A Top-Down Approach to Architecting CPI Component Performance Counters

Software developers can gain insight into software-hardware interactions by decomposing processor performance into individual cycles-per-instruction components that differentiate cycles consumed in active computation from those spent handling various miss events. Constructing accurate CPI components for out-of-order superscalar processors is complicated, however, because computation and miss event handling overlap. The authors' counter architecture, using an analytical superscalar performance model, handles overlap effects more accurately than existing methods.

Stijn Eyerman
Lieven Eeckhout
Ghent University

Tejas Karkhanis
Advanced Micro Devices

James E. Smith
University of Wisconsin-Madison

A key application of user-visible hardware performance counters is to give the software developer clear, accurate, and useful hardware-related performance information. This information provides guidance on software changes that can improve performance. An intuitively appealing way to represent the major performance components is to quantify their contributions to the average cycles per instruction (CPI). However, for out-of-order superscalar processors, conventional performance counters don't provide the type of information from which software developers can determine accurate CPI components. One reason is that hardware designers have historically constructed performance counters in a bottom-up fashion by focusing on individual events that affect performance, such as cache misses, without considering how to combine individual counts into a comprehensive picture of CPI components.

In contrast, by viewing performance in a top-down manner with accurate CPI measures as the goal, designers can define a set of performance counters that provide basic data from which software developers can build an accurate overall CPI picture. We have developed such a top-down approach, beginning with a superscalar processor performance model called interval analysis. Interval analysis gives an in-depth understanding of relationships among miss events and related performance penalties. We use insights from the performance model to design a novel hardware performance counter architecture for computing CPI components that are accurate to within a few percent of CPI components computed by detailed simulations. This approach is

Published by the IEEE Computer Society

significantly more accurate than previously proposed methods of breaking down CPI components (see the sidebar, ''Computing CPI stacks: Other approaches''). Moreover, the proposed counter architecture's hardware complexity is comparable to that of existing counter architectures.

Constructing CPI stacks

The average CPI for a computer program executing on a given microprocessor comprises a base CPI plus several CPI components that reflect lost cycle opportunities caused by miss events such as branch mispredictions and cache and translation look-aside buffer (TLB) misses. The breakdown of CPI into components is often called a CPI stack because the CPI data is typically displayed as a stack of histogram bars representing the CPI components, with the base CPI at the bottom. A CPI stack reveals valuable information about an application's behavior on a given microprocessor and provides more insight into that behavior than raw miss rates do.

Figure 1 shows an example CPI stack for the twolf benchmark executing on a four-wide superscalar out-of-order processor. The CPI stack reveals that the base CPI (in the absence of miss events) equals 0.3; other substantial CPI components are for L1 instruction cache misses (0.18), L2 data cache misses (0.56), and branch mispredictions (0.16). The overall CPI is 1.35 (the top level of the stack in Figure 1).

Application developers can use a CPI stack to optimize their software. For example, if the I-cache miss component is relatively high, improving instruction locality is the key to reducing CPI and increasing performance. Or, if the L2 D-cache miss component is high, as is true for the twolf example, changing the data layout or adding software prefetching instructions may be productive optimizations. Note that a CPI stack also shows the maximum performance improvement for a given optimization. For example, improving twolf's L2 D-cache behavior can improve overall performance by at most 41 percent—the L2 D-cache miss component divided by the overall CPI.

Computing CPI stacks: Other approaches

There are several approaches to computing CPI stacks for in-order architectures. For example, the Intel Itanium processor family provides a rich set of hardware performance counters for computing CPI stacks.[1] The Digital Continuous Profiling Infrastructure (DCPI) is another example of a hardware-performance-monitoring tool for an in-order architecture.[2] Computing CPI stacks for in-order architectures, however, is relatively simple compared with computing CPI stacks for out-of-order architectures.

The IBM Power5 is the only out-of-order microprocessor we know of that implements a dedicated counter architecture for computing CPI stacks.[3] The Power5 approach is an improvement over the naive approaches described in this article, but it doesn't accurately compute the instruction cache and instruction translation look-aside buffer CPI components or the branch misprediction penalty. The Intel Pentium 4 doesn't have a dedicated counter architecture for computing CPI stacks.[4] Rather, it features a mechanism for obtaining nonspeculative event counts—that is, it doesn't count miss events along mispredicted control-flow paths.

Researchers have recently proposed other hardware-profiling mechanisms for out-of-order architectures. However, the goal of those methods is quite different from ours. Our goal is to build simple, easy-to-understand CPI stacks, whereas the other approaches aim at detailed per-instruction profiling. For example, the ProfileMe framework randomly samples individual instructions and collects cycle-level information on a per-instruction basis.[5] ProfileMe collects aggregate CPI stacks by profiling many randomly sampled instructions and aggregating all of their individual latency information. An inherent limitation with this approach is that per-instruction profiling doesn't allow modeling of overlap effects. The ProfileMe framework partially addresses this issue by profiling pairs of potentially concurrent instructions.

Another per-instruction approach, Shotgun profiling, models overlap effects between multiple instructions by collecting miss event information within hot spots, using specialized hardware performance counters.[6] A ''postmortem'' analysis based on a simple processor model then determines the amount of overlap and interaction between instructions within these hot spots. Per-instruction profiling has three inherent limitations: It relies on sampling, which can introduce inaccuracy. It uses per-instruction information for computing overlap effects. And it uses interrupts for communicating miss event information from hardware to software, which can lead to overhead or perturbation issues.

References

1. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, 251110-003, Intel Corp., 2004.
2. J.M. Anderson et al., ''Continuous Profiling: Where Have All the Cycles Gone?'' *ACM Trans. Computer Systems*, vol. 15, no. 4, Nov. 1997, pp. 357-390.
3. A. Mericas, ''Performance Monitoring on the POWER5 Microprocessor,'' *Performance Evaluation and Benchmarking*, L.K. John and L. Eeckhout, eds., CRC Press, 2006, pp. 247-266.
4. B. Sprunt, ''Pentium 4 Performance-Monitoring Features,'' *IEEE Micro*, vol. 22, no. 4, July 2002, pp. 72-82.
5. J. Dean et al., ''ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors,'' *Proc. 30th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 30), 1997, pp. 292-302.
6. B.A. Fields et al., ''Interaction Cost and Shotgun Profiling,'' *ACM Trans. Architecture and Code Optimization*, vol. 1, no. 3, Sept. 2004, pp. 272-304.

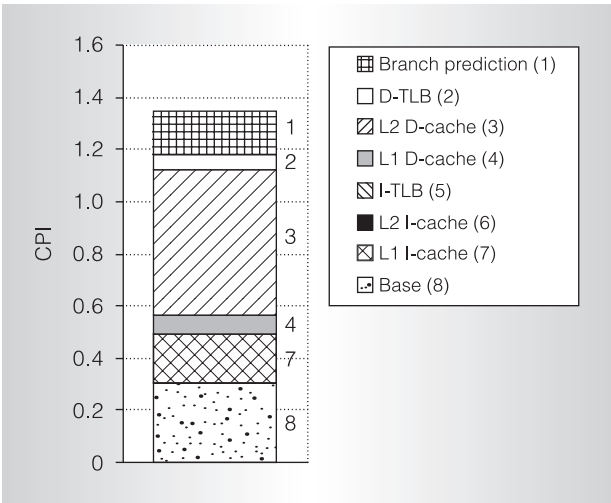


Figure 1. Example CPI stack for the twolf benchmark.

Underlying performance model

Although the basic idea of a CPI stack is simple, computing accurate CPI stacks on out-of-order superscalar processors is challenging because of the overlapped processing of independent operations and miss events. The interval performance model for superscalar performance evaluation is the basis of a top-down hardware performance counter architecture. Interval analysis provides insight into a superscalar processor's performance without requiring detailed tracking of individual instructions. In interval analysis, disruptive miss events (such as cache misses, TLB misses, and branch mispredictions) partition execution time into discrete intervals.

The model's basis is that a superscalar processor streams instructions through its pipelines and functional units, and, under optimal conditions (with no miss events), a well-balanced design sustains a performance level, in terms of instructions executed per cycle (IPC), approximately equal to its pipeline (dispatch and issue) width. We have verified that this is the case for the processor widths of practical interest—two-way to eight-way. We are not the first to observe this. Early studies showed that under ideal conditions, a processor with a sufficiently large instruction window can achieve high issue rates. The window is the block of consecutive instructions the control logic is considering for concurrent execution. In a superscalar processor, the window corresponds to instructions in the reorder buffer (ROB). Riseman and Foster showed a squared relationship between instruction window size and the number of instructions executed per cycle (IPC);[1] more recent studies have also made this observation.[2,3]

In practice, miss events often disrupt the ideal, smooth instruction flow. After a miss event, the issue of useful instructions eventually stops; then, no useful instructions are issued until the miss event is resolved and instructions can once again begin flowing.

Figure 2 depicts this interval behavior. The vertical axis represents the number of (useful) instructions issued per cycle (IPC), and the horizontal axis represents time (in clock cycles). The effects of miss events divide execution time into intervals. We define intervals as beginning and ending at the points where instructions just begin to dispatch (and issue) following recovery from the preceding miss event. That is, an interval includes the time period during which no useful instructions are issued following a particular miss event.

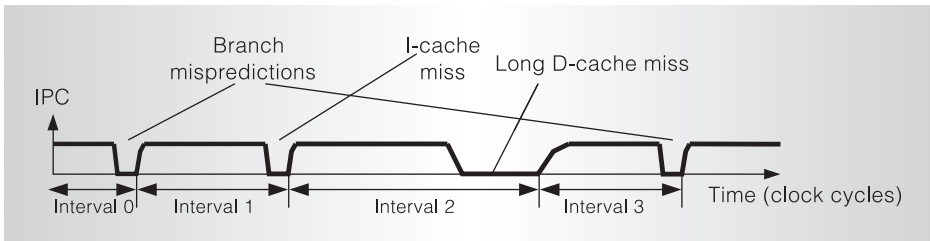


Figure 2. Basic idea of interval analysis: analyzing performance by dividing execution time into intervals between miss events.

By dividing execution time into intervals, we can analyze the performance behavior of intervals individually. In particular, we can describe and evaluate key performance characteristics on the basis of interval type (determined by the miss event that terminates it). Because underlying interval behavior is different for front-end and back-end miss events, we discuss them separately here. Then we discuss interactions between miss events.

Front-end misses

Front-end misses fall into two main categories: first, I-cache and I-TLB misses; second, branch mispredictions.

I-cache and I-TLB misses. Figure 3 shows the interval execution curve for an L1 or L2 I-cache miss; I-TLB misses exhibit similar behavior (the only difference is the amount of delay). This graph plots the number of instructions issued (on the vertical axis) versus time (on the horizontal axis); this is typical behavior, and we've smoothed the plot for clarity. At the beginning of the interval, instructions begin to fill the window at a sustained maximum dispatch width, and instruction issue and commit ramp up. As the window fills, the issue and commit rates increase toward the maximum value. Then, at some point, an I-cache miss occurs. Before the window starts to drain, all instructions already in the pipeline front end must first be dispatched into the window. This takes an amount of time equal to the number of front-end pipeline stages—that is, the number of clock cycles is equal to the front-end pipeline length. Offsetting this effect is the time required to refill the front-end pipeline after the missed line is accessed from the L2 cache (or main memory). Because the two pipeline-length delays exactly offset each other, the overall penalty for an instruction cache miss equals the miss delay.

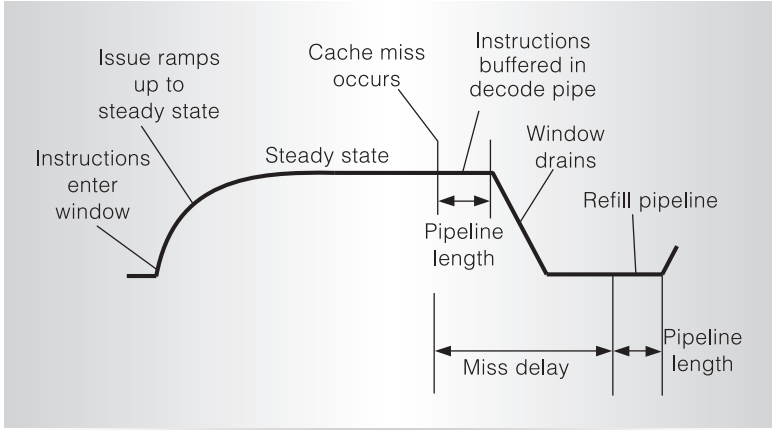


Figure 3. An I-cache miss interval.

Branch mispredictions. Figure 4 shows the timing of a branch misprediction interval. At the beginning of the interval, instructions begin to fill the window and instruction issue ramps up. Then, at some point, the mispredicted branch enters the window. At that point, the window begins draining useful instructions (those that will eventually commit). Mis-speculated instructions following the mispredicted branch continue filling the window, but they don't contribute to the issue of good instructions. Nor, generally speaking, do they inhibit the issue of useful instructions, if we assume that the oldest ready instructions are allowed to issue first. Eventually, when the mispredicted branch resolves, the processor flushes the pipeline and refills it with instructions from the correct path. During this refill time, there is a zero-issue region where no instructions issue or complete. The zero-region is approximately equal to the time it takes to refill the front-end pipeline.

From the interval analysis, it follows that the overall performance penalty of a branch misprediction equals the difference between the time the mispredicted branch first enters

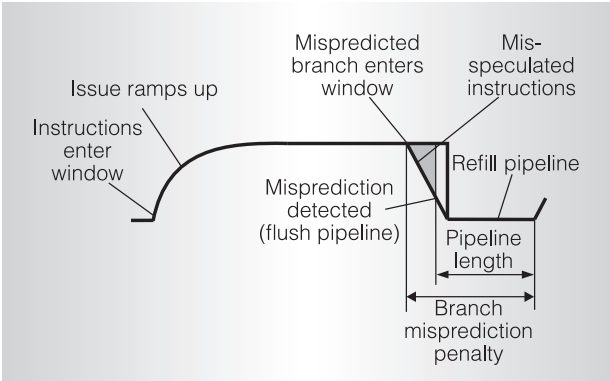


Figure 4. Interval behavior of a branch misprediction.

the window and the time the first correct-path instruction enters the window following discovery of the misprediction. In other words, the overall performance penalty equals the branch resolution time plus the front-end pipeline length.

Research has shown that the branch resolution time is subject to the interval length and the average length of critical data dependence paths in the program.[4] In other words, the longer the interval and the longer the data dependence paths, the longer the branch resolution time. For many programs, the branch resolution time is the main contributor to the overall branch misprediction penalty.

Finally, it follows that to accurately compute the branch misprediction penalty, a hardware performance counter mechanism requires knowledge of when a mispredicted branch enters the ROB. This is reflected in the hardware performance counter architecture proposed here.

Back-end misses

For back-end miss events, we distinguish between events of short and long duration. The short back-end misses are L1 data-cache misses; the long back-end misses are L2 data-cache misses and data-TLB misses.

Short misses. If the processor design is reasonably well-balanced, the ROB (and related structures) will be large enough for the out-of-order execution of independent instructions to hide (overlap) the latency of short D-cache misses. Thus, our performance model considers loads that miss in the L1 D-cache in much the same way that it considers instructions issued to long-latency functional units.

Long misses. When a long D-cache miss occurs (from L2 to main memory), the memory delay is typically long—on the order of a hundred or more cycles. D-TLB misses exhibit similar behavior. Hence, we handle both in the same manner.

On an isolated long D-cache miss, the ROB eventually fills because the load blocks the ROB head, then dispatch stops, and eventually issue and commit stop.[5] After the miss data returns from memory, instruction issue resumes. The total penalty for an isolated long back-end miss equals the time between the ROB fill and the time data returns from memory.

Now consider the influence of a long D-cache miss that closely follows another long D-cache miss; assume that the two misses are independent of each other—that is, the first load doesn't feed the second load. By "closely" we mean within the W (window or ROB size) instructions that immediately follow the first long D-cache miss. These instructions will make it into the ROB before it blocks. If additional long D-cache misses occur within the W instructions immediately following the first long D-cache miss, there is no additional performance penalty because their miss latencies overlap that of the first. Figure 5 illustrates this timing. Here, we assume that the second miss follows the first by S instructions. When the first load's miss data returns from memory, the first load commits and no longer blocks the ROB head. Then, S new instructions can enter the ROB. This takes approximately S/I cycles, with I being the dispatch width—just enough time to overlap the remaining latency from the second miss. This overlap holds regardless of the value of S; the only requirement is that S be less than or equal to W. The same will be true for any number of other long D-cache misses that occur within W instructions of the first long D-cache miss.

From this analysis, we conclude that the penalty for both an isolated miss and multiple, overlapping long D-cache misses equals the time between the ROB filling up and the data returning from main memory. This time is exactly what the proposed hardware performance counter mechanism measures.

Miss event interactions

So far, we've considered the various miss event types in isolation. In practice, however, miss events don't occur in isolation; they interact with other miss events. Accurately dealing with these interactions is crucial to building meaningful CPI stacks, because

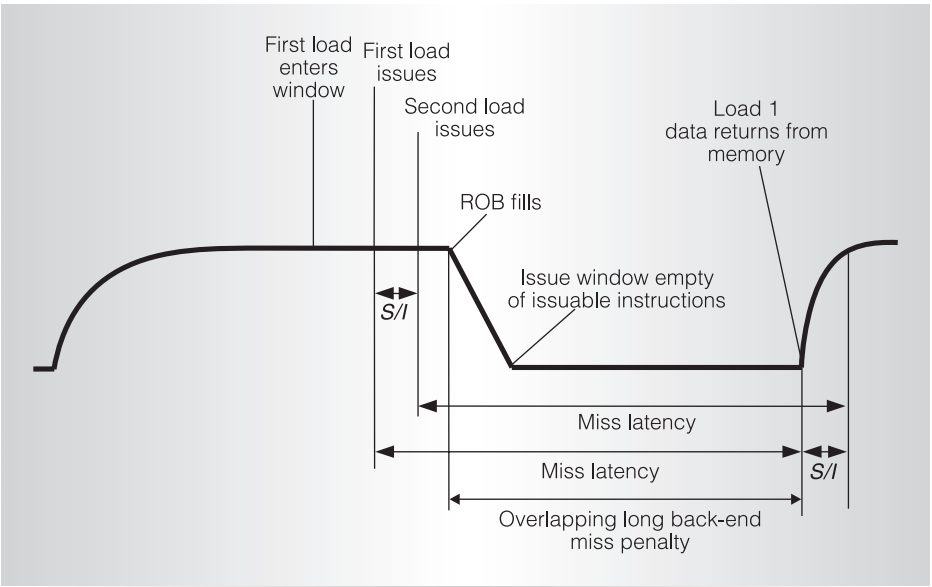


Figure 5. Interval timing of two overlapping long D-cache misses.

miss event penalties should not be double-counted.

Front-end miss event interactions. The degree of interaction between front-end pipeline miss events (branch mispredictions, I-cache misses, and I-TLB misses) is limited because the penalties don't overlap. That is, front-end pipeline miss events serially disrupt the flow of good instructions, so the negative effects of these miss events don't overlap. The only thing we must consider in building accurate CPI stacks is that the penalties of front-end pipeline miss events along mispredicted control-flow paths should not be counted. For example, the penalty of an I-cache miss along a mispredicted path should not be counted as such.

Front-end and long back-end miss event interactions. The interactions between front-end pipeline miss events and long back-end miss events are more complex because long back-end miss events can overlap front-end pipeline miss events. The question is, how do we account for both miss event penalties? For example, if a branch misprediction overlaps a long D-cache miss, do we account for the branch misprediction penalty, or do we ignore the branch misprediction penalty, saying that it is completely hidden under the long D-cache miss? The fraction of overlapped cycles is generally very small. It is no more than 1 percent for most of the SPECint2000 benchmarks and only as much as 5 percent for two of them.[6] Because the fraction of overlapped cycles is so small, any mechanism for dealing with them will result in relatively accurate and meaningful CPI stacks. Consequently, for simplicity, we opt for a hardware performance counter implementation that assigns overlap of front-end and long back-end miss penalties to the front-end CPI component, unless the ROB is full, triggering a count of the long back-end miss penalty.

Counter architecture

The insights obtained from interval analysis lead us to propose a counter architecture that computes the penalty or the number of cycles lost to L1 I-cache misses, L2 I-cache misses, I-TLB misses, L1 D-cache misses, L2 D-cache misses, D-TLB misses, branch mispredictions, and long-latency functional-unit stalls. The architecture includes a global CPI component counter for each of these categories, and the idea is to assign every cycle to one of these global CPI component cycle counters when possible; the steady-state (baseline)

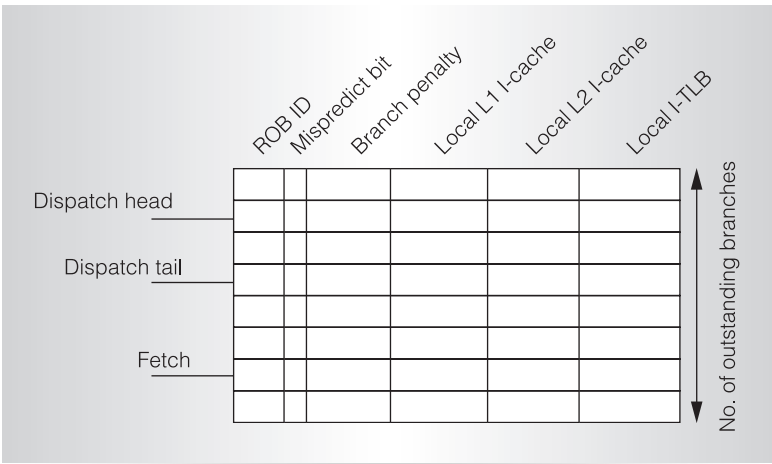


Figure 6. Front-end miss event table (FMT) for computing front-end miss penalties.

cycle count then equals the total cycle count minus the total sum of the individual CPI component cycle counters. The amount of hardware required to implement this counter architecture is small, and the amount of storage required is no more than a few hundred bits.

Front-end miss events

For front-end miss events, the counter architecture should provide three functions:

- counting the number of cycles lost on I-cache and I-TLB misses,
- computing the penalty of a branch misprediction as the time between the branch entering the ROB and new instructions entering the ROB after the branch's resolution, and
- ignoring front-end miss events along miss-speculated paths.

To provide these functions, we propose that the counter architecture include a front-end miss event table (FMT). Figure 6 shows its basic structure; Eyerman et al. present a space-efficient FMT implementation.[6] The FMT is a circular buffer that contains as many rows as the outstanding branches the processor supports. One entry is allocated per outstanding branch in the processor. FMT entries between the dispatch tail and the head pointers denote branches dispatched in the ROB; entries between the fetch pointer and the dispatch tail denote branches fetched but not yet dispatched.

The FMT computes the I-cache and I-TLB penalty as follows: For any cycle in which no instructions enter the pipeline because of an L1 or L2 I-cache miss or an I-TLB miss, the local counter in the FMT entry pointed to by the fetch pointer is incremented. For example, an L1 I-cache miss causes the local L1 I-cache miss counter in the FMT to increment every cycle until the cache miss resolves. Thus, the miss delay computed in the local counter corresponds to the actual I-cache or I-TLB miss penalty (according to interval analysis).

For computing a branch's misprediction penalty, the branch penalty counter keeps track of the number of lost cycles caused by a potential branch misprediction. Because it is unknown at dispatch time whether a branch is mispredicted, the proposed method computes the number of cycles during which each branch resides in the ROB. That is, the branch penalty counter increments every cycle for all branches between the dispatch head and tail pointers in the FMT. The FMT also holds the branch's ROB identification (ID). When a predicted branch resolves and turns out to be a misprediction, the branch's ROB ID is used to locate the corresponding FMT entry and set the mispredict bit.

When a branch instruction completes, the counter architecture updates the global front-end CPI component cycle counters. That is, it adds the local L1 I-cache, L2 I-cache, and I-TLB counters to the respective global cycle counters. If the branch is incorrectly predicted, the counter architecture adds the value in the branch penalty counter to the global branch misprediction cycle counter; from then on, the global branch misprediction cycle counter increments every cycle until new instructions enter the ROB. The resolution of a mispredicted branch also deallocates FMT entries by pointing the FMT dispatch tail pointer to the mispredicted branch entry and the FMT fetch pointer to the next FMT entry. This deallocation policy avoids the counting of cycles lost by front-end miss events along miss-speculated paths.

Back-end miss events

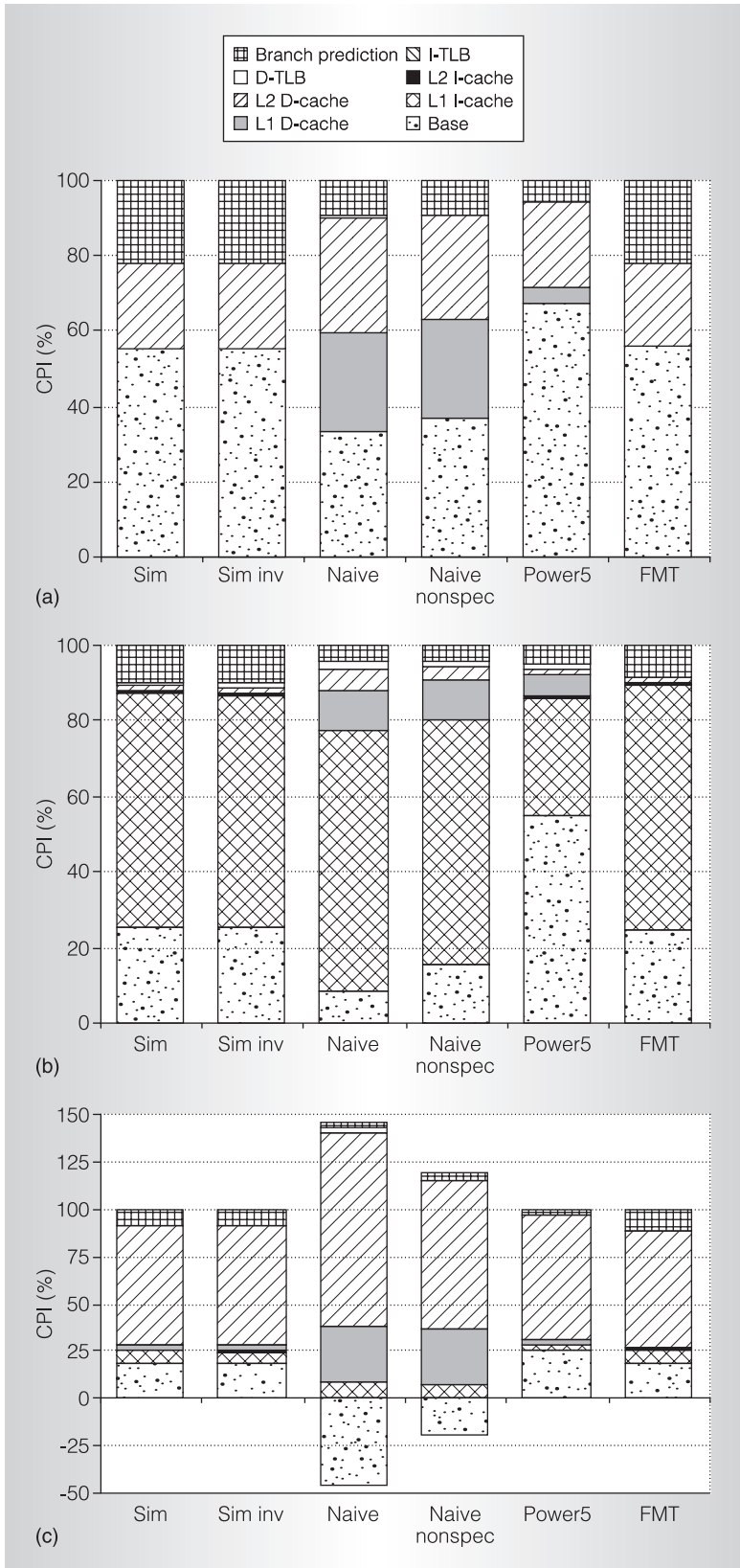
Hardware performance counters for computing cycles lost due to long back-end misses, such as long D-cache misses and D-TLB misses, are fairly easy to implement. These counters start counting when the ROB is full and the instruction blocking the ROB is a L2 D-cache miss or D-TLB miss. For every cycle in which both conditions hold, the respective cycle counter is incremented. Note that this approach handles isolated as well as overlapping long-latency misses.

The counter architecture computes resource stalls caused by long-latency functional-unit instructions and L1 D-cache misses by counting the cycle as a resource stall if the ROB is full and the instruction blocking the ROB head is a long-latency instruction or an L1 D-cache miss.

Evaluation

We evaluated the proposed hardware performance counter architecture by comparing it with two simulation-derived CPI stacks, two naive (simple) approaches, and the IBM Power5 mechanism. The simulation-derived CPI stacks serve as a reference for comparison. We used two simulation-derived stacks because of the difficulty in defining what a standard, correct CPI stack should look like. In particular, there might be cycles that one could reasonably ascribe to more than one miss event because of overlaps.

To compute the simulation-derived CPI stacks, we started with a processor executing with no miss events and then progressively computed lost cycles while adding miss events one at a time. The two simulation-derived stacks reflect opposite orderings of the sequence of adding miss events (Sim and Sim inv). The similarity of these two stacks supports our earlier contention that most overlap effects are minimal (over-



→

Figure 7. Comparison of normalized CPI stacks for three SPECint2000 benchmarks: bzip2 (a), crafty (b), and gcc (c).

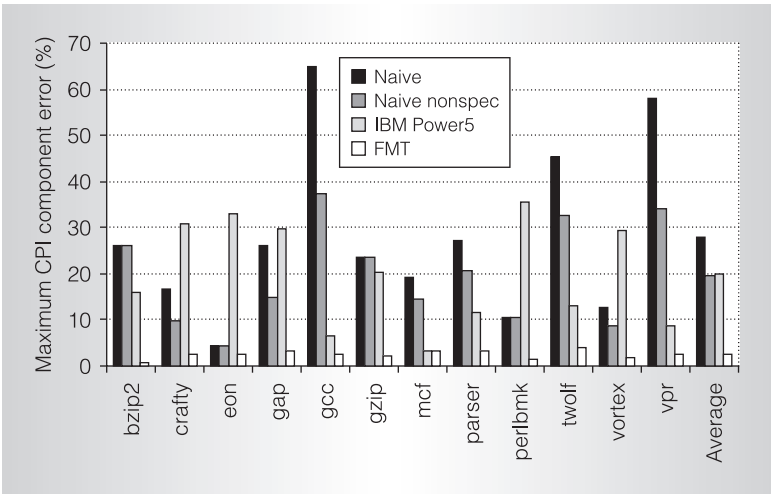


Figure 8. Maximum CPI component error for various approaches compared with the simulation-derived CPI stacks.

lapping L2 D-cache and D-TLB misses being the exceptions).

The first naive approach computes the CPI components by multiplying the number of miss events per instruction by the penalty for an isolated miss event. The second naive approach, which we call the naive nonspec approach, is similar, except that it doesn't compute penalties along mispredicted control-flow paths. The IBM Power5,[7] to the best of our knowledge, is the only out-of-order processor that implements a dedicated counter architecture for computing CPI components. The general Power5 approach is to inspect the pipeline's completion stage, and if no instructions can be completed in a given cycle, it increments the appropriate completion stall counter.

We evaluated performance for all the SPECint2000 benchmarks, but here we present results for three typical benchmarks (see Figure 7). For some of the benchmarks, the naive approach results in CPI stacks that are highly inaccurate, to the point of not being meaningful; see, for example, the CPI stack for gcc in Figure 7c. The sum of the miss event counts multiplied by the miss penalties is larger than the total cycle count. This causes the base CPI, which is the total cycle count minus the miss event cycle counts, to be negative. The reason the naive approach fails to build accurate CPI stacks is that it doesn't adequately deal with overlapped long back-end misses, doesn't accurately compute the branch misprediction penalty, and counts I-cache and I-TLB misses along mispredicted paths. The naive nonspec approach, which does not count miss events along mispredicted paths, is more accurate than the naive approach, but the CPI stacks are still not as accurate as the simulation-derived CPI stacks.

The IBM Power5 approach is clearly an improvement over the naive approaches. However, compared with the simulation-derived CPI stacks, the Power5 stacks—for example, bzip2 and crafty in Figure 7—are inaccurate. The Power5 mechanism falls short because it estimates the I-cache miss penalty or the branch misprediction penalty as the zero-issue region past an I-cache miss event or a branch misprediction. This is a significant underestimate of the actual miss penalties illustrated by Figures 3 and 4.

The CPI stacks we obtained using our hardware performance counter architecture track the simulation-derived CPI stacks very closely. Whereas both the naive and IBM Power5 mechanisms showed high inaccuracy for several benchmarks, the FMT architecture showed significantly fewer errors for all benchmarks. As Figure 8 shows, all maximum CPI component errors were less than 4 percent. The average error for our performance counter architecture was 2.5 percent.

As part of future research, we plan to apply the CPI stack building methodology developed in this article to gaining better insight into how specific compiler optimizations interact with superscalar processor implementations to affect performance. In addition, we plan on extending interval analysis into a performance model for predicting rather than analyzing overall performance. MICRO

Acknowledgments

Stijn Eyerman and Lieven Eeckhout are supported through fellowships from the Fund for Scientific Research, Flanders, Belgium.

References

1. E.M. Riseman and C.C. Foster, ''The Inhibition of Potential Parallelism by Conditional Jumps,'' *IEEE Trans. Computers*, vol. 21, no. 12, Dec 1972, pp. 1405-1411.
2. T.S. Karkhanis and J.E. Smith, ''A First-Order Superscalar Processor Model,'' *Proc. 31st Ann. Int'l Symp. Computer Architecture* (ISCA 31), IEEE CS Press, 2004, pp. 338-349.
3. P. Michaud, A. Seznec, and S. Jourdan, ''Exploring Instruction Fetch Bandwidth Requirement in Wide-Issue Superscalar Processors,'' *Proc. 8th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 99), IEEE CS Press, 1999, pp. 2-10.
4. S. Eyerman, J.E. Smith, and L. Eeckhout, ''Characterizing the Branch Misprediction Penalty,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 06), IEEE Press, 2006, pp. 48-58.
5. T. Karkhanis and J.E. Smith, ''A Day in the Life of a Data Cache Miss,'' *Proc. 2nd Ann. Workshop Memory Performance Issues* (WMPI 02), held in conjunction with ISCA 29, 2002; http://www.ece.wisc.edu/~jes/papers/wmpi02.tejas.pdf.
6. S. Eyerman et al., ''A Performance Counter Architecture for Computing Accurate CPI Components,'' *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS XII), ACM Press, 2006, pp. 175-184.
7. A. Mericas, ''Performance Monitoring on the POWER5 Microprocessor,'' *Performance Evaluation and Benchmarking*, L.K. John and L. Eeckhout, eds., CRC Press, 2006, pp. 247-266.

Stijn Eyerman is a PhD student in the Electronics and Information Systems Department at Ghent University, Belgium. His research interests include computer architecture in general and performance analysis and modeling in particular. Eyerman has an MS in computer science and engineering from Ghent University.

Lieven Eeckhout is an assistant professor in the Electronics and Information Systems Department at Ghent University, Belgium. His research interests include computer architecture, performance analysis and modeling, and workload characterization. Eeckhout has a PhD in computer science and engineering from Ghent University. He is a member of the IEEE.

Tejas Karkhanis is a microprocessor design engineer in the Boston Design Center of Advanced Micro Devices. His research interests include high-performance and low-power microprocessor design. Karkhanis has BS, MS, and PhD degrees in electrical engineering from the University of Wisconsin-Madison.

James E. Smith is a professor in the Department of Electrical and Computer Engineering at the University of Wisconsin-Madison. His research interests include high-performance and power-efficient processor implementations, processor performance modeling, and virtual machines. Smith has a PhD in computer science from the University of Illinois. He is a member of the IEEE and the ACM.

Direct questions and comments about this article to Lieven Eeckhout, ELIS—Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; leeckhou@elis.UGent.be.