A Performance Counter Architecture for Computing Accurate CPI Components

Stijn Eyerman[†] Lieven Eeckhout[†] Tejas Karkhanis[‡] James E. Smith[‡]
[†]ELIS, Ghent University, Belgium
[‡]ECE, University of Wisconsin–Madison
Email: {seyerman,leeckhou}@elis.UGent.be, {karkhani,jes}@ece.wisc.edu

ABSTRACT

Cycles per Instruction (CPI) stacks break down processor execution time into a baseline CPI plus a number of miss event CPI components. CPI breakdowns can be very helpful in gaining insight into the behavior of an application on a given microprocessor; consequently, they are widely used by software application developers and computer architects. However, computing CPI stacks on superscalar out-of-order processors is challenging because of various overlaps among execution and miss events (cache misses, TLB misses, and branch mispredictions).

This paper shows that meaningful and accurate CPI stacks can be computed for superscalar out-of-order processors. Using interval analysis, a novel method for analyzing out-of-order processor performance, we gain understanding into the performance impact of the various miss events. Based on this understanding, we propose a novel way of architecting hardware performance counters for building accurate CPI stacks. The additional hardware for implementing these counters is limited and comparable to existing hardware performance counter architectures while being significantly more accurate than previous approaches.

1. INTRODUCTION

A key role of user-visible hardware performance counters is to provide clear and accurate performance information to the software developer. This information provides guidance regarding the kinds of software changes that are necessary for improved performance. One intuitively appealing way of representing the major performance components is in terms of their contributions to the average cycles per instruction (CPI). However, for out-of-order superscalar processors, conventional performance counters do not provide the type of information from which accurate CPI components can be determined. The reason is that performance counters have historically been constructed in a bottom up fashion by focusing on the events that affect performance, for example the various cache miss rates, without regard for how the performance counts should be combined to form an overall picture of CPI.

In contrast, by viewing performance in a top down manner with accurate CPI measures as the goal, a set of performance counters can be defined that do provide basic data from which an accurate overall picture of CPI can be built. We have taken such a top down approach, using interval analysis, a superscalar processor performance model we have developed, as a guide. The performance model gives an in-depth understanding of the relationships among miss events and related performance penalties. The insights from the performance model are used to design a novel hardware performance counter architecture for computing accurate CPI components — within a few percent of the components computed by detailed simulations. This is significantly more accurate than previously proposed CPI breakdown approaches with errors that are higher than 30%. Moreover, the hardware complexity for our counter architecture is limited and comparable to existing hardware performance counter architectures.

We first revisit the CPI stack and existing approaches to measuring CPI stacks (section 2). We then use a performance model, interval analysis, to determine what the miss penalties are for the various miss events (section 3). Based on these insights we subsequently propose our hardware performance counter mechanism for building accurate CPI stacks (section 4). Subsequently, the proposed mechanism is evaluated (section 5) and related work is discussed (section 6). Finally, we conclude in section 7.

2. CONSTRUCTING CPI STACKS

The average CPI for a computer program executing on a given microprocessor can be divided into a base CPI plus a number of CPI components that reflect the 'lost' cycle opportunities because of miss events such as branch mispredictions and cache and TLB misses. The breakdown of CPI into components is often referred to as a CPI 'stack' because the CPI data is typically displayed as stacked histogram bars where the CPI components stack on top of one another with the base CPI being shown at the bottom of the histogram bar. A CPI stack reveals valuable information about how a given application behaves on a given microprocessor and gives more insight into an application's behavior than raw miss rates do.

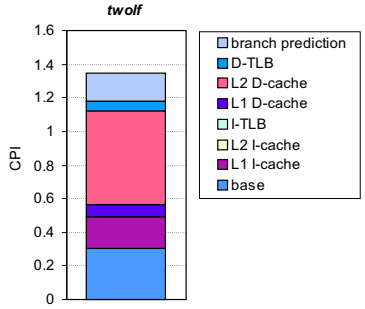

Figure 1: Example CPI stack for the twolf benchmark.

Figure 1 shows an example CPI stack for the twolf benchmark for a 4-wide superscalar out-of-order processor detailed in section 5.1. The CPI stack reveals that the base CPI (in the absence of miss events) equals 0.3; other substantial CPI components are for the L1 I-cache (0.18), the L2 D-cache (0.56) and the branch predictor (0.16). The overall CPI equals 1.35 which is shown by the top of the CPI stack. Application developers can use a CPI stack for optimizing their software. For example, if the intruction cache miss CPI is relatively high, then improving instruction locality is the key to reducing CPI and increasing performance. Or, if the L2 D-cache CPI component is high as is the case for the twolf example, changing the data layout or adding software prefetching instructions are viable optimizations. Note that a CPI stack also shows what the maximum performance improvement is for a given optimization. For example for twolf, improving the L2 D-cache behavior can improve overall performance by at most 41%, *i.e.*, the L2 D-cache CPI component divided by the overall CPI.

Although the basic idea of a CPI stack is simple, computing accurate CPI stacks on superscalar out-of-order processors is challenging because of the parallel processing of independent operations and miss events. A widely used naive approach for computing the various components in a CPI stack is to multiply the number of miss events of a given type by an average penalty per miss event [1, 10, 11, 17, 20]. For example, the L2 data cache miss CPI component is computed by multiplying the number of L2 misses with the average memory access latency; the branch misprediction CPI contributor is computed by multiplying the number of branch mispredictions with an average branch misprediction penalty. We will refer to this approach as the *naive approach* throughout the paper.

There are a number of pitfalls to the naive approach, however. First, the average penalty for a given miss event may vary across programs, and, in addition, the number of penalty cycles may not be obvious. For example, previous work [4] has shown that the branch misprediction penalty varies widely across benchmarks and can be substantially larger than the frontend pipeline length — taking the frontend pipeline length as an estimate for the branch misprediction penalty leads to a significant underestimation of the real branch misprediction penalty. Second, the naive approach does not take into consideration that some of the miss event penalties can be hidden (overlapped) through out-of-order processing of independent instructions and miss events. For example, L1 data cache misses can be hidden almost completely in a balanced, out-of-order superscalar processor. As another example, two or more L2 data cache misses may overlap with each other. Not taking these overlapping miss events into account can give highly skewed estimates of the CPI components. And finally, the naive approach makes no distinction between miss events along mispredicted control flow paths and miss events along correct control flow paths. A naive method may count events on both paths, leading to inaccuracy.

To overcome the latter problem, some processors, such as the Intel Pentium 4 [18], feature a mechanism for obtaining nonspeculative event counts. This is achieved by implementing a tagging mechanism that tags instructions as they flow through the pipeline, and the event counters get updated only in case the instruction reaches completion. In case the instruction is not completed, *i.e.*, the instruction is from a misspeculated path, the event counter does not get updated. We will refer to this approach as the *naive_non_spec* approach; this approach differs from the naive approach in that it does not count miss events along mispredicted paths.

In response to some of the above shortcomings, the designers of the IBM POWER5 microprocessor implemented a dedicated hardware performance counter mechanism with the goal of computing the CPI stack [12, 13]. And to the best of our knowledge, the IBM POWER5 is the only out-of-order processor implementing a dedicated hardware performance counter architecture for measuring the CPI components similar to our approach — in fact, other approaches have been proposed which we discuss in section 6. The IBM POWER5 has hardware performance counters that can be programmed to count particular completion stall conditions such as I-cache miss, branch misprediction, L2 D-cache miss, L1 D-cache miss, etc. The general philosophy for the IBM POWER5 CPI mechanism is to inspect the completion stage of the pipeline. And if no instructions can be completed in a given cycle, the appropriate completion stall counter is incremented. As such, the completion stall counters count the number of stall cycles for a given stall condition. There are two primary conditions for a completion stall.

- First, the reorder buffer (ROB) is empty. There are two possible causes for this.
  - First, an I-cache miss, or an I-TLB miss occurred, and the pipeline stops feeding new instructions into the ROB. This causes the ROB to drain, and, eventually, the ROB may become empty. When the ROB is empty, the POWER5 mechanism starts counting lost cycles in the I-cache completion stall counter until instructions start entering the ROB again.
  - Second, a branch is mispredicted. When the mispredicted branch gets resolved, the pipeline needs to be flushed and new instructions need to be fetched from the correct control flow path. At that point in time, the ROB is empty until newly fetched instructions have traversed the frontend pipeline to reach the ROB. The POWER5 mechanism counts the number of cycles with an empty ROB in the branch misprediction stall counter.

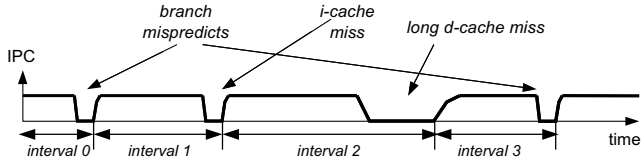

Figure 2: Basic idea of interval analysis: performance can be analyzed by dividing time into intervals between miss events.

- The second reason for a completion stall is that the instruction at the head of the ROB cannot be completed for some reason. The zero-completion cycle can be attributed to one of the following.
  - The instruction at the head of the ROB is stalled because it suffered a D-cache miss or a D-TLB miss. This causes the D-cache or D-TLB completion stall counter to be incremented every cycle until the memory operation is resolved.
  - The instruction at the head of the ROB is an instruction with latency greater than one cycle, such as a multiply, divide, or a long latency floating-point operation, and the instruction has not yet completed. The long latency completion stall counter is incremented every cycle until completion can make progress again.

These three CPI stack building approaches, the two naive approaches and the more complex IBM POWER5 approach, are both built in a bottom up fashion. These bottom up approaches are inadequate for computing the true performance penalties due to each of the miss events (as will be shown in detail in this paper). And as a result, the components in the resulting CPI stack are inaccurate.

3. UNDERLYING PERFORMANCE MODEL

We use a model for superscalar performance evaluation that we call *interval analysis* in our top down approach to designing a hardware performance counter architecture. Interval analysis provides insight into the performance of a superscalar processor without requiring detailed tracking of individual instructions. With interval analysis, execution time is partitioned into discrete intervals by the disruptive miss events (such as cache misses, TLB misses and branch mispredictions). Then, statistical processor and program behavior allows superscalar behavior and performance to be determined for each interval type. Finally, by aggregating the individual interval performance, overall performance is estimated.

The basis for the model is that a superscalar processor is designed to stream instructions through its various pipelines and functional units, and, under optimal conditions, a well-balanced design sustains a level of performance more-or-less equal to its pipeline (dispatch/issue) width. For a balanced processor design with a large enough ROB and related structures for a given processor width, the achieved issue rate indeed very closely approximates the maximum processor rate. This is true for the processor widths that are of practical interest – say 2-way to 6- or 8-way. We are not the first to observe this. Early studies such as by Riseman and Foster [] showed a squared relationship between instruction window size and IPC; this observation was also made in more recent work, see for example [] and []. There are exceptions though where there are very long dependence chains due to loop carried dependences and where the loop has relatively few other instructions. These situations are uncommon for the practical issue widths, however. Nevertheless, our counter architecture handles these cases as resource stalls or L1 D-cache misses, as described in section 4.3.

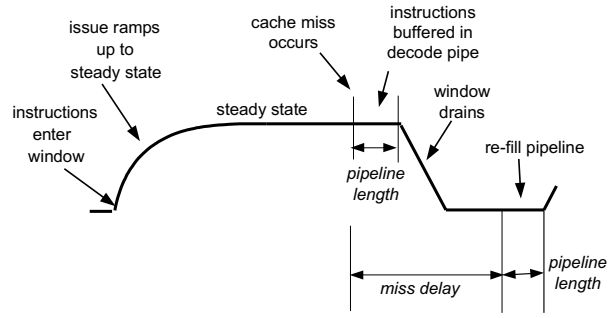

Figure 3: An I-cache miss interval.

However, the smooth flow of instructions is often disrupted by miss events. When a miss event occurs, the issuing of useful instructions eventually stops; there is then a period when no useful instructions are issued until the miss event is resolved and instructions can once again begin flowing. Here we emphasize *useful*; instructions on a mispredicted branch path are not considered to be useful.

This interval behavior is illustrated in Figure 2. The number of instructions issued per cycle (IPC) is shown on the vertical axis and time (in clock cycles) is on the horizontal axis. As illustrated in the figure, the effects of miss events divide execution time into intervals. We define intervals to begin and end at the points where instructions just begin issuing following recovery from the preceding miss event. That is, the interval includes the time period where no instructions are issued following a particular miss event. By dividing execution time into intervals, we can analyze the performance behavior of the intervals individually. In particular, we can, based on the type of interval (the miss event that terminates it), describe and evaluate the key performance characteristics. Because the underlying interval behavior is different for frontend and backend miss events, we discuss them separately. Then, after having discussed isolated miss events, we will discuss interactions between miss events.

3.1 Frontend misses

The frontend misses can be divided into I-cache and I-TLB misses, and branch mispredictions.

3.1.1 Instruction cache and TLB misses

The interval execution curve for an L1 or L2 I-cache miss is shown in Figure 3 — because I-cache and I-TLB misses exhibit similar behavior (the only difference being the amount of delay), we analyze them collectively as 'I-cache misses'. This graph plots the number of instructions issued (on the vertical axis) versus time (on the horizontal axis); this is

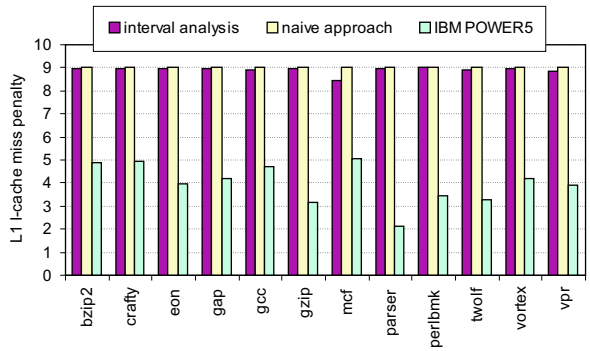

Figure 4: The penalty due to an L1 instruction cache miss; the access latency for the L2 cache is 9 cycles.

typical behavior, and the plot has been smoothed 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 instruction cache miss occurs. All the instructions already in the pipeline frontend must first be dispatched into the window before the window starts to drain. This takes an amount of time equal to the number of frontend pipeline stages, *i.e.*, the number of clock cycles equal to the frontend pipeline length. Offsetting this effect is the time required to re-fill the frontend 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. Simulation data verifies that this is the case for the L1 I-cache, see Figure 4; we obtained similar results for the L2 I-cache and I-TLB. The L1 I-cache miss penalty seems to be (fairly) constant across all benchmarks. In these experiments we assume an L2 access latency of 9 cycles. The slight fluctuation of the I-cache miss penalty between 8 and 9 cycles is due to the presence of the fetch buffer in the instruction delivery subsystem.

Our proposed hardware performance counter mechanism, which will be detailed later, effectively counts the miss delay, *i.e.*, it counts the number of cycles between the time the instruction cache miss occurs and the time newly fetched instructions start filling the frontend pipeline. These counts are then ascribed to either I-cache or I-TLB misses. The naive approach also computes the I-cache (or I-TLB) penalty in an accurate way multiplying the number of misses by the miss delay. The IBM POWER5 mechanism, in contrast, counts only the number of zero-completion cycles due to an empty ROB; this corresponds to the zero-region in Figure 3 after the ROB has drained. This means that the IBM POWER5 mechanism does not take into account the time to drain the ROB. This leads to a substantial underestimation of the real instruction cache (or TLB) miss penalty as shown in Figure 4. Note that in some cases, the IBM POWER5 mechanism may not ascribe any cycles to the instruction cache miss. This is the case when the window drain time takes longer than the miss delay, which can happen in case a largely filled ROB needs to be drained and there is low ILP or a significant fraction long latency instructions.

3.1.2 Branch mispredictions

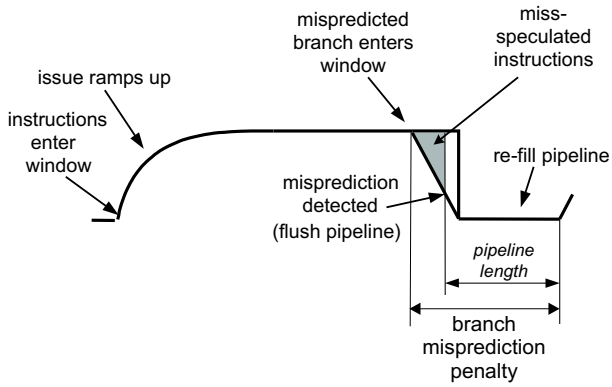

Figure 5: Interval behavior for a branch misprediction.

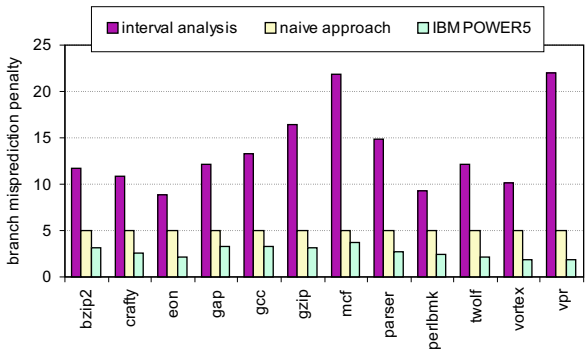

Figure 6: The average penalty per mispredicted branch.

Figure 5 shows the timing for 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 to be drained of useful instructions (*i.e.*, those that will eventually commit). Miss-speculated instructions following the mispredicted branch will continue filling the window, but they will not contribute to the issuing of good instructions. Nor, generally speaking, will they inhibit the issuing of good instructions if it is assumed that the oldest ready instructions are allowed to issue first. Eventually, when the mispredicted branch is resolved, the pipeline is flushed and is re-filled with instructions from the correct path. During this re-fill time, there is a zero-issue region where no instructions issue nor complete, and, given the above observation, the zero-region is approximately equal to the time it takes to re-fill the frontend pipeline.

Based on the above interval analysis, it follows that the overall performance penalty due to a branch misprediction equals the difference between the time the mispredicted branch first enters 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, *i.e.*, the time between the mispredicted branch entering the window and the branch being resolved, plus the frontend pipeline length. Eyerman *et al.* [4] have shown that the branch resolution time is subject to the interval length and the amount of ILP in the pro-

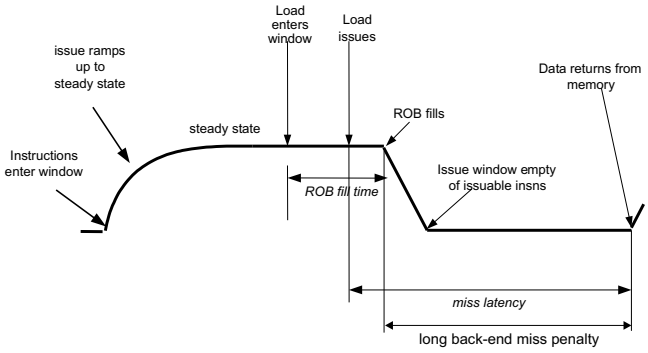

Figure 7: Interval behavior for an isolated long (L2) data cache miss.

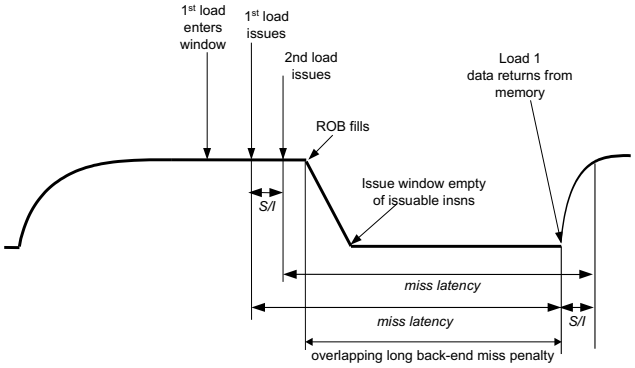

Figure 8: Interval timing of two overlapping long D-cache misses.

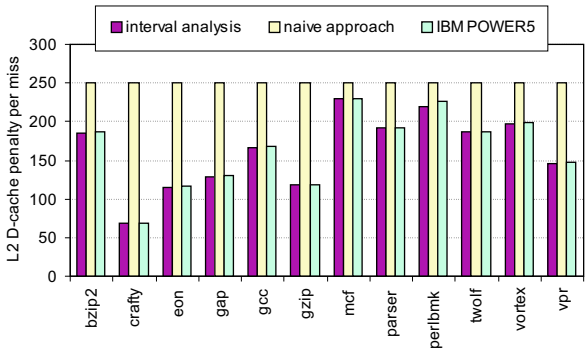

Figure 9: Penalty per long (L2) data cache miss.

gram; *i.e.*, the longer the interval and the lower the ILP, the longer the branch resolution time takes. For many benchmarks, the branch resolution time is the main contributor to the overall branch misprediction penalty.

Based on the interval analysis it follows that in order to accurately compute the branch misprediction penalty, a hardware performance counter mechanism requires knowledge of when a mispredicted branch entered the ROB. And this has to be reflected in the hardware performance counter architecture (as is the case in our proposed architecture). None of the existing approaches, however, employ such an architecture, and, consequently, these approaches are unable to compute the true branch misprediction penalty; see Figure 6. The naive approach typically ascribes the frontend pipeline length as the branch misprediction penalty which is a significant underestimation of the overall branch misprediction penalty. The IBM POWER5 mechanism only counts the number of zero-completion cycles on an empty ROB as a result of a branch misprediction. This is even worse than the naive approach as the number of zero-completion cycles can be smaller than the frontend pipeline length.

3.2 Backend misses

For backend miss events, we make a distinction between events of short and long duration. The short backend misses are L1 data cache misses; the long backend misses are the L2 data cache misses and D-TLB misses.

3.2.1 Short misses

Short (L1) data cache misses in general do not lead to a period where zero instructions can be issued. Provided that the processor design is reasonably well-balanced, there will be a sufficiently large ROB (and related structures) so that the latency of short data cache misses can be hidden (overlapped) by the out-of-order execution of independent instructions. As such, we consider loads that miss in the L1 data cache in a similar manner as the way we consider instructions issued to long latency functional units (see section 4.3).

3.2.2 Long misses

When a long data cache miss occurs, *i.e.*, from the L2 to main memory, the memory delay is typically quite large — on the order of a hundred or more cycles. Similar behavior is observed for D-TLB misses. Hence, both are handled in the same manner.

On an isolated long data cache miss, the ROB eventually fills because the load blocks the ROB head, then dispatch stops, and eventually issue and commit cease [8]. Figure 7 shows the performance of an interval that contains a long data cache miss where the ROB fills while the missing load instruction blocks at the head of the ROB. After the miss data returns from memory, instruction issuing resumes. The total long data cache miss penalty equals the time between the ROB fill and the time data returns from memory.

Next, we consider the influence of a long D-cache miss that closely follows another long D-cache miss — we assume that both L2 data cache misses are independent of each other, *i.e.*, the first load does not feed the second load. By 'closely' we mean within the W (window size 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 another long D-cache miss, there is no additional performance penalty because their miss latencies are essentially overlapped with the first. This is illustrated in Figure 8. Here, it is assumed that the second miss follows the first by S instructions. When the first load's miss data returns from memory, then the first load commits and no longer blocks the head of the ROB. Then, S new instructions are allowed to enter the ROB. This will take approximately S/I cycles with I being the dispatch width — just enough time to overlap the remaining

latency from the second miss. Note that this overlap holds regardless of S, the only requirement is that S is less than or equal to the ROB size W. A similar argument can be made for any number of other long D-cache misses that occur with W instructions of the first long D-cache miss.

Based on this analysis, we conclude that the penalty for an isolated miss as well as for overlapping long data cache misses, equals the time between the ROB filling up and the data returning from main memory. And this is exactly what our proposed hardware performance counter mechanism counts. In contrast, the naive approach ascribes the total miss latency to *all* long backend misses, *i.e.*, the naive approach does not take into account overlapping long backend misses. This can lead to severe overestimations of the real penalties, see Figure 9. The IBM POWER5 mechanism on the other hand, makes a better approximation of the real penalty and starts counting the long data cache miss penalty as soon as the L2 data cache miss reaches the head of the ROB. By doing so, the IBM POWER5 ascribes a single miss penalty to overlapping backend misses. This is a subtle difference with our mechanism; the IBM POWER5 approach starts counting as soon as the L2 data cache miss reaches the head of the ROB, whereas the method we propose waits until the ROB is effectively filled up. As such, our method does not count for the amount of work that can be done in overlap with the long D-cache miss, *i.e.*, filling up the ROB. This is a small difference in practice, however; see Figure 9.

3.3 Interactions between miss events

Thus far, we have considered the various miss event types in isolation. However, in practice, miss events do not occur in isolation; they interact with other miss events. Accurately dealing with these interactions is crucial for building meaningful CPI stacks since we do not want to double-count miss event penalties. We first treat interactions between frontend miss events. We then discuss interactions between frontend and backend miss events.

3.3.1 Interactions between frontend miss events

The degree of interaction between frontend pipeline miss events (branch mispredictions, I-cache misses and I-TLB misses) is limited because the penalties do not overlap. That is, frontend pipeline miss events serially disrupt the flow of good instructions so their negative effects do not overlap. The only thing that needs to be considered when building accurate CPI stacks is that the penalties due to frontend pipeline miss events along mispredicted control flow paths should not be counted. For example, the penalty due to an I-cache miss along a mispredicted path should not be counted as such. The naive approach does count *all* I-cache and I-TLB misses, including misses along mispredicted paths, which could lead to inaccurate picture of the real penalties. The naive_non_spec method, the IBM POWER5 mechanism as well as our method do not count I-cache and I-TLB misses along mispredicted paths.

3.3.2 Interactions between frontend and long backend miss events

The interactions between frontend pipeline miss events and long backend miss events are more complex because frontend pipeline miss events can be overlapped by long backend miss events. The question then is: how do we account for both miss event penalties? For example, in case a branch misprediction overlaps with 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? In order to answer these questions we measured the fraction of the total cycle count for which overlaps are observed between frontend miss penalties (L1 and L2 I-cache miss, I-TLB miss and branch mispredictions) and long backend miss penalties. The fraction of overlapped cycles is generally very small, as shown in Table 1; no more than 1% for most benchmarks, and only as much as 5% for a couple of benchmarks. Since the fraction overlapped cycles is very limited, any mechanism for dealing with it will result in relatively accurate and meaningful CPI stacks. Consequently, we opt for a hardware performance counter implementation that assigns overlap between frontend and long backend miss penalties to the frontend CPI component, unless the ROB is full (which triggers counting the long backend miss penalty). This implementation results in a simple hardware design.

benchmark	input	% overlap
bzip2	program	0.12%
crafty	ref	1.03%
eon	rushmeier	0.01%
gap	ref	5.40%
gcc	166	0.93%
gzip	graphic	0.04%
mcf	ref	0.02%
parser	ref	0.43%
perlbmk	makerand	1.00%
twolf	ref	4.97%
vortex	ref2	3.10%
vpr	route	0.89%

Table 1: Percentage cycles for which frontend miss penalties overlapped with long backend miss penalties.

4. COUNTER ARCHITECTURE

In our proposed hardware performance counter architecture, we assume one total cycle counter and 8 *global* CPI component cycle counters for measuring 'lost' cycles due 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 idea is to assign every cycle to one of the global CPI component cycle counters when possible; the steady-state (baseline) cycle count then equals the total cycle count minus the total sum of the individual global CPI component cycle counters. We now describe how the global CPI component cycle counters can be computed in hardware. We make a distinction between frontend misses, backend misses, and long latency functional unit stalls.

4.1 Frontend misses

4.1.1 Initial design: FMT

To measure lost cycles due to frontend miss events, we propose a hardware table, called the *frontend miss event table (FMT)*, that is implemented as shown in Figure 10. The FMT is a circular buffer and has as many rows as the processor supports outstanding branches. The FMT also has

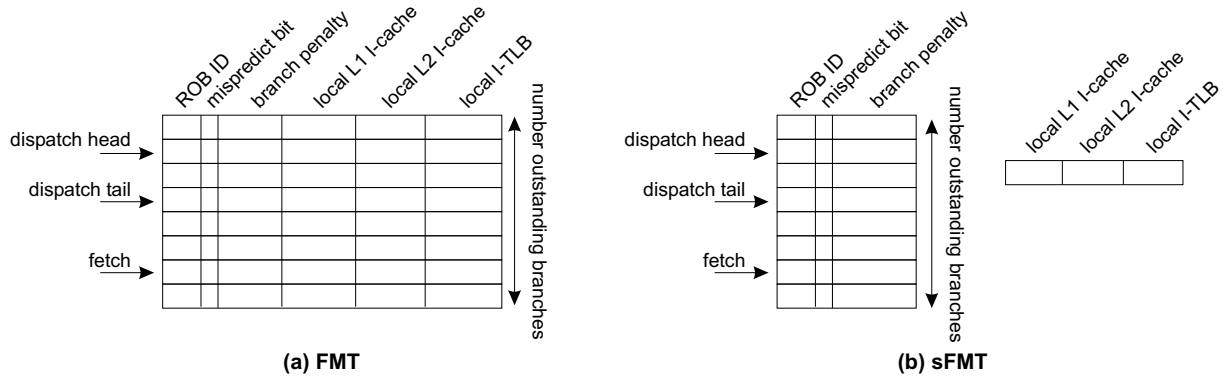

Figure 10: (a) shows the FMT, and (b) shows the sFMT for computing frontend miss penalties.

three pointers, the fetch pointer, the dispatch head pointer, and the dispatch tail pointer. When a new branch instruction is fetched and decoded, an FMT entry is allocated by advancing the fetch pointer and by initializing the entire row to zeros. When a branch dispatches, the dispatch tail pointer is advanced to point to that branch in the FMT, and the instruction's ROB ID is inserted in the 'ROB ID' column. When a branch is resolved and turns out to be a misprediction, the instruction's ROB ID is used to locate the corresponding FMT entry, and the 'mispredict' bit is then set. The retirement of a branch causes the dispatch head pointer to increment which de-allocates the FMT entry.

The frontend miss penalties are then calculated as follows. Any cycle in which no instructions are fed into the pipeline due to an L1 or L2 I-cache miss or an I-TLB miss causes the appropriate *local* counter in the FMT entry (the one pointed to by the fetch pointer) to be incremented. For example, an L1 I-cache miss causes the local L1 I-cache miss counter in the FMT (in the row pointed to by the fetch pointer) to be incremented every cycle until the cache miss is resolved. By doing so, the miss delay computed in the local counter corresponds to the actual I-cache or I-TLB miss penalty (according to interval analysis).

For branches, the local FMT 'branch penalty' counter keeps track of the number of lost cycles caused by a presumed branch misprediction. Recall that the branch misprediction penalty equals the number of cycles between the mispredicted branch entering the ROB and new instructions along the correct control flow path entering the ROB after branch resolution. Because it is unknown at dispatch time whether a branch is mispredicted, the proposed method computes the number of cycles each branch resides in the ROB. That is, the 'branch penalty' counter is incremented every cycle for all branches residing in the ROB, *i.e.*, for all branches between the dispatch head and tail pointers in the FMT. This is done unless the ROB is full — we will classify cycles with a full ROB as long backend misses or long latency misses; this is the easiest way not to double-count cycles under overlapping miss events.

The global CPI component cycle counters are updated when a branch instruction completes: the local L1 I-cache, L2 I-cache and I-TLB counters are added to the respective global cycle counters. In case the branch is incorrectly predicted, then the value in the local 'branch penalty' counter is added to the global branch misprediction cycle counter. And from then on, the global branch misprediction cycle counter is incremented every cycle until new instructions enter the ROB. The resolution of a mispredicted branch also places the FMT dispatch tail pointer to point to the mispredicted branch entry and the FMT fetch pointer to point to the next FMT entry.

4.1.2 Improved design: sFMT

The above design using the FMT makes a distinction between I-cache and I-TLB misses past particular branches, *i.e.*, the local I-cache and I-TLB counters in the FMT are updated in the FMT entry pointed to by the fetch pointer and the fetch pointer is advanced as each branch is fetched. This avoids counting I-cache and I-TLB miss penalties past branch mispredictions. The price paid for keeping track of I-cache and I-TLB miss penalties along mispredicted paths is an FMT that requires on the order of a few hundred bits for storing this information. The simplified FMT design, which is called the *shared FMT* or sFMT, has only one shared set of local I-cache and I-TLB counters; see Figure 10. The sFMT requires that an 'I-cache/I-TLB miss' bit be provided with every entry in the ROB — this is also done in the Intel Pentium 4 and IBM POWER5 for tracking I-cache misses in the completion stage. Since there are no per-branch I-cache and I-TLB counters in the sFMT, the sFMT only requires a fraction of storage bits compared to the FMT. The sFMT operates in a similar fashion as the FMT: the local I-cache and I-TLB counters get updated on I-cache and I-TLB misses. The completion of an instruction with the 'I-cache/I-TLB miss' bit set (i) adds the local I-cache and I-TLB counters to the respective global counters, (ii) resets the local I-cache and I-TLB counters, and (iii) resets the 'I-cache/I-TLB miss' bits of all the instructions in the ROB. In case a mispredicted branch is completed, the local 'branch penalty' counter is added to the global branch misprediction cycle counter and the entire sFMT is cleared (including the local I-cache and I-TLB counters).

Clearing the sFMT on a branch misprediction avoids counting I-cache and I-TLB miss penalties along mispredicted paths. However, when an I-cache or I-TLB miss is followed by a mispredicted branch that in turn is followed by an I-cache or I-TLB miss, then the sFMT incurs an inaccuracy because it then counts I-cache and I-TLB penalties along mispredicted control flow paths. However, given the fact that I-cache misses and branch mispredictions typically oc-

ROB	128 entries
LSQ	64 entries
processor width	decode, dispatch and commit 4 wide fetch and issue 8 wide
latencies	load (2), mul (3), div (20)
L1 I-cache	8KB direct-mapped
L1 D-cache	16KB 4-way set-assoc, 2 cycles
L2 cache	unified, 1MB 8-way set-assoc, 9 cycles
main memory	250 cycle access time
branch predictor	hybrid bimodal/gshare predictor
frontend pipeline	5 stages

Table 2: Processor model assumed in our experimental setup.

cur in bursts, the number of cases where the above scenario occurs is very limited. As such, the additional error that we observe for sFMT compared to FMT, is very small, as will be shown later in the evaluation section.

4.2 Long backend misses

Hardware performance counters for computing lost cycles due to long backend 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 if the instruction blocking the ROB is a L2 D-cache miss or D-TLB miss, respectively. For every cycle that these two conditions hold true, the respective cycle counters are incremented. Note that by doing so, we account for the long backend miss penalty as explained from interval analysis.

4.3 Long latency unit stalls

The hardware performance counter mechanism also allows for computing resource stalls under steady-state behavior. Recall that steady-state behavior in a balanced processor design implies that performance roughly equivalent to the maximum processor width is achieved in the absence of miss events, and that the ROB needs to be filled to achieve the steady state behavior. Based on this observation we can compute resource stalls due to long latency functional unit instructions (including short L1 data cache misses). If the ROB is filled and the instruction blocking the head of the ROB is an L1 D-cache miss, we count the cycle as an L1 D-cache miss cycle; or, if the instruction blocking the head of a full ROB is another long latency instruction, we count the cycle as a resource stall.

5. EVALUATION

5.1 Experimental setup

We used SimpleScalar/Alpha v3.0 in our validation experiments. The benchmarks used, along with their reference inputs, are taken from the SPEC CPU 2000 benchmark suite, see Table 1. The binaries of these benchmarks were taken from the SimpleScalar website. In this paper, we only show results for the CPU2000 integer benchmarks. We collected results for the floating-point benchmarks as well; however, the CPI stacks for the floating-point benchmarks are less interesting than the CPI stacks for the integer benchmarks. Nearly all the floating-point benchmarks show very large L2 D-cache CPI components; only a few benchmarks exhibit significant L1 I-cache CPI components and none of the benchmarks show substantial branch misprediction CPI components. The baseline processor model is given in Table 2.

5.2 Results

This section evaluates the proposed hardware performance counter mechanism. We compare our two hardware implementations, FMT and sFMT, against the IBM POWER5 mechanism, the naive and naive_non_spec approaches and two simulation-based CPI stacks.

The simulation-based CPI stacks will serve as a reference for comparison. We use two simulation-based stacks because of the difficulty in defining what a standard 'correct' CPI stack should look like. In particular, there will be cycles that could reasonably be ascribed to more than one miss event. Hence, if we simulate CPI values in a specific order, we may get different numbers than if they are simulated in a different order. To account for this effect, two simulation-based CPI stacks are generated as follows. We first run a simulation assuming perfect branch prediction and perfect caches, *i.e.*, all branches are correctly predicted and all cache accesses are L1 cache hits. This yields the number of cycles for the base CPI. We subsequently run a simulation with a real L1 data cache. The additional cycles over the first run (which assumes a perfect L1 data cache) gives the CPI component due to L1 data cache misses. The next simulation run assumes a real L1 data cache and a real branch predictor; this computes the branch misprediction CPI component. For computing the remaining CPI components, we consider two orders. The first order is the following: L1 I-cache, L2 I-cache, I-TLB, L2 D-cache and D-TLB; the second order, called the 'inverse order', first computes the L2 D-cache and D-TLB components and then computes the L1 I-cache, L2 I-cache and I-TLB CPI components. Our simulation results show that the order in which the CPI components are computed only has a small effect on the overall results. This follows from the small percentages of cycles that process overlapping frontend and backend miss event penalties, as previously shown in Table 1.

Figure 11 shows normalized CPI stacks for the SPECint2000 benchmarks for the simulation-based approach, the naive and naive_non_spec approach, the IBM POWER5 approach, and the proposed FMT and sFMT approaches. Figure 12 summarizes these CPI stacks by showing the maximum CPI component errors for the various CPI stack building methods.

Figure 11 shows that the naive approach results in CPI stacks that are highly inaccurate (and not even meaningful) for some of the benchmarks. The sum of the miss event counts times 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 count, to be negative. This is the case for a number of benchmarks, such as gap, gcc, mcf, twolf and vpr, with gcc the most notable example. The reason the naive approach fails in building accurate CPI stacks is that the naive approach does not adequately deal with overlapped long backend misses, does not accurately compute the branch misprediction penalty, and in addition, it counts I-cache (and I-TLB) misses along mispredicted paths. However, for benchmarks that have very few overlapped backend misses and very few I-cache misses along mispredicted paths, the naive approach can be fairly accurate, see for example eon and perlbmk. The naive_non_spec approach which does not count miss events

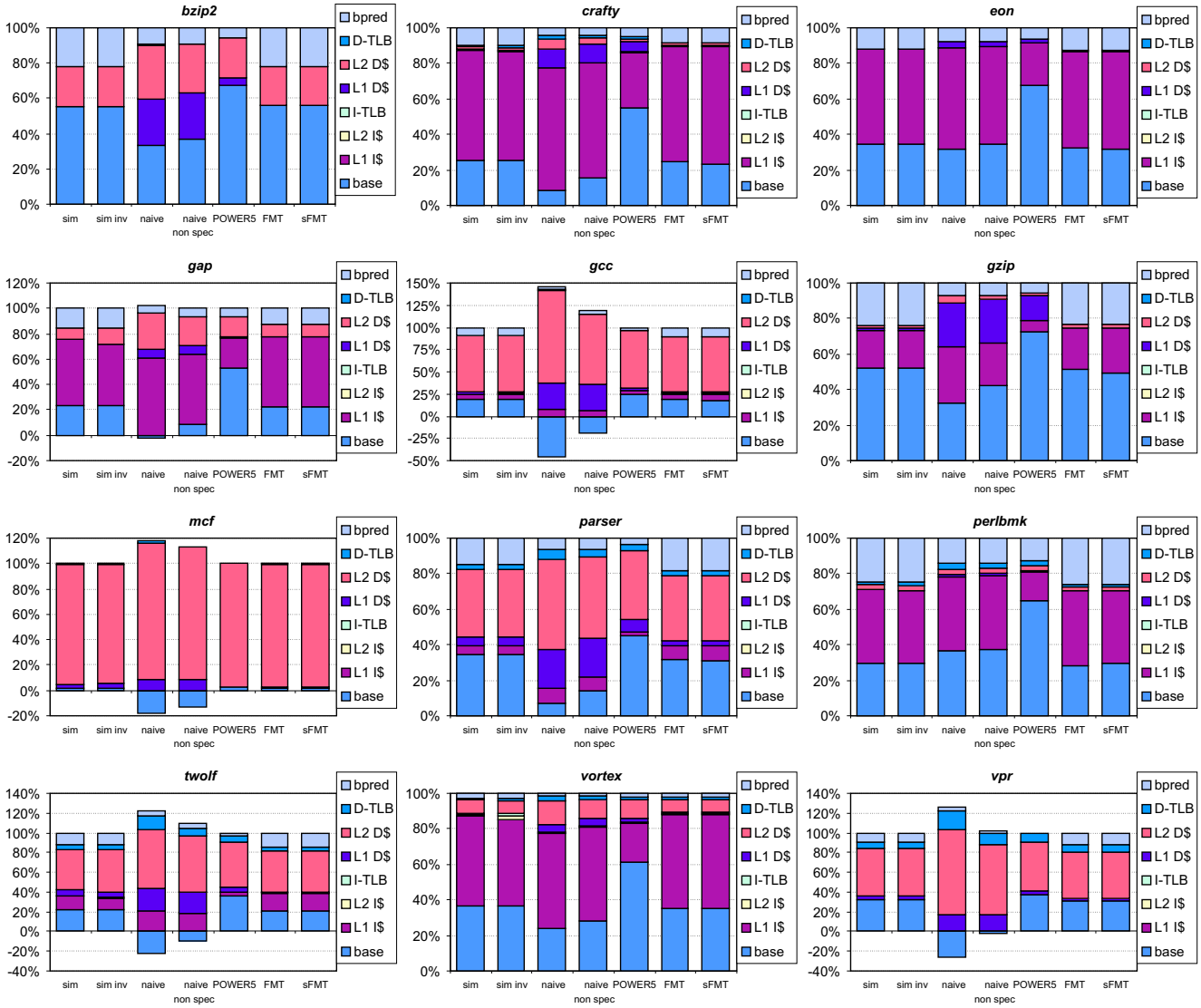

Figure 11: Normalized CPI breakdowns for the SPECint2000 benchmarks: the simulation-based approach, the inverse order simulation-based approach, the naive approach, the naive_non_spec approach, the IBM POWER5 approach and the FMT and sFMT approaches.

along mispredicted paths, is more accurate than the naive approach, however, the CPI stacks are still not very accurate compared to the simulation-based CPI stacks.

The IBM POWER5 approach clearly is an improvement compared to the naive approaches. For the benchmarks where the naive approaches failed, the IBM POWER5 mechanism succeeds in producing meaningful CPI stacks. However, compared to the simulation-based CPI stacks, the IBM POWER5 CPI stacks are still inaccurate, see for example crafty, eon, gap, gzip, perlbmk and vortex. The reason the IBM POWER5 approach falls short is that the IBM POWER5 mechanism underestimates the I-cache miss penalty as well as the branch misprediction penalty.

The FMT and sFMT CPI stacks track the simulation-based CPI stacks very closely. Whereas both the naive and IBM POWER5 mechanisms show high errors for several bench-

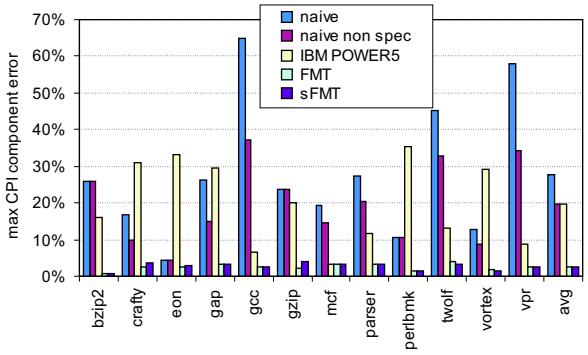

Figure 12: Maximum CPI component error for the naive approaches, the IBM POWER5 approach, FMT and sFMT compared to the simulation-based CPI stacks.

marks, the FMT and sFMT architectures show significantly lower errors for all benchmarks. All maximum CPI component errors are less than 4%, see Figure 12. The average error for FMT and sFMT is 2.5% and 2.7%, respectively.

6. RELATED WORK

The Intel Itanium processor family provides a rich set of hardware performance counters for computing CPI stacks [7]. These hardware performance monitors effectively compute the number of lost cycles under various stall conditions such as branch mispredictions, cache misses, etc. The Digital Continuous Profiling Infrastructure (DCPI) [2] is another example of a hardware performance monitoring tool for an in-order architecture. Computing CPI stacks for in-order architectures, however, is relatively easy compared to computing CPI stacks on out-of-order architectures.

Besides the IBM POWER5 mechanism, other hardware profiling mechanisms have been proposed in the recent past for out-of-order architectures. However, the goal for those methods is quite different from ours. Our goal is to build simple and easy-to-understand CPI stacks, whereas the goal for the other approaches is detailed per-instruction profiling. For example, the *ProfileMe* framework [3] randomly samples individual instructions and collects cycle-level information on a per-instruction basis. Collecting aggregate CPI stacks can be done using the ProfileMe framework by profiling many randomly sampled instructions and by aggregating all of their individual latency information. An inherent limitation with this approach is that per-instruction profiling does not allow for modeling overlap effects. The ProfileMe framework partially addresses this issue by profiling two potentially concurrent instructions. Shotgun profiling [5] tries to model overlap effects between multiple instructions by collecting miss event information within hot spots using specialized hardware performance counters. A postmortem analysis then determines, based on a simple processor model, the amount of overlaps and interactions between instructions within these hot spots. Per-instruction profiling has the inherent limitation of relying on (i) sampling which may introduce inaccuracy, (ii) per-instruction information for computing overlap effects, and (iii) interrupts for communicating miss event information from hardware to software which may lead to overhead and/or perturbation issues.

A number of researchers have looked at superscalar processor models [9, 14, 15, 16, 19], but there are three primary efforts that led to the interval model. First, Michaud *et al.* [14] focused on performance aspects of instruction delivery and modeled the instruction window and issue mechanisms. Second, Karkhanis and Smith [9] extended this type of analysis to all types of miss events and built a complete performance model, which included a sustained steady state performance rate punctuated with gaps that occurred due to miss events. Independently, Taha and Wilson [19] broke instruction processing into intervals (which they call 'macro blocks'). However, the behavior of macro blocks was not analytically modeled, but was based on simulation. Interval analysis combines the Taha and Wilson approach with the Karkhanis and Smith approach to miss event modeling. The interval model represents an advance over the Karkhanis and Smith 'gap' model because it handles short interval behavior in a more straightforward way. The mechanistic interval model presented here is similar to an empirical model of Hartstein and Puzak [6]; however, being an empirical model, it cannot be used as a basis for understanding the mechanisms that contribute to the CPI components.

7. CONCLUSION

Computing CPI stacks on out-of-order processors is challenging because of various overlap effects between instructions and miss events. Existing approaches fail in computing accurate CPI stacks, the main reason being the fact that these approaches build CPI stacks in a bottom up fashion by counting miss events without regard on how these miss events affect overall performance. A top down approach on the other hand, starts from a performance model, interval analysis, that gives insight into the performance impacts of miss events. These insights then reveal how the hardware performance counter architecture should look like for building accurate CPI stacks. This paper proposed such a hardware performance counter architecture that is comparable to existing hardware performance counter mechanisms in terms of complexity, yet it achieves much greater accuracy.

Acknowledgements

Stijn Eyerman and Lieven Eeckhout are Research and Postdoctoral Fellows, respectively, with the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen). This research is also supported by Ghent University, the IWT and the HiPEAC Network of Excellence.

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *Proceedings of the 25th Very Large Database Conference*, 1999.

[2] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, Nov. 1997.

[3] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. *ProfileMe*: Hardware support for instruction-level profiling on out-of-order processors. In *MICRO-30*, Dec. 1997.

[4] S. Eyerman, J. E. Smith, and L. Eeckhout. Characterizing the branch misprediction penalty. In *ISPASS-2006*, Mar. 2006.

[5] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Interaction cost and shotgun profiling. *ACM TACO*, 1(3):272–304, Sept. 2004.

[6] A. Hartstein and T. R. Puzak. The optimal pipeline depth for a microprocessor. In *ISCA-29*, pages 7–13, May 2002.

[7] Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, May 2004. 251110-003.

[8] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In *WMPI 2002 held in conjunction with ISCA-29*, May 2002.

[9] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA-31*, pages 338–349, June 2004.

[10] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *ISCA-25*, June 1998.

[11] Y. Luo, J. Rubio, L. K. John, P. Seshadri, and A. Mericas. Benchmarking internet servers on superscalar machines. *IEEE Computer*, 36(2):34–40, Feb. 2003.

[12] A. Mericas. POWER5 performance measurement and characterization. Tutorial at the IEEE International Symposium on Workload Characterization, Oct. 2005.

[13] A. Mericas. Performance monitoring on the POWER5 microprocessor. In L. K. John and L. Eeckhout, editors, *Performance Evaluation and Benchmarking*, pages 247–266. CRC Press, 2006.

[14] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *PACT-1999*, pages 2–10, Oct. 1999.

[15] D. B. Noonburg and J. P. Shen. Theoretical modeling of superscalar processor performance. In *MICRO-27*, pages 52–62, Nov. 1994.

[16] D. B. Noonburg and J. P. Shen. A framework for statistical modeling of superscalar processor performance. In *HPCA-3*, pages 298–309, Feb. 1997.

[17] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *ASPLOS-VIII*, Oct. 1998.

[18] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, July 2002.

[19] T. M. Taha and D. S. Wills. An instruction throughput model of superscalar processors. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP)*, June 2003.

[20] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Jan. 1996.