Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching

Jih-Kwon Peir
CISE Department
University of Florida
peir@cise.ufl.edu

Shih-Chang Lai
ECE Department
Oregon State University
laish@ece.orst.edu

Shih-Lien Lu
Jared Stark
Konrad Lai
Microprocessor Research
Intel Labs
shih-lien.l.lu@intel.com

ABSTRACT

A processor must know a load instruction's latency to schedule the load's dependent instructions at the correct time. Unfortunately, modern processors do not know this latency until well after the dependent instructions should have been scheduled to avoid pipeline bubbles between themselves and the load. One solution to this problem is to predict the load's latency, by predicting whether the load will hit or miss in the data cache. Existing cache hit/miss predictors, however, can only correctly predict about 50% of cache misses.

This paper introduces a new hit/miss predictor that uses a Bloom Filter to identify cache misses early in the pipeline. This early identification of cache misses allows the processor to more accurately schedule instructions that are dependent on loads and to more precisely prefetch data into the cache. Simulations using a modified SimpleScalar model show that the proposed Bloom Filter is nearly perfect, with a prediction accuracy greater than 99% for the SPECint2000 benchmarks. IPC (Instructions Per Cycle) performance improved by 19% over a processor that delayed the scheduling of instructions dependent on a load until the load latency was known, and by 6% and 7% over a processor that always predicted a load would hit the cache and with a counter-based hit/miss predictor respectively. This IPC reaches 99.7% of the IPC of a processor with perfect scheduling.

Categories and Subject Descriptors

C.1.1 [**Processor Architectures**]: Single Data Stream Architectures

General Terms

Algorithm, Design, Performance

Keywords

Bloom Filter, Data Cache, Data Prefetching, Instruction Scheduling, Data Speculation

ICS'02, June 22-26, 2002, New York, New York, USA.

1. INTRODUCTION

To achieve the highest performance, a processor must execute a pair of dependent instructions with no intervening pipeline bubbles. It must arrange for—or schedule—the dependent instruction to begin execution immediately after the instruction it depends on (i. e., the parent instruction) completes execution. Accomplishing this requires knowing the latency of the parent.

Unfortunately, a modern processor schedules an instruction well before it executes, and the latency of some instructions can only be determined by their execution. For example, the latency of a load depends on where in the cache/memory hierarchy its data exists, and can only be determined by executing the load and querying the caches. At the time the load is scheduled, its latency is unknown. At the time its *dependents* should be scheduled, its latency may *still* be unknown. Hence, the timely scheduling of the instructions that are dependent on a load is a problem in modern processors.

The Intel Pentium 4 illustrates this problem. On an Intel Pentium 4 [6, 7], a load is scheduled 7 cycles before it begins execution. Its execution (load-use) latency is 2 cycles. At the time a load is scheduled, its execution will not begin for another 7 cycles. Two cycles after the load is scheduled, if the load will hit the (first-level) cache, its dependent instructions must be scheduled to avoid pipeline bubbles. However, two cycles after the load is scheduled, the load has not yet even started executing, so its cache hit/miss status is unknown. A similar situation exists in the Compaq Alpha 21264 [9]. A load is scheduled 2 cycles before it begins execution, and its execution latency is 3 cycles. If the load will hit the (first-level) cache, its dependents must be scheduled 3 cycles after it has been scheduled to avoid pipeline bubbles. However, the load's cache hit/miss status is still unknown 3 cycles after it has been scheduled.

One possible solution to this problem is to schedule the dependents of a load only after the latency of the load is known. The processor delays the scheduling of the dependents until it knows the load hit the cache. This effectively increases the load's latency to the amount of time between when the load is scheduled and when its cache hit/miss status is known. This solution introduces bubbles into the pipeline, and can devastate processor performance. Our simulations show that a processor using this solution drops 17% of its performance (in Instructions Per Cycle [IPC]) compared to an ideal processor that uses an oracle to perfectly predict

load latencies and perfectly schedule their dependents.

A better solution—and the solution that is the focus of this work—is to use *data speculation*. The processor speculates that a load will hit the cache (a good assumption given cache hits rates are generally over 90%), and schedules its dependents accordingly. If the load hits, all is well. If the load misses, any dependents that have been scheduled will not receive the load's result before they begin execution. All these instructions have been erroneously scheduled, and will need to be rescheduled.

Recovery must occur whenever instructions are erroneously scheduled due to data (mis)speculation. Although misspeculation is rare, the overall penalty for all mis-speculations may be high, as the cost of each recovery can be high. If the processor only rescheduled those instructions that are (directly or indirectly) dependent on the load, the cost would be low. However, such a recovery mechanism is expensive to implement. The recovery mechanism for the Compaq Alpha 21264 simply reschedules all instructions scheduled since the offending load was scheduled, whether they are dependent or not. Although it's cheaper to implement, the recovery cost can be high with this mechanism due to the rescheduling and re-execution of the independent instructions. Regardless of which recovery mechanism is implemented, as processor pipelines grow deeper and issue widths widen, the number of erroneously scheduled instructions will increase, and recovery costs will climb.

To reduce the penalty due to data mis-speculations, the processor can predict whether the load will hit the cache, instead of just speculating that the load will always hit. The load's dependents are then scheduled according to the prediction. As an example of a cache hit/miss predictor, the Compaq Alpha 21264 uses the most significant bit of a 4-bit saturating counter as the load's hit/miss prediction. The counter is incremented by one every time a load hits, and decremented by two every time a load misses. Unfortunately, even with 2-level predictors [15], only about 50% of the cache misses can be correctly predicted.

In this paper, we describe a new approach to hit/miss prediction that is very accurate and space (and hence power) efficient compared to existing approaches. This approach uses a *Bloom Filter (BF)*, which is a probabilistic algorithm to quickly test membership in a large set using hash functions into an array of bits [2]. We investigate two variants of this approach: the first is based on *partitioned-address* matching, and the second is based on *partial-address* matching. Experimental results show that, for modest-sized predictors, Bloom Filters outperform predictors that used a table of saturating counters indexed by load PC. These *table-based* predictors operate just like the predictor for the Compaq Alpha 21264, except they have multiple counters instead of just one. As an example, for an 8K-bit predictor, the Bloom Filter mispredicts 0.4% of all loads, whereas the table-based predictor mispredicts 8% of all loads. This translates to an 7% improvement in IPC over the table-based predictor. Compared to a machine with a perfect predictor, a machine with a Bloom Filters has 99.7% of its IPC.

The remainder of the paper is organized as follows: The next section explains data speculation fundamentals and related work. Section 3 explains BFs and how they can be used as hit/miss predictors. Section 4 describes how the SimpleScalar microarchitecture [3] must be modified to support data speculation using a BF as a hit/miss predictor. Section 5 evaluates the performance of BFs, reporting their accuracy as hit/miss predictors and the performance benefit (in IPC) they can provide. Finally, Section 6 concludes.

2. DATA SPECULATION

2.1 The Fundamentals

To facilitate the presentation and discussion, we consider a baseline pipeline model that is similar to the Compaq Alpha 21264 [9]. In the baseline model, the front-end pipeline stages are: instruction fetch and decode/rename. After decode/rename, the ALU instructions go through the back-end stages: schedule, register read, execute, writeback, and commit. Additional stages are required for executing a load. After decode/rename, loads go through schedule, register read, address generation, two cache access cycles, an additional cycle for hit/miss determination (data access before hit/miss using way prediction [4]), writeback, and commit. Thus, there are a total of 7 and 10 cycles for ALU and load instructions, respectively.

Figure 1 shows the problem in scheduling the instructions that are dependent on a load. For simplicity, the front-end stages are omitted. In this example, the *add* instruction consumes the data produced by the *load* instruction. After the *load* is scheduled, it takes 5 cycles to resolve the hit/miss. However, the dependent *add* must be scheduled the third cycle after the load is scheduled to achieve the minimum 3-cycle load-use latency and allow back-to-back execution of these two dependent instructions. If the processor speculatively schedules the *add* assuming the *load* will hit the cache, the *add* will get incorrect data if *load* actually misses the cache. In this case, the *add* along with any other dependent instructions scheduled within the illustrated 3-cycle speculative window must be canceled and rescheduled.

To show the performance potential of using data speculation for scheduling instructions that are dependent on loads, we simulated the SPECint2000 benchmarks. We compare two scheduling techniques. The first is a *no-speculation* scheme: the dependents are delayed until the hit/miss of the parent load is known. The second uses a *perfect* hit/miss predictor that knows the hit/miss of a load in time to (perfectly) schedule its dependents to achieve minimum load latency. The performance gap (in IPC) between these two extremes shows the performance potential of speculatively scheduling the dependents of loads. Figure 2 shows the results. In these simulations, we modified the SimpleScalar out-of-order pipeline to match our baseline model; and doubled the default SimpleScalar issue width to 8, scaling the other parameters accordingly. A more detailed description of the simulation model is given in Section 5. On average, the IPC for perfect scheduling is 17% higher than the IPC for the no-speculation scheme. Thus, the main focus of this paper is to recover this 17% performance gap, by using mechanisms for efficient load data speculation.

2.2 Related Work

The Compaq Alpha 21264 uses a mini-restart mechanism to cancel and reschedule all instructions scheduled since a mis-speculated load was scheduled [9]. While this mini-restart is less costly than restarting the entire processor pipeline, it is still expensive to reschedule (and re-execute) both the dependent and the independent instructions. To alleviate this problem, the Compaq Alpha 21264 uses the

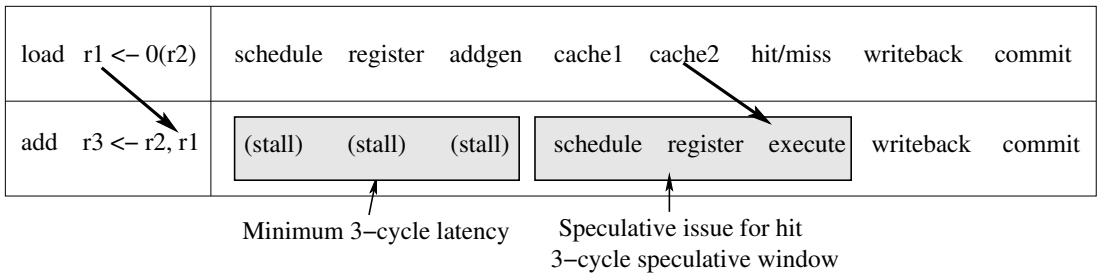

Figure 1: Example of Data Speculation for a Load

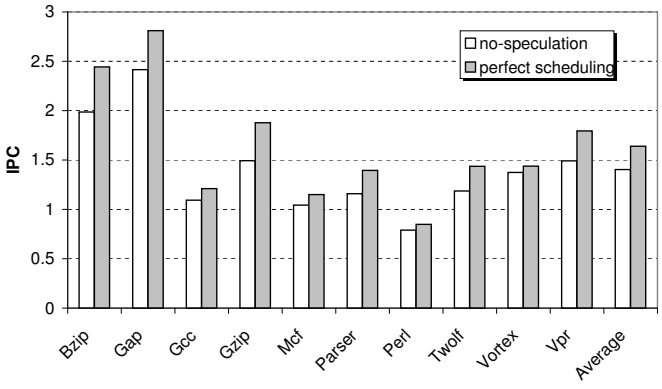

Figure 2: No-Speculation vs. Perfect Scheduling

most significant bit of a 4-bit saturating counter as the load's hit/miss prediction. The counter is incremented by one every time a load hits, and decremented by two every time a load misses. The load's dependents are scheduled according to the prediction. If the prediction is wrong, either the load was predicted to miss and it hit, in which case the execution of the dependents will be unnecessarily delayed; or the load was predicted to hit and it missed, in which case dependents may have been erroneously scheduled and will need to be rescheduled.

Yoaz et al. [15] used 2-level local predictors, 2-level global predictors, and hybrid predictors for cache hit/miss prediction. Their results show that these predictors only correctly identify half of the misses (for SPECint95), leaving the other half predicted as hits. Furthermore, they incorrectly identify a small percentage of the hits as being misses.

The MIPS R10000 speculatively issues instructions that are dependent on a load and reschedules them if the load misses the cache [14].

The Intel Pentium 4 achieves a minimum 2-cycle load-use latency by leveraging the fact that most accesses hit the first-level (L_1) cache. The scheduler issues the dependent micro-operations (called *uops*) before the parent load has finished executing [6, 7]. In most cases, the scheduler assumes the load will hit the L_1 cache. A '*replay*' mechanism is used to handle the case where the load misses the L_1 cache. The replay logic keeps track of the dependent *uops* of each speculative load. When a load misses, all its dependent *uops* are re-executed with the correct data when that data becomes available.

Morancho, Llabería, and Olivé describe a recovery mechanism for load latency misprediction [11]. A recovery buffer retains all speculatively scheduled instructions. After a latency misprediction, the load's dependent instructions can be re-scheduled directly from the recovery buffer as soon as the load data becomes available. The recovery buffer allows the processor to remove instruction from the scheduler early, providing more space for other instructions.

3. BLOOM FILTERS

A *Bloom Filter (BF)* is a probabilistic algorithm to quickly test membership in a large set using multiple hash functions into an array of bits [2]. A BF quickly filters (i. e., identifies) non-members without querying the large set by exploiting the fact that a small percentage of erroneous classifications can be tolerated. When a BF identifies a non-member, it is *guaranteed* to not belong to the large set. When a BF identifies a member, however, it is *not guaranteed* to belong to the large set. To put it more simply, the result of the membership test is either: it is definitely not a member, or, it is probably a member. In this paper, we consider two variants of the BF for filtering cache misses: one based on *partitioned-address* matching, and the other based on *partial-address* matching. To simplify our discussion, we first assume both the BF and the cache use physical addresses. Afterwards, we will describe using virtual addresses.

3.1 Partitioned-Address Bloom Filter

Consider a cache line address with n bits (ignoring the offset bits). A large, direct-mapped array of 2^n bits is required to precisely record whether each cache line address is in the cache. To reduce the space and allow a quick access, a *partitioned-address* BF can be constructed. Instead of using the entire line address, the address can be split into m partitions, with each partition using its own array of bits. The result is m sub-arrays with $2^{n/m}$ bits, each of which records the membership of the respective address partitions of lines stored in the cache. A cache miss is identified when one or more of the address partitions for the address of a requested line does not belong to the respective address partition of any line in the cache. A *filter error* is encountered when a cache miss cannot be identified. This situation happens when the line is not in the cache, but all m partitions of the line's address match address partitions of other cache lines. The *filter rate* represents the percentage of cache misses that can be identified.

Figure 3 illustrates how the *partitioned-address* BF works. A load address is partitioned, in this example, into 4 equally divided groups, *A1, A2, A3*, and *A4*. Each of the four address partitions is used to index separate BF arrays, *BF1, BF2, BF3*, and *BF4*, respectively. Each entry in the BF

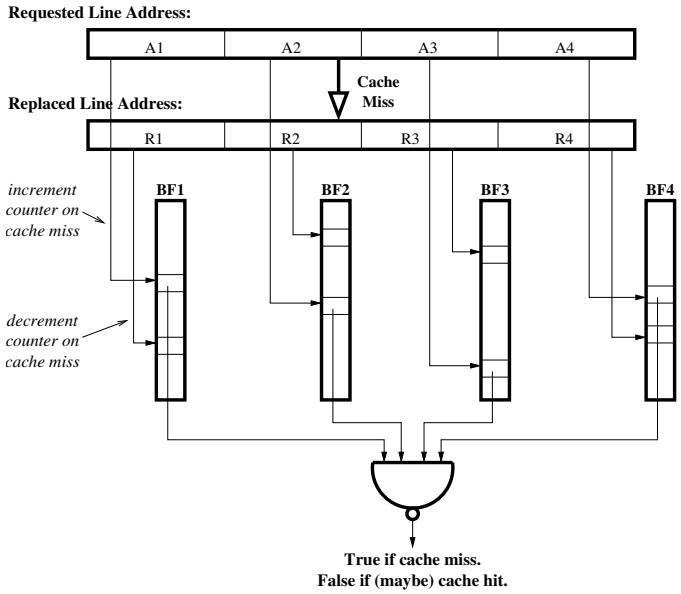

Figure 3: Partitioned-Address Bloom Filter for Cache Miss Detection

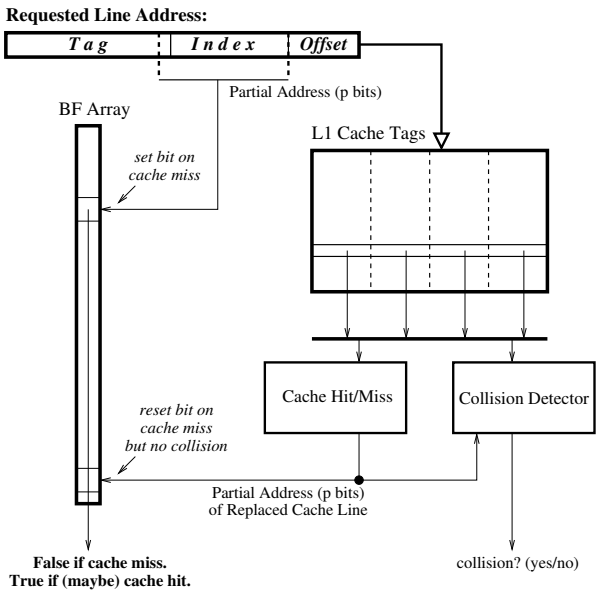

Figure 4: Partial-Address Bloom Filter for Cache Miss Detection

arrays contains the information of whether the address partition belongs to the corresponding address partition of any line in the cache. If any of the 4 BF arrays indicates one of the address partitions is absent from the cache, the requested line is not in the cache. Otherwise, the requested line is probably in the cache, but it's not guaranteed to be.

Given the fact that a single address partition can exist for multiple lines in the cache, the primary difficulty of the *partitioned-address* BF is to maintain the correct membership information. When a line is removed from the cache, an exhaustive search is necessary to check if the address partitions for the address of the removed line still exist for any of the remaining lines. To avoid such a search, each entry in the BF array contains a reference counter that keeps track of the number of cache lines with the entry's corresponding address partition. When a cache miss occurs, each counter for the address partitions for the address of the newly-requested line is incremented, while the counters for the address partitions for the address of the replaced line are decremented. A zero count indicates the corresponding address partition does not belong to any line in the cache. Although accurate, this counter technique requires extra space in the BF arrays for the counters along with adders to handle the updates. A similar idea has been considered to reduce the number of comparators for a set-associative cache [8] and to filter cache-coherence traffic in a multiprocessor environment [12].

3.2 Partial-Address Bloom Filter

The *partial-address* BF uses the least-significant bits of the line address to index a small array of bits. Each bit indicates whether the partial address matches any corresponding partial address of a line in the cache. The array size is reduced to 2^p bits, where p is the number of partial address bits. A *filter error* occurs when the partial address of the requested line matches the partial address of an existing cache line, but the other portion of the line address does not match. We call such cases *collisions*. The least-significant bits are selected rather than more-significant bits to reduce the chance of collisions. Due to memory reference locality, the more-significant line address bits tend to change less frequently. With a sufficient number of low-order partial address bits to represent cache line addresses, collisions are rare [10].

The design of a *partial-address* BF is illustrated in Figure 4. A *BF array* with 2^p bits indicates whether the corresponding partial address matches that of any cache line. The BF array is updated to reflect any cache content change. When a cache miss occurs, except for the caveat described in the paragraph below, the entry in the BF array for the replaced line is *reset* to indicate that the line with that partial address is no longer in the cache. Then, the entry for the requested line is *set* to indicate that a line with that partial address now exists in the cache.

If the partial address is wider than the cache index, when two cache lines share the same partial address, they must be in the same set in a set-associative cache. The BF array indicates which partial addresses exist in the cache, so if one of these lines is replaced, the BF entry for the replaced line should *not* be reset, since the partial address still exists for the line that was not replaced. When a cache line is replaced, the *collision detector* checks the remaining cache lines in the same set as the replaced line to see if any of them have the same partial address as the replaced line. If any do have the same partial address, the BF entry is not reset. Otherwise, the entry is reset. The collision detection is done in parallel with the cache hit/miss detection. The BF array is updated on the detection of a cache miss.

3.3 Bloom Filters using Virtual Addresses

The hit/miss prediction for a load must be done before the scheduling of its dependents. If the physical address is not available in time to perform the prediction, the virtual address must be used. When a virtual address is used to access a BF, it is called a *virtual-address BF*. If the cache is virtually indexed and tagged, the virtual-address BF op-

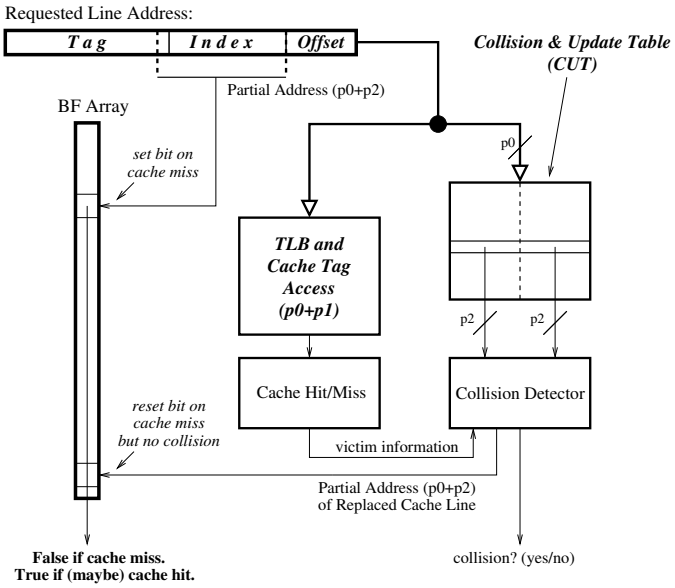

Figure 5: Partial-Virtual-Address Bloom Filter for Cache Miss Detection

erates analogously to the BF and cache that both use only physical addresses. However, if the cache is either virtually-indexed physically-tagged or physically-indexed physically-tagged, the BF array update for the virtual-address BF must be modified. In this section, we describe these modifications.

With virtual addresses, two virtual addresses can map to the same physical address, causing an *address synonym*. With a virtual-address BF, the BF might identify the first address as missing the cache, even though the line is in the cache set identified by the second address. That is, the BF identifies a load as missing the cache even though it hits. This situation can arise regardless of whether the cache is physically or virtually indexed. In this situation, the processor simply delays scheduling the load's dependent instructions. Since cache hits by synonyms are rare, the performance loss caused by the delayed scheduling is minimal. In fact, for some virtually-indexed caches, the load-use latency for a synonym hit is longer than for a non-synonym hit. For scheduling, the processor may initially treat the synonym hit as a cache miss, in which case the BF should identify the synonym hit as a cache miss anyway.

A more essential issue is correctly updating the BF array on cache misses. Let's first focus on the *partial-address* BF shown in Figure 5. To simplify our discussion, assume the cache is physically indexed and tagged with p_0+p_1 index bits, where p_0 bits are within the page offset and p_1 bits are beyond the offset. During a cache access, p_1 bits are translated. Also assume p_0+p_2 partial virtual address bits are used to access the BF, where p_2 bits are beyond the page offset. To correctly update the BF array, the p_2 bits of each cache line are stored in a *Collision and Update Table (CUT)*. When a line is replaced, its p_2 bits are read from the CUT. These p_2 bits are then combined with the requested line's p_0 bits to update the BF array.

The CUT is organized as a two-dimensional array and indexed by the p_0 bits. During each cache access, the set of p_2 bits indexed by p_0 are read from the CUT. If a cache miss is detected, the p_2 bits of the victim (e. g., LRU) line in the accessed cache set are compared to the p_2 bits for the other lines in that CUT set. If the victim's p_2 bits don't match any other line's p_2 bits, there is no collision, and the victim's p_2 bits are used along with the p_0 bits to reset the BF array to indicate that the line with the p_0+p_2 partial address is no longer in the cache. If the victim's p_2 bits do match another line's p_2 bits, the victim and the other line share the same partial address, and there is a collision. In this case, the BF entry for the victim line is left alone. Then, the BF entry for the requested line is set using the partial virtual address of the requested line. Note that when the cache is virtually-indexed physically-tagged, all the cache index bits are used to access the CUT. In this case, only the partial address bits beyond the virtual cache index bits need to be saved in the CUT and compared for collision detection.

Handling a virtual *partitioned-address* BF is straightforward. Virtual address tags must be stored in the cache tag array along with the physical tags. When a line is replaced, the replaced line's virtual address tag is used to update the counter in each partitioned BF.

For the remainder of the paper, we will assume virtual-address BFs. The virtual address needed to access the BF is available after the address generation cycle. Due to its rarity, we will omit discussions of synonym hits. If fact, for our benchmarks there are no synonyms.

4. THE MICROARCHITECTURE

In our baseline model, ALU instructions require a minimum of 7 cycles: instruction fetch (IFE), decode/rename (DEC), schedule (SCH), register read (REG), execute (EXE), writeback (WRB), and commit (CMT). Loads extend the execute stage to 4 cycles: address generation (AGN), two cache access cycles (CA1, CA2), and hit/miss determination (H/M). Assuming a load hits the L_1 cache, there is a 3-cycle speculative window in which the load's dependents and their children are scheduled. When a miss occurs, all of the dependent instructions and their children scheduled in these 3 cycles must be canceled and re-executed using the correct data when it becomes available.

4.1 Predictor Timing and Mini-Restart

If data cache misses can be predicted early enough and accurately enough, the processor's scheduler can avoid inserting pipeline bubbles between a load and its dependent instructions. To be effective, the load's cache hit/miss prediction must be done before its dependents must be scheduled. Thus, there are two basic issues: (1) when, and (2) how fast the hit/miss prediction can be performed. Hit/miss predictors that use saturating counters, like the one used by the Compaq Alpha 21264, can access the counter at the beginning of the pipeline. Since our pipeline has a minimum 3-cycle load latency, the prediction is available before any of the load's dependents need to be scheduled. If a miss is predicted, the dependents are blocked from scheduling until either the data comes back from the outer levels of the memory hierarchy or the prediction is found to be incorrect.

The proposed Bloom Filter approach, on the other hand, requires the load address to accurately identify (filter) misses. This filtering can only be performed after the load address is calculated in the address generation cycle. As shown in Figure 1, the load's dependent instructions must be scheduled the cycle after the load's address generation to avoid

pipeline bubbles. By using a small BF, cache misses can be filtered in the cycle after the address generation, which is two cycles before the hit/miss determination. However, it is still one cycle too late to prevent the dependent instructions from scheduling.

To reap the prediction accuracy benefit provided by the BF, the load's dependents are always aggressively scheduled assuming a cache hit. At the end of the cycle the dependents are scheduled, the parent load has finished accessing the BF. If a miss is identified, the dependents are canceled and recovered in the next cycle. Since there is only a single-cycle speculative window, a precise recovery of the load's dependents may be feasible without excessive hardware complexity. This could be achieved by preventing the load's scheduled dependents from broadcasting their tags to their dependents, inhibiting the wakeup of their dependents. All independent instructions scheduled during this single-cycle window would be allowed to continue.

The Compaq Alpha 21264 has a similar precise recovery scheme to handle the dependents of floating-point loads. It also has a 3-cycle minimum load-use latency (1 for address generation and 2 for cache access). The cache hit/miss detection is done in the second cache access cycle, so the speculative window is only 2 cycles. The dependents of floating-point loads are always delayed from scheduling by one cycle. Consequently, the two-cycle speculative window for integer loads is reduced to a one-cycle window for floating-point loads. When the dependents of a floating-point load are being scheduled, the hit/miss detection is being performed in the same cycle. If a miss is detected, the dependents in this one-cycle window are precisely recovered in the next cycle [1]. This recovery should incur minimum penalty, as these dependents have to wait for the load data to return from the outer levels of the memory hierarchy anyway. The only potential adverse impact is that these dependents unnecessarily occupy functional units.

If a load is predicted to hit the cache, and it is later identified by the normal cache access as a miss, all dependent instructions scheduled during the entire 3-cycle speculative window have been or will be incorrectly executed. It is not sufficient to only re-schedule those instructions that directly depend on the load. Descendants of those dependent instructions may have been scheduled, and also need to be canceled and re-scheduled. A simple and workable scheme is to squash all instructions scheduled during the 3-cycle speculative window, as is done by the Compaq Alpha 21264. This simple recovery scheme reduces the hardware complexity needed to track all the dependencies and speculative states. However, both dependent and independent instructions scheduled in this 3-cycle window are canceled and re-scheduled. Independent instructions are rescheduled the cycle after the misprediction is detected. Dependent instructions are rescheduled according to the correct completion time of the load, which in most cases is determined by the level 2 cache access time.

Figure 6 illustrates the recovery mechanism for data misspeculation. Again, the first two pipeline stages are omitted to simplify the figure. When the BF identifies a load as missing the cache, only those dependent instructions scheduled in the same cycle are canceled. The cancellation does not affect any independent instructions scheduled in this cycle, as shown in Part (a) of the figure. When a miss cannot be correctly filtered by the BF, and the miss is detected during

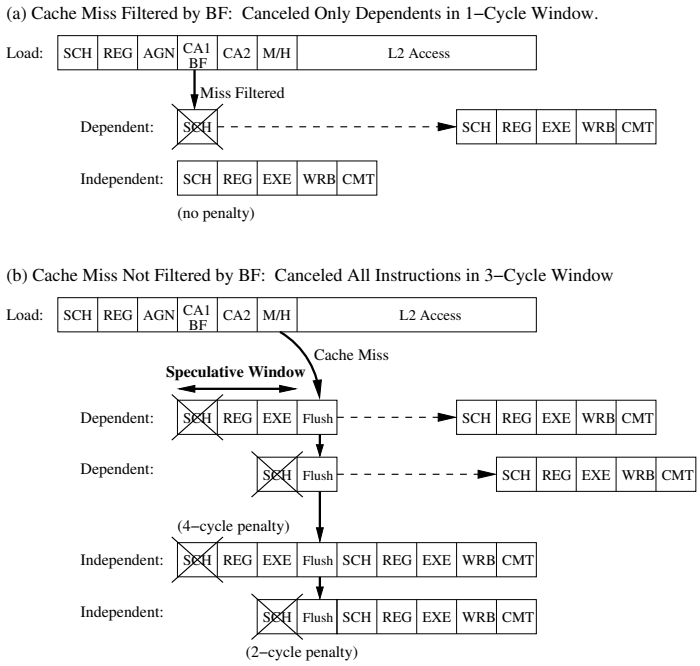

Figure 6: Recovery and Re-execution for: (a) Cache Miss Filtered by BF, and (b) Cache Miss Not Filtered by BF

the regular cache access, all of the instructions that were scheduled during the 3-cycle speculative window are canceled. Cancellation and re-execution involves resetting the canceled instructions' processor state. We assume it takes a separate *flush* cycle before the canceled instructions can be re-scheduled. Although independent instructions can be re-scheduled right away, they encounter a minimum 2–4 cycle penalty depending on where they reside in the speculative window. For example, a 4-cycle penalty occurs for those instructions that were scheduled in the first of the three speculative cycles as marked in Part (b) of the figure. Other factors such as data and resource dependencies may further increase the number of penalty cycles.

4.2 Prefetching and Memory Dependencies

Compared with other cache hit/miss predictors, the BF is unique in that misses that are identified must not exist in the L_1 cache. Therefore, once a miss is identified, it is safe to issue a miss request to the second-level cache (L_2). In our pipeline model, this effectively reduces the L_2 cache and memory latencies by two cycles. Although other predictors also allow early L_2 cache access, they may incorrectly identify some L_1 cache hits as being misses, introducing extra penalties and complexity into the processor.

In our simulator, a Load-Store Queue (LSQ) is used to detect and enforce memory dependencies. It also allows loads to fetch data directly from an aliasing store in the LSQ without accessing the cache. The memory dependence is detected after the address of the load is generated (AGN). The load is forced to wait if the address of any potentially aliasing store in the LSQ is unknown. In our processor model, we assume this memory dependence detection is done early and accurately in the pipeline, which we model with a perfect memory dependence predictor. This allows the scheduling

Fetch/Decode/Issue Width	8
Branch Predictor	8K-entry 4-way BTB 16-bit Gshare
RUU/LSQ Size	64
L_1 Inst/Data	16KB 4-way
L_2 Cache	4MB 8-way
Access Latency: L_1/L_2/Mem	2/7/100
Memory Ports	4
Integer Add/Mult ALU	4/2
Floating-P Add/Mult ALU	4/2

Table 1: Simulation parameters

Prediction Method	Array Size (in bits)
Partition-3	15360
Partition-4	4480
Partial-1x	512
Partial-4x	2048
Partial-16x	8192
Partial-64x	32768
Always-hit	0
Counter-1	4
Counter-128	512
Counter-512	2048
Counter-2048	8192
Counter-8192	32768

Table 2: Cache hit/miss predictors and their required storage

of the cache access to be inhibited if the load depends on a store in the LSQ. For the SPECint2000 benchmarks we tested, half of them have a very low percentage (1–3%) of loads which fetch data from the LSQ. However, the other half have higher percentages, indicating the importance of knowing memory dependencies before scheduling a cache access. If memory dependence detection can not be done early enough to avoid pipeline bubbles, cache accesses can be speculatively scheduled before memory dependencies are known. The speculative cache access—and any instructions dependent on the load that were scheduled/executed—are canceled and potentially re-scheduled and re-executed if a memory dependence is later detected. The BF may also be used in conjunction with a memory dependence predictor [5] to provide more accurate scheduling of loads and their dependents. Further discussion in this direction is out of the scope of this paper.

5. PERFORMANCE EVALUATION

To evaluate the potential performance benefit of using a BF as a cache hit/miss predictor, we modified SimpleScalar to support BFs and other hit/miss predictors, and then ran most of the SPECint2000 benchmarks through the simulator. Our evaluation will compare the proposed BF technique to the other hit/miss predictors. Our simulated machine is a general-purpose out-of-order processor capable of issuing 8 instructions per cycle. The branch predictor consists of an 8K entry 4-way set-associative BTB and a 16-bit Gshare predictor. As described in Section 4, the pipeline is a minimum of 7 stages for ALU instructions and 10 stages for loads. A small 64-entry reorder buffer (called the RUU in SimpleScalar) was used for our studies as a larger instruction window may affect the cycle time. We modeled a detailed memory hierarchy, with the size and latency at each level reflecting current trends. We slightly modified the original SimpleScalar L_1 cache: instead of updating the cache tag array when a miss is detected, the tag array is updated when the missed data comes back from the outer levels of the memory hierarchy. This modification more accurately simulates cache misses, since the LRU line is not removed until the new data comes in.

Table 1 summarizes the simulation parameters. We simulated 10 of the SPECint2000 benchmarks using the reference input file. For each benchmark, we skip the first 500 million instructions and collect result statistics on the next 500 million instructions. We collect both prediction accuracy and IPC for the BFs and other cache hit/miss predictors.

We simulated different sizes of the two BF variants. For the *partitioned-address* BF, we simulated three (*Partition-3*) and four (*Partition-4*) equal partitions of the line address (27 bits). Each entry in the BF array maintains a counter capable of counting the entire number of L_1 cache lines. To avoid overflow in our simulations, each counter was 10 bits. For the *partial-address* BF, the BF array size ranges from having only one entry per L_1 cache line (*Partial-1x*) all the way up to having 64 entries per L_1 cache line (*Partial-64x*). In our baseline model, the L_1 data cache is 16KB with a 32-byte line size. We also perform sensitivity studies on cache size.

We also evaluate two previously proposed hit/miss predictors and some simple extensions to them. The first is to always predict cache hit (*Always-hit*). This method does not require any prediction table. The second is the predictor in the Compaq Alpha 21264, which uses a single 4-bit saturating counter (*Counter-1*). We also evaluate using an untagged table of 4-bit saturating counters, indexed by the PC of the load. We vary the size of the table from 128 counters (*Counter-128*) to 8192 counters (*Counter-8192*). Since each counter is 4 bits, the total size of the *Counter-128* predictor matches the size of the *Partial-1x* predictor. For the counter-based predictors, the prediction is performed in the instruction fetch cycle, and the counters are updated after the cache hit/miss status is known. Table 2 summarizes the predictors we simulated and the amount of storage they require. Note that besides the predictor array tables, other logic such as adders and comparators are required to perform predictions.

5.1 Prediction Accuracy

Figure 7 plots the filtering rates of the BFs. Recall the filtering rate is the percentage of misses identified (filtered) by the BF. In general, *partitioned-address* BFs perform poorly. The average filtering rate of *Partition-3* is only about 45%. *Partition-4* (not shown) has a dismal 5% average rate. Due to memory reference locality, the lowest partition of an address provides the most information, with the upper partitions providing almost no information. This is evident when comparing *Partition-3* to *Partial-1x*. The lowest partition of *Partition-3* uses the same 9 bits as *Partial-1x*. Yet the upper two partitions used by *Partition-3* only help to identify an additional 3% of misses. For *partial-address* BFs, the filtering rate improves dramatically as the size of the BF array

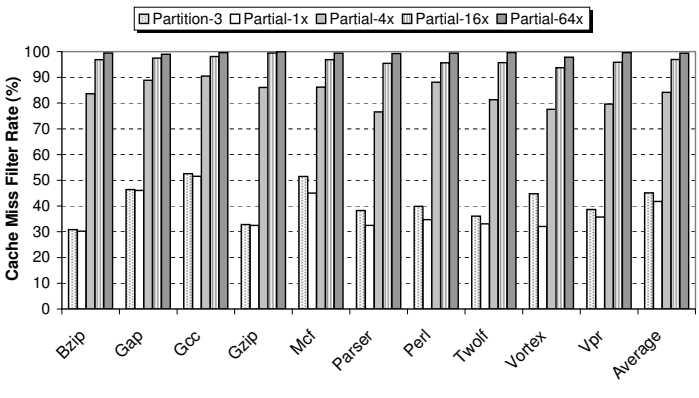

Figure 7: Cache Miss Filter Rate Using Partitioned and Partial Address Bloom Filters

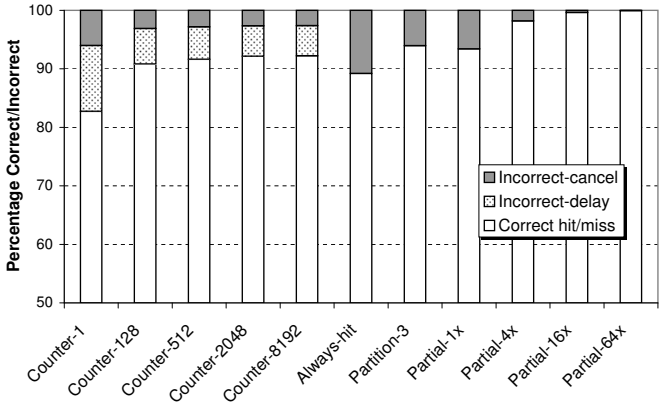

Figure 8: Prediction Accuracies for Different Cache Hit/Miss Predictors

increases. For a modestly sized 8K-bit BF array, the average filtering rate of *Partial-16x* is 97%.

Figure 8 shows the average (over all the benchmarks) correct and incorrect cache hit/miss prediction rates. It shows the prediction accuracy for BFs as well as other predictors. Correct predictions include both predict-hit-actual-hit and predict-miss-actual-miss cases. Incorrect predictions are separated into two groups. *Incorrect-cancel* is the case where a hit is predicted, but the load actually misses the cache. All speculatively scheduled dependents of the load must be canceled and rescheduled. *Incorrect-delay* is the case where a miss is predicted, but the load actually hits the cache. This misprediction unnecessarily delays the scheduling of the load's dependents and hence injects bubbles into the pipeline.

The predictors using saturating counters have a significant percentage of predictions in the *Incorrect-delay* group, and this percentage is insensitive to the predictor size. For *Counter-2048*, 5.2% of predictions are in the *Incorrect-delay* group and 2.7% are in the *Incorrect-cancel* group. The BFs, on the other hand, don't have any predictions in the *Incorrect-delay* group. In addition, the percentage in *Incorrect-cancel* decreases dramatically with larger BFs. The total misprediction rate is only 0.4% for *Partial-16x* using a moderately sized 8K-bit BF array. As expected, the simple *Counter-1* and *Always-hit* predictors have the two highest average misprediction rates.

5.2 IPC Improvement

Figure 9 compares the IPCs for several data speculation methods. In addition to the different types of hit/miss predictors, we include the IPC of a machine that doesn't use any data speculation and of a machine that uses a perfect hit/miss predictor (*Perfect-sch*). Also, the benefit of data prefetching using *Partial-16x* and *Perfect-sch* are shown (labeled with *-DP* in the legend of the figure). We show results for all the individual benchmarks since the IPC improvements are very different among them. *Partial-16x* without data prefetching shows a 17% improvement over *No-speculation* and a 4% improvement over *Always-hit*. Compared to *Counter-1* and *Counter-2048*, the improvements are 9% and 6%. With data prefetching, the improvements rise to 19%, 6%, 11% and 8%, respectively. It is important to point out that *Partial-16x* reaches 99.7% of the IPC of *Perfect-sch*, and *Partial-16x-DP* reaches 99.7% of the IPC of *Perfect-sch-DP*.

Among the benchmarks, *Gcc*, *Perl*, and *Vortex* show little difference between the different data speculation methods. Analysis reveals that these three programs have a large number of L_1 instruction cache (I-cache) misses. The high I-cache miss rate prevents instructions from entering the pipeline, reducing the benefit of data speculation. The lower instruction fetch rate greatly reduces the RUU occupancy, which is measured as the average number of RUU entries occupied. Since the RUU occupancy is much lower, loads and their dependents can stay in the RUU longer without blocking other instructions, so there is less of a difference between *No-speculation* and aggressive speculation such as *Partial-16x*. Table 3 summarizes the performance improvement of *Partial-16x* over *No-speculation* for 3 different I-cache sizes. Note the IPC improvement grows as the I-cache size increases. The IPC improvement for *Always-hit* also grows as I-cache size increases (not shown in the table), but not as quickly as it does for *Partial-16x*.

5.3 Sensitivity Studies

In this section we examine the effect of BFs on processor performance for various data cache sizes, RUU sizes, and different branch predictors.

Figure 10 plots the IPC improvement of *Always-hit*, *Partial-16x*, *Partial-16x-DP*, and *Perfect-sch-DP* over *No-speculation* for four different data cache sizes. We make three observations. First, the bigger the cache, the better the IPC improvement for all 4 data speculation methods. With bigger caches, scheduling becomes more important, because the performance bottleneck caused by data cache misses is reduced. Thus, delaying the scheduling of a load's dependents until after its cache hit/miss status has been determined (as is done by the *No-speculation* method) is a bigger loss of opportunity. Second, the IPC of *Always-hit* improves faster than the other methods as cache size increases. This is because its prediction accuracy is directly tied to the cache hit rate, so it sees the biggest improvement in prediction accuracy as the cache size increases. The IPC improvement of *Partial-16x-DP* over *Always-hit* reduces from 5.9% to 5.4% to 4.9% to 4.3% as the cache size is increased from 8KB to 64KB. Nevertheless, we expect future high-performance processors will use smaller first-level caches to enable higher clock frequencies. Third, due to high accuracy, *Partial-16x-*

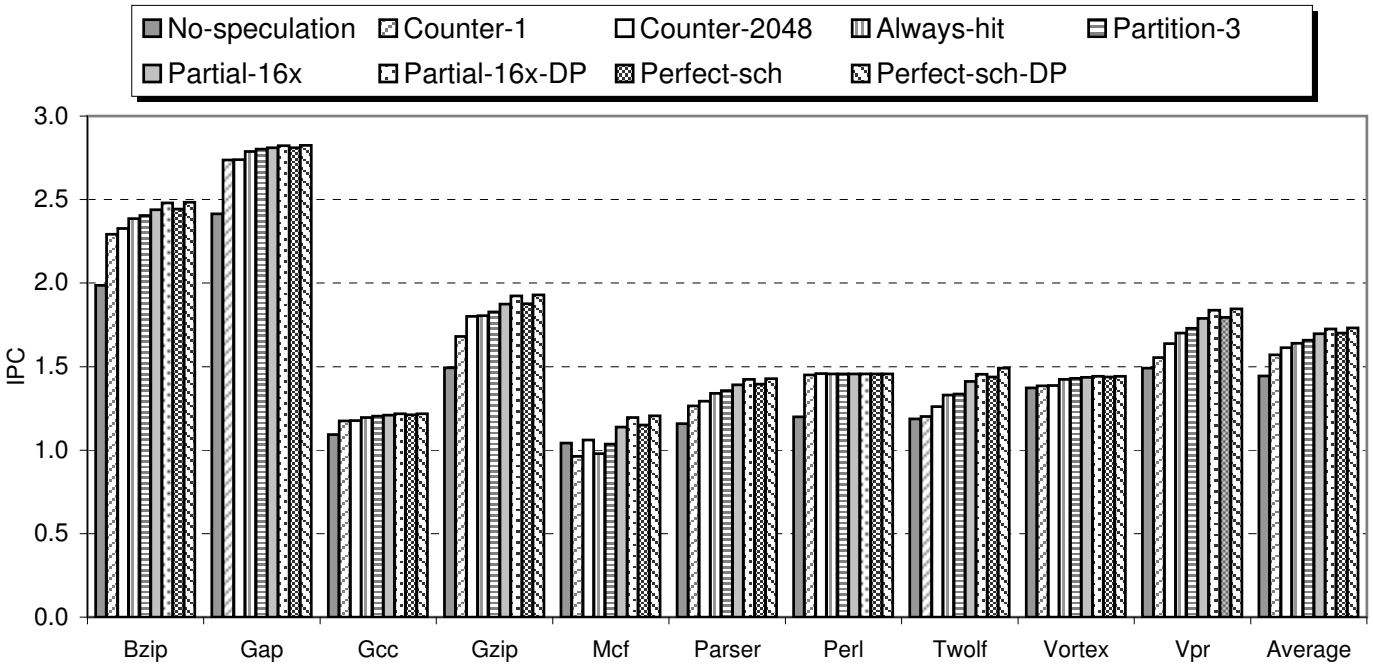

Figure 9: IPC Comparisons for Different Data Speculation Methods

	Gcc			Perl			Vortex		
I-cache	IPC%	I-miss%	RUU-ocu.	IPC%	I-miss%	RUU-ocu.	IPC%	I-miss%	RUU-ocu.
8KB	7.7	6.3	15.5	5.3	8.5	14.3	1.5	10.5	14.7
16KB	10.7	4.2	19.8	7.0	6.0	17.6	4.6	6.9	22.2
32KB	16.3	2.0	27.6	13.6	2.9	26.4	7.9	4.3	30.5

Table 3: Percent IPC improvement, I-cache miss rate, and RUU occupancy for 3 I-cache sizes

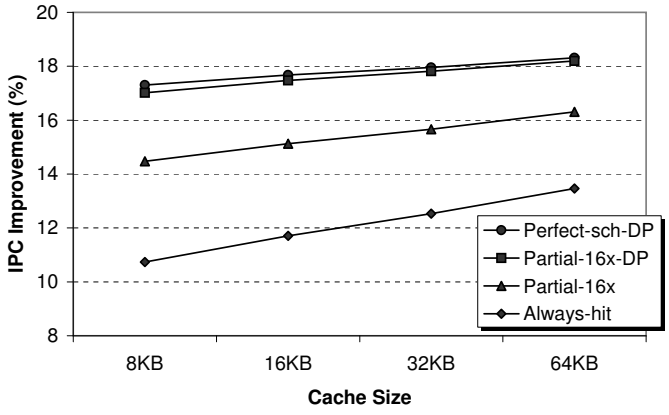

Figure 10: IPC Improvement Over No-speculation for Different Data Cache Sizes

DP achieves 99.9% of the IPC of the machine with a perfect scheduler for large caches.

Figure 11 shows the IPC improvement of *Partial-16x-DP*, *Partial-16x*, and *Always-hit* over *No-prediction* for three RUU sizes: 32, 64, and 128. We make several observations.

First, the IPC improvement is the greatest for the small RUU for all three methods. To achieve high performance with a small RUU, instructions need to flow through the RUU freely. Without data speculation, instructions that are dependent on loads block the flow. Thus, data speculation—even with all the rescheduling of dependent instructions due to mis-speculations—is essential for high performance when the RUU size is small.

Second, immediately prefetching the data when the BF identifies a miss improves IPCs by an additional 2–3%.

Third, the IPC improvement for *Always-hit* drops faster with increasing RUU size than the other two methods. And the performance gap between *Partial-16x* and *Always-hit* widens with bigger RUUs. To better illustrate this behavior, Figure 12 plots the IPC improvement of *Partial-16x-DP* and *Partial-16x* over *Always-hit*. In addition to the default Gshare predictor, the figure plots the performance of the two methods using a perfect branch predictor (labeled with *-perfectBR* in the legend of the figure). The results clearly show that the IPC improvement over *Always-hit* increases for bigger RUUs. Our simulation results show that with a small RUU, *Partial-16x* and *Always-hit* have similar RUU occupancies even though *Always-hit* produces more mispredictions. With a larger RUU, *Partial-16x* produces fewer RUU-full stalls than *Always-hit*. Effectively, *Partial-16x* has a larger instruction window in which to find instruction level

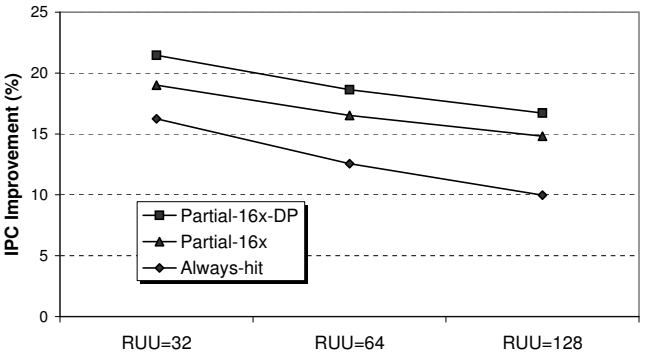

Figure 11: IPC Improvement Over No-speculation for Different RUU Sizes

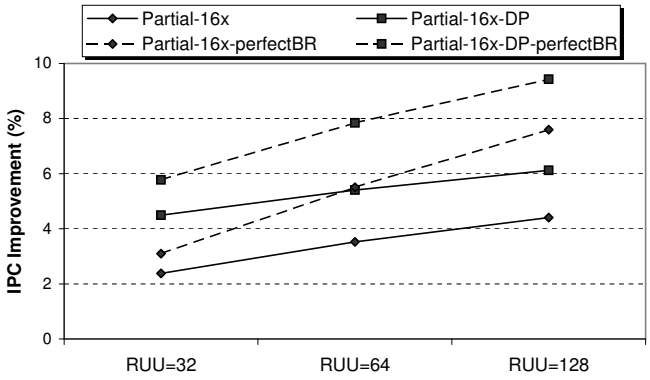

Figure 12: IPC Improvement Over Always-Hit for Different RUU Sizes

parallelism.

Lastly, in Figure 12, the IPC improvement of *Partial-16x* grows faster with increasing RUU size for the perfect branch predictor than for the default Gshare predictor. With a perfect branch predictor, the performance bottleneck due to branch mispredictions is eliminated, and instruction scheduling becomes more important. In addition, RUU occupancy is very high, since there are never any branch mispredictions that flush the RUU. A critical scheduling resource—RUU entries—becomes incredibly scarce. *Partial-16x* makes better use of this critical resource than *Always-hit*, as it cancels and reschedules fewer instructions. For a 128 entry RUU and a perfect branch predictor, the proposed *partial-address* BF improves IPC by more than 9% over the *Always-hit* method. Note that as branch prediction technology improves, the performance characteristics of real processors approach the performance characteristics of processors with perfect branch predictors.

6. CONCLUSION

Data speculation allows instructions that are dependent on a load to be scheduled before the latency of the load is known. A simple approach is to speculate (predict) that the load will always hit the L_1 cache and schedule its dependents accordingly. Unfortunately, whenever a prediction is wrong, the machine must recover all the mis-scheduled dependents, and performance suffers. In this paper we described how a Bloom Filter (BF) can be used to accurately predict cache misses. With a reasonably sized BF, we can correctly predict 99% of all misses. For the SPECint2000 benchmarks running on a modified SimpleScalar out-of-order model, the performance of a machine with a BF improved by 19% over a machine that delayed the scheduling of the load's dependents until the load's hit/miss status was known, and by 6% over a machine that speculated loads always hit the cache. We have also shown that this performance improvement grows as the window size and branch prediction accuracy increase. We expect that our BF technique will have an even greater performance advantage as pipelines deepen and cache latencies increase.

ACKNOWLEDGEMENTS

Jih-Kwon Peir thanks Windsor Hsu at IBM Almaden Research Center for early feedback on this paper. We also thank the anonymous referees for their helpful comments. This work is supported in part by NFS grants MIP-9624498 and EIA-0073473, and by Intel research donations.

REFERENCES

[1] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, 1999.

[2] B. Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Communications of The ACM*, 13(7):422–426, 1970.

[3] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report #1342, CS Department, Univ. of Wisconsin-Madison, June 1997.

[4] B. Calder, D. Grunwald, and J. Emer. Predictive Sequential Associative Cache. *Proc. of 2nd Int'l Symp. on High Performance Computer Architecture*, 1996.

[5] G. Chrysos and J. Emer. Memory Dependence Prediction using Store Sets. *Proc. of 25th Int'l Symp. on Computer Architecture*, 1998.

[6] P. Glaskowsky. Pentium 4 (Partially) Previewed. *Microprocessor Report*, Aug. 2000.

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technical Journal*, Q1 2001.

[8] R. Kessler, R. Jooss, A. Lebeck, and M. Hill. Inexpensive Implementations of Set-Associativity. *Proc. of 16th Int'l Symp. on Computer Architecture*, 1989.

[9] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[10] L. Liu. Cache Design with Partial Address Matching. *Proc. of 27th Int'l Symp. on Microarchitecture*, 1994.

[11] E. Morancho, J. Llabería, and A. Olivé. Recovery Mechanism for Latency Misprediction. *Proc. of 10th Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2001.

[12] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. Jetty: Filtering Snoops for Reduced Energy Consumption in SMP Servers. *Proc. of 7th Int'l Symp. on High Performance Computer Architecture*, 2001.

[13] D. Papworth. Tuning the Pentium Pro Microarchitecture. *IEEE Micro*, 16(2):8–15, 1996.

[14] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–41, 1996.

[15] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation Techniques for Improving Load Related Instruction Scheduling. *Proc. of 26th Int'l Symp. on Computer Architecture*, 1999.