Hybrid Analytical Modeling of Pending Cache Hits, Data Prefetching, and MSHRs

Xi E. Chen Tor M. Aamodt
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, BC, CANADA
Email: {xichen, aamodt}@ece.ubc.ca

Abstract

As the number of transistors integrated on a chip continues to increase, a growing challenge is accurately modeling performance in the early stages of processor design. Analytical models have been employed to rapidly search for higher performance designs, and can provide insights that detailed simulators may not. This paper proposes techniques to predict the impact of pending cache hits, hardware prefetching, and realistic miss status holding register (MSHR) resources on superscalar performance in the presence of long latency memory systems when employing hybrid analytical models that apply instruction trace analysis. Pending cache hits are secondary references to a cache block for which a request has already been initiated but has not yet completed. We find pending hits resulting from spatial locality and the fine-grained selection of instruction profile window blocks used for analysis both have non-negligible influences on the accuracy of hybrid analytical models and subsequently propose techniques to account for their effects. We then introduce techniques to estimate the performance impact of data prefetching by modeling the timeliness of prefetches and to account for a limited number of MSHRs by restricting the size of profile window blocks. As with earlier hybrid analytical models, our approach is roughly two orders of magnitude faster than detailed simulations. When modeling pending hits for a processor with unlimited outstanding misses we improve the accuracy of our baseline by a factor of 3.9, decreasing average error from 39.7% to 10.3%. When modeling a processor with data prefetching, a limited number of MSHRs, or both, the techniques result in an average error of 13.8%, 9.5% and 17.8%, respectively.

1. Introduction

To design a new microprocessor architects typically create a cycle-accurate simulator and run numerous simulations to quantify performance trade-offs. Not only is the task of creating such a simulator time-consuming, but also running such simulations can be slow. Both are significant components of overall design-to-market time. As microprocessor design cycles stretch with increasing transistor budgets, architects effectively start each new project with less accurate information about the eventual process technology that will be used, leading to designs that may not achieve the full potential of a given process technology node.

An orthogonal approach to obtaining performance estimates for a proposed design is analytical modeling [36]. An analytical model employs mathematical formulas that approximate the performance of the microprocessor being designed based upon program characteristics and microarchitectural parameters. One of the potential advantages of analytical modeling is that it can require much less time than crafting and running a performance simulator. Thus, when an architect has analytical models available to evaluate a given design, the models can help shorten the design cycle. While cycle-accurate simulator infrastructures exist that leverage reuse of modular building blocks [10], [34], and workload sampling [29], [35] can reduce simulation time, another key advantage of analytical modeling is its ability to provide insights that a cycle-accurate simulator may not.

Several analytical models have been proposed before [8], [16], [19], [23], [24], [25], [26], [27], [28] and Karkhanis and Smith's first-order model [19] is relatively accurate. Their first-order model separately estimates the *cycles per instruction* (CPI) component due to branch mispredictions, instruction cache misses, and data cache misses, then adds each CPI component to an estimated CPI under ideal conditions to arrive at a final model for the performance of a superscalar processor. However, little prior work has focused on analytically modeling the performance impact of data prefetching and the performance impact of hardware support for a *limited* number of overlapping long latency data cache misses due to finite MSHR resources. In this paper we explore how to accurately model these important aspects of modern microprocessor designs.

One significant aspect of long latency data cache misses is the large effect of *pending data cache hits* (PH) on overall performance. A pending data cache hit is a memory reference to a cache block for which a request has already been initiated by another instruction but has not yet completed (i.e., the requested block is still on its way from memory). Figure 1 compares actual CPI due to long latency data cache misses to modeled CPI due to long latency data cache misses for mcf with increasing memory access latency. The first bar (actual) shows the result from a cycle-accurate simulator whose configuration is described in Section 4. The second bar (baseline) shows the result from a careful re-implementation of a previously proposed hybrid analytical model [19][1]. The third bar (SWAM w/ PH) illustrates the result from a new technique that we propose in this paper (see Section 3.5.1). In this paper, a modeled CPI due to long latency data cache misses ($CPI_{D\$miss}$) is always derived by dividing the total extra cycles due to long latency data cache misses by the total number of *instructions committed*. From Figure 1 we observe that the error becomes increasingly significant as memory latency grows. Therefore, to accurately model the performance of future superscalar microprocessors, it is necessary to carefully model pending data cache hits.

This paper makes the following contributions:

- It shows that the performance impact of pending data cache hits is non-negligible for memory intensive applications and describes how to model their effect on performance in the context of a trace driven hybrid analytical model (Section 3.1).
- It presents a novel technique to more accurately compensate for the potential overestimation of the modeled $CPI_{D\$miss}$, which relies upon analysis of a program's individual characteristics (Section 3.2).
- It proposes a technique to model the $CPI_{D\$miss}$ when a data prefetching mechanism is applied in a microprocessor, without requiring a detailed simulator (Section 3.3).
- It describes a technique to analytically model the impact of a limited number of outstanding cache misses supported by a memory system (Section 3.4).
- It also proposes two novel profiling techniques to better analyze overlapped data cache misses (Section 3.5).

We evaluate our approach for modeling data prefetching by using it to predict the performance impact of three different prefetching strategies and find the average error of our model is 13.8% versus 50.5% when not using the technique we propose. As the instruction window of future microprocessors becomes larger [9], a limited number of *Miss Status Holding Registers* (MSHRs) can have a dramatic impact on the performance of the whole system [33]. Our technique reduces the arithmetic mean of the absolute error of our baseline from 33.6% to 9.5% when the maximum number of outstanding misses supported is limited. As with earlier hybrid modeling approaches [19], [18], [20], we find our model is two orders of magnitude faster than detailed simulations. Our improvement increase the realism (and hence applicability) of analytical models for microprocessor designers.

1. Our analysis shows that pending data cache hits can significantly impact model accuracy. However, we note that the prediction error of the CPI due to long latency data cache misses we report for our ***baseline*** modeling technique (described in Section 2) is in some cases large relative to those reported in [19]. Our baseline modeling technique is our implementation of the first-order model described in [19] based upon the details described in that paper and the follow-on work [18], [20]. We believe the discrepancy is partly due to our use of a smaller L2 cache size of 128 KB versus 512 KB used in [19], and partly due to their use of a technique similar to the one we describe in Section 3.5.1 [32].

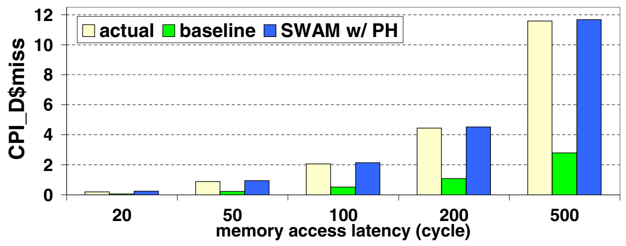


Figure 1. Comparison of CPI component due to long data cache miss versus modeled result for mcf with different memory access latencies

In this paper, we use arithmetic mean of the absolute error to validate the accuracy of an analytical model, which we argue is the correct measure since it always reports the largest error numbers and is thus conservative in not overstating the case for improved accuracy. Note that we are interested in averaging the error of the CPI prediction on different benchmarks, not the average CPI predicted for an entire benchmark suite, which often allows errors on individual benchmarks to "cancel out" in a way that suggests the modeling technique is more accurate than it really is. We also report the geometric mean and harmonic mean of the absolute error to allay any concerns that these numbers might lead to different conclusions. In all cases the improvements resulting from applying our new modeling techniques are robust enough that the selection of averaging technique does not impact our conclusions.

The rest of this paper is organized as follows. Section 2 reviews the first-order model [18], [19], [20]. Section 3 describes how to accurately model the effects of pending data cache hits, data prefetching, and hardware that supports a limited number of outstanding data cache misses. Section 4 describes the experimental methodology and Section 5 presents and analyzes our results. Section 6 reviews related work and Section 7 concludes the paper.

2. Background: First-Order Model

Before explaining the details of our techniques introduced in Section 3, it is necessary to be familiar with the basics of the first-order model of superscalar microprocessors. Karkhanis and Smith's first-order model [19] leverages the observation that the overall performance of a superscalar microprocessor can be estimated reasonably well by subtracting the performance losses due to different types of miss-

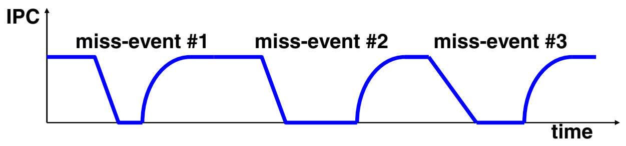


Figure 2. Useful instructions issued per cycle (IPC) over time used in the first order model [19]

events from the processor's sustained performance under the absence of miss-events. The miss-events considered include long latency data cache misses (e.g., L2 cache misses for a memory system with two-level cache hierarchy), instruction cache misses, and branch mispredictions.

Figure 2 illustrates this approach. When there are no miss-events, the performance of the superscalar microprocessor is approximated by a stable IPC, expressed through a constant useful instructions issued per cycle (IPC) over time. When a miss-event occurs, the performance of the processor falls and the IPC gradually decreases to zero. After the miss-event is resolved, the decreased IPC ramps up to the stable value under ideal conditions. A careful analysis of this behavior leads to the first-order model [19].

While Figure 2 shows that a miss-event occurs only after the previous miss-events have been resolved, in a real processor it is possible for different types of miss-events to overlap. For example, a load instruction can miss in the data cache a few cycles after a branch is mispredicted. However, it has been observed (and we confirmed) that overlapping between *different types* of miss-events is rare enough that ignoring it results in negligible error in typical applications [19], [12].

This paper focuses on improving the accuracy of the modeled $CPI_{D\$miss}$ (i.e., CPI component due to long latency data cache misses) since it is the component with the largest error in prior first-order models [18], [19]. Note that short latency data cache misses (i.e., L1 data cache misses that hit in the L2 cache in this paper) are not regarded as miss-events in prior first-order models [18], [19] and they are modeled as long-execution-latency instructions when modeling the base CPI. In the rest of this paper, we use the term "cache misses" to represent long latency data cache misses. As noted by Karkhanis and Smith [19], the interactions between microarchitectural events of the same type cannot be ignored.

Our baseline technique for modeling data cache misses, based upon Karkhanis and Smith's first-order model [19], analyzes dynamic instruction traces created by a cache simulator. To differentiate such models, which analyze instruction traces, from earlier analytical models [8], [23], [1], [26], [16], [27] that do not, we also refer to them as *hybrid analytical models* in this paper. In each *profile step*, a ROB_{size} number of consecutive instructions in the trace are put into the *profiling window* (or block) and analyzed, where ROB_{size} is the size of the re-order buffer. If all of the loads missing in the data cache in a profile step are data independent of each other, they are considered overlapped (i.e., the overlapped misses have the same performance impact as a single miss). When data dependencies exist between misses, the maximum number of misses in the same data dependency chain is recorded and the execution of all the other misses are modeled to be hidden under this dependency chain.

In the rest of this paper, $num\_serialized\_D\$miss$ represents the sum of the maximum number of misses measured in any single data dependency chain in a block of instructions, accumulated over all blocks making up the entire instruction trace. When all instructions in the trace have been analyzed, the $CPI_{D\$miss}$ can be estimated as

$$CPI_{D\$miss} = \frac{num\_serialized\_D\$miss \times mem\_lat}{total\_num\_instructions} \quad (1)$$

where mem_lat stands for the main memory latency and $total\_num\_instructions$ is the total number of instructions committed (of any type).

The $CPI_{D\$miss}$ modeled in Equation 1 often overestimates the actual $CPI_{D\$miss}$ since out-of-order execution enables overlap of computation with long latency misses. A simple solution proposed by Karkhanis and Smith [19] is to subtract a fixed number of cycles per serialized data cache miss based upon ROB size to compensate. The intuition for this compensation is that when a load issues and accesses the cache, it can be the oldest instruction in the ROB, the youngest instruction in the ROB, or somewhere in between. If the instruction is the oldest or nearly the oldest, the performance loss (penalty of the instruction) is the main memory latency. On the other hand, if the instruction is the youngest or nearly the youngest one in the ROB and the ROB is full, its penalty can be partially hidden by the cycles required to drain all instructions before it, and can be approximated as $mem\_lat - \frac{ROB_{size}}{issue\_width}$ [19]. It has been observed that loads missing in the cache are usually relatively old when they issue [19]; and thus, perhaps the simplest (though not most accurate) approach is to use no compensation at all [19]. The mid-point of the two extremes mentioned above can also be used (i.e., a load missing in the cache is assumed to be in the middle of ROB when it issues), and the numerator in Equation 1 becomes $num\_serialized\_D\$miss \times (mem\_lat - \frac{ROB_{size}}{2 \times issue\_width})$ [18].

3. Modeling Long Latency Memory Systems

In this section, we describe how we model pending cache hits, data prefetching, and a limited number of MSHRs.

3.1. Modeling Pending Data Cache Hits

The method of modeling long latency data cache misses described in Section 2 profiles dynamic instruction traces generated by a cache simulator [19]. Since a cache simulator provides no timing information, it classifies the load or store bringing a block into the cache as a miss and all subsequent instructions accessing the block before it is evicted as hits.

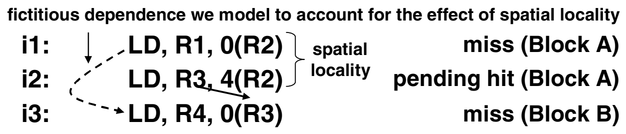

fictitious dependence we model to account for the effect of spatial locality
i1: LD, R1, 0(R2) miss (Block A)
i2: LD, R3, 4(R2) pending hit (Block A)
i3: LD, R4, 0(R3) miss (Block B)
spatial locality

Figure 3. An example showing how two data independent misses (i1, i3) are connected by a pending hit (i2), upon which i3 is data dependent.

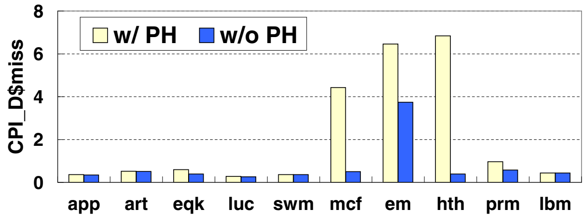


Figure 4. Impact of pending data cache hit latency on the CPI due to long latency cache misses (measured over all instructions committed from a detailed simulator)

However, the actual latency of many instructions classified as a hit by a cache simulator is much longer than the cache hit latency. For example, if there are two close load instructions accessing data in the same block that is not currently in the cache, the first load will be classified as a miss by the cache simulator and the second load as a hit, even though the data would still be on its way from memory in a real processor implementation. Therefore, since the second load is classified as a hit in the dynamic instruction trace, it is ignored in the process of modeling $CPI_{D\$miss}$ using the approach described in Section 2.

More importantly, a significant source of errors results when two or more data independent load instructions that miss in the data cache are connected by a *third* pending data cache hit. We elaborate what "connected" means using the simple example in Figure 3. In this example, i1 and i3 are two loads that miss and they are data *independent* of each other, while i2 is a pending hit since it accesses the data in the same cache block as i1. The model described in Section 2 classifies i1 and i3 as overlapped and the performance penalty due to each miss using that approach is estimated as half of the memory access latency (total penalty is the same as if there is a single miss). However, this approximation is inaccurate since i3 is data dependent on the pending data cache hit i2 and i2 gets its data when i1 obtains its data from memory (i.e., i1 and i2 are waiting for the data from the same block). Therefore, in the actual hardware, i3 can only start execution after i1 gets its data from memory although there is no true data dependence between i1 and either i2 or i3. This scenario is common since most programs contain significant spatial locality. The appropriate way to model this situation is to consider i1 and i3 to be serialized in our analytical model, even though they are data independent and access distinct cache blocks.

Figure 4 shows the impact that pending data cache hits combined with spatial locality have on overall performance for processors with long memory latencies. The first bar (w/ PH) illustrates measured $CPI_{D\$miss}$ for each benchmark on the detailed simulator described in Section 4 and the second bar (w/o PH) shows the measured $CPI_{D\$miss}$ when all the pending data cache hits are simulated as having a latency equal to the L1 data cache hit latency. From this figure, we observe that the difference is significant for eqk, mcf, em, hth [37], and prm.

To model the effects of pending data cache hits analytically, we first need to identify them without a detailed simulator. At first, this may seem impossible since there is no timing information provided by the cache simulator. We tackle this by assigning each instruction in the dynamic instruction trace a sequence number in program order and labeling each memory access instruction in the trace with the sequence number of the instruction that first brings the memory block into the cache. Then, when we profile the instruction trace, if a hit accesses data from a cache block that was first brought into the cache by an instruction still in the profiling window, it is regarded as a pending data cache hit.

For every pending hit identified using this approach (e.g., i2 in Figure 3), there is a unique instruction earlier in the profiling window that first brought in the cache block accessed by that pending hit (e.g., i1 in Figure 3). When we notice a data dependence between a later cache miss (e.g., i3 in Figure 3) and the pending hit (i2), we model a dependence between the early miss (i1) and the instruction that is data dependent on the pending hit (i3) since the two instructions (i1 and i3) have to execute serially due to the constraints of the microarchitecture.

3.2. Accurate Exposed Miss Penalty Compensation

While the model described in Section 2 uses a fixed number of cycles to adjust the modeled $CPI_{D\$miss}$, we found that compensation with a fixed number of cycles (a constant ratio of the reorder buffer size) does not provide consistently accurate compensation for all of the benchmarks that we studied, resulting in large modeling errors (see Figure 10). To capture the distinct distribution of long latency data cache misses of each benchmark, we propose a novel compensation method. The new method is motivated by our observation that the number of cycles hidden for a load missing in the cache is roughly proportional to the distance between the load and the immediately preceding load that missed in the cache (we define the distance between two instructions to be the difference between their instruction sequence number). This is because when a load instruction misses in the cache, most of the instructions between that load and the immediately preceding long latency miss are independent of that load. Therefore, we approximate the latency of the later load that can be overlapped with useful computation as the time used to drain those intermediate instructions from the instruction window, which we estimate as the distance between the two loads divided by the issue width. When we profile an instruction trace, the average distance between two consecutive loads missing in the cache is also collected and used to adjust the modeled $CPI_{D\$miss}$. If the distance between two misses exceeds the window size, it is truncated since the miss latency can be overlapped by at most $ROB_{size} - 1$ instructions.

Equation 2, below, shows how the $CPI_{D\$miss}$ is adjusted by subtracting a compensation term, $\frac{dist}{issue\_width} \times num\_D\$miss$, from the numerator in Equation 1.

$$CPI_{D\$miss} = \frac{num\_serialized\_D\$miss \times mem\_lat - comp}{total\_num\_instructions}$$

$$comp = (\frac{dist}{issue\_width} \times num\_D\$miss) \quad (2)$$

Here $dist$ is the average distance between two consecutive loads that miss in the cache and the term $\frac{dist}{issue\_width}$ represents the average number of cycles hidden for each cache miss. The product of this term and the total number of loads missing in the cache ($num\_D\$miss$) becomes the total number of cycles used to compensate for the overestimation of the baseline profiling method.

3.3. Modeling Data Prefetching

Data prefetching is a technique to bring data from memory into the cache before it is required so as to hide (or partially hide) long memory access latency. Many hardware data prefetching strategies have been proposed before [2], [14], [17], [30]. In this section, we demonstrate how to extend our model described in Section 3.1 to estimate the $CPI_{D\$miss}$ when a data prefetching technique is employed *without running detailed simulations*.

To model the $CPI_{D\$miss}$ when a particular prefetching method is applied, a cache simulator implementing that prefetching method is needed to generate a dynamic instruction trace. While this does require some coding, we found that the resulting analytical model obtains very accurate results and is two orders of magnitude faster than detailed simulations. As described in Section 3.1, when a cache simulator generates an instruction trace, each memory access instruction in the trace is labeled with the sequence number of the instruction that first brought the data into the cache. If the data required by a load was brought into the cache by a prefetch, then the load is labeled with the sequence number of the previous instruction that triggered the prefetch.

Recall that, when no prefetching mechanism is applied, an instruction trace generated by a cache simulator is divided into blocks of instructions and each block is analyzed in a profile step. In each profile step, the maximum number of loads that are in a dependence chain and miss in the cache is recorded. However, when an effective prefetching method is implemented, many loads that would have missed in the cache become hits. To be more specific, many of them become pending hits given that some of the prefetches cannot fully hide the memory access latency. We found that to accurately model prefetch performance, it is necessary to approximate the timeliness of the prefetches and consequently the latencies of these pending hits relatively accurately.

Figure 5 illustrates how we analyze a pending hit in an instruction trace when a particular prefetching mechanism is applied. Here a pending hit can either be due to a prefetch or a demand miss and, in both cases, it is analyzed using the algorithm in Figure 5. For each pending hit (`crntInst` in Figure 5), we find the instruction (`prevInst` in Figure 5) that brought `crntInst`'s required data into the cache. We approximate `crntInst`'s latency based upon the observation that typically the further `prevInst` is from `crntInst`, the more latency of `crntInst` can be hidden. The *hidden latency* of `crntInst` is estimated as the number of instructions between `crntInst` and `prevInst` divided by the issue width of the microprocessor being modeled. Note that we employ the approximation of ideal CPI equal to 1/issueWidth in this calculation. Then, `crntInst`'s latency is estimated as the difference between the memory access latency and the hidden latency, or zero if the memory latency is completely hidden. This latency is in cycles, and we normalize it by dividing it by the main memory latency since the accumulated $num\_serialized\_D\$miss$ after each profile step is represented in units of main memory latency.

The part of the code marked B in Figure 5 models a significant phenomenon (late or tardy prefetches) that we observed in our study of various hardware prefetching mechanisms. Since the instruction trace being analyzed is generated by a cache simulator that is not aware of the out-of-order execution of the superscalar microprocessor being modeled, a pending hit due to prefetching indicated by the cache simulator is often actually a miss during out-of-order execution. Figure 6 shows a simplified example illustrating how this may happen. In this example, there are eight instructions and they are labeled from i1 to i8 in program order. Figure 6 shows the data dependency graph constructed during profiling according to an instruction trace generated by a cache simulator *assuming the pseudo-code marked B in Figure 5 is not included*. In Figure 6, i1 and i5 are loads missing in the data cache (represented by the shaded circles) and i6 triggers a prefetch that brings the data accessed by a load i8 into the cache when i6 issues (represented by the broken line arrow labeled "prefetch" from i6 to i8). For each instruction, the longest normalized length of the data dependency chain up to and including that instruction is shown (in units of main memory latency). For example, "i3.length=1" above i3 in Figure 6 means that it takes one memory access latency from when i1 (the first instruction in the profile step) issues until i3 finishes execution since i3 is data dependent on i1, which missed in the cache. Since i8 is a pending hit (represented by the circle

```
if ( the instruction (crntInst) is a pending hit (e.g., i8 in Fig 6) ) {
    find the most recent instruction (prevInst) in profiling window (e.g., i6 in Fig 6) that brings crntInst's required data into the cache
                                          estimated hidden latency
 A  { crntInst.lat = max(memLat - (crntInst.iseq – prevInst.iseq) / issueWidth, 0)    // calculate the latency of the current instruction
    { crntInst.lat = crntInst.lat / memLat    // normalize the crntInst.lat to the memory latency

    crntInst.length = max(inst.length) where inst is an instruction on which crntInst directly depends
                                          (true data dependency exists, e.g., i7 → i8 in Fig 6)
    if (crntInst.length < prevInst.length – prevInst.lat) {      }
        crntInst.length = critInst.length + 1                    }  B  tardy
        crntInst.lat = 1                                          }     prefetch
    } else {
        accmLength = prevInst.length – prevInst.lat  + crntInst.lat
        if (accmLength > crntInst.length) {
  C             crntInst.lat = accmLength – crntInst.length
                crntInst.length = accmLength
timely  }
prefetch else
                crntInst.lat = 0
    }
}
```

Notation

crntInst:	the current instruction being analyzed
prevInst:	the instruction bringing the data required by the current instruction into the cache
crntInst.iseq:	the instruction sequence number of the current instruction
issueWidth:	the issue width of the microprocessor
memLat:	the memory access latency
crntInst.lat:	the normalized time interval between the issue and the writeback of the current instruction
crntInst.length:	the normalized length of the data dependency chain up to the current instruction

Figure 5. Algorithm for analyzing a pending hit in an instruction trace when a prefetching mechanism is applied

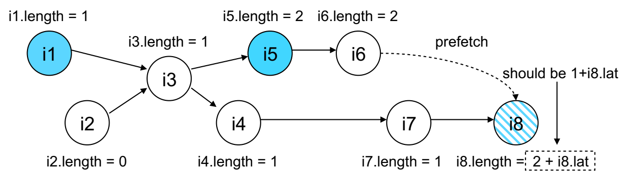


Figure 6. An example motivating Figure 5 part B

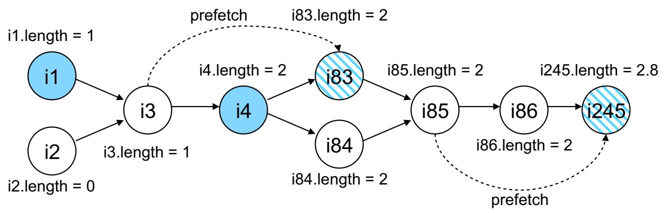


Figure 7. An example explaining Figure 5 part C

filled with hatching) and the associated prefetch is started when i6 issues, i8.length is calculated, without B, as the sum of i6.length and i8.lat, where i8.lat is estimated in part A in Figure 5 as $\frac{memLat - \frac{i8.iseq - i6.iseq}{issueWidth}}{memLat}$. In this example, i8.lat is almost equal to 1.0 since i8 is very close to i6.

Although the data accessed by i8 is regarded as being prefetched by the algorithm in Figure 5 without B, i8 is actually (as determined by detailed simulation) a miss rather than a pending hit due to out-of-order execution. In Figure 6, we observe that i6.length is bigger than i7.length. Therefore, before i6 (e.g., a load instruction) issues (and hence triggers a hardware generated prefetch), i8 has already issued and missed in the data cache. Thus, the prefetch provides no benefit. The code marked B in Figure 5 accurately takes account of this significant effect of out-of-order scheduling by checking if crntInst (i8) issues before the prefetch is triggered. We observed that removing part B in Figure 5 increases the average error for the three prefetching techniques that we model from 13.8% to 21.4% while adding part B slows our model by less than 2%.

An example in Figure 7 shows how the part of code marked C in Figure 5 models the case when a useful prefetch occurs in out-of-order execution (i.e., a prefetch which lowers CPI). In Figure 7, only nine relevant instructions are shown out of the 256 instructions included in a profile step (assuming ROB_{size} is 256). Among these nine instructions, i1 and i4 are loads that miss in the data cache and both i3 and i85 trigger prefetches, making i83 and i245, respectively, pending hits. The number of cycles hidden in the prefetch triggered by i3 is estimated as $\frac{i83.iseq - i3.iseq}{issueWidth} = \frac{83-3}{4} = 20$ (when issue width is four), and then the remaining latency after normalization is calculated as $\frac{memLat-20}{memLat} = 0.9$ (we assume throughout our examples that memory access latency is 200 cycles). However, since i83 is data dependent of i4 and i4.length=2, when i83 issues, its prefetched data has already arrived at the data cache and its real latency becomes zero (this case corresponds to the "else part" inside of part C in Figure 5). The number of cycles hidden by the prefetch for i245 is estimated (from part A in Figure 5) as $\frac{i245.iseq - i85.iseq}{issueWidth} = \frac{245-85}{4} = 40$ with remaining normalized latency of $\frac{memLat-40}{memLat} = 0.8$. Since the instruction triggering the prefetch (i85) and the instruction that i245 directly depends on (i86) finish execution around the same time (i.e., i85.length=i86.length), i245.length becomes 2.8 and i245.lat becomes 0.8 (this case corresponds to the "if part" inside of part C in Figure 5).

3.4. Modeling a Limited Number of MSHRs

The method of analytically modeling the CPI due to long latency data cache misses described in Section 2 assumes that

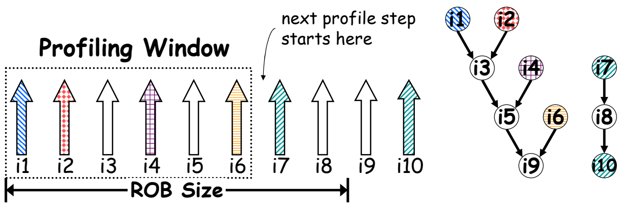


Figure 8. An example showing profiling with $ROB_{size} = 8$ and $N_{MSHR} = 4$. Each arrow corresponds to a dynamic instruction in the trace. Data cache misses are filled with patterns. Corresponding data dependency graph is shown to the right.

at most ROB_{size} cache misses can be overlapped. However, this assumption is unreasonable for most modern processors since the maximum number of outstanding cache misses the system supports is limited by the number of *Miss Status Holding Registers* (MSHRs) [21], [13], [3] in the processor. In a real processor, the issue of memory operations to the memory system has to stall when available MSHRs run out.

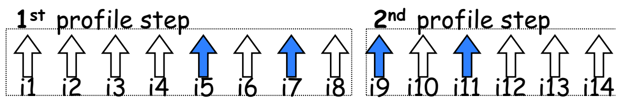
Based upon the technique described in Section 2, the profiling window with the same size as the instruction window is always assumed to be full when modeling $CPI_{D\$miss}$. In order to model a limited number of outstanding cache misses, we need to refine this assumption. During a profile step, we first stop putting instructions into the profiling window when the number of instructions that miss in the data cache and have been analyzed is equal to N_{MSHR} (number of MSHRs) and then update $num\_serialized\_D\$miss$ only based upon those instructions that have been analyzed to that point[2].

Figure 8 illustrates how the profiling technique works when the number of outstanding cache misses supported is limited to four. Once we encounter N_{MSHR} (four) cache misses in the instruction trace (i.e., $i1$, $i2$, $i4$, and $i6$), the profile step stops and $num\_serialized\_D\$miss$ is updated (i.e., the profiling window is made shorter). In the example, the four misses are data independent of each other (and not connected with each other via a pending hit as described in Section 3.1), thus $num\_serialized\_D\$miss$ is incremented by only one. Although $i7$ also misses in the cache, it is included in the next profile step since all four MSHRs have been used.

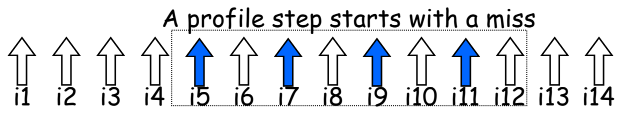
3.5. Profiling Window Selection

In this section, we present two important refinements to the profiling technique described in Section 2 (which we will refer to hereafter as *plain profiling*) to better model the overlapping between cache misses.

2. In real execution, cache misses that are regarded as not present in the profiling window simultaneously due to lack of available MSHRs could actually be in the instruction window simultaneously. Reducing the profiling window size only approximates the performance loss due to a limited number of MSHRs. We leverage this observation in Section 3.5.2.



(a) **Plain Profiling**



(b) **SWAM Profiling**

Figure 9. An example comparing plain profiling and SWAM profiling with $ROB_{size} = 8$. Each arrow is a dynamic instruction. Data cache misses are shaded.

3.5.1. Start-with-a-miss (SWAM) Profiling. We observe that often the plain profiling technique described in Section 2 does not account for all of the cache misses that can be overlapped, due to the simple way in which it partitions an instruction trace. Figure 9(a) shows a simple example. In this example, we assume that all the cache misses (shaded arrows) are data independent of each other for simplicity. Using the profiling approach described in Section 2, a profile step starts at pre-determined instructions (for example, $i1$, $i9$, $i17$..., when ROB_{size} is eight). Therefore, although the latency of $i5$, $i7$, $i9$, and $i11$ can be overlapped, the plain profiling technique does not account for this.

By making each profile step start with a cache miss, we find that the accuracy of the model improves significantly. Figure 9(b) illustrates this idea. Rather than starting a profile step with $i1$, we start a profile step with $i5$, so that the profiling window will include $i5$ to $i12$. Then, the next profile step will seek and start with the first cache miss after $i12$. We call this technique start-with-a-miss (SWAM) profiling and in Section 5.1 we will show that it decreases the error of plain profiling from 29.3% to 10.3% with unlimited MSHRs.

We explored a sliding window approximation (start each profile window on a successive instruction of any type), but found it did not improve accuracy while begin slower. SWAM improves modeling accuracy because it more accurately reflects what the contents of the instruction window of a processor would be (a long latency miss would block at the head of the ROB).

3.5.2. Improved SWAM for Modeling a Limited Number of MSHRs (SWAM-MLP). The technique for modeling MSHRs proposed in Section 3.4 can be combined with SWAM to better model the performance when the number of outstanding cache misses supported by the memory system is limited. The basic idea is to have each profile step start with a miss and finish either when the number of instructions that have been analyzed equals the size of the instruction window or when the number of cache misses that have already been analyzed equals the total number of MSHRs. However, choosing a profiling window independent of whether a cache miss is data dependent on other misses (or connected to other

Table 1. Microarchitectural Parameters

Machine Width	4
ROB Size	256
LSQ Size	256
L1 D-Cache	16KB, 32B/line, 4-way, 2-cycle latency
L2 Cache	128KB, 64B/line, 8-way, 10-cycle latency
Memory Latency	200 cycles

misses via pending hits as described in Section 3.1) leads to inaccuracy because data dependent cache misses cannot simultaneously occupy an MSHR entry.

To improve accuracy further, we stop a profile step when the number of cache misses that are data independent of misses that have been analyzed in the same profile step (rather than the number of cache misses being analyzed) equals the total number of MSHRs. In the rest of this paper we call this technique SWAM-MLP since it improves SWAM by better modeling memory level parallelism. When a miss depends on an earlier miss in the same profiling window, the later miss cannot issue until the earlier one completes and SWAM-MLP improves model accuracy because it takes into account that out-of-order execution can allow another independent miss that is younger than both of the above misses to issue. Therefore, the number of instructions that miss in the data cache and that should be analyzed in a profile step should, in this case, be more than the total number of MSHRs.

4. Methodology

To evaluate our analytical model, we have modified SimpleScalar [5] to simulate the performance loss due to long latency data cache misses when accounting for a limited number of MSHRs. We compare against a cycle accurate simulator rather than real hardware to validate our models since a simulator provides insights that would be challenging to obtain without changes to currently deployed superscalar performance counter hardware [12]. We believe the most important factor is comparing two or more competing (hybrid) analytical models against a single detailed simulator provided the latter captures the behavior one wishes to model analytically. Table 1 describes the microarchitectural parameters used in this study. Note we are focusing on predicting only the CPI component for data cache misses using our model. We verified that the CPI contribution due to overlapping miss events is small for our benchmarks with realistic branch prediction and instruction caches [7] so our comparisons in Section 5 is to a detailed cycle accurate simulator in which instruction cache misses have the same latency as hits and all branches are predicted perfectly. In the rest of this paper, we focus on how to accurately predict $CPI_{D\$miss}$, which is the performance loss due to long latency data cache misses when both branch predictor and instruction cache are ideal (this is the same methodology applied to model $CPI_{D\$miss}$ described in [19]).

Table 2. Benchmarks

Benchmark	Label	Miss rate	Suite
173.applu	app	31.1MPKI	SPEC 2000
179.art	art	117.1MPKI	SPEC 2000
183.equake	eqk	15.9MPKI	SPEC 2000
189.lucas	luc	13.1MPKI	SPEC 2000
171.swim	swm	23.5MPKI	SPEC 2000
181.mcf	mcf	90.1MPKI	SPEC 2000
em3d	em	74.7MPKI	OLDEN
health	hth	45.7MPKI	OLDEN
perimeter	prm	18.7MPKI	OLDEN
470.lbm	lbm	17.5MPKI	SPEC 2006

To evaluate the technique proposed in Section 3.3 for estimating the $CPI_{D\$miss}$ when a prefetching mechanism is applied, we have applied our modeling techniques to predict the performance benefit of three different prefetching mechanisms: prefetch-on-miss [30], tagged prefetch [14], and stride prefetch [2]. When prefetch-on-miss [30] is applied, an access to a cache block that results in a cache miss will initiate a prefetch for the next sequential block in memory given that the block is not in the cache. The tagged prefetch mechanism [14] adds a tag bit to each cache block to indicate whether the block was demand-fetched or prefetched. When a prefetched block is referenced, the next sequential block is prefetched if it is not in the cache. The stride prefetch technique [2] uses a *reference prediction table* (RPT) to detect address referencing patterns. Each entry in the RPT is assigned a state and a state machine is applied to control the state of each entry. Whether a prefetch is initialized or not depends on the current state of the entry [2]. In this study, we modeled a 128-entry, 4-way RPT that is indexed by the microprocessor's program counter (PC).

To stress our model, we simulate a relatively small L2 cache compared to contemporary microprocessors. We note that the size of the L2 cache that we simulated is close in size to those employed in microprocessors shipped at the time when those benchmarks we use were released. The benchmarks chosen are ones from SPEC 2000 [31] and OLDEN [6] that have at least 10 long latency data cache misses for every 1000 instructions simulated (10MPKI). Table 2 illustrates the miss rates of these benchmarks and the labels used to represent them in figures. Moreover, for each benchmark, we select 100M representative instructions to simulate using the Sim-Point toolkit [29].

5. Results

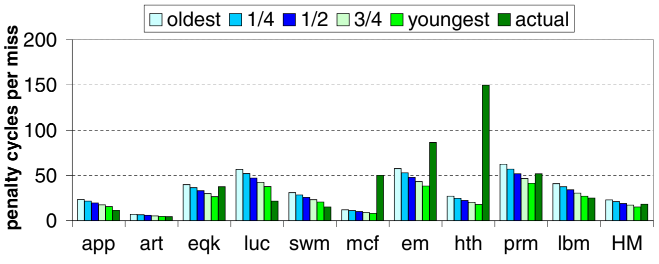
This section summarizes our experimental results.

5.1. Modeling Pending Data Cache Hits

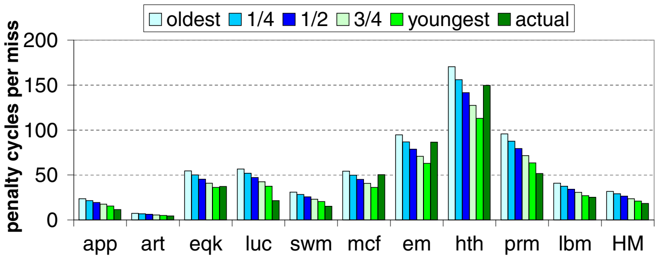
Section 2 describes prior proposals for compensating for the overestimation of modeled penalty cycles per serialized miss using a *fixed* number of cycles. Figure 10(a) and Figure 10(b) illustrate the modeled results after compensation with constant cycles both *without*, and *with* the pending hit compensation technique described in Section 3.1, respectively. In these two figures, we show results using five different constant compensation factors. The first bar (oldest) corresponds to the assumption that an instruction that misses in the cache is always the oldest one in the instruction window when it issues (accesses the first level cache). The second bar (1/4) corresponds to the assumption that there are always $\frac{1}{4}ROB_{size} = 64$ in-flight instructions older than a cache miss when it issues and it is similar to the next two bars, (1/2) and (3/4). The fifth bar (youngest) corresponds to the assumption that there are always $ROB_{size} - 1$ older instructions in the window when the instruction issues (i.e., the instruction is always the youngest one in the window when it issues). The last bar (actual) shows the simulated penalty cycles per cache miss from cycle accurate simulation. From this data, we observe that there is no one fixed cycle compensation method that performed consistently the best for all of the benchmarks we studied. For example, in Figure 10(a) we observe that error is minimized using "youngest" for *app*, *art*, *luc*, *swm*, and *lbm*, but minimized using "oldest" for *em*, *mcf*, and *hth*, while, *eqk* and *prm* requires something in-between. The harmonic mean for each fixed cycle compensation method is also shown and we notice that, due to the fact that positive and negative errors cancel out, the harmonic means of some fixed cycle compensation methods appear close to the detailed simulation results. However, it is important to recognize that their accuracy on individual benchmarks is quite poor. By using the fixed cycle compensation method, we find that the smallest arithmetic mean of absolute error is 43.5% when not modeling pending hits and 26.9% when modeling pending hits, resulting when employing "youngest" compensation.

To account for the distinct behavior of each benchmark, we use the average distance between two consecutive cache misses to compensate for the overestimation of the modeled extra cycles due to long latency data cache misses as described in Section 3.2. Figure 11(a) compares the $CPI_{D\$miss}$ for both the plain profiling technique described in Section 2 and the start-with-a-miss (SWAM) profiling technique described in Section 3.5.1 (with pending hits modeled) to the results from detailed simulation. The first bar (Plain w/o comp) and the third bar (SWAM w/o comp) correspond to the modeled results without any compensation; the second bar (Plain w/ comp) and the fourth bar (SWAM w/ comp) are the modeled results with the compensation technique described in Section 3.2.

Figure 11(a) and Figure 11(b) show that for benchmarks with heavy pointer chasing such as mcf, $em3$, and hth, ignoring the effects of pending data cache hits results in a dramatic underestimate for $CPI_{D\$miss}$. As discussed in Section 3.1, the reason for this is that many data independent misses are connected by pending cache hits, which must be appropriately modeled. Moreover, as we expect, SWAM profiling is more accurate than plain profiling since it can capture



(a) Not modeling pending data cache hits



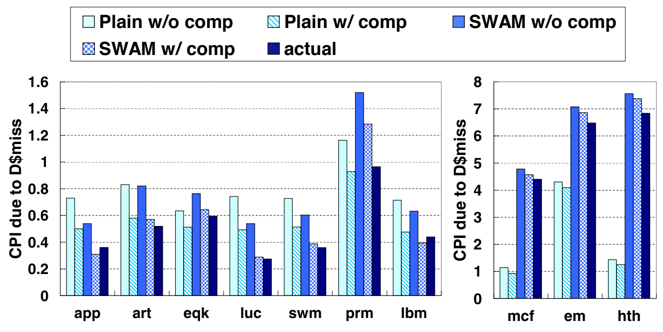
(b) Modeling pending data cache hits

Figure 10. Penalty cycles per miss with fixed number of cycles compensation for plain profiling (Unlimited MSHRs)

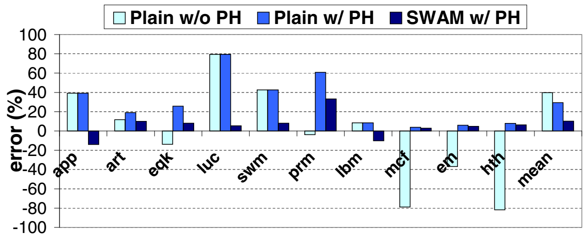
more overlapping data cache misses. Figure 11(b) illustrates the error of each modeling technique after compensation. From Figure 11(b) we observe that the arithmetic mean of the absolute error (mean) decreases from 39.7% to 29.3% when modeling pending cache hits. We also observe from Figure 11(b) that the arithmetic mean of the absolute error for SWAM profiling when pending data cache hits are modeled (SWAM w/ PH) is about 3.9 times lower than plain profiling when pending hits are not modeled (Plain w/o PH): the arithmetic mean of the absolute error decreases from 39.7% to 10.3%. Geometric mean of the absolute error decreases from 26.4% to 8.2%, and harmonic mean of the absolute error decreases from 15.3% to 6.9%. Accuracy also improves, and not just for "micro-benchmarks" [37]: In Figure 11(b), comparing "Plain w/o PH" to "SWAM w/ PH", we find that, on average, the arithmetic mean of the absolute error decreases from 31.6% to 9.1% for the five SPEC 2000 benchmarks *excluding* mcf.

5.2. Modeling Different Prefetching Techniques

In this section, we evaluate $CPI_{D\$miss}$ when modeling the three prefetching techniques mentioned in Section 4 (with unlimited MSHRs). Figure 12(a) compares the actual $CPI_{D\$miss}$ to the modeled one for the three prefeching methods. For each prefetching method, both the prediction when each pending hit is analyzed according to the algorithm described in Figure 5 (labeled "w/ PH") and the prediction when pending hits are treated as normal hits (labeled with "w/o PH") are shown. We use SWAM in both cases. When employing the algorithm in Figure 5, we apply SWAM as



(a) CPI due to D$miss



(b) Modeling error

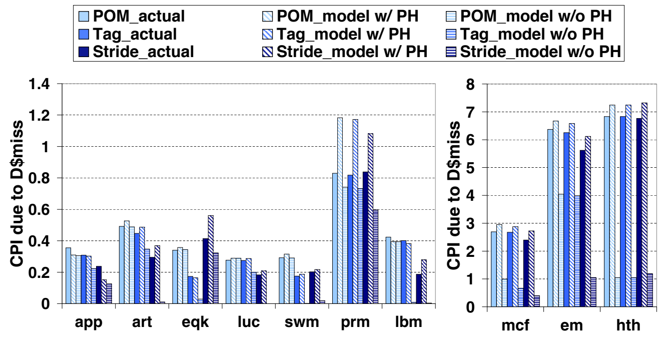
Figure 11. CPI due to D$miss and modeling error for different profiling techniques (Unlimited MSHRs)

follows: When we analyze the trace we let each profile step start with a miss or a hit due to a prefetch. The latter refers to a demand request whose data was brought into the data cache by a previous prefetch (we start with it since its latency may not be fully hidden and thus it may stall commit). Figure 12(b) shows the error of the model for each benchmark.

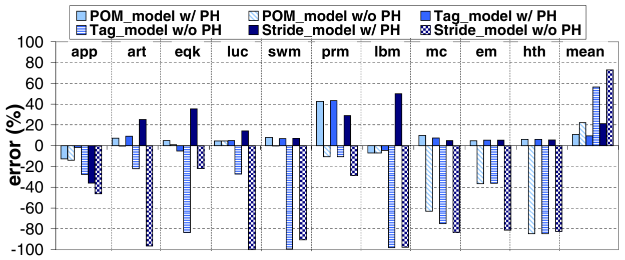
From Figure 12(b) we observe that if pending hits are not appropriately modeled (i.e., a pending hit is simply treated as a hit and not analyzed based upon the algorithm in Figure 5), the modeled $CPI_{D\$miss}$ always underestimates the actual $CPI_{D\$miss}$. The reason is that with a prefetching technique applied, a large fraction of the misses occurring when there is no prefetching become pending hits since prefetches generated by that prefetching technique cannot fully hide the memory access latency of those misses. By using the method of analyzing pending hits that we propose in Section 3.3 to model prefetching, the arithmetic mean of the absolute error decreases from 22.2% to 10.7% for prefetch-on-miss, from 56.4% to 9.4% for tagged prefetch technique, and from 72.9% to 21.3% for stride prefetch technique (i.e., the arithmetic mean of the absolute error decreases from 50.5% to 13.8% overall for the three data prefetching techniques modeled).

5.3. Modeling Limited Number of MSHRs

All of the results that we have seen thus far are for modeling a processor with an unlimited number of MSHRs. This section compares modeled $CPI_{D\$miss}$ when the number of available MSHRs is limited. Figure 13(a), (b), and (c) compare the modeled $CPI_{D\$miss}$ to the simulated results when the maximum number of MSHRs in a processor is sixteen, eight, and four, respectively. We show data for eight



(a) CPI due to D$miss with different prefetching techniques



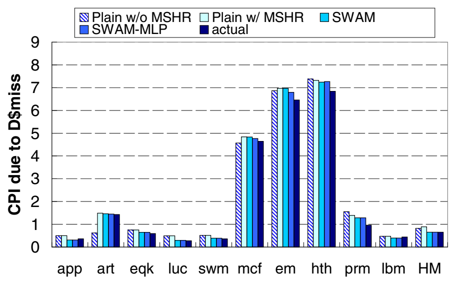
(b) Modeling error with different prefetching techniques

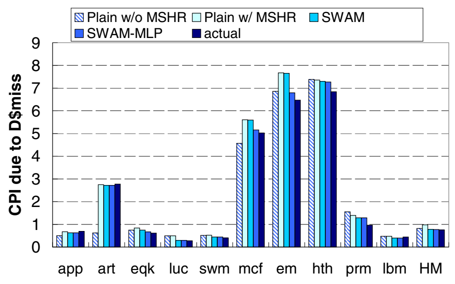
Figure 12. CPI due to D$miss and modeling error while prefetch-on-miss (POM), tagged prefetch (Tag), or stride prefetch (Stride) technique is applied.

MSHRs and four MSHRs since we note that Prescott has only eight MSHRs [4] and Williamette has only four MSHRs [15]. For each benchmark, the first bar (Plain w/o MSHR) shows the modeled $CPI_{D\$miss}$ from plain profiling (i.e., it is not aware that there are a limited number of MSHRs and always provides the same result for each benchmark) and the second bar (Plain w/ MSHR) shows the modeled $CPI_{D\$miss}$ from plain profiling with the technique of modeling a limited number of MSHRs (Section 3.4) included. The third and the fourth bar illustrates the modeled $CPI_{D\$miss}$ from SWAM (Section 3.5.1) and SWAM-MLP (Section 3.5.2), respectively. For these four profiling techniques, pending hits are modeled using the method described in Section 3.1. The modeling error based on the data in Figure 13(a)–(c) is illustrated in Figure 14(a)–(c).

SWAM-MLP is consistently better than SWAM. We observe that as the total number of MSHRs decreases, the advantage of SWAM-MLP over SWAM becomes significant, especially for eqk, mcf, em, and hth, for which it is more likely to have data dependence among cache misses thus affecting the size of the profiling window that SWAM-MLP chooses. SWAM decreases the arithmetic mean of the absolute error from 32.6% (Plain w/o MSHR) to 9.8%, from 32.4% to 12.8%, and from 35.8% to 23.2%, when the number of MSHRs is sixteen, eight, and four, respectively[3].

3. Geometric mean is reduced from 19.4% to 7.4%, from 21.5% to 9.7%, and from 21.8% to 10.9%; harmonic mean is reduced from 8.5% to 5.8%, from 14.5% to 7.0%, and from 10.2% to 5.1%


(a) $N_{MSHR} = 16$


(b) $N_{MSHR} = 8$

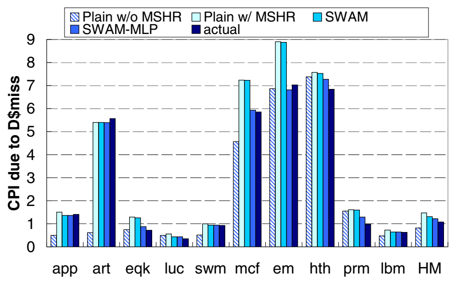
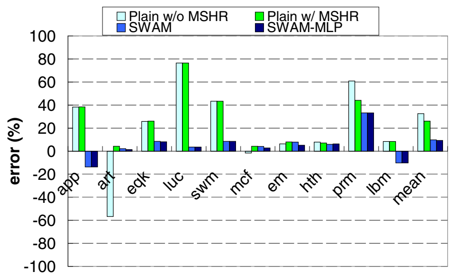
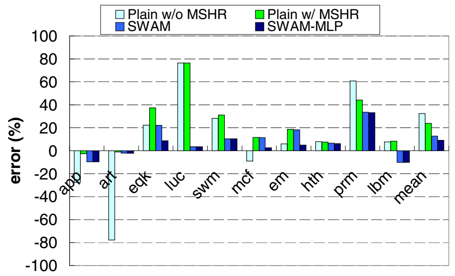

(c) $N_{MSHR} = 4$

Figure 13. CPI due to D$miss for $N_{MSHR} = 16$, $N_{MSHR} = 8$, and $N_{MSHR} = 4$.


(a) $N_{MSHR} = 16$


(b) $N_{MSHR} = 8$

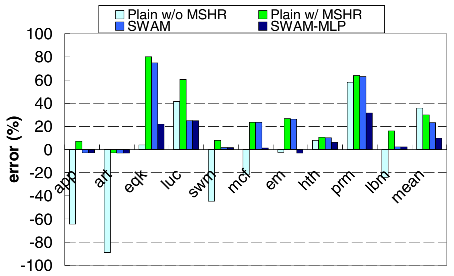

(c) $N_{MSHR} = 4$

Figure 14. Error of the modeled $CPI_{D\$miss}$ for $N_{MSHR} = 16$, $N_{MSHR} = 8$, and $N_{MSHR} = 4$.

SWAM-MLP further decreases the error to 9.3%, 9.2%, and 9.9%[4] (i.e., SWAM-MLP decreases the error of plain profiling (Plain w/o MSHR) from 33.6% to 9.5% when the number of MSHRs is limited). Note that accuracy improves not only for pointer-chasing benchmarks: For the SPEC 2000 benchmarks excluding mcf, average error reduces from 48.1% to 7.0% comparing Plain w/o MSHR to SWAM-MLP.

5.4. Putting It All Together

We also evaluated the combination of the techniques for modeling prefetching (Section 3.3) and SWAM-MLP to model the performance of the three prefetching methods with limited MSHRs. On average, the error of modeling prefetching is 15.2%, 17.7%, and 20.5%, when the number of MSHRs is sixteen, eight, and four, respectively (average of 17.8% across all three prefetch methods).

5.5. Speedup of the Hybrid Analytical Model

One of the most important advantages of the hybrid analytical model we present in this paper versus detailed simulations is its fast speed of analysis. On average our model is 150, 156, 170, and 229 times faster than the detailed simulator when the number of MSHRs is unlimited, sixteen, eight, and four, respectively, with a minimum speedup of 91×. Moreover, for estimating the performance impact of prefetching, on average our model is 184, 185, 215, and 327 times faster than the detailed simulator when the number of MSHRs is unlimited, sixteen, eight, and four, respectively, with a minimum speedup of 87×. These speedups were measured on a 2.33 GHz Intel Xeon E5345 processor.

4. Geometric mean of the absolute error is 6.5%, 6.7%, 5.2%, and harmonic mean of the absolute error is 4.6%, 5.2%, 3.3%, when the number of MSHRs is sixteen, eight, and four, respectively.

6. Related Work

There exist many analytical models proposed for superscalar microprocessors [24], [25], [26], [28]. A common limitation of early models is that they assume a perfect data cache. As the gap between memory and microprocessor speed increases, data cache misses must be properly modeled to achieve reasonable accuracy.

Agarwal et al. [1] present an analytical cache model estimating cache miss rate given cache parameters. However, in a superscalar, out-of-order execution microprocessor, the cache miss rate itself is not enough to predict the real performance of the program analyzed. Jacob et al. [16] propose an analytical memory model that applies a variational calculus approach to determine a memory hierarchy optimized for the average access time given a fixed hardware budget with an (implicit) assumption that memory level parallelism does not occur. Noonburg and Shen present a superscalar microprocessor Markov chain model [27] that does not model long memory latency. The first-order model proposed by Karkhanis and Smith [19] is the first to attempt to account for long latency data cache misses. Our analytical model improves the accuracy of our re-implementation of their technique described in Section 2 (based upon details available in the literature) by modeling pending data cache hits, and extends it to estimate the performance impact of data prefetching techniques and a limited number of outstanding cache misses.

Some earlier work has investigated the performance impact increasing the number of MSHRs using detailed simulations [13], [3]. The hybrid analytical model that we propose can be used to estimate the performance impact of a limited number of MSHRs without requiring a detailed simulator.

Concurrent with our work, Eyerman proposed an approach similar to SWAM described in Section 3.5.1, except that the profile window slides to begin with each successive long latency miss [11]. Reportedly, a pending hit compensation mechanism was used [22].

7. Conclusions

In this paper, we proposed and evaluated several improvements to an existing analytical performance model for superscalar processors. We showed the importance of properly modeling pending data cache hits and proposed a technique to accurately model their performance impact, while not requiring a performance simulator. We then extended this model to estimate the performance of a microprocessor when an arbitrary data prefetching method is applied. Moreover, we proposed a technique to quantify the impacts of a limited number of MSHRs. Overall, the techniques we introduced in this paper can reduce the modeling error of our baseline from 39.7% to 10.3% for a set of memory intensive benchmarks. The average error of our model is 13.8% when modeling several prefetching strategies and 9.5% when modeling a limited number of supported outstanding cache misses.

Acknowledgment

We would like to thank Tejas Karkhanis, Lieven Eeckhout, Ali Bakhoda, Wilson Fung, Johnny Kuan, Andrew Turner, Henry Wong, George Yuan, and the anonymous reviewers for their valuable comments on this paper. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] A. Agarwal, M. Horowitz, and J. Hennessy. An Analytical Cache Model. *TOCS*, 7(2):184–215, 1989.

[2] J.-L. Baer and T.-F. Chen. An Effective On-chip Preloading Scheme to Reduce Data Access Penalty. In *SC*, pages 176–186, 1991.

[3] S. Belayneh and D. R. Kaeli. A discussion on non-blocking/lockup-free caches. *SIGARCH Comput. Archit. News*, 24(3):18–25, 1996.

[4] D. Boggs, A. Baktha, J. Hawkins, D. T. Marr, J. A. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The Microarchitecture of the Intel® Pentium® 4 Processor on 90nm Technology. *Intel® Technology Journal*, 8(1), 2004.

[5] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[6] M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, June 1996.

[7] X. E. Chen and T. M. Aamodt. An improved analytical superscalar microprocessor memory model. In *4th Workshop on Modeling, Benchmarking and Simulation (MoBS 2008)*, pages 7–16, June 2008.

[8] C. K. Chow. On Optimization of Memory Hierarchies. *IBM J. Res. Dev.*, (2):194–203, 1974.

[9] A. Cristal, O. Santana, M. Valero, and J. F. Martínez. Toward Kilo-instruction Processors. *ACM TACO*, 1(4):389–417, 2004.

[10] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A Performance Model Framework. *IEEE Computer*, 35(2):68–76, 2002.

[11] S. Eyerman. *Analytical Performance Analysis and Modeling of Superscalar and Multi-threaded Processors*. PhD thesis, Ghent, 2008.

[12] S. Eyerman, L. Eeckhout, T. S. Karkhanis, and J. E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *ASPLOS-XII*, pages 175–184, 2006.

[13] K. I. Farkas, N. P. Jouppi, and P. Chow. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *HPCA-1*, page 78, 1995.

[14] J. D. Gindele. Buffer Block Prefetching Method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, 1977.

[15] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium® 4 Processor. *Intel® Technology Journal*, 5(1), 2001.

[16] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge. An Analytical Model for Designing Memory Hierarchies. *IEEE Trans. Computer*, 45(10):1180–1194, 1996.

[17] N. P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *ISCA-17*, pages 364–373, 1990.

[18] T. S. Karkhanis. *Automated Design of Application-specific Superscalar Processors*. PhD thesis, University of Wisconsin, 2006.

[19] T. S. Karkhanis and J. E. Smith. A First-order Superscalar Processor Model. In *ISCA-31*, pages 338–349, 2004.

[20] T. S. Karkhanis and J. E. Smith. Automated Design of Application Specific Superscalar Processors: An Analytical Approach. In *ISCA-34*, pages 402–411, 2007.

[21] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA-8*, pages 81–87, 1981.

[22] Lieven Eeckhout. Personal Communication. Sept, 2008.

[23] J. E. MacDonald and K. L. Sigworth. Storage Hierarchy Optimization Procedure. *IBM J. Res. Dev.*, 19(2):133–140, 1975.

[24] P. Michaud, A. Seznec, and S. Jourdan. Exploring Instruction-Fetch Bandwidth Requirement in Wide-issue Superscalar Processors. In *PACT*, pages 2–10, 1999.

[25] P. Michaud, A. Seznec, and S. Jourdan. An Exploration of Instruction Fetch Requirement in Out-of-Order Superscalar Processors. *Int'l Journal of Parallel Programming*, 29(1):35–38, 2001.

[26] D. B. Noonburg and J. P. Shen. Theoretical Modeling of Superscalar Processor Performance. In *MICRO-27*, pages 52–62, 1994.

[27] D. B. Noonburg and J. P. Shen. A Framework for Statistical Modeling of Superscalar Processor Performance. In *HPCA-3*, pages 298–309, 1997.

[28] D. J. Ofelt. *Efficient Performance Prediction for Modern Microprocessors*. PhD thesis, Stanford University, 1999.

[29] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *ASPLOS-X*, pages 45–57, 2002.

[30] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[31] Standard Performance Evaluation Corporation. SPEC CPU2000 benchmarks. http://www.spec.org.

[32] Tejas S. Karkhanis. Personal Communication. Sept., 2008.

[33] J. Tuck, L. Ceze, and J. Torrellas. Scalable Cache Miss Handling for High Memory-Level Parallelism. In *MICRO-39*, pages 409–422, 2006.

[34] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural Exploration with Liberty. In *MICRO-35*, pages 271–282, 2002.

[35] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *ISCA-30*, pages 84–97, 2003.

[36] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. The Future of Simulation: A Field of Dreams. *Computer*, 39(11):22–29, 2006.

[37] C. B. Zilles. Benchmark health considered harmful. *SIGARCH Comput. Archit. News*, 29(3):4–5, 2001.