
THE BASICS OF PERFORMANCE-MONITORING HARDWARE

PERFORMANCE-MONITORING FEATURES PROVIDE DATA THAT DESCRIBE HOW AN APPLICATION AND THE OPERATING SYSTEM ARE PERFORMING ON THE PROCESSOR. THIS INFORMATION CAN GUIDE EFFORTS TO IMPROVE PERFORMANCE.

..... Most modern, high-performance processors have special, on-chip hardware that monitors processor performance. Data collected by this hardware provides performance information on applications, the operating system, and the processor. These data can guide performance improvement efforts by providing information that helps programmers tune the algorithms used by the applications and operating system, and the code sequences that implement those algorithms.

Performance event monitoring

Performance events can be grouped into five categories: program characterization, memory accesses, pipeline stalls, branch prediction, and resource utilization. Program characterization events help define the attributes of a program (and/or the operating system) that are largely independent of the processor's implementation. The most common examples of these events are the number and type of instructions (for example, loads, stores, floating point, branches, and so on) completed by the program. Memory access events often comprise the largest event category and aid performance analysis of the processor's memory hierarchy. For example, memory events can count references and misses to various caches and transactions on the processor memory bus. Pipeline stall event information helps users analyze how well the

program's instructions flow through the pipeline. Processors with deep pipelines rely heavily on branch prediction hardware to keep the pipeline filled with useful instructions. Branch prediction events let users analyze the performance of branch prediction hardware (for example, by providing counts of mispredicted branches). Resource utilization events let users monitor how often a processor uses certain resources (for example, the number of cycles spent using a floating-point divider).

Performance-monitoring hardware

Performance-monitoring hardware typically has two components: performance event detectors and event counters. By properly configuring the event detectors and counters, users can obtain counts of a variety of performance events under various conditions.

Users can configure performance event detectors to detect any one of several performance events (for example, cache misses or branch mispredictions). Often, event detectors have an event mask field that allows further qualification of the event. For example, the Intel Pentium III's event to count load accesses to the level 2 cache (L2_LD) has an event mask that lets event detectors monitor only accesses to cache lines in a specific state—modified, shared, exclusive, or invalid.¹

The event detector configuration also allows qualification by the processor's current

Brinkley Sprunt
Bucknell University

privilege mode. Operating systems use supervisor and user privilege modes to prevent applications from accessing and manipulating critical data structures and hardware that only the operating system should use directly. When the operating system is executing on the processor, the privilege mode is supervisor; when an application is executing on the processor, the privilege mode is user. As such, the ability to qualify event detection by the processor's privilege mode allows counting of events caused only by the operating system or only by an application. Configuring the event detector to detect events for both privilege modes counts all events.

In addition to counting events detected by the performance event detectors, users can configure performance event counters to count only under certain edge and threshold conditions. The edge detection feature is most often used for performance events that detect the presence or absence of certain conditions every cycle. For these events, an event count of one represents a condition's presence and zero indicates its absence. For example, a pipeline stall event indicates the presence or absence of a pipeline stall on each cycle. Counting the number of these events gives the number of cycles that the pipeline stalled.

However, the edge detection feature can also count the number of stalls (more specifically, the number of times a stall began) rather than just the total number of cycles stalled. With edge detect enabled, the performance counter will increment by one only when the previous number of performance events reported by the event detector is less than the current number being reported. So when the event detector reports zero events on a cycle followed by one event on the next cycle, the event counter has detected a rising edge and will increment by one. It's usually possible to invert the sense of the edge detection to count falling edges. For these events, disabling the edge detection feature counts stall durations, and enabling edge detection counts the number of stalls. Dividing the total stall duration by the number of stalls gives the average number of cycles stalled for a particular stall condition.

The event counter's second major feature is threshold support. This capability lets the event counter compare the value it reports each cycle to a threshold value. If the report-

ed value exceeds the threshold, the counter increments by one. The threshold feature is only useful for performance events that report values greater than one each cycle. For example, superscalar processors can complete more than one instruction per cycle. Selecting instructions completed as the performance event and setting the counter threshold to two would increment the counter by one whenever three or more instructions complete in one cycle. This provides a count of how many times three or more instructions completed per cycle.

Performance profiles

Although performance event detectors and counters can easily detect the presence of a performance problem and let the user estimate the severity of the problem, it's often necessary to find the locations of the code (whether in the application or the operating system) that are causing the performance problem. Knowing the source of the performance problem lets programmers alter the high-level algorithms used by the application and/or the low-level code to avoid or reduce the problem's impact. To illustrate how performance counters can help create a profile that identifies the major sources of performance problems, let's first review the goals and techniques used to create time-based profiles.

Time-based profiles

A common technique to identify areas upon which to focus tuning efforts is to obtain a time-based profile of the application. A time-based profile estimates the percentage of time an application spends in its major sections.² Focusing tuning efforts on the application's most frequently executed sections maximizes the benefits of any performance-tuning changes made to the code.

A time-based profile relies upon interrupting an application's execution at regular time intervals. During each interrupt, the interrupt service routine saves the value of the program counter. Once the application completes, the user can create a histogram that shows the number of samples collected for each program counter value. Assuming that the histogram draws from many program counter samples, it will show the application's most frequently executed sections.

Event-based profiles

A technique similar to that for creating a time-based profile can help collect an event-based profile. An event-based profile is a histogram that plots performance event counts as a function of code location. Instead of interrupting the application at regular time intervals (as is done to create a time-based profile), the performance-monitoring hardware interrupts the application after a specific number of performance events has occurred. So just as a time-based profile indicates the most frequently executed instructions, an event-based profile indicates the most frequently executed instructions that cause a given performance event.

To support event-based sampling (EBS), performance-monitoring hardware typically generates a performance monitor interrupt when a performance event counter overflows. To generate an interrupt after N performance events, the performance counter is initialized to a value of overflow minus N before being enabled. A performance monitor interrupt service routine (ISR) handles these interrupts. The ISR saves sample data from the program (for example, the program counter) and re-enables the performance event counter to cause another interrupt after N occurrences of a certain performance event. After the application finishes executing, the user can plot the data samples saved by the ISR to create an event-based profile.

Improving performance

To illustrate the use and benefits of performance-monitoring hardware, let's examine two examples. The first example demonstrates how performance-monitoring data help tune the memory accesses of a high-level algorithm for better performance on a specific processor. The second example demonstrates how performance-monitoring data can find a low-performing code sequence, which programmers can then modify to improve performance.

Identifying memory access problems

Sometimes, algorithms developed for a particular processor generation don't perform as well on a successive generation. Consider the following simple algorithm for finding prime numbers: The algorithm creates a large array where each element in the array represents an integer and begins by initializing all array ele-

ments to 0. To find prime numbers, the algorithm repeatedly steps through the array with successively greater strides starting with a stride of 2. On all but the initial steps, storing a value of 1 into the element marks the array elements. This mark indicates that the number corresponding to the array element is not prime. For example, the first pass through the array would use a stride of 2 and would mark locations 4, 6, 8, and so on. The second pass would use a stride of 3 and mark locations 6, 9, 12, and so on. Once the algorithm passes through the entire array, the array elements that are still 0 correspond to prime numbers.

Now, consider how this algorithm performs on the original Pentium and Pentium Pro processors.³ Both processors have an 8-Kbyte L1 data cache, but the Pentium Pro has an additional 256 Kbytes of unified L2 cache. In general, it seems that the overall performance of the Pentium Pro would surpass that of the Pentium because of its more advanced design and the presence of the large, unified L2 cache. However, a subtle difference in the allocation policies for these caches can cause some performance surprises. The Pentium does not allocate a new cache line for a store that misses the cache, but the Pentium Pro does. When the Pentium Pro allocates a new line to the caches, it must load the whole line—32 bytes of data—from memory.

In steady state, the prime number algorithm, described previously, accesses a large array (much larger than the caches) using a large stride. Thus, almost every store to the prime number array will miss the caches. On the Pentium, each store miss results in one bus transaction between the processor and memory because the Pentium does not allocate a new cache line. However, because the Pentium Pro allocates a new cache line on each store miss, each store miss causes four bus transactions to load the 32-byte cache line using the 8-byte-wide processor memory bus. Also note that, in steady state, most of the lines in the Pentium Pro caches are dirty (they have data that is more recent than that in main memory), because they are holding the 1s stored in the prime number array. As such, when the Pentium Pro allocates a new line, it must write a dirty line back to memory, requiring another four bus transactions. As such, the steady-state behavior of this algorithm causes only one

bus transaction for each store to the prime number array for the Pentium, but causes eight bus transactions for the Pentium Pro.³ This eight-times difference in required bus bandwidth is a bottleneck for the Pentium Pro, and its performance on this algorithm is low compared to that of the Pentium.

Initially, the reasons for the Pentium Pro's lower performance on the prime number algorithm were far from obvious. However, Intel's analysis of the cache and bus performance data collected using the hardware performance monitors made this problem apparent. Once the Intel computer architects understood the problem, they realized that a slight change in the algorithm would substantially reduce the processor memory traffic for the Pentium Pro. Instead of blindly storing a 1 in the array elements on each algorithm step, the algorithm should first test the array value. It should only write a 1 if the existing array value is 0. This simple, one-line change to the algorithm eliminates the majority of dirty cache lines because it only writes each element in the prime number array once. This change significantly reduces processor memory traffic for the Pentium Pro by eliminating the write back of dirty cache lines to main memory. It brings the Pentium Pro's performance to a level on par with or better than that of the Pentium. Without the performance-monitoring capabilities of these processors, the investigation and resolution of this performance problem would have been much more difficult.

Eliminating partial register stalls

For another example of the usefulness of hardware performance-monitoring features, consider the performance problem of partial register stalls on P6-based Pentium processors (these include the Pentium Pro, Pentium II, and Pentium III along with their Xeon and Celeron versions).⁴ On these processors, a partial register stall occurs when an instruction writes the lower 8 or 16 bits of a 32-bit register and another, closely following instruction reads the full 32 bits from the register. Here, register renaming⁵ causes each write of a register to go to a different physical register. Consequently, the newly written 8 or 16 bits reside in a different physical register than the unchanged upper bits. A subsequent instruction cannot read the register's full 32 bits until

the partial components of the register's value are merged as the instructions retire. This situation causes a stall of at least seven cycles, but typically lasts from 10 to 14 cycles.

To determine whether or not an application causes partial register stalls, a user can configure the performance event detectors and event counters to count the number of partial-register-stall cycles as the application runs. If the frequency of these cycles as a percentage of the application's total cycles is significant (that is, greater than 3 percent), finding and eliminating the partial register stalls will probably increase performance.

Once users identify partial register stalls as a significant performance problem, they can use the EBS support of the performance counters to locate the most frequent occurrences of these stalls. Users can use the collected samples to create a histogram of sample counts versus code location. This histogram identifies the specific locations in the code that cause the majority of partial register stalls so programmers can modify the code to eliminate them.

Alternative approaches

Several other approaches to collecting processor performance data don't rely on performance-monitoring hardware. These approaches typically require modifying the application to collect data about its own execution, as with the Etch tool.⁶ This instrumentation of the application requires either rebuilding it from source code or modifying its executable version. The modified version adds additional code to collect various data, which typically include instruction trace and memory reference data. Other tools then process these data to produce performance data. However, this approach is sometimes difficult to use—it's not always feasible to rebuild an application. The source code and build tools are not always available, for example. Also, these approaches can disturb the application's behavior, bringing into question the validity of the collected data.

Another way to collect processor performance data is by using a simulator to model the processor as it executes the application. This simulation approach can yield a wealth of data (for example, detailed data on pipeline stalls, branch prediction, cache performance, and so on). However, processor manufactur-

Software tools for performance-monitoring hardware

A variety of software tools provide a high-level interface and analysis capabilities for performance-monitoring hardware. The Performance Application Programming Interface (PAPI) tool suite (<http://icl.cs.utk.edu/projects/papi/>) provides a common interface to performance-monitoring hardware for many different processors, including Alpha, Athlon, Cray, Itanium, MIPS, Pentium, PowerPC, and UltraSparc. Intel's VTune Performance Analyzer (<http://developer.intel.com/software/products/vtune/vtune60/index.htm>) supports the performance-monitoring hardware of all Intel Pentium and Itanium processors, and provides additional performance analysis tools such as call graph profiling and processor-specific tuning advice. The Digital (now Hewlett-Packard) Continuous Profiling Infrastructure (<http://www.tru64unix.compaq.com/dcp/index.html>) for Alpha processors uses performance-monitoring hardware to provide a low-overhead facility for performance monitoring of an entire system. The Rabbit Performance Counter Library (<http://www.scl.ameslab.gov/Projects/Rabbit>) provides a high-level interface to P6-based Pentium processors and AMD Athlon processors on Linux systems. The Brink and Abyss tools (http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtml) provide a high-level interface to the Pentium 4 performance-monitoring features on a Linux system.

ers do not usually provide simulators for advanced processor designs. These simulators are also difficult to develop without detailed information about the processor's implementation, details of which the manufacturer does not typically divulge. Even if a simulator is available, its accuracy is always in question, and it's often difficult to get a complex application to run in a simulated environment. Simulation speed is also significantly slower than running the application on the real processor, slowing data collection and making it infeasible to simulate some applications.⁷

Using performance-monitoring hardware has several distinct advantages over these other approaches. Having the processor itself actually collect performance data as it executes an application avoids several problems. First, the application and operating system remain largely unmodified, apart from the addition of drivers in the operating system to enable access to the hardware performance counters. As such, using hardware performance counters does not require tools to instrument the application, and the disturbance to the system under analysis is minimal. Second, not using a simulation of the application, operating system, or processor ensures that the accuracy of the collected event counts. Third, performance-monitoring hardware collects data on the fly as the application executes, allowing full-speed data collection and avoiding the slowness of simulation-based approaches. Fourth, this approach can collect

data for both the application and the operating system.

These advantages often make hardware performance monitoring the preferred, and sometimes only, choice for collecting processor performance data.

Performance-monitoring capabilities today

Many of today's high-performance processors have hardware performance-monitoring capabilities. The "Software tools for performance-monitoring hardware" sidebar contains a discussion of software interface and analysis tools for some of these processors.

Advantages

Intel's P5-based processors (these include the original Pentium and Pentium MMX) and P6-based processors provide two performance counters that support privilege mode qualification, threshold comparisons, and interrupt generation on counter overflow.¹ These processors also provide a large set of performance events (more than 80 on the Pentium III). AMD's Athlon processor⁸ provides four performance counters with capabilities similar to the Pentium III, but with a smaller set of performance events (about 25). IBM's PowerPC 750 processor⁹ provides four counters with capabilities similar to the Pentium III and supports more than 40 events. Motorola's PowerPC 7450 provides six counters and over 200 events.¹⁰ An additional feature of the PowerPC performance-monitoring hardware is the ability to count performance events only for marked processes on a system running multiple processes. This feature lets a process indicate that its events should be counted, excluding those events generated by any unmarked process.

Intel's Itanium processor provides four performance counters and over 90 performance events.¹¹ In addition to supporting privilege-mode qualification, threshold comparisons, and interrupt generation on counter overflow, Itanium allows event qualification by opcode as well as instruction and/or data address range. Itanium events also account for all major single- and multicycle stall and flush conditions. These events let the user determine the percentage of all cycles spent on various activities, such as execution, data and instruction accesses, and branch mispredic-

tions. Unlike most other processors, Itanium contains a set of event address registers that records the instruction and data addresses for cache and transition look-aside buffer (TLB) misses. These event address registers identify data addresses and structures that cause frequent cache and TLB misses.

Sun's UltraSparc I and II provide two performance counters and qualification by privilege mode for over 20 events, but they have no threshold support and cannot generate interrupts on counter overflow.¹² Compaq's Alpha 21264 provides two performance counters that support privilege-mode qualification and interrupt generation on counter overflow.¹³ The Alpha 21264 was also the first processor to support ProfileMe, a novel profiling technique, discussed in the "Alpha's ProfileMe approach" sidebar.¹⁴

Limitations

The performance-monitoring support for these processors all suffer from a common set of problems.

Too few counters. First, the processors have only a limited number of counters (ranging from two to six) that can count concurrently. Therefore, collecting a variety of performance data requires running the application many times, because the hardware can only concurrently count a small subset of events. Even the Motorola PowerPC 7450 with six performance counters requires many application runs to collect a complete set of data for all performance events.

For many events, users might want to calculate the ratio of event count to the number of instructions retired. This ratio helps gauge a performance problem's severity, and the instruction count can confirm the consistency between different runs of the same application. So, it's often the case that one of the few counters available is used repeatedly to collect instruction counts. In general, having more counters would reduce the number of application runs needed to collect performance data.

Speculative counts. Second, for processors that detect performance events for instructions that do not complete, the problem of speculative versus nonspeculative counts arises. Most high-performance processors have deep

Alpha's ProfileMe approach

Digital Equipment Corp. (now Hewlett-Packard) developed the ProfileMe¹ method for instruction-level profiling as part of the Digital Continuous Profiling Infrastructure (<http://www.tru64unix.compaq.com/dcpi/index.html>). ProfileMe differs from the profiling support offered by other processors described in this article. Instead of capturing the program counter's value upon the overflow of a performance counter, the ProfileMe approach uses hardware that, when triggered, causes the processor to collect performance event information for a single instruction as it flows through the processor pipeline. Thus, this approach collects the entire performance lifetime for one execution of the instruction. Internal registers hold this information, which includes data such as the program counter, pipeline stage latencies, cache hit/miss information (including addresses), branch target addresses, and whether the instruction retires or is cancelled. Once the instruction completes, the ProfileMe hardware generates an interrupt, and the interrupt service routine collects the information as a sample for building a performance profile. The first implementation of ProfileMe appears in the Alpha 21264/EV67 processor.²

Although collecting independent samples of instruction lifetimes can reveal much about an instruction's typical performance, on more advanced processors, the interaction of multiple instructions executing concurrently can often reveal more information about the stalls and bottlenecks that an application encounters. To better depict the interaction of concurrently executing instructions in the processor's pipeline, ProfileMe also supports paired sampling, where, for each instruction profiled, a second instruction that may execute concurrently with the first instruction is also profiled. These paired samples enable the creation of various concurrency and utilization metrics, such as an estimate of the number of cycles during a particular instruction's lifetime in which the processor did not issue other instructions.

ProfileMe's more detailed picture of how instructions flow through the pipeline permits a wider variety of metrics than is possible with more traditional profiling approaches. This information can help find and analyze performance bottlenecks to improve application performance as well as aid in the design of future processors.

References

1. J. Dean et al., "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Symp. Microarchitecture (Micro-30)*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 292-302.
2. *Alpha 21264/EV67 Microprocessor Hardware Reference Manual*, order no. DS-0028B-TE, Compaq Computer, Houston, Texas, 2000; http://www.support.compaq.com/alpha-tools/documentation/current/21264_ev67/21264ev67_hrm.pdf.

pipelines to improve performance. However, branches in a program often change the flow of instructions into the pipeline, causing a significant performance loss when the processor flushes the pipeline and fetches the correct instruction stream. To reduce the frequency of pipeline flushes, most processors use a form of branch prediction to predict the outcome of conditional branch instructions. Consequently, these processors execute most instructions speculatively (that is, before program execution determines which execution path is

correct). Some of these speculatively executed instructions are along incorrect execution paths and therefore never complete. The instructions on the correct path complete and then retire. Because speculatively executed instructions that do not complete can still cause performance events (for example, cache misses and pipeline stalls), it's often important to separate events caused by speculatively executed instructions that never retire from those that do retire. The speculative count for an event comprises events caused by all instructions, independent of whether or not the instruction that caused an event retires. Nonspeculative counts refer to event counts caused by instructions that retire.

Sampling delay. The third and probably the worst problem with most performance-monitoring hardware is the imprecision of EBS support. EBS lets the user identify the set of instructions causing the majority of occurrences of a specific performance event. Once the instruction is identified, the programmer can attempt to recode the application to avoid or reduce the performance problem's impact. Most EBS hardware simply monitors a particular performance counter and signals a performance monitor interrupt (PMI) when the counter overflows. The PMI ISR then saves the value of the program counter for the interrupted instruction. However, the current instruction is usually not the instruction that caused the performance counter to overflow.

The processor's pipeline and the delay between counter overflow and the signaling of an interrupt combine to create an arbitrary distance (measured in terms of retired instructions) between the instruction causing the performance event and the instruction whose address the PMI ISR saves. For example, consider an event-causing instruction that creates a counter overflow during the fifth stage in a 15-stage pipeline. If the event immediately caused an interrupt, at least 10 earlier instructions (assuming only one instruction per pipeline stage) remain in the pipeline. The PMI ISR would incorrectly identify an instruction prior to the event-causing instruction.

Alternatively, consider an event-causing instruction in the pipeline's last stage that makes the counter overflow. If only one instruction was in this last stage and the

processor signaled the interrupt immediately, then the PMI ISR would correctly identify the event-causing instruction. However, this situation is rarely the case. In a superscalar processor, the pipeline's last stage can contain multiple instructions. Also, several cycles elapse before the processor generates an interrupt request, allowing additional instructions to retire. In these cases, the PMI ISR would incorrectly identify an instruction that followed the event-causing instruction. Consequently, EBS typically does not correctly identify the instructions causing the performance problem. The team that developed ProfileMe found that an EBS profile of a single source of data cache misses on Intel's Pentium Pro resulted in a widely distributed profile that spanned 25 instructions.¹⁴ Inaccuracies like these make it difficult to use the EBS data and often an expert on the processor's implementation is needed to interpret the data properly.

Lack of data-address profiling. A fourth problem with most performance-monitoring hardware is that there is no support for collecting profiles of data addresses used by the program that cause performance events in the memory hierarchy (a notable exception to this is Itanium's event address register support). Just as profiles of a problematic instruction address (obtained via EBS) help identify problems in the application or operating system code, profiles of data addresses that cause performance problems (for example, cache and TLB misses) could help characterize and avoid performance problems in the memory hierarchy. To support the creation of address profiles, the processor must either capture samples of problematic data addresses or capture enough state—the instruction and the register values, for example—to compute these addresses.

The general capabilities of performance-monitoring hardware described in this article have been used extensively to analyze application, operating system, and processor performance. These analyses have helped improve not only application and operating system code but also compilers and next-generation processor designs. However, as discussed previously, the performance-monitoring support offered by most processors is limited

(too few counters, lack of support for distinguishing between speculative and nonspeculative event counts, imprecise event-based sampling, and lack of support for creating data-address profiles). The recent introduction of Intel's Pentium 4 and Pentium 4 Xeon processors provides performance-monitoring capabilities that overcome these limitations, while also providing full support for the simultaneous multithreading capabilities of the Pentium 4 Xeon processor. A companion article, *Pentium 4 Performance-Monitoring Features*, on page 72 of this issue details these performance-monitoring improvements and describes their implementation and use.

MICRO

References

1. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, order no. 245472, Intel, Santa Clara, Calif., 2002; <http://developer.intel.com/design/pentium4/manuals/>.
2. *GNU Gprof*, Free Software Foundation, Boston, 1998; <http://www.gnu.org/manual/gprof-2.9.1/gprof.html>.
3. *Intel Architecture Optimizations Manual*, order no. 242816-003, Intel, Santa Clara, Calif., 1997; <http://developer.intel.com/design/pro/manuals/>.
4. *Intel Architecture Software Optimization Reference Manual*, order no. 245127-001, Intel, Santa Clara, Calif., 1999; <http://developer.intel.com/design/PentiumIII/manuals/>.
5. D. Sima, "The Design Space of Register Renaming Techniques," *IEEE Micro*, vol. 20, no. 5, Sept./Oct. 2000, pp. 70-83.
6. T. Romer et al., "Instrumentation and Optimization of Win32/Intel Executables Using Etch," *Proc. Usenix Windows NT Workshop*, Usenix Assoc., San Diego, Calif., 1997, pp. 1-8.
7. K. Keeton et al., "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads," *Proc. 25th Int'l Symp. Computer Architecture*, CS Press, Los Alamitos, Calif., 1998, pp. 15-26.
8. *AMD Athlon Processor, x86 Code Optimization Guide*, AMD, Sunnyvale, Calif., 2002; <http://www.amd.com/products/cpg/athlon/techdocs/pdf/22007.pdf>.
9. *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*, publication no. GK21-0263-00, IBM, White Plains, N.Y., 1999; <http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7785256996006C28E2>.
10. *MPC7450 RISC Microprocessor Family User's Manual*, no. MPC7450UM/D, Motorola, Schaumburg, Ill., 2001; <http://www.motorola.com/brdata/PDFDB/docs/MPC7450UM.pdf>.
11. *Intel Itanium Architecture Software Developer's Manual*, rev. 1.1, vols. 1-3, Intel, Santa Clara, Calif., 2000.
12. *Ultra-SPARC-III User's Manual*, part no. 805-0087-01, Sun Microsystems, Palo Alto, Calif., 1997; <http://www.sun.com/oem/products/manuals/805-0087.pdf>.
13. *Alpha 21264 Microprocessor Hardware Reference Manual*, order no. EC-RJRZA-TE, Compaq Computer, Houston, Texas.
14. J. Dean et al., "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Symp. Microarchitecture (Micro-30)*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 292-302.

Brinkley Sprunt is an assistant professor of electrical engineering at Bucknell University. His research interests include computer performance modeling, measurement, and optimization. Sprunt has a PhD in electrical and computer engineering from Carnegie Mellon University. He previously worked at Intel where he was a member of the architecture teams for the 80960, Pentium Pro, and Pentium 4 projects. He is a member of the IEEE and ACM.

Direct questions and comments about this article to Brinkley Sprunt, Bucknell Univ., Electrical Engineering Dept., Moore Ave., Lewisburg, PA 17837; bsprunt@bucknell.edu.

For further information on this or any other computing topic, visit our Digital Library at <http://computer.org/publications/dlib>.

PENTIUM 4 PERFORMANCE-MONITORING FEATURES

THE INTEL PENTIUM 4'S UNIQUE PERFORMANCE-MONITORING FEATURES OVERCOME MANY LIMITATIONS AND PROBLEMS FOUND IN PREVIOUS PROCESSORS. PENTIUM 4 XEON PERFORMANCE MONITORING SUPPORTS SIMULTANEOUS MULTITHREADED EXECUTION FEATURES.

Brinkley Sprunt
Bucknell University

..... Most modern, high-performance processors have special, on-chip hardware that can monitor performance. The features of this monitoring hardware typically include event detectors and counters, qualification of event detection and counting by privilege mode and event characteristics, and support for event-based sampling. However, these features often suffer from a common set of problems including a small number of counters, inability to distinguish between speculative and nonspeculative events, and imprecise event-based sampling. With the introduction of the Pentium 4 processor, Intel has overcome many of the performance monitoring limitations of previous processors. The Pentium 4 supports 48 event detectors and 18 event counters, enabling the concurrent collection of a significantly larger set of performance event counts than any other processor. The Pentium 4 also provides several instruction-tagging mechanisms that enable counting of nonspeculative performance events (that is, events generated by instructions that retire). In addition to the support for imprecise event-based sampling (IEBS), the Pentium 4 also provides precise event-based sampling (PEBS) that unambiguously identifies instructions that cause key performance events. PEBS support also lets users create data address profiles for various memory

events. Because the Pentium 4 Xeon processor is the first implementation of the x86 architecture to support simultaneous multithreading (SMT), its performance-monitoring support includes qualification of event detection by thread ID and qualification of event counting by thread mode.¹

Event detectors and counters

At more than 42 million transistors, the Pentium 4 is significantly larger than its immediate predecessor, the Pentium III, which has 28 million transistors. With such a large design, the previous approaches used by Intel for organizing the performance event detectors and counters became impractical. These processor designs dispersed the event detectors across the chip, close to the units where the performance events occur; event counters were in a central location. This approach required routing all event count signals to the centrally located event counters.

To follow a similar approach in the larger Pentium 4 design would have required significantly more silicon area to route the event count buses to the event counters. Intel's computer architects considered two alternatives: incorporating a counter into the implementation of each event detector or dispersing several blocks of counters across the chip and letting geographically close event detectors

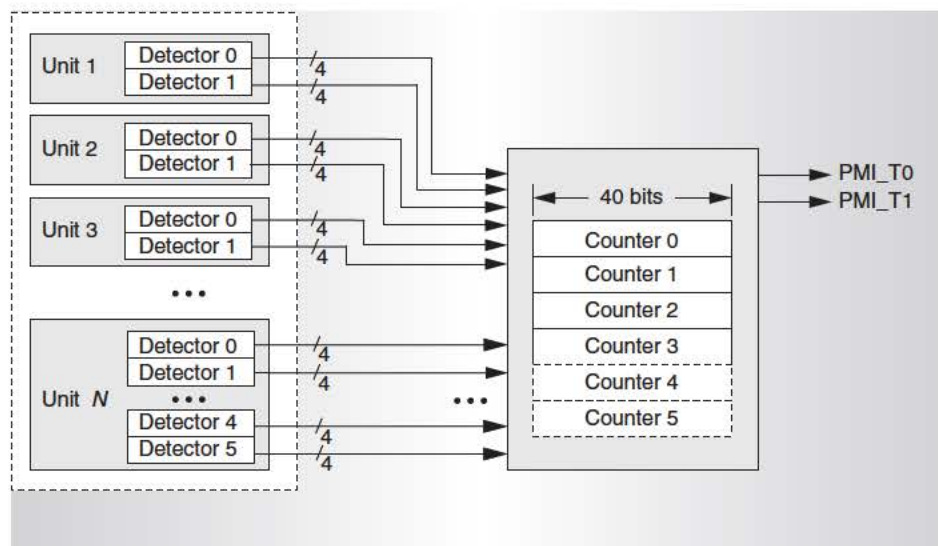


Figure 1. The general structure of the Pentium 4 event counters and detectors. The event detectors control the selection of events and qualification of event detection by privilege mode (OS and/or USR) and thread ID. Each Pentium 4 unit contains two or six event detectors. Each event counter block contains four or six counters, and each counter can select an event detector, perform threshold comparisons and edge detection, and qualify event counting by thread mode. The counters in the counter block can also signal performance monitor interrupts on counter overflow.

share these blocks. The first approach would eliminate signal routing among event detectors and counters while providing one event counter for each event detector. The second approach would reduce signal routing among event detectors because of the event detectors' close proximity to a counter block. The first approach would have eliminated routing costs and provided the user with more counters that they could use concurrently. Because the silicon area required to implement one counter per event detector was greater than that for implementing several shared groups of counters, Intel's computer architects chose the second approach.

Structure and function

Figure 1 shows the structure and function of these event detector and counter groups. The units in Figure 1 are logic sets that perform a significant function in the Pentium 4's design, such as the branch prediction or the trace cache units. Implementing the event detectors in pairs supports concurrent detection of the same event for different threads in a simultaneous multithreaded system. Most units have one pair of equally capable event

detectors, although the checker/retire unit has three event detector pairs.

Each event detector has a maximum 4-bit-wide output bus, providing a maximum increment per cycle of 15. The Pentium 4 implementation routes output buses from each event detector to an event counter block. Most event counter blocks contain four counters with the exception of the counter block in the instruction queue unit, which has six counters. Each counter block has two output signals for requesting interrupts on counter overflow for EBS. With this organization, many event detectors share a set of four (or six) counters. Therefore, one of these groups can only count four (or six) events concurrently.

The event detectors allow the selection and masking of an event and can qualify event detection by privilege mode and thread ID. Event counters support threshold comparison, edge detection, and thread mode qualification. They can also generate performance monitor interrupts on counter overflow to support EBS. Users can configure these features using event select control registers (ESCRs) and counter configuration control registers (CCCRs).

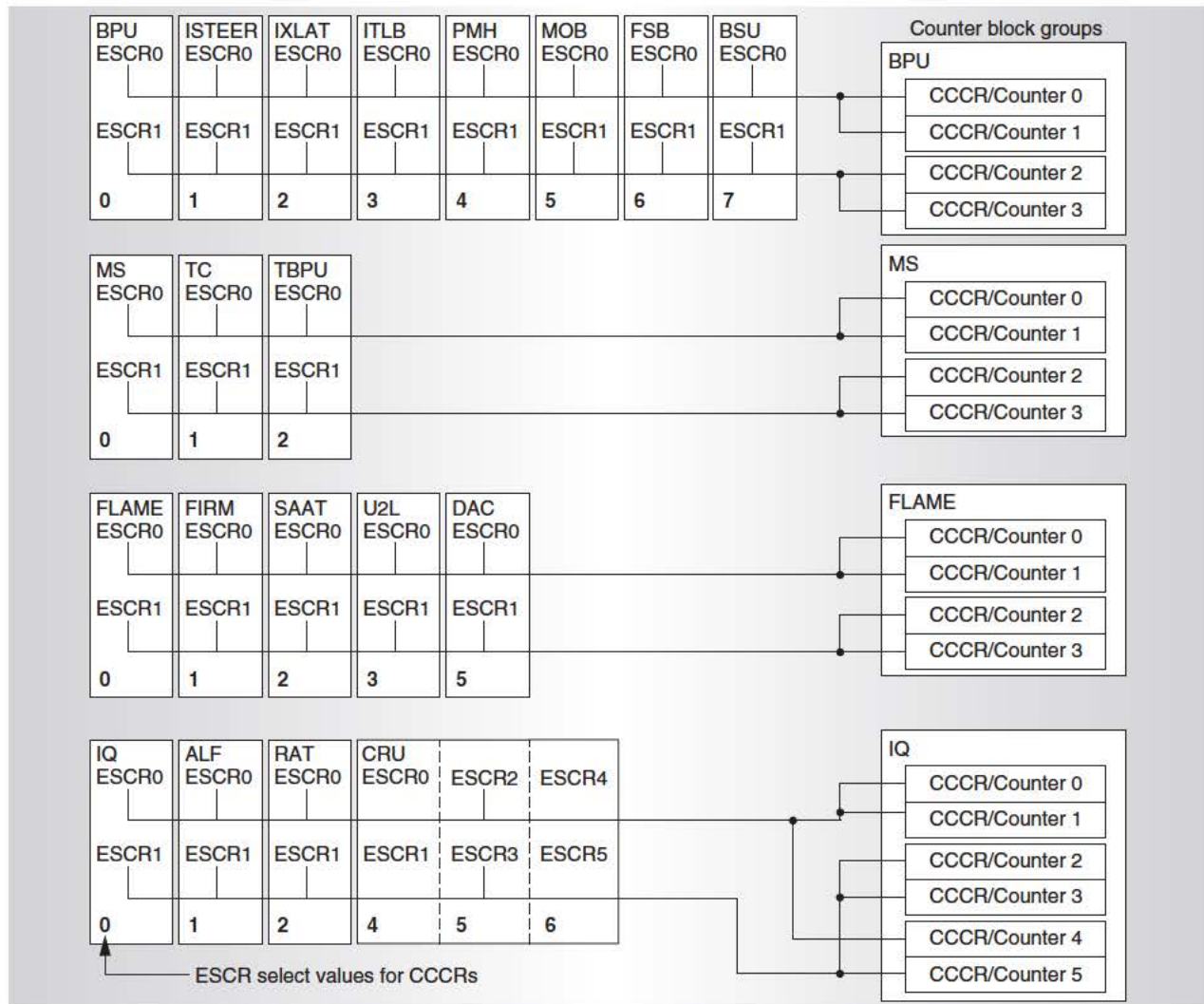


Figure 2. Interconnections among event detectors and event select control registers (ESCRs), and their associated counters and counter configuration control registers (CCCRs).

Groups

Figure 2 shows the four groups of event detectors and counters implemented in the Pentium 4. Each group consists of event detectors (containing ESCRs), shown for several units in the figure, and a block of counters (containing CCCRs). Designers named each group according to the Pentium 4 unit that contains the counter block for that group. For example, the branch prediction unit gives its name to the group that contains event counters in the BPU, ISTEER, IXLAT, ITLB, PMH, MOB, FSB, and BSU units, and these detectors connect to a counter block in the BPU that contains four CCCR-counter pairs. Counters in the counter blocks are grouped

into pairs, such as counters 0 and 1 in the BPU group. The paired counters share the same set of event detectors and ESCRs. As Figure 2 shows, although the Pentium 4 contains 44 event detectors, it can only count 18 events concurrently because only 18 counters are available. Even so, the ability to count 18 events concurrently is a substantial increase over the capabilities of other processors.

Event selection

An ESCR, shown in Figure 3, configures an event detector. The 7-bit event select field selects the desired event, and the 16-bit event mask field selects a subset of the desired event. For example, the Pentium 4 `branch_retired`

event defines 4 bits in the event mask field to select from the following branch types: taken, not taken, predicted, and mispredicted. To count mispredicted branches, the user would select the branch_retired event and then set the branch-taken-mispredicted and branch-not-taken-mispredicted bits in the event mask.

The ESCR's low-order 4 bits allow qualification of event detection by privilege mode and thread ID. The thread ID identifies one of the possible two threads concurrently executing via the Pentium 4's SMT capability. See the "Simultaneous multithreading: Improving instruction throughput for high-performance processors" sidebar (next page) for a brief introduction to the concepts, goals, and performance implications of the Pentium 4's SMT features. For example, to count all operating system events for both threads, the user would set both the T0_OS and T1_OS events. Defining each of these bits as a combination of both thread and privilege mode provides a greater degree of control than is possible if the event detector used 2 bits (T0 and T1) to select thread qualification and the other 2 bits (operating system, OS, and user, USR) to select privilege mode. For example, the user could count all user events for thread 0 and all operating system events for thread 1 (by setting the T0_USR and T1_OS bits), which is not possible with individual T0, T1, operating system, and USR bits in the ESCR.

Counter configuration

Figure 4 shows the bits and fields in a CCCR that

- enable the counter,
- select the event detector to use as the source for counter increments,
- qualify event counting by the processor's current thread mode,
- configure the counter's threshold and edge detection capabilities,
- configure the interrupt generation on counter overflow, and
- enable the cascading of paired counters.

The CCCR also has one status bit, OVF, to indicate whether an overflow has occurred.

Using an event counter requires enabling the counter and selecting the event detector to supply increment values to the counter. The

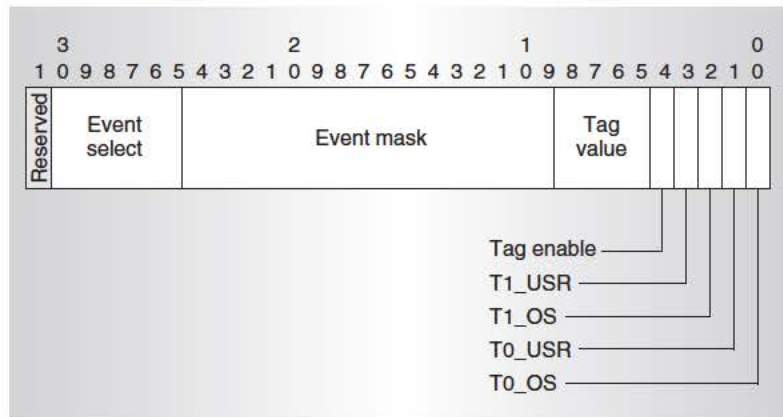


Figure 3. Event select control register.

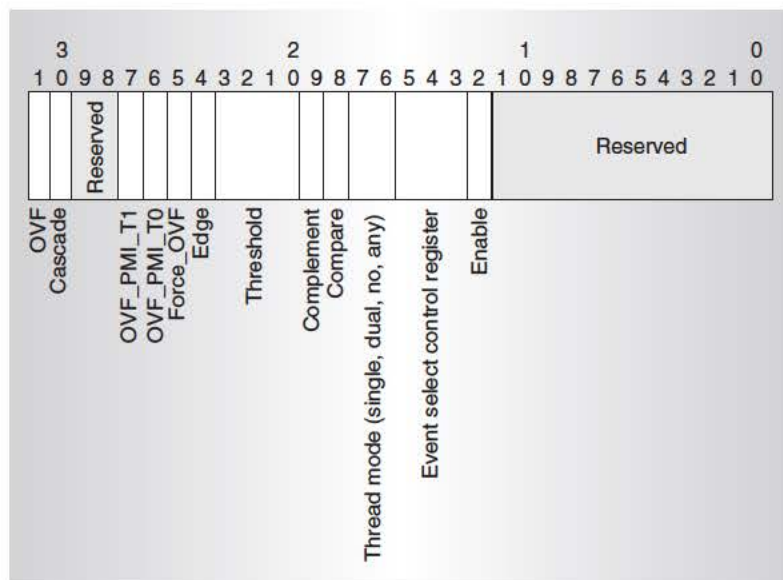


Figure 4. Counter configuration control register.

CCCR's enable bit turns on the counter, and the ESCR select field chooses the event detector output that the counter should use. In Figure 2, the bold numbers in the boxes for each pair of event detectors represent the value that the CCCR's ESCR select field uses to pick the indicated event detector.

The CCCR's compare, complement, and threshold fields control the counter's threshold detection capabilities. When the compare bit is set, the counter compares the incoming value from an event detector to the value in the threshold field. The complement bit determines the type of comparison. A zero complement bit selects a greater-than comparison. Setting the complement bit to one

Simultaneous multithreading: Improving instruction throughput for high-performance processors

Although modern, high-performance processors employ a wide variety of techniques to improve performance,¹ these processors often stall waiting for various conditions to resolve. These stalls leave the processor's pipeline and its multiple execution units idle for many cycles. Probably the most egregious of these stall conditions is a cache miss that can take tens or even hundreds of cycles to resolve. Although compiler writers and processor designers do their best to schedule useful work during a cache miss, these efforts are often not successful. Consequently, the processor seldom realizes its potential throughput (measured in the maximum number of possible instructions completed per cycle). In cases where only one task is ready to execute on the processor, nothing more can be done except to wait for the stall to clear. However, when more than one task is ready to execute (as is often the case for servers), the processor can improve its overall instruction throughput by concurrently sharing resources between two or more ready-to-run tasks.

Simultaneous multithreading

Consider a processor design that lets two tasks concurrently share resources. Such a processor would concurrently have instructions from both tasks flowing through its pipeline and using its resources. If neither task incurs a stall, each task would consume roughly half the processor's resources and execute at roughly half the rate it could if it were the only task on the system. Although this concurrent processing halves each task's execution rate, the processor's instruction throughput is basically the same as the throughput attained with only one task that does not stall.

Now consider the case where one of the tasks stalls. At this point, the other task can consume most of the processor's resources and approach

the execution rate it would attain if it were the only task. With this sharing technique, the processor's instruction throughput remains high, even though one task stalls. This dynamic sharing by two tasks lets the processor maintain a high instruction throughput in the face of various stall conditions.

This concurrent sharing of a processor's resources by two or more threads is called *simultaneous multithreading* (SMT),² and the Pentium 4 Xeon is the first implementation of the x86 architecture that supports SMT.³ A single Pentium 4 Xeon with SMT enabled can concurrently support two execution threads. As such, an SMT Pentium 4 Xeon appears to the operating system as a dual processor, even though it's physically only one processor. Because the operating system believes the system has two processors, it will assign ready-to-run tasks to both processors, allowing the sharing technique described previously to improve the overall instruction throughput.

Although SMT can improve overall throughput, it can also degrade throughput if concurrent sharing of processor's resources by multiple tasks causes excessive resource conflicts. Simple benchmarks can demonstrate both SMT-induced improvements and degradations in throughput.

Benchmark results

For example, let's examine the performance of two benchmarks running in single- and dual-thread modes on the Pentium 4 Xeon. These benchmarks are `fp_add_latency` and `l1_miss`. The `fp_add_latency` benchmark consists of a long series of serially dependent floating-point (FP) adds. The `l1_miss` benchmark generates a steady stream of independent memory loads, all of which miss the L1 cache. I obtained the throughput data discussed here using Pentium 4 Xeon's performance-monitoring support.

selects a less-than-or-equal-to comparison. When the comparison of the input and threshold values is true, the counter increments by one. When the threshold feature is enabled, the counter's edge detection capability can also be activated via the edge bit. With the threshold comparisons enabled and the edge bit set, the edge detection hardware compares the last and current threshold comparison results. The counter will increment by one only when the previous threshold comparison was false, and the current threshold comparison is true, thus detecting a rising edge on the threshold filter.

Since the Pentium 4 processor is the first implementation of the x86 architecture to support SMT, it also has performance-monitoring support that allows qualification of event counting by the processor's current thread mode. The CCCR's 2-bit thread mode field qualifies event counting by the processor's current thread mode. The four encod-

ings for this field represent each of the processor's possible thread modes.

- In *single-thread* mode, only one thread is active, and the processor dedicates all of its resources to that thread.
- In *dual-thread* mode, two threads are active, and they share the processor's resources.
- In *no-thread* mode, no threads are active. However, some resources must respond to other events in the system, and this activity can cause performance events. For example, consider the case of a dual-processor system where one processor is in no-thread mode, and the other processor is in dual-thread mode. The processor in dual-thread mode could initiate memory transactions that the cache of the processor in no-thread mode might need to service. An example would be a read-for-ownership transaction necessary

Consider the `fp_add_latency` benchmark. On the Pentium 4 Xeon, the FP adder can start a new FP add every cycle—each FP add has a latency of five cycles. Because the serial dependency between the FP adds is the bottleneck for the `fp_add_latency` benchmark, the benchmark attains a throughput of 0.21 microoperations (μ ops) per cycle (roughly one FP add every five cycles) when running in single-thread mode. When two copies of this benchmark run in dual-thread mode, the combined throughput is 0.41 μ ops per cycle, almost twice the throughput attained in single-thread mode. This example shows how SMT can significantly improve overall throughput by essentially doubling processor performance compared to running tasks sequentially in single-thread mode. The `fp_add_latency` benchmark improves performance because of the FP adder's long latency and the serial dependency between the adds keeps the FP adder utilization low. These features let two tasks share the FP adder and other processor resources with essentially no interference.

However, the performance of the `l1_miss` benchmark running in single-thread mode compared with two copies of the `l1_miss` benchmark running in dual-thread mode shows a very different throughput result. Because the memory loads in the `l1_miss` benchmark are all independent and each one misses the L1 cache, the memory system is quickly flooded with demand fetch requests from the L1 cache to the L2 cache.

In single-thread mode, the `l1_miss` benchmark attains a throughput of 1.23 μ ops per cycle. However, when the processor executes two copies of the `l1_miss` benchmark in dual-thread mode, the combined throughput is only 0.65 μ ops per cycle. This is roughly half the throughput attainable if the processor ran two copies of the benchmark sequentially in single-thread mode. This happens because the memory system is already significantly loaded with only one copy of the `l1_miss` benchmark running. Doubling the demand on the memory system by running another

copy of the benchmark in dual-thread mode results in excessive conflicts in the memory system, significantly degrading performance.

Thus, the Pentium 4 Xeon's SMT features can be both an advantage and a detriment to overall performance. To obtain the full potential of SMT while avoiding its pitfalls, operating systems must carefully select the tasks that will concurrently share an SMT processor. Researchers have proposed a new task-scheduling approach—*symbiotic task scheduling*—and investigated the use of hardware performance-monitoring data to guide the job scheduling on SMT processors.⁴ This study showed symbiotic task scheduling to improve response time performance by as much as 17 percent.

References

1. K. Diefendorff, "PC Processor Microarchitecture, A Concise Review of the Techniques Used in Modern PC Processors," *Microprocessor Report*, 12 July 1999.
2. D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, 1995, pp. 392-403.
3. D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology J.*, vol. 6, no. 1, 14 Feb. 2002, pp. 1-12.
4. A. Snively and D.M. Tullsen, "Symbiotic Jobscheduling for a Simultaneous Multithreading Processor," 9th Int'l Conf. Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, 2000, pp. 234-244; <http://www-cse.ucsd.edu/users/tullsen/aspl00.pdf>.

for modifying the cache line of the processor in no-thread mode.

- The *any* mode enables event counting independent of the current thread mode.

Other features

To support EBS, Pentium 4 counters can generate interrupts on counter overflow using the `OVF_PMI_T0` and `OVF_PMI_T1` fields in the CCCR (see Figure 4). Setting the `OVF_PMI_T0` bit will generate a thread 0 interrupt on counter overflow. Setting the `OVF_PMI_T1` bit will generate a thread 1 interrupt on counter overflow. Additionally, setting the `FORCE_OVF` bit configures the counter to overflow on every counter increment. This feature is useful when the user wants to collect profile information about all occurrences of an infrequently occurring event. Because several counters can generate interrupts on overflow, each CCCR contains an `OVF` bit that indicates that the corre-

sponding counter has overflowed. Upon an interrupt, the performance monitor interrupt's interrupt service routine (ISR) checks the `OVF` bits in the active CCCRs to determine which counter(s) overflowed. Note that EBS implemented in this manner suffers from the same inaccuracies observed for previous processors.

The Pentium 4 supports cascading counters, which the CCCR's cascade bit controls (see Figure 4). When two counters are cascaded, the first counter counts normally, but the second counter does not begin counting until the first counter overflows. By initializing the first counter to overflow after M events, the second counter to overflow after N events, and the second counter to generate an interrupt on overflow, the user can collect information on correlated events with a fairly fine degree of control. It's also possible to cascade corresponding counters in different counter pairs within a counter block. For example, BPU

counter 2 can start counting when BPU counter 0 overflows by setting the cascade bit and clearing the enable bit in counter 2's CCCR. When BPU counter 0 overflows, BPU counter 2 will be enabled to count.

Obtaining nonspeculative event counts

When counting events, it's often helpful to distinguish between events caused by speculatively executed instructions that never retire versus events caused by instructions that do retire. For example, instructions that are executed in the shadow of a frequently mispredicted branch often do not retire. These nonretiring instructions can significantly increase the counts for certain events when compared to event counts obtained only from instructions that retire. These counts can even exceed the number of events caused by instructions that retire. Using the total event count rather than the separated speculative and nonspeculative event counts can lead a performance analyst to draw flawed conclusions. For example, consider a program that has a frequently accessed data structure that is small enough to fit in the data cache. However, this program also has a frequently executed branch instruction that is often mispredicted. The instructions in the shadow of this mispredicted branch (which never retire) access data that is not needed by the program and that is not in the data cache. If performance analysts couldn't distinguish between the nonspeculative and speculative data cache miss counts, they could draw the erroneous conclusion that data needed by the program don't fit in the cache. This erroneous conclusion could then lead to futile efforts to tune the data structure's size and access patterns in an effort to improve cache performance. However, with the nonspeculative and speculative data counts properly separated, analysts could focus efforts on improving the prediction rate for the problematic branch and/or adjust the code such that speculatively executed instructions in the mispredicted branch's shadow do not miss the data cache.

The Pentium 4 uses tagging mechanisms to provide nonspeculative event counts. As the processor decodes each x86 instruction, it breaks it into a sequence of one or more simple operations called microoperations, or μ ops. The Pentium 4's tagging mechanisms enable tagging these μ ops when they cause certain performance events. Once tagged, a μ op

retains the tag until the μ op retires successfully or is cancelled. As the μ ops pass through the retirement logic, tagged μ ops can be counted, providing a nonspeculative event count.

The Pentium 4 implements three tagging mechanisms: front end, execution, and replay. The front-end tagging mechanism tags the μ ops responsible for events occurring early in the pipeline related to instruction fetch, instruction types, and μ op delivery from the trace cache. The execution tagging mechanism tags certain classes of μ ops as they write their results back to the register file. The replay tagging mechanism tags μ ops that are reissued (replayed) because of conditions such as cache misses, branch mispredictions, dependence violations, and resource conflicts. Both the front-end and replay tagging mechanisms have essentially one tag bit per μ op and thus can only handle one event at a time. However, the execution tagging mechanism has four tag bits available per μ op and, as such, can concurrently track up to four events. Several machine-specific registers and, in some cases, the tag and tag value bits of the ESCRs (see Figure 3) enable these tagging mechanisms.

Precise event-based sampling

The Pentium 4's PEBS support is a significant performance-monitoring advantage. Previous processors only supported IEBS, and the inaccuracy of these profiles often made them useless.

To provide PEBS support, the Pentium 4 takes a different approach to collecting sample data. Previous processors would merely issue a macroinstruction interrupt after a performance event counter overflowed, and the ISR collected the sample data. However, deep pipelines, superscalar execution, and latency between the counter overflow and the actual interrupt cause wide variations in the accuracy of samples collected in this manner. The Pentium 4 uses a microassist and a microcode-assist service routine to capture sample data for PEBS. A microassist changes the source of the next μ ops to be executed in a fashion analogous to the way an interrupt changes the source of the next instructions to be executed. Microassists typically handle infrequent or problematic conditions that occur during instruction execution (such as raising an exception when a divide-by-zero

error occurs). When a microassist is signaled, a microcode-assist service routine handles the condition that caused the microassist. The use of a microassist and a microcode service to collect samples for PEBS avoids the inaccuracies of IEBS.

Implementation

PEBS support on the Pentium 4 works in the following manner. The user allocates a PEBS buffer in memory to hold the samples to collect and sets a bit in the PEBS_ENABLE machine-specific register (MSR) to enable PEBS. The user then configures a μ op tagging mechanism to tag certain μ ops as they flow through the pipeline. The user also configures a counter to count tagged μ ops as they retire. Once the counter overflows, the Pentium 4's retirement logic examines all retiring μ ops. When it finds a tagged retiring μ op, it forces a microassist to occur just before the tagged μ op retires. A microassist is similar to a macroinstruction interrupt in that it halts the normal execution of instructions. However, unlike a software interrupt, the processor handles the microassist entirely in microcode, and no instructions after the microassist-causing instruction will retire before completion of the microassist service routine. The microassist service routine collects the actual sample data by storing the current program counter and the values in the general-purpose registers in the PEBS buffer allocated for the samples. After completing the microassist, the processor resumes normal execution and the event-causing instruction retires.

Using a microassist and a microcode-assist service routine to collect the sample data instead of a macroinstruction interrupt and an ISR avoids all of the inaccuracies of IEBS. Since the microassist is invoked immediately prior to the retirement of the event-causing instruction, the association of the sample data with the event-causing instruction is insured to be precise. Also, because the microassist is taken just before the event-causing instruction retires, the inaccuracies observed with IEBS caused by the latency between the retirement of the event-causing instruction and the invocation of an ISR to collect the sample data are no longer a factor that can affect the accuracy of the sample data.

The microassist checks a high watermark

for the PEBS buffer after storage of each sample. Once the buffer reaches this high watermark, an interrupt is signaled and the corresponding ISR copies the samples from the PEBS buffer to a more permanent location. With PEBS, the ISR copies samples already collected in the PEBS buffer to another location for analysis. It then empties the PEBS buffer to enable collection of more samples. Because the ISR does not collect the actual sample data when using PEBS, the latency of the interrupt no longer affects the accuracy of the event-based samples.

Buffering

Pentium 4 PEBS support has another advantage over IEBS: The overall overhead of event-based sample buffering is lowered because the PEBS ISR processes many samples in one invocation of the ISR. In contrast, IEBS requires a new invocation of the ISR for each sample. This improvement in sample-processing efficiency can either reduce the interference experienced by the monitored system or allow collection of more samples for the same level of interference.

Benchmark program

To demonstrate the advantages of PEBS over IEBS, I created a simple benchmark program. This benchmark uses a load instruction in a nested loop to cause many L1 data cache misses on the Pentium 4. Although only one load (the target load) in the nested loop actually causes a cache miss, the cache-missing load is preceded and followed by large sequential blocks of loads that always hit the cache. I then created event-based profiles for L1 data cache misses using PEBS and IEBS support on the Pentium 4.

Figure 5 (next page) summarizes the sampling results from this benchmark. For PEBS, 100 percent of the samples correctly identify the load instruction that misses the L1 data cache (identified as "+0" in Figure 5). In contrast, none of the IEBS samples identify the correct load instruction. Moreover, the instructions identified by IEBS were 65 or more sequential instructions after the target instruction (for example, 46.5 percent of all the IEBS samples identified a load instruction 69 instructions after the load instruction that misses the cache). This inaccuracy of IEBS on

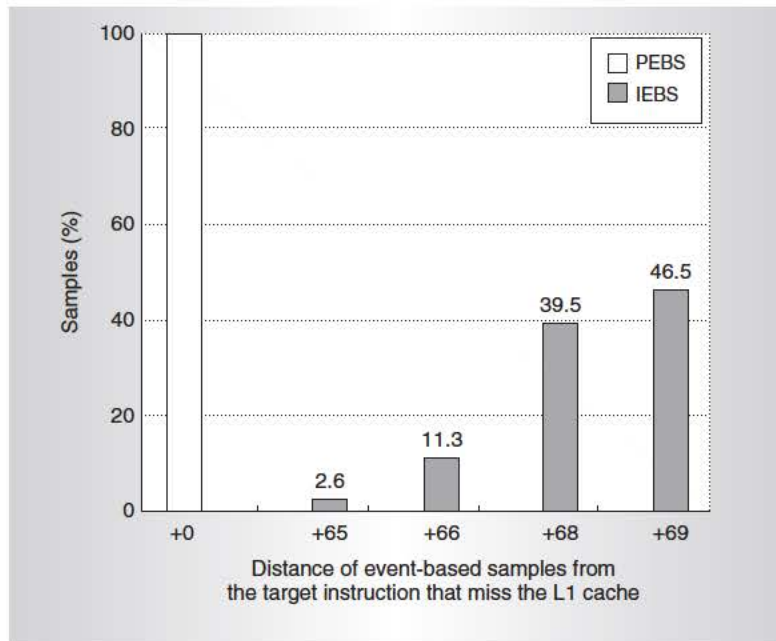


Figure 5. Comparison of the sample accuracy for precise and imprecise event-based sampling.

the Pentium 4 is much worse than that for the Pentium Pro processor as assessed by the ProfileMe team.² This greater inaccuracy arises from the Pentium 4's more aggressive design and the increasing difference between processor cycle time and main-memory access time. This greater inaccuracy also emphasizes the importance of the Pentium 4 PEBS support.

Pentium 4 PEBS support also enables the creation of data address profiles, a capability not possible with IEBS. A data address profile identifies locations in data memory that are associated with various memory system performance events, such as cache and translation look-aside buffer misses. The user can recreate the data memory address used by the load or store μ op instruction that caused the performance event. Doing so requires decoding the instruction identified by the PEBS sample and using the sampled register values to compute the effective address of the instruction's operands. By building a profile of these data memory addresses, programmers can identify the regions in data memory that cause common memory system performance problems. Once programmers identify these regions, it's often possible to rearrange an application's memory data structures to reduce the frequency of memory system performance

problems. In previous processors, it was not possible to create these profiles with IEBS because the IEBS samples often incorrectly identified the instructions causing memory system performance events.

Let's return to the `l1_miss` benchmark used previously in the IEBS and PEBS discussion to demonstrate the creation and use of data address profiles. To create an address profile for the `l1_miss` benchmark, I configured the replay tagging mechanism to tag load μ ops that miss the L1 cache. I also configured a counter to count tagged μ ops as they retire. This benchmark generated 80 million L1 cache misses. A second benchmark, with PEBS enabled, collected a sample every 100,000 L1 cache misses, generating 800 samples. As mentioned previously, all 800 samples identify the same instruction as missing the L1 cache. The address, opcode, and operands for this instruction are

```
80484c3: mov (%EAX), %EAX
```

This load instruction uses contents of the EAX register as the address for a load access to memory whose result is written into the EAX register. Therefore, simply identifying the EAX values for each sample provides the memory addresses responsible for the majority of the L1 misses in this benchmark. Table 1 includes a count of unique EAX values for each sample. In this table, the sample count column indicates the number of samples that contain the same EAX value. The extended-instruction-pointer values (0x080484c3) for these samples are all the same as that for the instruction shown previously. The last column indicates the EAX value that corresponds to the memory address used by the loads causing the L1 cache misses. The table rows are sorted from lowest to highest EAX value.

By examining the memory address data in Table 1, I can deduce the behavior of the `l1_miss` benchmark. First, note that the lower 12 bits of each EAX value are the same. Second, the difference between successive EAX values is equal to 8 Kbytes. The Pentium 4 L1 cache is an 8-Kbyte, four-way set associative cache. By repeatedly accessing a set of eight locations in memory that are each 8 Kbytes apart, the benchmark thrashes the Pentium 4's L1 cache, which can only hold the memory

contents of four accesses for this set of addresses at any one time. This analysis matches the benchmark's behavior, which causes L1 cache misses while performing one million executions of an eight-iteration loop that moves through a large array with an 8-Kbyte stride. If this were a data address profile from a real application, I could now use this analysis to reduce the frequency of cache misses by either changing the order in which the instructions access the data structure or by rearranging the data structure layout in memory.

Problems, limitations, and opportunities

Although the Pentium 4's performance-monitoring support significantly improves upon that of its predecessors, its current implementations do have problems and limitations. The most significant of these is the lack of documentation for performance-monitoring capabilities, implementation bugs, and undisclosed features. The sheer complexity of the Pentium 4 microarchitecture makes the creation of clear and accurate descriptions of its performance-monitoring events difficult. Early releases of this documentation provided cryptic descriptions of events and also omitted key information. For example, early descriptions of the front-end tagging mechanism did not list any events detectable by the front-end tagging mechanism.

However, Intel routinely updates this documentation online, making significant improvements with each release.³ The most recent releases of this documentation (which also include descriptions of the Xeon processor's hyperthreading capabilities) fill several omissions in the originals and also make numerous attempts to clarify the meaning and proper use of performance-monitoring events and features.

Implementation bugs in the performance-monitoring features have also made some features difficult to use. For example, the logic equations used to detect the IOQ_allocation event differ for different model versions and the qualification of event counting by user and supervisor privilege levels is ignored for this event. Again, as Intel designers find these bugs, the documentation is updated accordingly.

Lastly, several performance-monitoring features remain undocumented. For example, Intel has only documented one front-end tag-

Table 1. Counts for each unique EAX sampled value.

Sample count	EAX
102	0x08049760
88	0x0804b760
106	0x0804d760
102	0x0804f760
100	0x08051760
100	0x08053760
101	0x08055760
101	0x08057760

ging event: `μop_type`. These features remain undocumented primarily because Intel designers have not yet sufficiently validated them for external release. Although performance-monitoring support is important and useful, it does not significantly impact the number of processors sold. Hence, the company dedicates fewer resources to implementing, validating, and maintaining performance-monitoring features than to admittedly more important processor features, such as functional correctness and overall performance.

Three areas for improvement will become increasingly important as symmetric multiprocessing and hyperthreading become more common and the use of performance-monitoring features grows.

First, the interface and mechanisms that support qualification of event detection by thread ID and event counting by thread mode are not easily extended to support more than two tasks executing concurrently. If future implementations increase the level of hyperthreading, providing these capabilities might require a different approach.

Second, the current performance-monitoring features don't provide support for attributing event counts to specific tasks executed on a machine with a time-sharing operating system. The performance counters count events for all tasks that execute on the processor (here, the word *tasks* refers to multiple applications ready to run on the processor, not the number of tasks that can concurrently execute on the processor via hyperthreading). As a result, when the operating system shares the processor among several tasks, it's not possible to attribute event counts to the tasks that caused them without altering the operating

system. This is also a problem for the buffering of PEBS samples by the processor. With the current Pentium 4 implementation, if the operating system switches tasks before the PEBS ISR empties the PEBS buffer, the buffer will contain samples from multiple tasks with no way to identify the source of each sample.

Third, the definition of performance-monitoring events, the capabilities of the performance-monitoring support, as well as the software interface to select events and configure various performance-monitoring mechanisms change with each new microarchitecture. For example, the most recent x86 processor implementations from Intel are based upon three distinct microarchitectures, all supporting different events and capabilities:

- the P5 microarchitecture, which debuted as the original Pentium and evolved into the Pentium MMX;
- the P6 microarchitecture, which debuted as the Pentium Pro and evolved into the Pentium II and Pentium III processors along with their Celeron and Xeon versions; and
- the Willamette microarchitecture, which debuted as the Pentium 4 and has evolved into the hyperthreaded Pentium 4 Xeon.

Although many performance events are by definition microarchitecture specific, the changes in event definitions and interfaces make it difficult to develop and maintain software that uses performance-monitoring capabilities across different microarchitectures. A few software tools manage to do this, such as Intel's Vtune tool (<http://developer.intel.com/software/products/vtune/vtune60/index.htm>) and the PAPI tools (<http://icl.cs.utk.edu/projects/papi/>). However, it would be a significant improvement if a core set of key performance events, mechanisms, and interfaces could be defined and maintained for each new microarchitecture developed.

Intel's Pentium 4 processor provides performance-monitoring features that overcome

many of the limitations of the performance-monitoring hardware of previous processors. The new features of the Pentium 4 greatly enhance collection of accurate processor performance data and will enable a new class of performance-tuning tools and capabilities, such as the dynamic tuning of applications and the operating system.

References

1. D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology J.*, vol. 6, no. 1, 14 Feb. 2002, pp. 1-12; http://developer.intel.com/technology/itj/2002/volume06issue01/art01_hyper/vol6iss1_art01.pdf.
2. J. Dean et al., "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Symp. Microarchitecture (Micro-30)*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 292-302.
3. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, order no. 245472, Intel, Santa Clara, Calif., 2002; <http://developer.intel.com/design/pentium4/manuals/>.

Brinkley Sprunt is an assistant professor of electrical engineering at Bucknell University. His research interests include computer performance modeling, measurement, and optimization. Sprunt has a PhD in electrical and computer engineering from Carnegie Mellon University. He previously worked at Intel where he was a member of the architecture teams for the 80960, Pentium Pro, and Pentium 4 projects. He is a member of the IEEE and ACM.

Direct questions and comments about this article to Brinkley Sprunt, Bucknell Univ., Electrical Engineering Dept., Moore Ave., Lewisburg, PA 17837; bsprunt@bucknell.edu.

For further information on this or any other computing topic, visit our Digital Library at <http://computer.org/publications/dlib>.