Core Monitors: Monitoring Performance in Multicore Processors

Paul E. West
Florida State University
west@cs.fsu.edu

Yuval Peress
Florida State University
peress@cs.fsu.edu

Gary S. Tyson
Florida State University
tyson@cs.fsu.edu

Sally A. McKee
Cornell University
sam@cs.cornell.edu

ABSTRACT

As we reach the limits of single-core computing, we are promised more and more cores in our systems. Modern architectures include many performance counters per core, but few or no intercore counters. In fact, performance counters were not designed to be exploited by users, as they now are, but simply as aids for hardware debugging and testing during system creation. As such, they tend to be an "after thought" in the design, with no standardization across or within platforms. Nonetheless, given access to these counters, researchers are using them to great advantage [17]. Furthermore, evaluating counters for multicore systems has become a complex and resource-consuming task. We propose a Performance Monitoring System consisting of a specialized CPU core designed to allow efficient collection and evaluation of performance data for both static and dynamic optimizations. Our system provides a transparent mechanism to change architectural features dynamically, inform the Operating System of process behaviors, and assist in profiling and debugging. For instance, a piece of hardware watching snoop packets can determine when a write-update cache coherence protocol would be helpful or detrimental to the currently running program. Our system is designed to allow the hardware to feed performance statistics back to software, allowing dynamic architectural adjustments at runtime.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.1.3 [**Processor Architectures**]: Other Architecture Styles

General Terms

Design

Keywords

Performance Monitoring, Multicore, Cache Coherency, Scheduling, Profiling, Debugging, and Realtime

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CF'09, May 18–20, 2009, Ischia, Italy.
Copyright 2009 ACM 978-1-60558-413-3/09/05 ...$5.00.

1. INTRODUCTION

Performance counters in CPUs are being reevaluated in light of multicore architectures. Hardware manufacturers are investing in multicore architectures to improve processor performance as an alternative to higher clock speeds and deeper pipelines. To take advantage of these additional cores, programmers must understand and evaluate the operation of multi-process or multi-threaded applications. Current performance counters monitor single-core characteristics and little, if any, interprocess communication. Extracting data from performance counters is already a daunting task, and adding additional counters exacerbates this problem.

Dynamic Optimizations, such as those in Java's JIT Compiler, are appearing for optimizations/load balancing that can be evaluated during the runtime of the application. Unfortunately, reading and evaluating performance counters during runtime imposes a performance overhead and changes the concurrency behavior of the running application. A study in 2006 [2] showed the newer techniques could lower overhead to 67%, from a previous 176% in one of their benchmarks.

Performance counters must evolve to efficiently account for concurrency between cores. This paper proposes a programmable multicore monitor system to aid the programmer and the Operating System by providing information on how to utilize the available cores more effectively.

2. PERFORMANCE MONITORING SYSTEM ARCHITECTURE

The Performance Monitoring System architecture is divided into two main parts: Performance Monitor Detectors (PMDs) and Performance Monitor Cores (PMCs). These components, along with the execution cores, communicate via new Performance Monitor signals and standard interrupts passed on the processor's interconnect. Some PMDs are extensions of performance counters, whereas others are more complex units that monitor memory traffic. When a PMD detects an event, a Notify Signal is sent to the PMC. If the PMC must inform the kernel, it sends an interrupt to an execution core to signal a change in architectural or software policy.

2.1 Performance Monitor Detectors

Performance counters can be used by programmers to find bottlenecks. Normally, a programmer begins profiling by clearing the counters, running the program, and then reading the counters at the end of execution. Another method, ProfileMe [7], watches a pseudo-randomly selected instruction and records every event that instruction triggers. Once this ProfileMe instruction completes, the kernel interrupts execution and records the events so the user can

later retrieve statistics. In general, most processors implement the former approach, since ProfileMe does not detect all occurrences, changes the reference behavior, and has significant performance penalties.

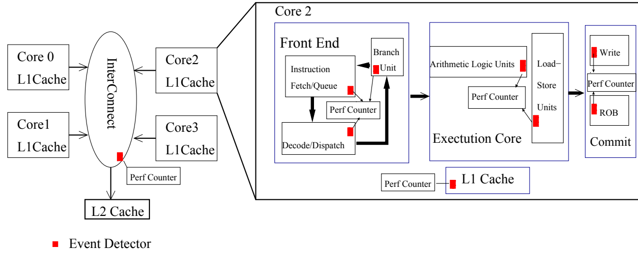

Figure 1: Typical placement of performance counters.

Due to high clock rates and wiring concerns, most counters are placed near the events they can count. Some systems (like BlueGene) have performance counters on the interconnect to help profile some interconnect communication. For instance, in Figure 1, the performance counter near the ROB is used to count the number of committed instructions, but is not used for branch mispredictions, due to distance. This limits the number of events that a performance counter can detect, which may create a need for multiple executions to acquire all needed information. PMDs differ in that all system events can be sent as Notify Signals to the PMC.

To save area, most hardware manufacturers create few performance counters to count the large number of events. For example, the AMD Opteron has four performance counters, but can detect over 80 unique events. This design choice lowers die area, but as density increases with Moore's law, area becomes less of a concern. PMDs differ in that they can send Notify Signals for all wanted events because they are Content Addressable Memories (CAMs).

Currently, these performance counters evaluate single-core performance and very little inter-core communication. As we move towards multicore processor designs, performance counters must evolve to account for additional factors affecting performance, such as core interconnects. Multicore designs need inter-core counters along with CPU time to process events triggered by inter-core events. We therefore propose a separate monitor core, the PMC, to evaluate such events (Figure 2). This new monitoring core should also take into account single-core events from each execution core in order to correlate information. Hence, on-core Performance Monitor Detectors must interact with the PMC.

Performance measurements are made by traditional performance counters (that count discrete events like cache misses) and by monitoring bus activity on address or data buses. CAMs placed on important bus structures can match complete or partial patterns. Bus monitors and event counters can be combined, but exploring that functionality is beyond the scope of this work.

To avoid over saturating the system, the counter in a PMD sends an event every n^{th} detection. These event counters work in conjunction with the CAMs to send Notify Signals to the PMC. In addition to these performance measurements, bus monitors watch memory traffic, e.g., by capturing memory accesses to important locations and reporting information on cache coherence traffic.

2.2 PMC Architecture

Traditionally, in single core systems, a program must stop execution to read counters and then evaluate the running process. This evaluation processing comes at the cost of execution time, and it changes memory reference behavior. Programs usually re-

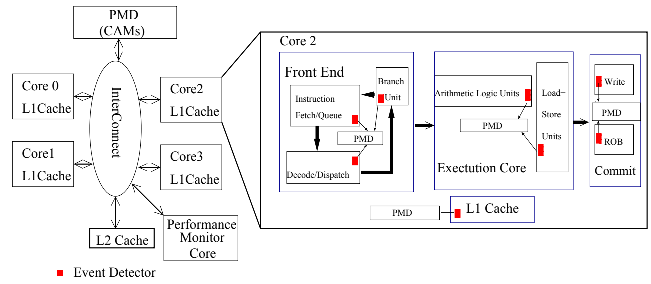

Figure 2: Placement of the Performance Monitoring System within a Multicore Environment.

quire multiple runs in order to collect the necessary system event counts because performance counters are unable to monitor more than one event at a time. With the proposed monitoring system, we can evaluate the running system without interrupting any executing processes. A separate, simple event driven processor, the PMC, evaluates events to potentially make architectural and software changes.

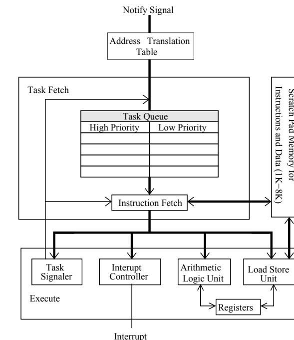

Figure 3: Performance Monitor Core (PMC).

The PMC, depicted in Figure 3, is designed similarly to a normal two-stage pipeline, fetching from the PC and executing instructions through the ALU, load/store unit, and interrupt controller. Instead of the standard integration of a local cache into the processor cache hierarchy, the PMC has its own specialized scratch-pad memory unit. The scratch-pad memory stores sequences of instructions as well as data to be fetched when designated Notify Signals are caught.

When a Notify Signal is caught by the PMC, it is placed into a low or high priority Task Queue. The priority of the signal is defined by the programmer, and can change which statistics are collected sooner by the Performance Monitoring System (PMS). Giving priority to certain performance counters can ensure that signals generated from high priority counters are not ignored due to queue overflow.

Assuming the PMC is not handling a signal and at least one signal exists in the Task Queue, a new PC will be fetched from the scratch-pad. Instructions will continue to be fetched from the scratch-pad until a stop signal is received from the Task Signaler in the Execute stage, at which point the next task should be fetched from the Task Queue. In addition to sending a stop signal, the Task Signaler can place another task for later processing into the Task Queue. If no tasks are present in the Task Queue, the PMC can be switched to a low power state until another task enters the Task Queue.

Evaluating Notify Signals requires a small amount of memory

and a small number of instructions. Most evaluations require output in the form of an average; for example, average cache miss rate, average number of cycles per instruction, and average mispredictions per branch. If the average exceeds a certain threshold, then we consider it a problem. For example, if the average miss rate in an L1 data cache is 75% and CPI is high, then poor L1 cache use is causing poor performance.

With the PMS, the PMC evaluates cache miss rate and CPI instead of a program on an execution core. Evaluating Notify Signals requires calculating averages which require little memory. Such behavior is what allows the monitor to utilize such a small scratch-pad memory, which is independent of the system memory space. This enables applications to continue without affects from the monitor.

The PMC's operations are simple, often involving only a few instruction, so numerous tasks can complete in a fairly short period of time. Due to the simplicity of the design, floating-point computation was found to be unnecessary for the PMC, and was removed from the PMC's ISA. Keeping perfect averages is costly due to complexity of hardware division. Fortunately, performance counters never guarantee perfect numbers, nor are they required to. To achieve a weighted average, we weight 50% to the current average and 50% to the old average. When an average is computed, the current sum is generated from the summation of the current value plus the last value divided by two; which is a single downshift. The current average and the old average are then summed and divided by two. This gives an efficient weighted average without the use of division. More precise averages are achievable, but were never required in our experiments.

Programming an event driven processor differs from coding for conventional processors. When a Notify Signal enters the PMC, it indexes into an Address Translation Table (ATT) which then maps the signal to the proper instruction address for execution. For example, in Figure 4, an incoming Data Cache Miss signal is translated into the proper address and placed on the Task Queue.

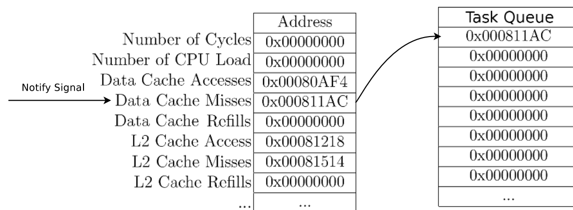

Figure 4: Address Translation Table of the Monitor.

Execution processors must be able to program the PMC to capture specific Notify Signals. A programmer can achieve this via writes to the ATT. The ATT is part of the monitor's internal memory system, which is set through a default ROM or the kernel using memory mapped I/O. To allow direct writes into the scratch-pad, the PMC is exposed to the Operating System through memory mapped I/O, allowing the Operating System to read/write directly into the scratch-pad. When a write signal is sent from an execution core to the Interrupt Controller for the purpose of modifying the ATT, current PMC execution is suspended and the Load/Store unit of the PMC is used to modify the content of the ATT. An execution processor invalidates an address in the ATT to alleviate processing of unnecessary Notify Signals. Alternatively, when an address is set in the ATT, the PMC instructs the appropriate PMD to begin collecting data for the event that was set in the ATT.

If the PMC's Task Queue overflows, then Notify Signals that could potentially be useful are lost. This possibility is due to the numerous system events available [1] and the ability for a single instruction to trigger multiple system events. The number of Notify Signals received must be controlled, otherwise, the Task Queue will always be full and the PMC will not be responsive. First, if the Notify Signal is not needed, we can disable its detection (which is the default start state stored in the ROM). To reduce the number of system events placed on the Task Queue, certain counters only let the nth instance of each system event to issue a Notify Signal. Hence, located on the PMDs on the path from system event detection to the PMC, there is a mask that prevents all but the nth event through.

While simple, these masks still take up space and must be interfaced with (to allow the mask to be set). Also, having the counter-mask as a component of the PMD means fewer signals are sent to the PMC, but more wires must be placed in order to interface with each PMD. Alternatively, counter-masks placed in the PMC means fewer wires that are used more often. Thus, if heat is an issue of great concern, counters can be placed on the PMDs to dissipate heat across the chip. If area needs to be conserved and heat is not a concern, then the counter's mask can be placed in the PMC. Due to modern heat dissipation concerns, it is logical to assume that the counter masks should be located on the PMDs.

When initiated, the PMC runs its own simplified kernel to initialize memory and initiate the default event handlers in the ATT. This kernel resides in a ROM next to the scratch-pad memory.

2.3 Monitor Instructions

Simulation was done in M5 [3] using the Alpha ISA. Alpha instructions are 32 bits wide and start with a 6-bit op code [5]. Since opcode 0x07 is unused, all monitor instructions start with 0x07. Since more than one instruction is needed, the next four bits describe which monitor instruction to execute. This leaves 22 bits to be used as fields for 16 individual instructions. Only the PMC needs the extra instructions because the execution cores interact with the system using regular read/write instructions to the reserved address space.

The PMC must be able to set Notify Signal tasks, check/set hardware, and signal when a task has completed. Below are the additional PMC instructions.

wcmpm: Write Counter Mask Performance Monitor: This instruction sets the mask in the PMD to filter the number of Notify Signals sent to the PMC.

wepm: Write Event Performance Monitor: This instruction places the proper address into the ATT. If the address placed was NULL, then the respective PMD is switched off.

pmhalt: Performance Monitor Halt: This instruction tells the PMC that processing of the current task has ended. Hence, the monitor may stop executing instructions and is now free to execute another task or enter a low power state.

cpm: Clear Performance Monitor: This instruction is issued when all tasks and PMDs need to be switched off. This instruction is always run at system start up to ensure everything is cleared.

pmchk: Performance Monitor Check Signal: This instruction queries the system to see what architecture features are used, and places 1 or 0 into the return register. For example, if the current Cache Coherency model does not produce any Write Update packets, there is no need to count them.

pmsig: Performance 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.

It is the kernel's job to change hardware state, but sometimes the decision is better left to the monitor. Hence, **pmchk** and **pmsig** were created. These two instructions take care of architectural pol-

icy changes. Software changes must be passed to the kernel, and sometimes this is preferred so the kernel keeps track of the current hardware state. When the monitor decides that the kernel needs to take over execution, the **pmsig** instruction generates an interrupt for the kernel to take over.

An interrupt occurs when a system-wide change is needed or the kernel needs information. 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 briefly interrupt each execution core and be used to determine which processor contains the conflicting process. The kernel may then record the problematic process and schedule accordingly.

As the number of cores increases, a single PMS 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 PMCs may exist throughout the system. Inter-Monitor communication is simple; one PMC sends a Notify Signal to another for processing. For instance, if multiple PMCs are detecting Cache Protocol invalidates, each one can be programmed to send their "cycles per invalidate" every expected number of cycles. The receiving PMC can then figure out the number of cycles per invalidate and make changes if needed.

Note that the receiving PMC has a higher chance of having its task queue overflowed. In normal performance monitoring, if few low priority Notify Signals are dropped it is considered acceptable because we are studying general trends. With monitor communication, lost signals may be unacceptable. Hence, the instruction to send a Notify Signal may write to a buffer that clears itself when the Notify Signal is successfully written to the Task Queue. Another possibility is to wait for the signal to be successfully received before executing the next instruction. This can unfortunately lead to deadlock, so a buffered write is preferred. If the buffer is full, then the Notify Signal is ignored and the programmer of the monitor must rewrite the code to prevent buffers from overflowing.

3. MANAGING ARCHITECTURAL FEATURES WITH THE PERFORMANCE MONITORING SYSTEM

Performance counters provide a mechanism to detect bottlenecks. Changing certain hardware policies can avoid bottlenecks in programs. For instance, if a multithreaded program shares data often, a certain type of Cache Coherence scheme may provide better performance over another. With current performance counters, a program would have to check the system state at intervals. These periodic checks change the reference behavior and slow down useful execution which in turn changes the performance counter's data. The PMS, on the other hand, is passive to the executing process and can make decisions without changes to memory or execution behavior. Hence, the monitor makes the decision of what Cache Coherency Protocol to use during execution.

Write-update Cache Coherence Protocols were created to deal with the conflict between 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 consumer 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, write-update protocols cause more memory traffic than is sometimes necessary. For example, in 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, write-update protocols would generate more memory traffic on the cache interconnect than a write-invalidate one. Since memory is often a bottleneck, this effect is exacerbated. For our simulation, and data collection, we chose both MOESI(write-invalidate) and Dragon(write-update) due to their similarities in hardware requirements [8].

Some problem domains cannot be rewritten to benefit from implemented Cache Coherency policy. Current hardware manufactures decide what type of Cache Coherence scheme to implement and provide little or no options once printed. Instead of forcing designers to make such an early decision, both MOESI and Dragon were implemented in hardware with a PMS that can adaptively switch between these two algorithms, depending on which appears to be most efficient for the task at hand. Since all MOESI states are valid Dragon states, and all Dragon states are valid MOESI 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.

3.1 Implementation

Both MOESI and Dragon contain 5 states. For this paper we will refer to the 5 states through the MOESI protocol: $Modified$, $Owned$, $Exclusive$, $Shared$, and $Invalidate$. These two protocols are identical, except when a write request hits a cache line that is $Owned$ or $Shared$. In MOESI, the write first invalidates all other cache lines, sets its cache line to $Modified$, and completes the write. In Dragon (write-update), the write broadcasts the request to other caches, changes its state to $Owned$, and completes the write. Cache lines receiving a write-update request change their state to $Shared$. Further details of Cache Coherency protocols are left to the reader.

Each L1 cache in the system is capable of switching between MOESI and Dragon. Since the states of MOESI and Dragon are the same, only the cases of a write during the $Owned$ or $Shared$ cache line states must be modified. In that case, a switch in the Cache will determine which packet is sent: invalidate for the MOESI protocol or write for the Dragon protocol. As mentioned before, Dragon eliminates write invalidates while increasing traffic on the cache interconnect. Hence, if the caches are using MOESI and there is a significant amount of write invalidates sent on the cache interconnect, the cache 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 Dragon is placing unneeded traffic on the cache interconnect.

The PMS records the number of cycles, number of write-invalidates, number of accesses to $Owned$ state cache lines, and number of accesses to $Shared$ state cache lines. The average of write-invalidates signify how often possession of data moves from one cache to another. When the PMS detects a significant high rate of write-invalidates, it will signal a switch to the Dragon protocol. On the other hand, a low rate of accesses to $Owned$ state lines with little or no accesses to $Shared$ state cache lines signifies superfluous write-updates. In this case the PMS changes Cache Coherence protocol to MOESI.

While both MOESI and Dragon are similar, they can not be immediately switched. All cache traffic must stop and clear out before a switch can be made due to conflicts between write-update and write-invalidate. Luckily, since all valid Dragon states are valid

MOESI states, the caches do not need to be flushed. Hence, when a switch is made the caches stall until the bus is empty and then the switch is made.

Simulation was done in the M5 simulator using the ALPHA ISA. Linux kernel 2.6.13 with a small number of applications were run on 4 execution cores, while a hand-written kernel ran on the PMC. Each execution core has a 32kB L1 instruction cache and a 64kB L1 data cache. Both of these L1 caches were connected to a bus which was connected to a 2MB L2 cache which is then connect to a 512MB physical memory. Both protocols were implemented on the L1 cache level for each execution core.

	Dragon	MOESI	Monitor
Barnes	0.00000%	23.9870%	23.9871 %
LU	25.7403%	0.00000%	25.7404 %
Python	24.0111%	0.00000%	24.0120 %
Cholesky	23.2743%	0.00000%	23.2743 %
Combined	0.00000%	3.45476%	15.6111 %
Average	14.6051%	5.28835%	22.525 %

Figure 5: Cache coherency performance improvement.

Each program was run with both Dragon and MOESI separately. Then, each program ran with the PMS having the initial Cache Coherence protocol set to the worse performing of the two available. We found the PMS quickly detected the poor performance of its initial protocol and changed the Cache Coherency accordingly. Figure 5 show the performance improvements of MOESI, Dragon, and the PMS compared to the slowest running policy for that benchmark. A final configuration was used, *combined*, in which both BARNES and LU benchmarks from SPLASH2 were run sequentially on the same hardware configuration. Then, the same benchmarks were run using the PMS which allows a quick detection and switch between the policies thus outperforming both Dragon and MOESI.

Since a wide variety of applications are commonly run on such multicore processors, architectural changes are needed to ensure the smooth running of each program and the entire multicore system. The *combined* configuration shows a real world example of two different programs executing (LU and BARNES). The PMS performs better then either MOESI or Dragon because it can adaptively change an architectural feature when needed.

We have shown that the PMS can change the Cache Coherency scheme to ensure Cache coherency is not the bottleneck in multithreaded applications. Grahn and Stenström [10] explored combining a write-update and write-invalidate protocol. We are doing the same thing except using a general purpose monitor, which reduces logic needed in caches. Also, we can change other architectural features, such as, prefetch policies, voltage scaling, or select the most appropriate network-on-chip topology available. Each change can be done independent of the OS, which removes overhead.

4. KERNEL LEVEL SUPPORT

Scheduling is the process of deciding which task to run next and for how long. Naturally, a scheduler should provide a processes enough time to be responsive without making other processes seem unresponsive or cause performance problems due to context switching. The Linux kernel takes many factors into account including process priority, frequency of interrupts, and NUMA when selecting the next process to run. Unfortunately, some behaviors of processes are still hidden from the kernel. For instance, the kernel cannot detect how often a process requests memory outside of its L1 cache. 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.

We propose using the monitor to help aid the kernel in determining memory bound processes. 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 identified processes more efficiently.

The kernel must first determine if a process is memory-bound or CPU-bound. Unfortunately, the kernel does not have a means to distinguish memory-bound processes. 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 time frame 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. Fortunately, a kernel knows how much I/O occurs and can give a good estimate of when *load-up* is complete.

Thus it is the kernel's responsibility to profile the running program. For efficiency concerns, the kernel wants to interact with the PMS as little as possible. Hence the monitor is always calculating if a certain CPU is running a memory bound process or not. When the kernel believes that enough time has passed to determine if a process is memory bound or not, then the kernel reads the value from the monitor.

A device driver was written for the PMC and the Linux kernel's scheduler was augmented to take memory bound programs into account. In the new Memory Bound Kernel (MBK), when a process is created it is flagged as not memory bound. After two time slices have passed, the MBK queries the PMC if the process is memory bound and flags the current process accordingly. Now, when a memory bound process enters the run queue, a counter representing the number of running memory bound processes is incremented. When a memory bound process leaves, the counter is decremented. If a memory bound process is about to enter the run queue and the global counter is not 0, then the kernel looks into the CPU's run queue for non-memory bound program that can run. If one is found, then the scheduler schedules the non-memory bound program. Otherwise, the memory bound program is scheduled. There are mechanisms in the kernel to move programs from one CPU's run queue to another, but these were not implemented.

To evaluate overhead of the new kernel, we compare the standard kernel with our modified one. The performance overhead was below .1%. To create a real-life example we ran a four-threaded instance of OCEAN from Splash2 and four-threaded WATER-NSQUARED on a quad processor machine as describe in the cache coherency section. OCEAN generates significant bus traffic that slows other OCEAN threads while WATER-NSQUARED produces little bus traffic. The modified kernel was able to detect the memory-bound programs and schedule accordingly to create a 17% performance improvement for the overall system.

We have shown that the PMS is capable of providing feedback

to software to enable better performance. The kernel in our example rearranges the scheduling of processing to efficiently use the memory system.

5. DEBUGGING SUPPORT

Software debugging is an important part of the software life cycle. Multicore debugging is hard because of unrepeatable errors. When a thread is interrupted to be debugged, the change in cache behavior and the pausing of the thread can affect the execution of other threads. Unfortunately, most recent architectural improvements do not provide any hardware support to help multicore debugging. Instead, software must handle most debugging facilities which incurs a significant performance penalty of 10-100 times original execution length and may not always provide replicable results in a multicore system [31].

The PMS in this paper can provide debugging support. Instead of injecting code to look for an event, the monitor is capable of looking for events and doing some analysis. This analysis determines if any actions need to be taken.

One vital aspect of debugging is setting watchpoints to allow a programmer to pause execution and inspect the state of the program. Hardware-Assisted Watchpoints allow debuggers to set addresses that trigger an exception. These exceptions pause execution and give control to the kernel. The kernel does the necessary processing and gives execution control to the debugging software. Unfortunately, most hardware architectures provide few hardware-assisted watchpoints and do not provide a range of addresses to watch. Hence, a significant number of watchpoints must be implemented in software which incurs a large performance penalty.

To combat the significant performance penalty, the monitor system can assist the debugger in handling watchpoints. A debugger can program the monitor to look at certain memory addresses. Once programmed, the monitor will throw an exception when the proper memory address is accessed.

5.1 Architectural Design

Implementing watchpoints requires detection of watchpoint addresses and preserving the processor state when the watchpoint is triggered. Detection is handled through the monitor detectors and the monitor core, while preserving processor state is done through stalling the processor itself.

PMDs located in the memory system contain a CAM to identify when an event has occurred. This CAM can detect when a certain address matches a set of watchpoints. When a match is found, the performance counter sends a Notify Signal to the PMC and pauses the execution core. The monitor receives the Notify Signal and sends an interrupt to the processor to handle the watchpoint or continue execution.

The performance counter CAMs are not large, and thus have a small set of addresses that they can watch. Luckily, CAMs can be implemented with a “do not care" bit which can be used to specify a range of addresses. Hence the CAM can detect a wide range of watchpoint while using very little storage. While such an approach will generate false hits, the monitor can check that the watchpoint hit was correct and notify the execution processor to continue executing or have the kernel take over. This provides a trade-off between detecting more addresses and a number of false hits.

This model may still be slow due to the high amount of other events that can enter the PMC. To combat this, the monitor’s high priority Task Queue is used for watchpoints since the execution of a process is stalled until the task in the PMC is completed.

6. COMPILER FEEDBACK

A modern compiler implements numerous code improving transformations. Many of the optimizations may only be useful for certain programs. For example, software pipelining can improve the performance of long running loops. Unfortunately, if the program consists mainly of small short lived loops, this “optimization" is not effective and increases code size. Some other code improving transformations like “if-conversion" can negatively impact the use of other optimizations. Usually it is the programmers job to select the proper switches for the given program. Currently, for aggressively optimized code, the compilation process is a simple repetitive loop: several optimizations are selected and the code is compiled, the code is then executed and metrics (ex: cycles) are collected, the metrics are compared to the previous executions, finally new optimizations are selected and testing begins again. Chow and Wu [4], aimed to reduce the number of executions required before achieving the optimal or near optimal optimized binary.

While such advances in reduction of steps is helpful, the end product is essentially the result of a wide range trial and error. The PMS described in our research allows the addition of a new dimension to compiler optimization selection. As previously mentioned, optimizations such as software pipelining may be beneficial for some loops, while detrimental to others. In such a scenario, watchpoints described in the previous section can be set around loops thus allowing data collection for each. Data specific to code segments can then allow selection of optimizations via correlation between statistics and known optimization benefits. This means that at the end of the program’s execution, one loop having sequential data access and a single backwards taken branch would be an ideal candidate for software pipelining. Fine granularity optimization selection is the next logical step to whole program compilation flags.

While similar data can be collected in utilizing modern methods is possible, the PMS allows such collection with unmatched speed. To collect fine grain statistics required for such optimization selection requires either a simulation of the execution or several runs using modern performance counters. The first, simulation, is inherently slow but allows the collection of all the required data in one execution (similar to the PMS). The second alternative, performance counters, would slow the execution of the program and, as mentioned earlier, potentially skew some of the statistics due to interference with the data collection. The natural choice then is utilizing the transparent wide range data collection of the PMS.

7. RELATED WORK

IWatcher [4] is a system which provides architectural support for watchpoints in SMT machines. IWatcher adds a table to the Re-Order Buffer to identify when an address is in a set of watchpoints, or in a range of watchpoints. When a hit occurs, the original thread stalls and a microthread spawns to check if the address is a watchpoint. Then, one of the threads is squashed and the other continues. If the SMT CPU supports TLS (Thread Level Speculation) the original thread does not stall, because TLS allows a rollback if a watchpoint is found.

While TLS provides some performance improvement, it still comes at the cost of a second thread creation. Alternatively, we don’t foresee any loss in performance when using the PMS. The CAMs in the PMD filter most address, and when the processor is paused, caches are still fetching the data for the processor. Furthermore, the creation and execution of microthreads can have a significant performance impact on execution processes.

Most modern architectures include hardware assistance for mon-

itoring, usually via performance counter registers. These suffer common shortcomings [27]: usually too few counters, sampling delay, and lack of address profiling. Some systems address these deficiencies by augmenting hardware capabilities. For instance, the Pentium 4 [15, 28] contains 18 performance counters that tag μops when they trigger certain events. These events are counted only when the μops retire, ensuring that events from speculative instructions do not pollute samples. This microinstruction-targeted solution also overcomes sampling delay. However, such mechanisms only collect aggregate statistics via sampling: they cannot react to single events or collect additional data, e.g., load addresses, which prohibits direct correlation of observed events with the data structures causing them.

The Itanium Processor Family [14] allows single-event detection, such as memory accesses or branch mispredictions. Access mechanisms provide microarchitectural data, but exceptions are required on a per-event basis to deliver these data to consuming software. The process using the data experiences frequent interrupts, causing system perturbation at many levels. Overhead costs of these mechanisms, particularly with respect to time, limit the extent to which software can reasonably exploit them.

Owl [26] is a framework that deploys programmable elements throughout a system, placing hardware monitors at event sources. These monitors run and write back results autonomously with respect to the CPU, avoiding large system overheads of interrupt-driven monitoring, and removing the need to communicate irrelevant events to higher level software. The framework applies to a broad range of monitoring approaches and techniques, including memory fault isolation, heap activity analysis, and complex memory access pattern detection. Owl is not intended for design debugging during hardware development or fault recovery (although it may have uses in these arenas): instead it seeks to improve understanding of whole-system behavior.

Martonosi et al. [18] use the coherence/communication system in the Stanford FLASH multiprocessor as a performance monitoring tool. *FlashPoint* is embedded in the software coherence protocol to monitor and report memory behavior in cache miss counts and cache miss latencies, inducing a 10% execution time overhead. Performance monitoring boards in each SHRIMP node enable histograms of incoming packets from the network interface [17]. A threshold-based interrupt signals application software, and the operating system takes proper action in response to specified events. The SMiLE project [16, 13] develops a hardware monitor to detect memory layout problems for SCI-based clusters,using this monitor to guide data layout visualization and transparent data migration [19].

Helms et al. [12] present a novel approach for monitoring disjoint, concurrent operations in heavily queued systems. A non-obtrusive activity monitor acts as an on-chip tracing unit. In contrast to conventional tracing units that record information from one or more functional units for later analysis, they directly record an operation's path through the system, enabling immediate analysis of inconsistencies. The unit generates a unique signature for each path followed, and time stamps are used to measure and debug queuing effects and system timing behavior. The tracing method is used on the I/O chip in IBM's S/390 G5 and G6 Systems.

The MemorIES [20] project also exploits reconfigurable logic (FPGAs) for flexible monitoring, It plugs into an SMP bus to perform on-line emulation of multiple cache configurations and protocols while the system is running real workloads, and does not slow the running applications. This server design tool is not intended for online data analysis and application optimization, and is restricted to observing the memory bus. Stratified sampling [25] provides profiled support for dynamic optimizations via hardware that samples then hashes the event stream into substreams, and samples those, yielding very low overhead. ProfileMe [6] provides hardware mechanisms for instruction-based profiling such that the hardware selects instructions and collects information on them as they flow through the pipeline. In both systems, information is post-processed by software. They are not programmable, and do not perform semantic data reduction, thus they rely on sampling. Zhou et al. [30] track the page miss ratio curve (the page miss rate vs. memory size curve) to analyze efficiency of memory usage and then use this information to improve memory allocation and energy usage.

The BlueGene/P system includes a novel hybrid performance counter architecture that supports tracking of many concurrent events. Infrequently changing high-order bits are stored in dense SRAM, while frequently updated low-order counter bits are tracked using latches with dedicated 12-bit incrementers. The performance monitoring architecture includes support for per-event thresholding and fast event notification, using a two-phase interrupt-arming and triggering protocol. A first implementation provides 256 concurrent 64-bit counters, offering many more counters with higher resolution compared to counters typically found in current off-the-shelf microprocessors. [22]

Fields et al. [9] present an architectural model based on interaction costs, proposing performance monitoring hardware (a *shotgun profiler* similar to ProfileMe [6]) to measure costs by execution sampling. Hardware records information for one instruction at a time, and then assembles post-mortem graph fragments of specific instruction sequences in the same manner as shotgun genome sequencing.

Other projects focus on hardware monitoring for debugging and software reliability. iWatcher [32] uses a memory range table in hardware to detect memory accesses outside of allocated memory regions. If a violation occurs, iWatcher calls a registered software handler for the event. Xu et al. [29] introduce a flight data recorder that can replay coherence events in symmetric multiprocessors, and Prvulovic et al. [21] use thread-level speculation mechanisms to roll back the state of threads when data races occur.

Phoenix [24] performs much the same functionality that we do, but is limited to defect detection. Phoenix taps key logic signals, and, based on downloaded defect signatures, combines signals into conditions that flag defects. On defect detection, Phoenix flushes the pipeline and either retries the instruction or invokes a customized recovery handler. Like Owl, it relies on FPGAs to process monitored data and can correlate condition signals that may be related and signal defects.

Heil and Smith [11] presented a per core performance monitor to reduce overhead in profiling VMs such as JAVA. The design (Relation Profiling Architecture, or RPA) filters incoming events based on number of events that satisfy a simple Boolean expression. For simple profiling, such as garbage collection and edge profiling, RPA reduces overhead to $.5\%$. RPA does not perform any analysis by itself, instead it must interrupt the Operating System to do analysis of the data. While useful for simple single threaded profiling, RPA does not tackle multi-core profiling such as Cache Coherency.

8. FUTURE WORK

Providing a mutex lock is an expensive task required by many multi-threaded applications. The PMS can aid in more efficiently acquiring and releasing these locks. When a process enters a mutex, the kernel takes control and requests a lock from the PMC. The PMC, being a sequential core, can quickly grant/deny the request and inform the kernel of the appropriate action.

Just In Time Compilers (JIT) use profilers to help provide more efficient code production. While the net result is one providing faster executing code, the use of profilers comes at a cost in performance since clock cycles are required for profiling. The PMS can profile the running application and provide sufficient data to the JIT Compiler with minimal performance impact.

The availability of multiple cores on a single chip has called attention to heterogeneous core design and pushed the task of selecting the optimal execution core on to the compiler/programmer. The PMS can detect, at run-time, process characteristics and inform the kernel which core is best suited for the given application. For example, many processes have a load-up phase that is best run on a general purpose processor, but then enter a specialized phase unique to the application, such as a data processing phase which could be best served by a streaming processor. The monitor can detect this change during runtime and inform the kernel that the process would run more efficiently on a different core.

Buffer overflow detection can create both security issues and bugs in program execution. The PMS can assist in detection of these overflows by watching memory in a similar fashion to the watchpoints discussed earlier. A significant difference is that the PMD does not have to stall a CPU when a possible overflow occurs, thus having no affect on performance. In general, this approach will not create security risks, but there may still be enough time for a malicious program to break into a system before the PMS could inform the kernel to intervene. If security is of high priority, the PMD is capable of stalling the CPU by treating the overflow as a watchpoint. Similar software solutions exist, but are costly, unlike the PMS.

Event driven programming is the corner stone for modern software and GUI applications. Generally, a *tick* loop is written to update the state of the application and check for many conditions that may have changed since the last iteration of the loop. If a condition is found to be true, a thread is assigned to handle the appropriate function correlated with the event. The PMS provides event detection and thus can create a more intuitive programming model for event driven function calls and thread creation. If for example, a user could replace the *if* statement checking the condition for the function call with a global *when* statement, the PMS can be programmed to watch for the provided conditions and handle the thread creation. While such an approach does not improve performance, it does allow for a more intuitive method for event detection thus allowing programmers to write cleaner code.

9. CONCLUSION

The plethora of research on advanced performance monitors shows that existing mechanisms are insufficient for providing introspective data required by future generations of multicore computing systems. Most modern performance counters are limiting (and, in fact, were not designed for public consumption, but rather as hardware debugging aids during system "bring-up" time[1]). On mainstream systems with over 80 different events but as few as four to six counters, many program executions are required to collect all needed statistics. The alternative, simulation, can potentially track all possible events, but this "accuracy" is rarely verified, and comes at the cost of severe degradation in execution times. Our solution, the Performance Monitoring System, tracks all events with no performance penalty. Further, the PMS scales for multicore designs and can track inter-core events, something impossible with existing performance counters.

The Performance Monitor System can improve performance in architectural policies (like cache behavior) and kernel-level software policies in real time. Furthermore, we have discussed how assisting debugging (like watchpoints) and compiler profiling (for optimization selection) can be efficiently implemented in current and future multicore designs.

[1]A noteworthy exception is the Performance Monitoring Unit in the BlueGene/P series, which supports 256 counter [23].

10. REFERENCES

[1] S. B. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, Jul/Aug 2002.

[2] W. Binder. Portable and accurate sampling profiling for java. *Softw. Pract. Exper.*, 36(6):615–650, 2006.

[3] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[4] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. *2nd Workshop on Feedback Directed Optimization*, 1999.

[5] Compaq. Alpha architecture handbook. *whitpaper*, October 1998.

[6] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. IEEE/ACM 30th International Symposium on Microarchitecture*, pages 292–302, Dec. 1997.

[7] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. 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.

[8] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Computer Aided Verification*, pages 53–68, Dec. 2006.

[9] B. Fields, R. BodÂĂik, M. Hill, and C. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proc. IEEE/ACM 36th International Symposium on Microarchitecture*, pages 228–239, Dec. 2003.

[10] H. Grahn and P. Stenström. Evaluation of a competitive-update cache coherence protocol with migratory data detection. *J. Parallel Distrib. Comput.*, 39(2):168–180, 1996.

[11] T. Heil and J. E. Smith. Relational profiling: Enable thread-level paralelism in virtual machines. *Microarchitecture, IEEE/ACM International Symposium on*, 0:281, 2000.

[12] M. Helms, T. Büchner, R. Fritz, T. Schlipf, and M. Walz. Event monitoring in a system-on-a-chip. In *Proc. 12th Annual IEEE International ASIC/SOC Conference*, Sept. 1999.

[13] R. Hockauf, J. Jeitner, W. Karl, R. Lindhof, M. Schulz, V. Gonzales, E. Sanquis, and G. Torralba. Design and implementation aspects for the SMiLE hardware monitor. In G. Horn and W. Karl, editors, *Proc. of SCI-Europe 2000, The 3rd International Conference on SCI-Based Technology and Research*, pages 47–55. SINTEF Electronics and Cybernetics, Aug. 2000. ISBN: 82-595-9964-3, Also available at http://wwwbode.in.tum.de/events/.

[14] Intel. *Intel Itanium Architecture Software Developer's Manual*, 2000.

[15] Intel. *Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2002.

[16] W. Karl, M. Leberecht, and M. Schulz. Optimizing data locality for SCI-based PC-clusters with the SMiLE monitoring approach. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 169–176, Oct. 1999.

[17] M. Martonosi, D. W. Clark, and M. Mesarina. The SHRIMP performance monitor: Design and applications. In *ACM SIGMETRICS Performance Evaluation Review*, pages 61–69, May 1996.

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

[19] T. Mu, J. Tao, M. Schulz, and S. McKee. Interactive locality optimization on NUMA architectures. In *Proc. ACM 2003 Symposium on Software Visualization (SoftVis)*, pages 133–142,214, July 2003.

[20] A. Nanda, K. Mak, K. Sugavanam, R. Sahoo, V. Soundararajan, and T. Smith. MemorIES: a programmable, real-time hardware emulation tool for multiprocessor server design. *SIGPLAN Not.*, 35(11):37–48, 2000.

[21] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proc. 30th IEEE/ACM International Symposium on Computer Architecture*, pages 110–121, June 2003.

[22] V. Salapura. Bluegene/p performance counters. Personal Communication: Paper in Submission, Nov. 2007.

[23] V. Salapura, K. Ganesan, A. Gara, M. Gschwind, J. Sexton, and R. Walkup. Next-generation performance counters: Towards monitoring over thousand concurrent events. *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 139–146, April 2008.

[24] S. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware. In *Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture*, pages 26–37, Dec. 2006.

[25] S. Sastry, R. Bodík, and J. Smith. Rapid profiling via stratified sampling. In *Proc. 28th IEEE/ACM International Symposium on Computer Architecture*, pages 278–289, July 2001.

[26] M. Schulz, B. White, S. McKee, H. Lee, and J. Jeitner. Owl: Next generation system monitoring. In *Proc. ACM Computing Frontiers Conference*, May 2005.

[27] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, pages 64–71, July/August 2002.

[28] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, pages 72–82, July/August 2002.

[29] M. Xu, R. Bodik, and M. Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay. In *Proc. 30th IEEE/ACM International Symposium on Computer Architecture*, pages 122–135, June 2003.

[30] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. 11th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 177–188, Oct. 2004.

[31] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: efficient architectural support for software debugging. *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 224–235, June 2004.

[32] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iWatcher: Efficient architectural support for software debugging. In *Proc. 31st IEEE/ACM International Symposium on Computer Architecture*, pages 224–237, June 2004.