprof

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

Under the assumption that a subroutine always takes the same length of time, the subroutine counts are used to divide its running time among its callers. This information is presented to the user in two ways: first, subroutines are ranked by execution time, and second, by the call-tree hierarchy. The execution time ranking is useful for identifying slow

routines, while the call-tree format shows which parent routine calls a subroutine most often. In this way, a programmer can optimize a routine to run faster and also improve the parent routine so that fewer subroutine calls are made. As a performance tuning tool, gprof is good for general code and algorithm development, especially when the time complexity of a routine is not known beforehand.

A disadvantage of gprof is that the general nature of the execution profile provides little insight into sources of performance loss. For example, if a floating-point matrix multiply routine was labeled as being very slow, it would be unclear whether the floating-point operations were slow or if poor cache behaviour was encountered. Additionally, the inserted bookkeeping code overhead can be intrusive; we have observed code in which the bookkeeping routine accounts for 30% of the run time. This amount of intrusion can cause gprof data to be misleading. For example, consider that cache interference from gprof's inserted code can inflate the execution time of a procedure beyond any other and then misreport it as being the bottleneck. Additionally, Varley reports that when a routine is called by more than one parent, gprof can incorrectly attribute the running time among its callers because of the invalid assumption made above [Varley93]. Moreover, since sampling is done at a relatively slow rate, the program must run long enough to capture statistically reliable data. Consequently, although gprof is useful for general code and algorithm improvement, it does not provide enough insight into the source of performance loss and may occasionally report incorrect or misleading results.

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

2.1.2 Mtool

Similar to gprof, Mtool [Goldberg93] automatically instruments a program at the object-code level by timing the execution of important routines and adding counters to *basic blocks*, a term that describes small segments of straight-line code which have only one

entry point and only one exit. In addition, it supports parallel programs by measuring synchronization overhead and extra parallel work not present in sequential code. Mtool further breaks up runtime by classifying execution time as either compute time or memory overhead. This taxonomy aids the programmer during code tuning.

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

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

Mtool times loops and procedures via program counter sampling (in a manner similar to gprof), high-resolution hardware timers, or both. By using both, it tries to balance the drawbacks of sampling with the overhead of inserting timer probes. Using an initial basic block count profile, Mtool automatically selects memory-intensive loops and procedures that should be instrumented with timers. It also measures synchronization procedure calls with timers, but only if they are available on-processor or if it is requested by the user. Otherwise, Mtool uses program counter sampling. Despite the variety of timing methods supported, the authors of Mtool encourage that high-resolution timers be made available

on-processor with access latencies of 1-2 cycles because they are the preferred way to instrument code.

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

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

2.1.3 Paradyn

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

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

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

2.1.4 QPT and CProf

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

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

(for eqntott) and 3.46 (for vpenta, a benchmark kernel) times faster, with an average improvement of 1.69.

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

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

2.1.5 MemSpy

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

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

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

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

2.1.6 Summary

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

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

2.2 Hardware Monitoring in Processors

As system integration increases, monitoring hardware is able to observe less and less of the system. For example, SUPERMON [Bacque91] was able to trace every instruction ref-

erence because the instruction cache was on a separate chip, but all current microprocessors integrate this cache into the processor. In fact, current systems are even putting the second level cache in the same chip (the Alpha 21164 or Pentium Pro, for example). Fortunately, microprocessor designers also see the need for performance observation and include dedicated measurement hardware in the latest designs. This section describes the measurement facilities present in recent processors.

2.2.1 Intel Pentium and Pentium Pro

Work by Mathisen [Mathisen94], which was later updated by Ludloff [Ludloff94] and Collins [Collins95], reverse engineered² some of the Pentium instruction set and revealed that there are two 40-bit registers dedicated for performance measurement. Each of these registers is capable of counting one of the 42 possible events listed in Table 2.1. Additionally, the counters can be arranged to count events or the number of CPU cycles spent performing the events. Unfortunately, a program must be in supervisor mode to access them, meaning a (costly) system call is usually required.

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

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

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

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

TABLE 2.1. Intel Pentium performance-monitoring counter inputs.

2.2.2 DEC Alpha 21064

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

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

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

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

2.2.3 MIPS R4400 and R10000

Rather than provide on-chip counters, the MIPS R4400 [Heinrich94] brings pipeline status information to externally observable pins. Although it requires some hardware design effort, off-chip counters can be built to keep track of the 15 different pipeline events listed in Table 2.3. This feature is convenient because system designers can build 15 dedicated hardware counters and capture all information about a program in one pass, compared to most other microprocessors which only provide two counters. The R4400 also provides a 32-bit cycle counter for high-precision timing. An interrupt can be raised when the cycle counter reaches the value stored in a compare register — this is very useful for high-resolution program counter sampling, for example.

Interestingly, the next-generation MIPS processor, the MIPS R10000 [MIPS95], removes the pipeline status pins and provides two dedicated on-chip 32-bit counters instead. The events these counters monitor, shown in Table 2.4, are similar to the R4400 events except for a few additions. First, a difference is drawn between issued instructions and graduated instructions. This distinction is important because not all instructions that are issued will graduate due to speculative execution. These measurements give feedback to the processor and compiler designers on the effectiveness of the dynamic and static

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

TABLE 2.3. MIPS R4400 externally observable events.

scheduling of instructions. A second addition to the event table is the *way prediction* entry. The secondary cache is two-way set-associative, with the two ‘way’ tag comparisons done sequentially in time (in other products, they are done in parallel). This event measures the effectiveness of the *way prediction table* to predict which way to check first. The third addition is the inclusion of cache state events, such as counting external intervention or invalidate hits. These events may indicate performance loss due to multiprocessor data sharing conflicts and can be useful for tuning parallel programs³.

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

2.2.4 Hewlett-Packard PA-RISC

Hewlett-Packard has deemed performance monitoring hardware important enough to reserve room in the PA-RISC instruction set architecture (ISA) [HP94] and to define a performance monitor coprocessor and interrupt. Two specific instructions are currently defined, `PMDIS` and `PMENB`, to disable and enable data collection respectively. The performance monitor coprocessor is left as an implementation-defined unit, but additional room is left in the ISA for future standardized extensions. Further details about the HP

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

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

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

performance monitors are kept confidential, but some insight may be gained by looking at the Convex SPP systems described in Section 2.3.5.

2.2.5 Sun SPARC

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

Like the R4400, the SuperSPARC I and II contain special-purpose pins that permit observation of pipeline events. A total of 10 signal pins are defined in SuperSPARC I and this is extended into 48 signals, divided into 4 groups and multiplexed onto 12 pins, in

SuperSPARC II. Despite the larger number of signals, the functionality of these pins is roughly similar to the MIPS R4400 outputs.

2.2.6 IBM/Motorola/Apple PowerPC 604

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

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

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

2.2.7 IBM POWER2

Of all current processors, IBM's POWER2 contains the most performance monitoring features [Welbon94]. A total of 22 counters are available: one cycle counter, one 'correctable memory error' counter (which relates more to reliability-performance), and five counters

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

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

in each of four functional units. The 20 functional unit counters can each observe one of 16 different items, for a total of 320 observable events. In addition, a special mode exists

where 21 counters (all except the cycle counter) are dedicated to memory performance metrics of the storage control unit (SCU).

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

2.2.8 Processor Summary

A summary of the performance-monitoring features found in current processors is presented in Table 2.6. Where information about a processor is not known, the entry is shown as a question mark. As seen from the table, two performance-event counters and a timestamp counter are common. Unique features among processors include the Pentium's output pins, Alpha's input pins, PowerPC's sampling registers and threshold conditions, and POWER2's reliability counter. Positive trends are providing user-level access to counters and, as a result, instruction set architecture definitions for accessing them.

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

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

2.3 Hardware Monitoring in Systems

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

2.3.1 University of Toronto Hector

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

Hector Hardware Monitoring: SUPERMON

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

A block diagram of SUPERMON is shown in Figure 2.1. SUPERMON data collection is based upon two banks of 32K deep by 48-bit wide SRAM to be used as counters or for trace data storage. In trace mode, the memory is triggered by a programmable state machine which takes inputs from control signals and the virtual address. The trace data contains either the 30-bit address and a 16-bit timestamp or 8-bits of user-defined state data and a 40-bit timestamp. The state data is a function of address and control signals. The logic to control the state machine, control signals, and address trigger is all performed

with additional cascaded SRAMs. Performance of the cascaded SRAM is maintained by inserting registers between them and pipelining the design. This approach was chosen over using PALs because SRAM provides good logic capacity, predictable performance and is easier to reprogram.

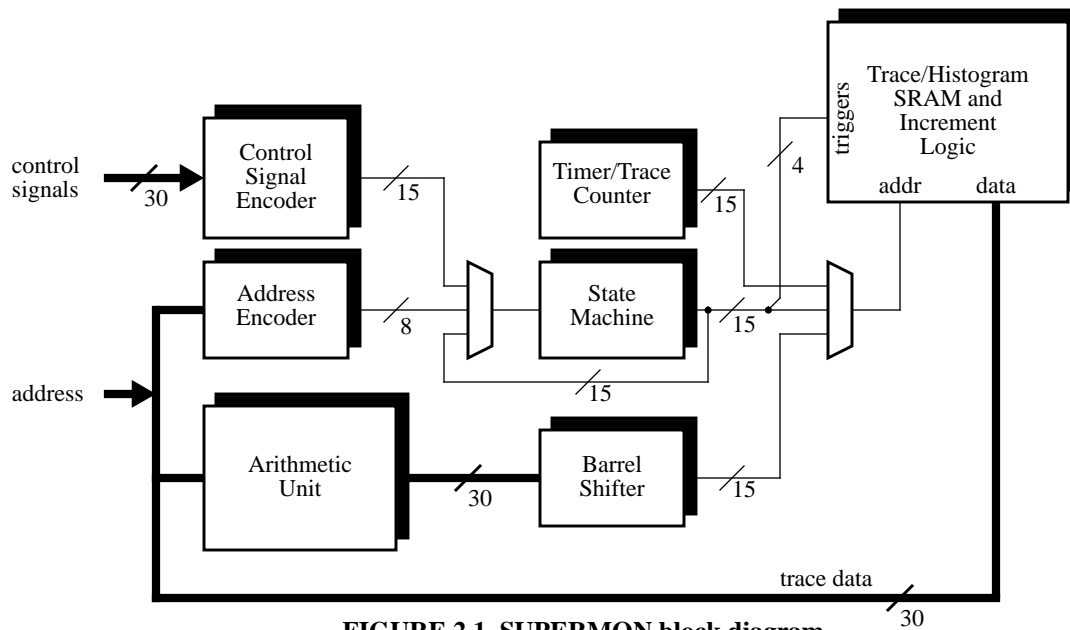


FIGURE 2.1. SUPERMON block diagram.

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

- how much time is spent in a certain user-defined state (there are 32K possible states),
- histogram latency; how often an operation takes 1 cycle, 2 cycles, ..., up to 32K cycles, or
- histogram absolute addresses, addresses relative to a fixed offset, or sequential address differences.

By building a histogram of addresses relative to a fixed offset, it is easier to monitor accesses to data structures like an array which can have an arbitrary starting address. Additionally, an interesting feature is provided while histogram data on addresses is collected: the bucket size can be constant, by shifting the address through a barrel shifter, or it can be variable by re-encoding the address into a floating-point representation. This

encoding extends the dynamic range of the histogram to cover the entire memory space while still providing fine-sized buckets in a region of interest (denoted by the fixed offset).

Using SUPERMON

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

2.3.2 Stanford DASH

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

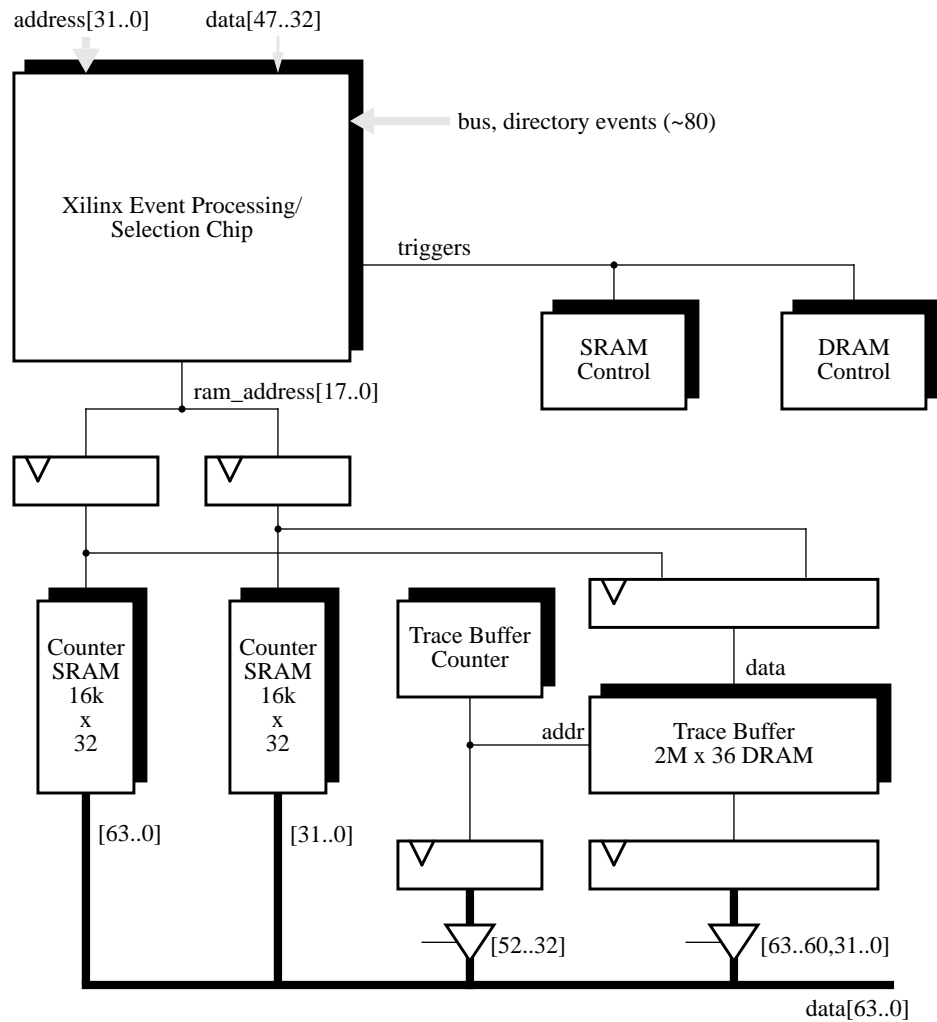


FIGURE 2.2. DASH performance-monitoring hardware.

The SRAM-based counters take two cycles to operate: one to fetch and one to increment and write back. The two banks can be interleaved to count an event that may change every cycle, or they can be used independently to count less frequent events. Each bank is 16K deep and 32 bits wide, providing approximately four and a half minutes of continuous operation before overflowing. The counters can be used to count events, where the address is formed by concatenating different event bits, or as a histogram array where the address comes from a counter in the programmable controller.

In addition to the counters, a trace buffer provides a detailed history of bus activity. The trace buffer DRAM, organized as 2M by 36 bits, is deep enough to capture information for over one-half a second; if longer traces are desired, the operating system must sus-

pend all process activity, dump the buffer to disk, and resume the processes. A common use of the trace buffer is to capture read and write requests, including the address and processor number. Another use adds the directory controller's PROM address and the time since the last request to each trace entry, thus halving the effective buffer size.

The programmable controller is an SRAM-based FPGA, the Xilinx XC3090. To change the monitoring mode or triggering conditions, the FPGA is reconfigured with a different circuit. For example, the SRAM can count 14 independent events by directly forming the address with the 14 signals. Alternatively, SRAM can be used as a histogram array by implementing counters in the FPGA and using them to form the address. Although this is a flexible approach, there are two drawbacks. First, reconfiguration time is on the order of 100 ms, so a program cannot frequently change the monitoring mode. Second, a software programmer must design a new hardware circuit if one doesn't already exist.

2.3.3 MIT M-Machine

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

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

numerous mask fields, the event counter can be configured to count one or a combination of events. The mask fields select:

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

2.3.4 Cray Research Inc.

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

The classic line of Cray supercomputers, from the CRAY X-MP up to the CRAY Y-MP C90, have all had performance monitoring capabilities [Cray92]. These computers are shared-memory vector processors. Rather than cache memory, they have instruction buffers and vector registers; no coherence is maintained. The main memory is highly interleaved and tuned to deliver large blocks of data at very high bandwidths to quickly fill the instruction buffer and vector registers. Instructions are obtained from the instruction buffer and can operate either on normal scalar data or on vectors up to 64 elements long⁴. If an instruction demands use of a particular vector register or functional unit, for instance, it is delayed until the resources are free; this is called *holding issue*, and is typical because vector operations can take many cycles to complete.

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

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

2.3.5 Convex Computer Corporation

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

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

Exemplar SPP-1000

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

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

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

Exemplar SPP-1200

The SPP-1200 hypernode uses the more recent Hewlett-Packard PA-7200 processors and has more performance monitoring features. A total of four registers can count from a variety of events. One of these registers counts any combination of local, remote, read, and write cache misses (this is similar to one of the Exemplar counters). The other three counters select from the seven events in Table 2.8. Additionally, one of these three registers can also count instruction or data cache miss latencies.

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

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

2.3.6 Kendall Square Research

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

The *local cache* coupled to each processor, akin to main memory, is 32 MB of DRAM. This should not be confused with the processor's *data subcache*, which is essentially a normal data cache. When a processor accesses memory and it misses in the local cache, a page-sized chunk of 16 KB is allocated in the local cache. Cache-coherence hardware searches for the address throughout the system, first on the local ring and then on remote

rings. If found, a copy of the 128 byte *subpage*, similar to a cache line, is copied or moved to the local cache. For details concerning actions taken when the address is not found, or what is done to the displaced page, the reader is referred to [KSR92a].

Performance Monitoring

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

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

TABLE 2.9. KSR performance-monitoring counters.

2.4 Summary

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

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

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