

Florida State University Libraries

Electronic Theses, Treatises and Dissertations

The Graduate School

2008

Core Monitors: Monitoring Performance in Multicore Processors

Paul E. West



Follow this and additional works at the [FSU Digital Library](#). For more information, please contact lib-ir@fsu.edu

Petitioner Samsung Ex-1013, 0001

CORE MONITORS: MONITORING PERFORMANCE IN MULTICORE PROCESSORS

Name: Paul E. West
Department: Computer Science Department
Major Professor: Gary Tyson
Degree: M.S.
Term Degree Awarded: Spring, 2008

Performance counters are becoming more complex as multi-core systems are becoming more wide spread. Consequently, evaluating these counters has become more complex as well. We propose providing hardware that monitors performance counters, namely in multi-core systems, in order to make decisions for improving performance. For instance, a piece of hardware watching snoop packets may be able to determine when a write-update cache coherence protocol would be helpful or detrimental to the current running program. Furthermore, watching memory traffic through a shared cache can determine if a program on a certain CPU is memory-bound. Once a program on a CPU is determined memory bound, the kernel can be informed to schedule accordingly. Finally, these new counters may be used to facilitate obtaining profile data for the compiler.

We have implemented monitors in a full system simulator and found performance improvement.

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

CORE MONITORS: MONITORING PERFORMANCE IN MULTICORE
PROCESSORS

By

PAUL E. WEST

A Thesis submitted to the
Computer Science Department
in partial fulfillment of the
requirements for the degree of
M.S.

Degree Awarded:
Spring Semester, 2008

The members of the Committee approve the Thesis of Paul E. West defended on April 9, 2008.

Gary Tyson
Professor Directing Thesis

David Whalley
Committee Member

Andy Wang
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

The members of the Committee approve the Thesis of Paul E. West defended on April 9, 2008.

Gary Tyson
Professor Directing Thesis

David Whalley
Committee Member

Andy Wang
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

ACKNOWLEDGEMENTS

I acknowledge the years of hard work humans have placed into science, and life given to me.

TABLE OF CONTENTS

List of Tables	vi
List of Figures	vii
Abstract	viii
1. INTRODUCTION	1
2. Performance Counters	2
2.1 Hardware Counters in Existing Machines	2
2.2 Performance Counter Use	8
3. Memory Monitor Architecture	11
3.1 Special Memory Monitor Instructions	14
4. Applications of Memory Monitor	18
4.1 Overview of Cache Coherence	18
4.2 Analysis and Use	21
4.3 Scheduling	23
5. Evaluation of Cache Coherence and Scheduling	25
5.1 Cache Coherence	25
6. Conclusion	29
6.1 Future Work	29
REFERENCES	31
BIOGRAPHICAL SKETCH	33

LIST OF TABLES

2.1 Number of counters and events in certain processors 3

2.2 Common detectable events in processors 4

LIST OF FIGURES

2.1 CESR for Pentium I. 5

2.2 Groups of counters on the Pentium 4. 7

3.1 Placement of the Monitor in a CPU chip. 13

3.2 Address and Queue of the Monitor. 14

5.1 CESR for Pentium I. 26

ABSTRACT

Performance counters are becoming more complex as multi-core systems are becoming more wide spread. Consequently, evaluating these counters has become more complex as well. We propose providing hardware that monitors performance counters, namely in multi-core systems, in order to make decisions for improving performance. For instance, a piece of hardware watching snoop packets may be able to determine when a write-update cache coherence protocol would be helpful or detrimental to the current running program. Furthermore, watching memory traffic through a shared cache can determine if a program on a certain CPU is memory-bound. Once a program on a CPU is determined memory bound, the kernel can be informed to schedule accordingly. Finally, these new counters may be used to facilitate obtaining profile data for the compiler.

We have implemented monitors in a full system simulator and found performance improvement.

CHAPTER 1

INTRODUCTION

Standard performance counters in CPUs are being reevaluated in light of multicore architectures. Hardware manufacturers have started moving to multicore architectures in order to improve processor performance. To take advantage of these new cores, programmers must understand and evaluate the operation of multiprocess applications. Current performance counters monitor single core characteristics and very little, if any, interprocess communication. Currently, extracting data from performance counters is a daunting task, and additional counters exacerbates this problem. This thesis proposes a programmable monitor for multicore systems to aid the programmer and the operating system by using the available hardware more effectively.

The monitoring system is inherently an event driven system which in turn changes the nature of the monitor processor from that of a conventional processor. Any incoming events are stored on a queue so that future events can be processed later. In order to handle an event, the monitor must identify the type of event and know which instructions to execute.

Following a more detailed explanation in Chapter 3, this thesis outlines two examples for using the monitor in a multicore environment. The first example controls the type of Cache Coherency protocol by examining the Cache Coherence traffic. The second example examines memory traffic to inform the kernel of memory bound programs. Once memory bound programs are identified, the scheduler can avoid over saturating the memory system. Lastly, future work covering this monitor is discussed.

CHAPTER 2

Performance Counters

Hardware performance counters are special registers on a CPU used to identify bottlenecks and other performance hindering characteristics of a running program. These counters are used to identify how often certain hardware events such as cache misses occur during the execution of a program. In order to improve the performance of an application, first the program is loaded, counters are reset, the program runs and counters are read. From the values in these counters a programmer can determine potential performance improvements. These improvements alter the code to improve the use of the branch predictor, memory, or any other piece of hardware. Once identified, the programmer can rewrite the code and rerun the program to look for more bottlenecks.

2.1 Hardware Counters in Existing Machines

Most modern processors contain hardware performance counters. While these counters are useful to help identify problems for a given program on a certain processor, hardware performance counters are processor dependent. This means that programmers can tune their code for a specific processor but not for any arbitrary implementation of the ISA. The way Intel executes x86 performance counters is different from the way AMD does. Even within Intel, performance counters have changed over time; the counters on the P4 are interfaced differently than the ones on the Intel Core Duo. Nevertheless, performance counters have a standard method of implementation that is used in almost all CPUs. Table 2.1 is a sample of the number of counters and event detectors for certain processors.

A performance counter is a special register in the CPU that is set up to increment when it receives a specific event signal. As depicted, the number of hardware counters is less than the number of events detected. Since performance counters do not directly improve performance

Table 2.1: Number of counters and events in certain processors

CPU	# Counters	# Events
Pentium	2	38
Pentium 4	18	44
21164 Alpha	3	66
AMD Opteron	4	84

(or save power) they are one of the last features designed for a processor. Furthermore, because processor area is conserved (to cut costs) the number of performance counters is low. In general, most processors have less than 6 counters, with the rare exception of the Pentium 4, which has 18.

Event detectors are used to detect performance oriented events, such as cache misses and branch predictions. Table 2.2 portrays common events that happen in modern processors.

A signal is generated from an event detector, which is a component of the hardware for which events are being detected. For instance, the invalidation of a cache line can be sent to a counter. Unfortunately, there are more events than counters. Therefore, the signal is routed to a central controller, which either ignores the signal or routes it to the proper register. To save power, most processors implement an off switch for all event detectors implemented as an “all or nothing” switch to avoid area complexity.

The Pentium I has a 32-bit control-and-event-select monitoring register (CESR) that is used to control incoming events. Control Counter Fields (CC0 and CC1) are set to control which signals are set to the counter. The CC0 and CC1 set what the Current Privileged Level (CPL) must be for the event to propagate to the counter. The pin control flags (PC0 and PC1) are used to identify when the CPU asserts the counter. “Off” means asserting every increment, and “on” means asserting only when there is an overflow. The Event Select bits (ES0 and ES1) identify which event increments the respective counter. The Pentium I shows the standard method of performance counters before major complexity problems arise.

For the Pentium 4, Intel chose a slightly different approach to the implementation of performance counters. Due to increased complexity of processor chips, the Pentium 4 and subsequent Intel processors have their counters distributed in order to save area on

Table 2.2: Common detectable events in processors

Event Name	Event Description
Number of Cycles	Keeps track of the number occurred cycles
Number of CPU Loads	The number of CPU loads
Data Cache Accesses	Number of accesses to the L1 Data cache
Data Cache Misses	Number of data cache misses. Usually you can configure to only count when in a certain Cache Coherency state (MOESI) and/or reads/writes.
Data Cache Refills	Number of times the Data cache has been refilled by the L2 cache or System memory. Usually this counter is configurable to count only refills from either L2 or System and the state of the cache line was in (MOESI).
L2 Cache Accesses	Number of accesses to the L1 Data cache
L2 Cache Misses	Number of data cache misses. Usually you can configure to only count when in a certain coherency state.
L2 Cache Refills	Number of times the Data cache has been refilled by the L2 cache or System memory. Usually this counter is configurable to count only refills from either L2 or System and the state of the cache line was in (MOESI).
Instruction Cache Fetches	The number of instruction fetch for the instruction cache.
Instruction Cache Misses	Number of misses to the instruction cache. Usually this can be broken up into misses that hit the L2 cache or must go to System Memory.
Instruction Fetch Stall	Number of cycles the fetcher stalled. Usually this happens from branch misses or cache misses.
Retired Instructions	Number of retired, or committed, instructions.
Retired μ ops	Number of retired, or committed, μ ops
Retired Branches	Number of retired, or committed, branches
Retired Mispredicted Branches	Number of retired, or committed, Mispredicted branches
Retired Mispredicted Taken Branches	Number of retired, or committed, mispredicted taken branches
Interrupts	Number of Interrupts.
Dispatched Stalls	Number of stalls generated. Generally this event can be configured for different reasons such as full reservation stations or aborted branches.

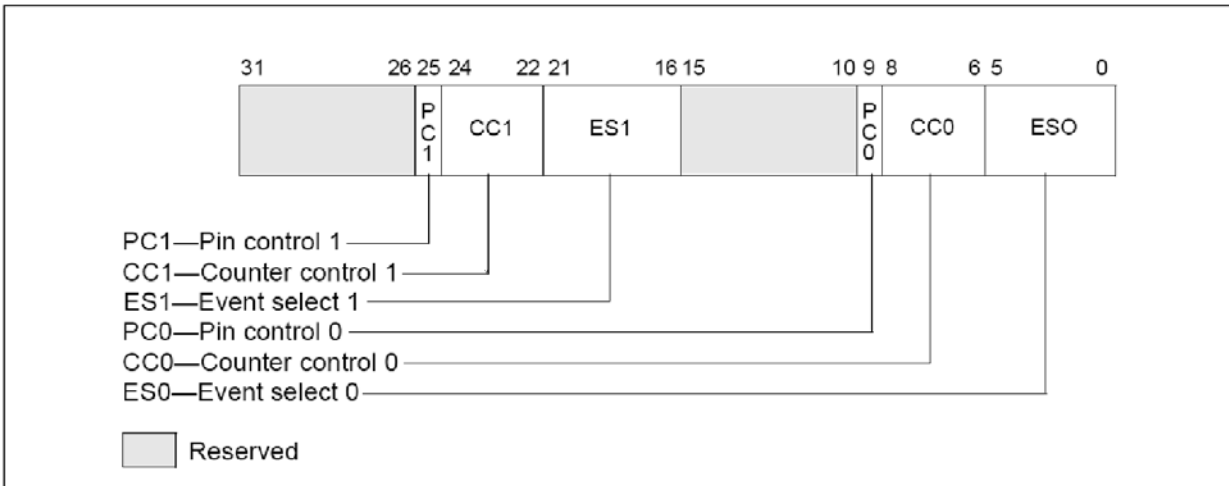


Figure 2.1: CESR for Pentium I.

routing signals [1]. Intel considered two designs: one which contained a counter in the implementation of each event detector; or another, composed of groups of counters spread out throughout the entire processor. The latter was chosen based on area conservation which in turn cut down on production costs.

To actually use these counters a programmer must first set the CESR for the proper events which can only be executed under privilege level 0. Hence, the kernel must first set up the counters to accumulate the data. To do this, the kernel must write to the CESR using the wrmsr command. Here is an example of setting up the Intel Processor processor to use performance counters:

```
wrmsr 0x11, 0x00690063 ;Sets the CESR (0x11) to have:
                        ;counter 0 count number of read misses
                        ;to the data cache, and
                        ;counter 1 count number of memory accesses
                        ;to the data cache.
                        ;Both are set to always read no matter the
                        ;privileged level.

wrmsr 0x12, 0x0        ;Clears counter 0(0x12)
wrmsr 0x13, 0x0        ;Clears counter 1(0x13)
```

Now, in user space, the programmer must read these counters before and after the code is profiled.

```
mov ecx, 0 ;select counter 0
```

```

rdpmc          ;read counter specified by eac into register eax
mov ebx, eax   ;save counter into register ebx

...           ;the profiled code executes

mov ecx, 0     ;select counter 0
rdpmc          ;read counter specified by eac into register eax
sub eax,ebx    ;write the total number of events into register eax
               ;This assumes that register ebx was not touched.
               ;You may have to save ebx to memory.

```

Now register *eax* has the number of read data cache misses that occurred in the profiled code. If the coder also used counter 1 for total cache accesses, the coder can deduce the average cache miss rate. If the miss rate is significant then the programmer can alter the profiled code to take more care of data cache misses, through better cache alignment or overall memory use. Once the corrections are in place, the coder runs the profiled code again to see if any changes occur.

The Pentium 4 features 18 performance counters and 22 event detectors per thread. Only a certain number of allocated performance counters can track a certain event. For example, there are 4 counters set aside to track 8 of a single thread's Branch Prediction Unit events. The choice for the number of counters per group was determined by Intel's prediction of consumers' need.

Grouping these performance counters helped reduce area and increase the number of performance counters. Even with fewer performance counters than event detectors, the Pentium still supports more counters than most modern chips, which have less than 6. Since counters and the wires to counters take up area, more companies are producing processors with fewer counters to cut down on production costs.

2.1.1 Alpha

The Alpha uses a different approach to performance counters. ProfileMe [2] is a method of recording events on a per instruction basis in order to gauge performance more accurately. The idea is to provide more accurate data for out-of-order CPUs, since other instructions in the pipeline can affect the previously started counters. Since counters in ProfileMe do not record the number of events triggered but only their presence, it is the Operating System's

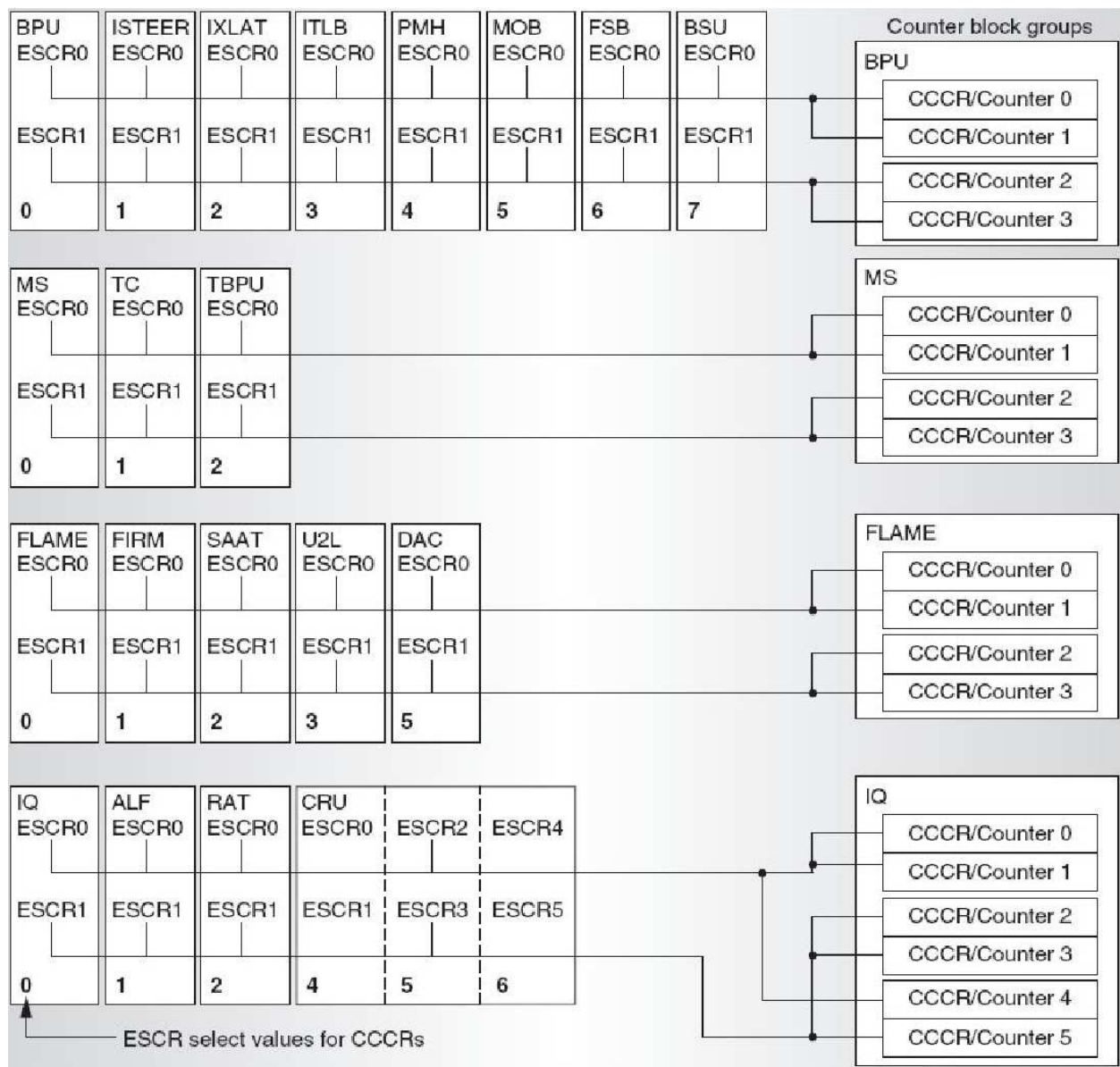


Figure 2.2: Groups of counters on the Pentium 4.

job to record the data and provide the correct amount to the user.

To begin profiling, the user tells the OS what events it wants profiled. The OS uses Alpha's PALCodes [3] to set the proper flags to record certain events and to determine how often the profiling will happen. Once profiling begins, the hardware picks a pseudo-randomly selected instruction to profile (based on the OS's provided sampling rate). As the instruction travels through the pipeline, any events triggered by the instruction are recorded. Once the

instruction retires, the CPU issues an interrupt. The OS records which events happened, clears the registers, and resumes execution. When the user requests the counters, it is the OS's job to provide the correct number per event.

An advantage of ProfileMe is that almost all events can be recorded at the same time. Unfortunately, the OS must handle the recording of data, which can interfere with some counters like Cache Misses. Furthermore, ProfileMe only operates on its pseudo-randomly selected instructions and can miss important one-time event problems. For example, a protocol violation or an important interrupt may happen only once or a few times during the execution of a program. ProfileMe also has a long wait time (about 1000 instructions) until it profiles an instruction. Hence, these important but rare events can not be dependably detected.

2.2 Performance Counter Use

Most users of these counters are programmers searching for bottlenecks in their programs. With PAPI [4] a programmer must ensure proper kernel configurations are “on”, then call PAPI's functions to start, stop, and collect data. Obviously, one side effect is the increase in code size and execution time from these calls. Furthermore, since these counters are moved into memory, a small memory footprint may replace “hot” data. While these effects are small, extensive use can intrude on memory and slow down execution. Hence, most of these counters are used to tune the code before large runs.

2.2.1 Accuracy

Korn [5] points out that the use of performance counters can lead to incorrect data. They also compare performance counters used in a binary loader, library calls in the source code, counts given by a simulator, and hand counted data as a baseline. As expected, the binary loader performs the worst: with over ten times performance penalty against the calculated baseline. Library calls perform much better with the worst case scenario less than 100% . SimpleScalar's Sim-out-of-order, the simulator used, was always less than 3.0%, and in most cases was within .1%.

These discrepancies come from different sources. SimpleScalar induces the insertion of arguments and other environmental variables into the simulation. The library calls have

overhead with the kernel, and furthermore, the kernel runs a scheduler on top of the executing program. The binary loader is even worse than the library calls because in addition to overhead of system calls and the kernel, the loader has to tell the kernel to execute the program. As noted in [5] the binary loader will approach the accuracy of the library calls injected into the code as program execution length increases.

On most hardware, a super-user mode is available. In general, this privileged mode is only available to the kernel. Some processors can detect when the *su* has started, and stop taking performance counters as a response to the mode change. Certain kernels, like Linux, can keep track of these counters in case of a context switch. After the kernel is done (and the correct performance counters are in place), the hardware will resume counting once *su* mode has exited.

Some researchers have given concern that the hardware of event detection can intrude on the running program. Zagha [6] developed a way to prevent the intrusion of running hardware counters within the running process. Naturally, reading from the OS level will still cause some overhead, but other hardware's reads will not affect the current running program. Also, [6] stated that total transistor count of performance counters was less than 0.2% of the total non-cache transistors. This paper shows that performance counters can be used to significantly improve performance by identifying bottlenecks in code.

A study presented at MICRO40[7] showed that to achieve accurate results, numerous runs must be done to interpolate the problems from software interrupting the running process. Many solutions have been researched to combat the low number of hardware performance counter registers. One solution is to run the program multiple times to acquire all event information. This approach is inadequate for the Monitor design since we are targeting on line execution. Multiplexing requires switching which counters are used during execution, but this approach can avoid certain one time events that are important to the evaluation. Many excellent methods exist, but all of these add complexity to a system which needs to be small and efficient. In order to evaluate what is causing performance problems, the collected hardware counter data must be correlated. Such computations further exacerbate the performance problem.

These counters record the events of one processor as opposed to recording events that affect other processors or the system as a whole. Memory contention between processors in certain architectures can cause the performance degradation that programmers look for

in performance counters. Unfortunately, inter-process bottlenecks (like memory) cannot be deciphered efficiently from single core hardware counters. Furthermore, certain one-time events can easily elude profiling techniques given by ProfileMe. Even if all needed events are available, it may take multiple runs to get every piece of information needed due to a low number of counters. Multiplexing attempts to fix this, but these one time events can occur and cause further problems. Lastly, as the number of cores in machines increases, the problem of correlating the data increases as well.

CHAPTER 3

Memory Monitor Architecture

Performance counters are used by programmers to find the bottlenecks for a certain program on a certain processor. Unfortunately for the standard approach, a low number of counters results in a need to multiplex data. Techniques like ProfileMe can capture all events for an individual instruction, but can easily miss certain one time events that can affect the entire system. Furthermore, both techniques incur a run time overhead which slows down the running program. Thus, performance problems may arise from the interaction of programs on multiple cores as opposed to those on a single core. Performance counters in multi-core environments are still useful for sequential execution and thread analysis. However, as the number of cores increases so does the number of event signals. This causes even more data to be correlated to profile the machine. Hence, performance counters need to incorporate interactions that encompass the entire system as opposed to individual processing elements.

Now these inter-process counters must be processed. In both ProfileMe and the standard approach the processor that was being profiled handles the counting. For these inter-process counters, one of the operating processors could handle the counters, but that could skew the selected processor's performance since data will travel from one core to the dedicated one. Furthermore, more counters need to be on the operative core to handle incoming data. Multiplexing of data could combat this, but could miss one time events. Also, wiring up one individual processor for all inter-process counters takes up a significant amount of area.

Another solution would be to spread out the counters. The benefit is that the recording of counters is spread throughout the system. The hardware would control which CPUs control which inter-process events. A down side is that a program must utilize every CPU in the system in order to read each counter. This increases software complexity because the program must find out which core is handling which event.

Even with a solution to processing inter-process counters, the programmer must interpret the data, figure out how to rewrite the code, and then implement. Furthermore, some decisions, like which Cache Coherency scheme to choose can change during the execution of a program. To implement a running change the performance counters must be read and interpreted during runtime. This increases execution time and can negatively impact cache memory performance. Instead of code running on the working processors, the performance counting code runs on a separate processor designed for handling data given by its performance counter.

This thesis will show a Monitor processor that evaluates performance counters of online programs, then changes hardware and software configuration in to order to more effectively use the hardware present. For example, varied cache coherence schemes provides a different performance for applications. The Monitor can detect when one scheme causes a great amount of congestion, then switch to another to improve performance.

The monitor must have access to all performance counters of a CPU but still be efficient. Placing the monitor off chip causes long latency for communication. Placing it on a core means many wires must be laid to an individual core. Hence, the monitor exists on chip, but independent of each processing core.

The monitor interfaces with the core's events as well as the L1 and L2 cache events. If more then one chip exists, then communication with other monitor units travels through the L2 cache. A configuration containing multiple monitor will be discussed after the single monitor architecture is explained.

Since the Memory Monitor acts upon events and not on sequential instructions, the Monitor is more related to an Event Driven CPU than to a conventional processor. At its core, the Monitor is a simple CPU triggered when events are placed into a queue. Events are generated from event detectors (part of hardware performance counters) that send information to the Monitor to be queued. The Monitor has its own piece of memory that is part of the system address space, where data is stored and computed. Once on the queue, the CPU takes the event and processes it with instructions provided by the running program.

Processors must be able to program the memory monitor to look for a specific events. The monitor has a reserved piece of memory which contains the address to jump to when an event signal arrives. Hence, a processor simply writes an address, which is derived from an

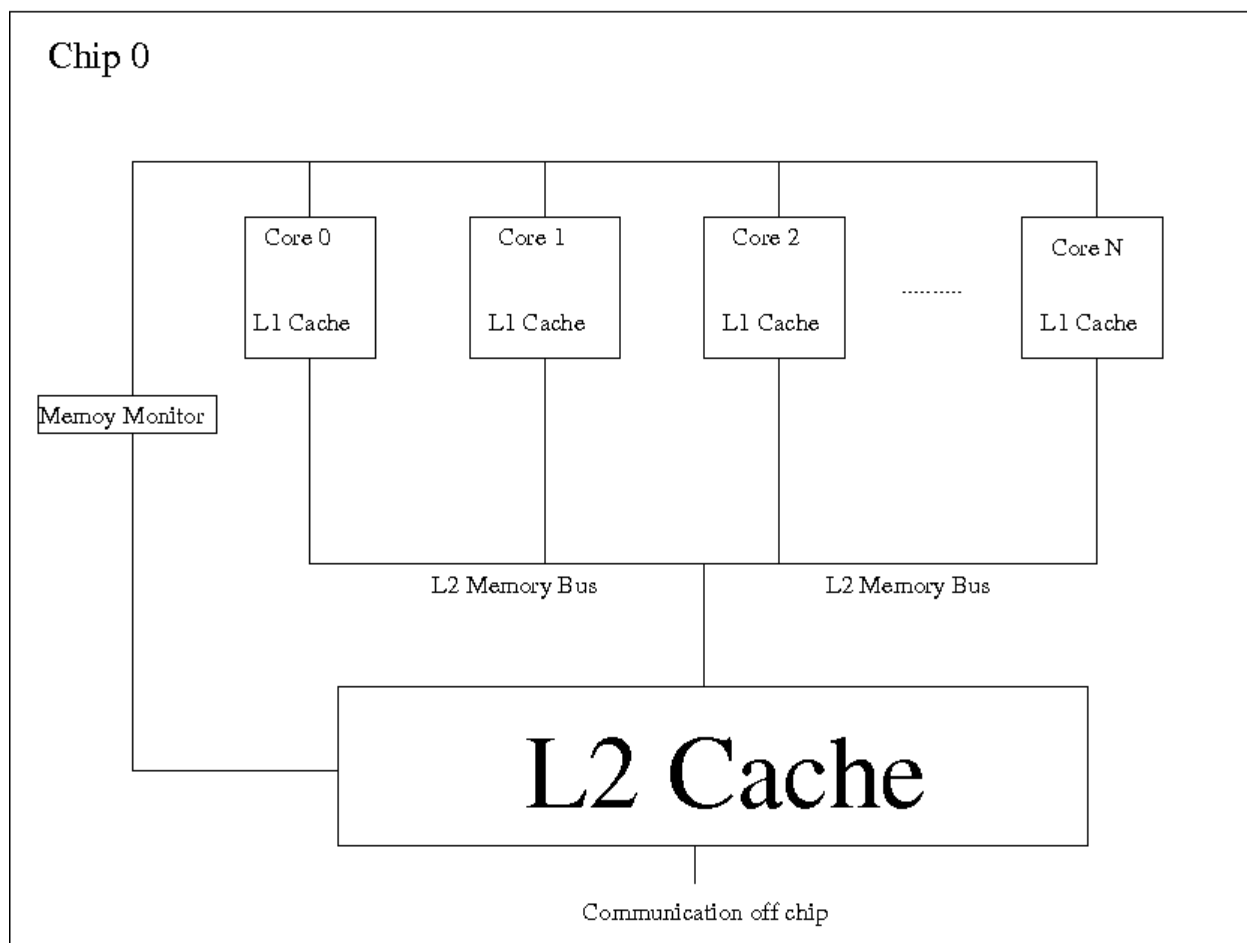


Figure 3.1: Placement of the Monitor in a CPU chip.

array indexed at the event, to signal which code is to handle the event. As a convention, an index with a NULL address means to ignore the incoming event signal. Thus, a processor can invalidate an address to alleviate processing of unnecessary events.

Programming for an event driven processor differs from coding for conventional processors. The Monitor may spend a lot of time idle, waiting for an event to process. This is not necessarily bad, because if the system is already optimal there is not a point in reconfiguring the hardware. On the other hand, if the system is loaded with events that start to overflow the event queue, then the processor is missing important data that could be useful. This is due to the numerous events available [1] and the ability for an instruction to fire multiple events, the number of events received by the Memory Monitor must be controlled. Otherwise,

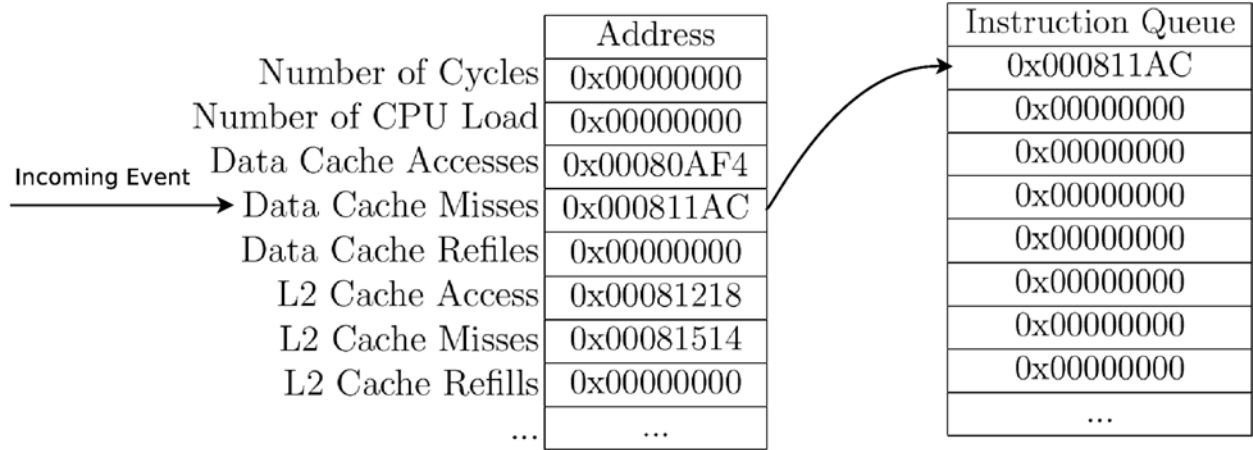


Figure 3.2: Address and Queue of the Monitor.

the event queue will always be full and the Memory Monitor will not be responsive. First, if the event is not being used, we can disable its detection (which is the default start state). To reduce the number of events placed on the event queue, counters that only let the n^{th} event onto the queue are used. Hence, somewhere on the path from event detection to monitor, there is a mask that prevents all but the n^{th} event through.

While simple, these masks still take up space and must be interfaced with (to set the mask). Also, having the counter-mask at the event detector means fewer signals sent to the Monitor, but more wires in order to interface with event detector. On the other hand, counter-masks in the event monitor means fewer wires that are used more often. So, if heat is the problem, counters can be placed on the event detectors to dissipate heat across the chip. If area needs to be conserved and the processor can take the heat, then the counter's mask can be on the Memory Monitor. Due to modern heat concerns, it is logical to assume the counter masks are located on the event detectors.

3.1 Special Memory Monitor Instructions

Our simulator used the Alpha ISA and thus the running example will use the ALPHA ISA. Alpha instructions are 32-bit wide and start with a 6-bit op code. [8] Since opcode $0x07$ is unused, all Memory Monitor instructions start with $0x07$. Since more than one instruction is needed, the next four bits describe which Monitor instruction to execute. That leaves 22 bits

as fields for 16 individual instructions. It is important to note that only the Monitor needs the extra instruction because the individual cores interact by writing to reserved address space. Below are the proposed monitor instructions.

wcmmm: Write Counter Mask Memory Monitor: This instruction sets the event described mask by sending the last field to the counter mask. If the original value of the mask was 0, then the event is turned on.

wemm: Write Event Memory Monitor: The address in the last field is written into the event array, indexed at the event field. This sets the address to jump to when an event fires.

mmhalt: Memory Monitor Halt: Not to be confused with a regular halt, this instruction tells the Monitor that processing of the current event has ended. Hence, the Monitor may stop executing instructions and is now free to execute another event.

cmm: Clear memory monitor: No fields are needed. This instruction is issued by the kernel when all events and detections need to be invalidated, which runs at the start up of a program to ensure everything is cleared.

mmchk: Memory Monitor Check signal: This instruction queries the system to see if a certain hardware device is on and places 1 or 0 into the return register. Thus, the Memory Monitor can detect which events it should be handle turn off/on certain event detectors.

mmsig: Memory Monitor Signal: This instruction signals the hardware of the system to change state or to send an interrupt to the CPU. For example, the signal may change from a write-invalidate Cache Coherence Scheme to a write-through one.

Usually it is the kernel's job to change hardware state, but sometimes the decision is better left to the monitor. Hence, **mmchk** and **mmsig** were created. These two instructions do not take care of all system changes. For instance, if the scheduling of processes is causing a problem, then the kernel must be informed. Furthermore, some systems may want the kernel to handle all system changes. In either case, the monitor must inform the kernel. When the monitor decides that the kernel needs to take over execution, the **mmsig** instruction is executed. When executing, the memory monitor sends an interrupt to a processor assigned by the hardware manufacturer. The interrupt then acts as a regular interrupt and the kernel can read the monitor's memory.

Interrupts should only happen when a system-wide change must happen. For instance, if the monitor identified the current running process as an interference to other processes, the kernel may be able to schedule the process more efficiently. The signal can interrupt and

be used to determine which processor contains the conflicting process. The kernel may then signal the process and schedule accordingly.

As the number of cores increases, a single memory monitor may require a significant amount of area to run all the wires. (This is similar to the problem in the Pentium 4.) To solve this, multiple Monitors may exist throughout the system. Hence, a larger address space is reserved for the monitors because each monitor has its own address space. Communication is simple for a Monitor. One monitor sends an event to another one for processing. For instance, if multiple monitors are detecting Cache Protocol invalidates, each monitor can be programmed to send their “cycles per invalidate” every expected number of cycles. The receiving monitor processor can then figure out the number of cycles per invalidate and make changes if needed.

Note that the receiving monitor has a higher chance of being overloaded. In normal performance monitoring if few event signals are dropped that is fine because we are studying general trends. With monitor communication this may be unacceptable. Hence, the instruction to send an event may write to a buffer that clears itself when the event is successfully written. Another possibility is to wait for the message to be successfully received before executing the next instruction. This can unfortunately lead to deadlock, so writing to a buffer is preferred. If the buffer is full, then the event is ignored and the programmer of the monitor must rewrite the code to prevent buffers from overflowing.

The Monitor requires both address space in the system and memory to do its own computation. Every Monitor’s address space to other CPUs consists of an array of addresses for the events. An internal memory structure is accessed only by the particular Monitor. The amount of memory needed is dependent on the number of events and number of processors. The number of events received must be retained so that future event calculation may depend on them. For example, to calculate the average miss rate, the monitor must know both the number of cycles and the number of misses.

Since the monitor must keep track of state, more memory is required. In the next example, a counter for the average miss rate of a certain type of miss in the system is recorded. For this example, only one extra counter is needed. In another example, the monitor records the memory traffic of a running process. Since the data is processor specific, a counter for each processor is required.

So the minimum amount of memory required for a monitor on an existing system is:

$$Amt_m = (\#events) + (\#monitoring_1) + \#processors * \#(monitoring_p) \quad (3.1)$$

where Amt_m represents the amount of memory, $monitoring_1$ is the number of events that require one counter, and $monitoring_p$ is the number of events requiring a counter for each processor.

CHAPTER 4

Applications of Memory Monitor

Our first and primary example consists of managing multiple Cache Coherence schemes on a system. A wide variety of cache coherence protocols exist, including Synapse, Illinois[9], Berkeley[10], Firefly[11], Dragon[12], and MOESI. Some of these protocols have different behavior than others. For instance, the Firefly protocol broadcasts writes to other shared cache lines to keep data consistent, whereas Illinois invalidates other cache lines to ensure that only one individual cache line is actually written to. Both the Illinois and Firefly protocols share similar states, but a broadcast of a write or an invalidate can yield different performance results. It is for this reason that Illinois and Firefly are the two protocols selected for the initial case study.

4.1 Overview of Cache Coherence

Maintaining consistency of shared data is the primary function of cache coherence schemes. Thus all caches that *share* a cache line must have (or will soon have) the same data for that particular cache line. For instance, CPU₀ may write to its L1 cache line *A* that is shared in CPU₁'s L1 cache. Before CPU₀'s L1 cache can invalidate the line in CPU₁'s L1 cache, CPU₁ reads from the cache line *A*. This causes a race condition that is not solved by the cache coherence model, but is instead managed in software by mutexes or semaphores.

For simplicity, this paper associates both Firefly and Illinois algorithms with *MOSI* (*Modified, Owned, Shared, and Invalid*) states. While different from the original papers describing these schemes, this should ease understanding of the similarities and differences of the two cache coherence algorithms.

Both algorithms are write-allocate, meaning an initial write pulls data from memory into the cache line instead of performing a write-through. Both algorithms also associate two

bits for each cache line: one denotes if the block is shared with other memory structures (*shared*), and the other denotes if the cache block is dirty (*dirty*). Furthermore, both cache coherence algorithms are write-back, meaning that data is sent to main memory only when a dirty cache line is invalidated.

Each cache line is initialized as “not shared” ($\neg shared$) and “not dirty” ($\neg dirty$). This means that the cache line is not contained within any other memory structures, nor has the cache line been written to. We denote this state as *Invalid*.

4.1.1 Illinois Read and Write Miss

If the CPU requests a read to an *Invalid* cache line (denoted a read miss), then the cache puts a read request on the cache interconnect (which is a bus). Other caches will service the request (if able) or the data may come from main memory. In either case, once the read request is serviced, the cache turns on the *shared* bit while leaving the *dirty* bit off. We denote the state of *shared* and $\neg dirty$ as *Shared* state.

On the other hand, if a cache block is *Invalid* and a write is sent to the cache line, then the cache will place a read request on the cache interconnect. Having serviced the read request, the cache will broadcast an invalidate signal to all other caches to ensure only one cache can write. Then, the *dirty* bit is turned on while the *shared* bit is left off. We denote the *dirty* and $\neg shared$ state as *Modified*. The broadcasted invalidate on a write miss in the Illinois protocol is required to ensure that *Modified* caches have exclusive ownership of the cache line. Hence, further writes to a *Modified* cache line may fulfill the request without sending any data to the cache interconnect.

4.1.2 Illinois Snooping

Snooping occurs when one cache broadcasts a request/response to other caches on the cache interconnect. For instance, on a read miss, the cache places a read request on the cache interconnect. If a cache has the data requested then the cache will send a response containing the data to the requesting cache. In this example the cache snooped a read, because a different cache had the requested data.

When a cache in state *Shared* receives a read snoop, the cache sends the response containing the data without changing state. If the cache is *Modified*, then the cache will service the request and turn on the *shared* bit. We denote this fourth and final state of

shared and *dirty* as *Owned*. *Owned* cache lines may also service a read request. When more than one cache line can service a read request, it is left to the hardware to decide which cache actually services the read request. Fortunately, each of the cache lines is the same, so it does not matter which one actually completes the request. Let it be known that *MOSI* has been used to denote a specific cache coherence protocol and a class of cache coherence protocols. For the remainder of this paper, *MOSI* will refer to the class of protocols.

Finally, if a *Shared* cache snoops an invalidate request, then the cache changes state to *Invalid*. If a *Modified* or *Owned* cache line snoops an invalidate request, then the cache must perform a write-back and change its state to *Invalid*.

4.1.3 Illinois Read and Write Hits

A read hit is always immediately serviced without the need to send any request on the cache interconnect. A write hit on a *Modified* cache is written to the cache without any need to place a request on the bus. A hit on a *Shared* or *Owned* cache must first broadcast an invalidate to all other caches, then write the data, and change its state to *Modified*. The broadcasted invalidate is sent to ensure that only one cache can be written to at any given point. This ensures that future write hits on a *Modified* cache are quickly serviced.

4.1.4 Firefly

Firefly was written to prevent frequent invalidates caused by a CPU writing to a memory address that is shared among many CPUs' caches. Hence, the only difference between Firefly and Illinois happens on a write hit to a cache in the state of *Shared* or *Owned*.

In the instance of a write hit to an *Owned* or *Shared* cache, Firefly sends a write to all other caches and performs a write-through to the next level in the memory hierarchy. *Owned* and *Shared* caches that snoop a write will write to their corresponding cache line. In either case the operations done by Firefly are completed without any change to the state of the cache. An algorithm that broadcasts writes to other caches, like Firefly, is referred to as a *write update* algorithm.

4.2 Analysis and Use

Firefly was created to deal with the conflict created by multiple CPUs attempting to acquire exclusive privileges to a cache line. This contention can dominate performance in the producer/consumer paradigm where one thread sends information for another thread to receive. The producer has to write the data to an address, then the reading thread must read the data and finally notify the producer that the data has been processed (if needed). Another problem may arise during a physical simulation of particles: if two threads are operating on particles that are contained within the same cache line, both caches will try to write to that particular line and cause contention.

On the other hand, Firefly may cause more memory traffic than is absolutely necessary. For multiprocessing ring models, where one thread passes the data to another, there is no need to keep the memory in every cache. In fact, once a processor starts writing to the cache line, other processors will be working on something else. In this case, Firefly would generate more memory traffic on the cache interconnect than Illinois. Since memory is often a bottleneck, this effect is exacerbated.

The problem lies in that there is no hardware mechanism able to switch between the two cache coherence policies, nor does there exist some software layer to change a ring algorithm to and from a producer/consumer algorithm. Current hardware manufactures must decide carefully whether or not it is worthwhile for their cache coherence algorithms to support *write-update*.

Instead of forcing designers to make such an early decision, both Illinois and Firefly were implemented in hardware with a *Memory Monitor* that can adaptively switch between these two algorithms, depending on which appears to be most efficient for the task at hand. Since all Illinois states are valid Firefly states, and all Firefly states are valid Illinois states, there is no need to flush the caches. Although, since write-updates can conflict with write-invalidates, communication on the bus must end before the switch can be done.

The Memory Monitor is a hardware enhancement that must be able to determine when to switch between these two algorithms. Firefly was designed to prevent frequent invalidates from a write to a *Shared* or *Owned* cache line. Therefore, a frequency of invalidates due to writes to *Shared* or *Owned* over the number of cycles should provide a reasonable estimate of the memory traffic pattern. Unfortunately, Firefly creates more memory traffic than Illinois.

Thus, when many write updates are being issued from only one CPU, it is more appropriate to switch back to Illinois.

4.2.1 Implementation

Each cache in the system hierarchy must be capable of switching between Illinois and Firefly. The states of Illinois and Firefly are the same; the only difference occurs when there is a write to an *Owned* or *Shared* cache line. In that case, a switch will determine which packet is sent. Each switch must be hooked up to the monitor that is managing these caches. As mentioned before, Firefly reduces write invalidates while increasing traffic on the cache interconnect. Hence, if the caches are using Illinois and there is a significant amount of write invalidates sent on the cache interconnect, the caches coherence protocol should be switched. On the other hand, if one cache is broadcasting a write to other caches lines that are never read or written, then Firefly is placing unneeded traffic on the cache interconnect.

The memory monitor must have a line of communication to each individual L1 cache. When deciding to change from Illinois to Firefly, the monitor is configured to increment a counter whenever a write-invalidate is received. Additionally, each 100th write-invalidate records its cycle time in a 10 entry circular buffer. When the buffer is full, each write to the buffer triggers the monitor to calculate the average number of cycles between the past 1,000 write invalidates. Once the average drops below a programmable limit (100 in the simulation), then the monitor sends a signal to the caches to switch to the Firefly policy. The switch simply changes the broadcast of invalidates to the broadcasting of writes.

The change from Firefly to Illinois occurs when a CPU attempts to access a shared cache line. This is easily recognized by detecting write updates that are written to the cache but never used. Luckily, through empirical experiment, detecting that only one CPU is broadcasting writes suffices. Now this does make some sense, because generally a program will read a cache line and then later write something to it. To detect this, the monitor is signaled when a write-broadcast is sent and will calculate the number of write-broadcasts a CPU has performed without another CPU sending a write-broadcast. Hence, one entry is the CPU identifier of the last broadcasted write, and a counter for the number of writes. When the monitor receives the signal, if the write ID is the same as the last CPU's ID, then the counter is incremented. Otherwise, the counter is set to 1 and the CPU ID entry is set as the new CPU ID. When the number of writes reaches a certain limit (20 in the simulation),

the monitor sends a signal to the caches to switch from Firefly to Illinois.

4.3 Scheduling

In operating systems, scheduling is the process of deciding which task to run next. Naturally, a scheduler should provide processes enough time to be responsive without making other processes seem unresponsive. Furthermore, real-time processes have absolute deadlines that the kernel must ensure are met. The Linux kernel takes priority, frequency of interrupts, and NUMA into account when selecting the next process to run. For instance, if a process is flagged as I/O (which can be determined because the kernel handles I/O) the kernel knows to switch out the process, but have it ready for execution when the I/O returns. In a multi-core environment, the 2.6.13 Linux kernel starts each CPU with a run queue containing only a swapper process. The swapper process simply loops, waiting for another process to enter the run queue and replace the swapper. When more than one process is on a CPU, the Linux kernel has to determine which process has precedence, and the decision is based on the aforementioned reasons.

Unfortunately, some behaviors of processes are still hidden from the kernel. For instance, how often a process requests memory outside of its L1 cache cannot currently be determined by the kernel. Thus, a memory-bound process seems the same as any other process to the kernel. This is fine until the system has scheduled multiple memory-bound processes to run together simultaneously. These memory-bound processes then generate additional traffic on the memory system, further slowing the execution of all processes.

Naturally, we propose implementing a counter to tally the number of accesses from a CPU that leave the L1 cache. The Linux kernel has mechanisms to determine when a process is I/O-bound, and when this case occurs, the kernel can handle the process using special rules. Similar to the I/O-bound case, the kernel is capable of scheduling memory-bound processes more efficiently.

The kernel must first determine if a process is memory-bound or CPU-bound. Unfortunately, the kernel does not currently have a means to distinguish memory-bound processes from software. The kernel must have some component to watch for increases in traffic on the cache's interconnect. Once it has a way to measure traffic, the kernel can designate a threshold of how many outbound L1 requests are needed in a certain timeframe in order to constitute a memory-bound process. The scheduler then tries to run only one memory

bound program at a time.

There are some problems that need careful consideration when dealing with process characterization. When a process begins there is a certain amount of *load-up* time that is spent fetching memory to start the program. For detecting memory-bound programs, the kernel should not consider the *load-up* time window. Hence, that statistic can not be considered accurate until it is known that the application was loaded into memory and is ready for execution. Furthermore, some processes perform a great deal of I/O on startup. This can be combated by polling at regular intervals to see if the processes should be flagged as memory-bound or CPU-bound.

CHAPTER 5

Evaluation of Cache Coherence and Scheduling

Evaluation work was done on the M5 simulation using System Call Emulation and Full System mode. For Cache Coherency, the monitor was implemented for M5[13] using system call emulation for the Alpha architecture. For scheduling, the monitor was implemented as a PCI device for the kernel to interface to. Four 2.0ghz Atomic CPUs were created, each with a 64K 2-cycle latency L1 cache. Each L1 cache was connected to a bus that was connected to a 10-cycle latency 2M L2 cache. The memory system used timings to acquire accurate latency of memory accesses. The processors would stall when waiting on memory to return.

5.1 Cache Coherence

To accurately portray results, the monitor was started with the slower coherence protocol and then switched. Firefly and Illinois were run through the Splash 2 Benchmark[14] suite. Once performance results were taken, the monitor was started with the “losing” cache coherence state.

In the overall experiment, Firefly outperformed Illinois. However, this was not always the case. Barnes is a simulation of interacting bodies (like galaxies or particles) using the Barnes-Hut hierarchical N-body method. Since data is represented as an oct-tree, accesses are spread out in memory. Hence, many writes will update other caches that are not using the node, and performance will degrade.

Ocean simulates eddy and boundary currents through contiguously allocated 4-D array. Each processor is responsible for a rectangular box of ocean. Since data is contiguous, memory that represents the boundaries of these rectangles will likely be referenced in the near future. In our result, Firefly outperforms Illinois.

As depicted, the monitor was always able to choose the better performing coherence

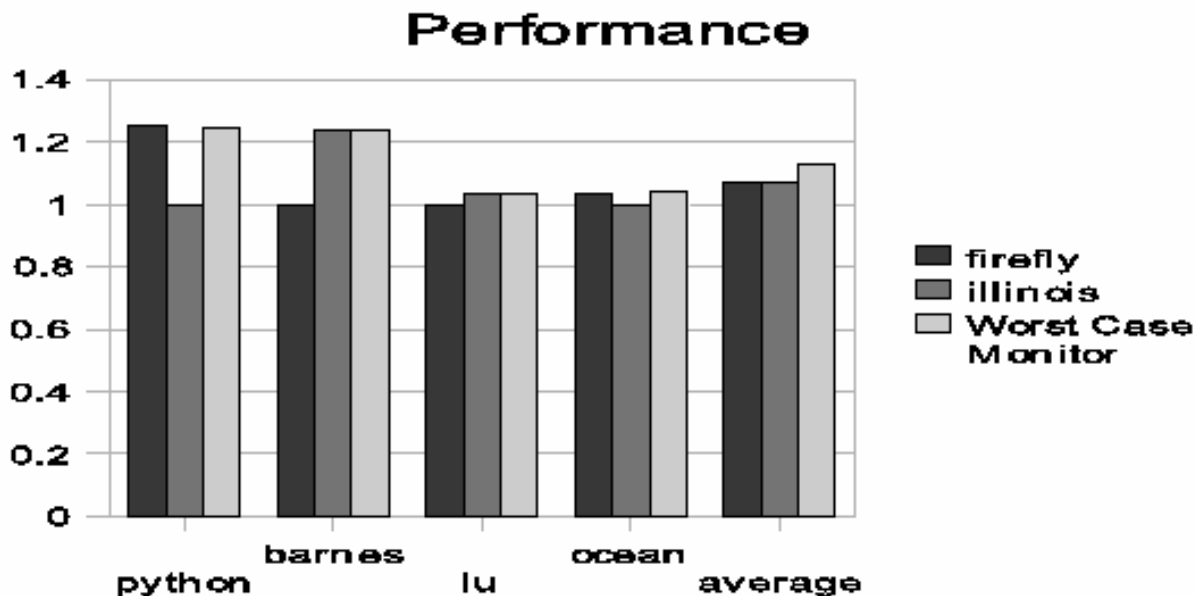


Figure 5.1: CESR for Pentium I.

policy. Overall the monitor outperformed each cache coherency policy individually.

An interesting exception was Ocean. The monitor version starting with the worse performer (Illinois) yet was able to slightly outperform Firefly. During simulation, the monitor switched between the two protocols. It is likely that the switch occurred due to eddies that are near the boundaries of these rectangles and then move into the center of the CPU's grid. When the eddies are close to the boundaries, FireFly works the best because data moves more smoothly from one CPU to another. On the other hand, when data was in the center of CPU's allocated memory, FireFly caused a lot of traffic problems and hence Illinois was the better performer.

5.1.1 Scheduling

The Linux 2.6.13 kernel was edited to have every task contain a memory-bound flag. In order for the kernel to read a piece of hardware that doesn't exist yet, a PCI device was implemented in the M5 full system simulator [13]. A real memory monitor does not necessarily need to be a PCI device, but the simulator provides robust facilities for creating and handling PCI devices. Furthermore, a PCI device in M5 may be connected to the bus

between the L2 cache and the L1 caches. So bus performance is not hindered. Unfortunately, there is more overhead in the kernel for handling a PCI device, but overall results still show benefits despite PCI overhead.

All event detectors on cache ports are connected to a single monitor. The monitor keeps a counter of the number of memory references for each CPU. When a request leaves the L1 caches, the monitor increments the proper CPU counter. Since, this is a PCI device, the counters are located at a base address. (More specifically the address is stored in Base Address Register, which we denote as *BAR*.)

The monitor mentioned above was implemented in M5 and a device driver was written for the 2.6.13 kernel. The device driver supplied a base address to the scheduler of the kernel to read and write. The offset from the base determines which CPU's counter the kernel was reading from. So, for example, *inb(BAR + 0)*; reads from CPU₀, *inb(BAR+1)* reads from CPU₁, etc... The *scheduler_tick()* function was edited to reset the counter (done by a write to the address) at 10 ticks and then read from the monitor at 15 ticks. If the number returned from the Monitor was above a limit (200 in the simulation), then a memory bound flag was set in the process structure. Now with a memory traffic-aware kernel, the scheduler must find a way to deal with running memory-bound processes.

The *Schedule()* function is called to attempt to replace the current running process with another process that needs to be executed. Obviously, when only one memory-bound process is running then the process can be run with no fear of contention. In fact, it would be an efficient use of the memory system if one memory-bound process was running at all times.

On the other hand, if a memory-bound process is scheduled next for execution, but a different CPU is currently running a memory-bound function, the scheduler has some choices to make. The scheduler could look for another process to run inside its own run queue. While not that expensive, the 2.6.13 kernel has 140 linked lists of processes to search through, and this could be detrimental to the scheduler's performance. Another option would be to migrate the currently scheduled process to the CPU with the running memory-bound process. All memory-bound processes would then always be executed on the same CPU. The downside here is that each memory-bound process would completely flush out the cache for every other memory-bound process. A third option would be to mark the memory-bound process as expired (having already used its time slice), and hope that the next process to execute is CPU-bound.

Our simulation uses the second option of moving the memory bound program from one CPU to another. The M5 full system simulator was used with timings for 4 Alpha CPUs and the same memory system previously mentioned in Cache Coherence. Ocean was run with 4 threads (memory bound) and then water-nsquared with 4 threads (CPU bound). As a base line it was found that both ocean and water-nsquared ran at closely the same execution time. Next, ocean and water-nsquared were run simultaneously through a non-memory traffic aware kernel, which took 1.243313 seconds. Next the kernel was modified and the experiment took 1.062417 seconds, which is an improvement of 17%. To see the associated overhead, the same experiment was run with the unmodified kernel in the scheduler and received 1.260846 seconds, which is a loss of 1.4%.

Cache Coherency and Scheduling are two different methods to improve performance of a multi-core system. The monitor is able to extract from performance counters when to either switch coherence, or inform the scheduler of the offending program.

CHAPTER 6

Conclusion

Performance counters are used to evaluate the performance of an application. As industry moves to multicore architecture, more counters must be added to evaluate the interprocess communication, and to find bottlenecks in multiprocessor applications. Managing these new counters is a problem because the traditional method in which a processor handles its own counters does not apply. This is due to the involvement of the system as a whole instead of individual cores. This thesis proposes a special processor which not only handles these new counters, but uses them to inform programmers and the Operating System of possible performance improvements.

6.1 Future Work

This thesis is a proof of concept. The architecture of the monitor has been explained and two possible uses have been implemented in the M5 simulator. While changing Cache Coherence and informing the kernel of memory bound applications are implemented, an actual Monitor Processor itself was not implemented in these experiments. The Monitor must be implemented and integrated with Cache Coherency and Memory Bound scenarios and the experiments rerun.

The problems of multicore architecture have not been solved. Future work will involve a more detailed study of these multicore applications and where they are going. From this work multicore applications can be evaluated to find performance improving characteristics. For instance, handling performance counters for interprocess communication has not been implemented in modern processors.

Mutexes in certain applications can take a significant amount of time to ensure mutual exclusion. The locking time required can be reduced with monitor assisted locking. Another

significant overhead is thread creation. By setting the monitor to watch certain memory locations, the monitor can interrupt an idle CPU to start executing a thread. This takes the computation of thread creation off the hands of a thread that could be doing useful work instead.

Debugging parallel applications is more difficult than single threaded programs. When any error occurs, the programmer must discover which thread of execution caused the error. This is in contrast to a single threading application where the programmer knows the error occurred in a single thread. The problem is more difficult when in a non-deterministic system where the error can not always be replicated. Furthermore a monitor is capable of handling watch points itself by waiting for a particular instruction address. When the instruction address is found and interrupt is sent and the debug can take over. This allows programs being debugged to run at full speed instead of a debugger watching every instruction.

Computer systems are becoming even more complex everywhere. It is difficult to judge what type of hardware will always be best because many different applications can run on a system. To improve performance, a monitor needs to help direct the uses of its many processing cores.

REFERENCES

- [1] Sprunt B. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, Jul/Aug 2002. 2.1, 3
- [2] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Wehl, and George Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society. 2.1.1
- [3] Alpha architecture handbook. <http://www.alphalinux.org/docs/alphaahb.html>, 2007. pg 332. 2.1.1
- [4] Performance application programming interface. <http://icl.cs.utk.edu/papi/>, 2008. 2.2
- [5] Korn W.; Teller P.J.; Castillo G. Just how accurate are performance counters? *Performance, Computing, and Communications, 2001. IEEE International Conference on.*, pages 303–310, Apr 2001. 2.2.1
- [6] Zagha M.; Larson B.; Turner S.; Itzkowitz M. Performance analysis using the mips r10000 performance counters. *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 16–16, 1996. 2.2.1
- [7] FIXME. Fix this. *FIXME*, FIXME 1998. 2.2.1
- [8] Compaq. Alpha architecture handbook. *whitpaper*, October 1998. 3.1
- [9] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM. 4
- [10] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *ISCA '85: Proceedings of the 12th annual international symposium on Computer architecture*, pages 276–283, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. 4
- [11] Charles P. Thacker, Lawrence C. Stewart, and Jr. Edwin H. Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Trans. Comput.*, 37(8):909–920, 1988. 4

- [12] Anant Agarwal, Richard Simoni, John L. Hennessy, and Mark Horowitz. An evaluation of directory schemes for cache coherence. In *25 Years ISCA: Retrospectives and Reprints*, pages 353–362, 1998. 4
- [13] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006. 5, 5.1.1
- [14] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, 1995. 5.1

BIOGRAPHICAL SKETCH

Paul E. West

Paul E. West was born on November 3rd of 1982 in San Antonio, Texas. In the Spring of 2005 he completed his Bachelors of Science degree in Computer Science and Mathematics at Trinity University. He entrolled for his masters and doctoral degree in the fall of 2005.

Paul's research interests include computer architecture, compilers, programming languages, and high performance computing.

Paul lives in Tallahassee, Florida, with his wife and a cat.