In Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32), November 1999.

Fetch Directed Instruction Prefetching

Glenn Reinman[†] Brad Calder[†] Todd Austin[‡]

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Electrical Engineering and Computer Science Department, University of Michigan

Abstract

Instruction supply is a crucial component of processor performance. Instruction prefetching has been proposed as a mechanism to help reduce instruction cache misses, which in turn can help increase instruction supply to the processor.

In this paper we examine a new instruction prefetch architecture called Fetch Directed Prefetching, *and compare it to the performance of next-line prefetching and streaming buffers. This architecture uses a decoupled branch predictor and instruction cache, so the branch predictor can run ahead of the instruction cache fetch. In addition, we examine marking fetch blocks in the branch predictor that are kicked out of the instruction cache, so branch predicted fetch blocks can be accurately prefetched. Finally, we model the use of idle instruction cache ports to filter prefetch requests, thereby saving bus bandwidth to the L2 cache.*

1 Introduction

At a high-level, a modern high-performance processor is composed of two processing engines: the *front-end processor* and the *execution core*. The front-end processor is responsible for fetching and preparing (*e.g.*, decoding, renaming, etc.) instructions for execution. The execution core orchestrates the execution of instructions and the retirement of their register and memory results to non-speculative storage. Typically, these processing engines are connected by a buffering stage of some form, *e.g.*, instruction fetch queues or reservation stations – the front-end acts as a producer, filling the connecting buffers with instructions for consumption by the execution core. This producer/consumer relationship between the front-end and execution core creates a fundamental bottleneck in computing, *i.e.*, execution performance is strictly limited by fetch performance. Efficient instruction cache performance is critical in order to keep the execution core satisfied. Instruction cache prefetching has been shown to be an effective technique for improving instruction fetch performance [2, 9, 6, 7, 13, 17, 18, 20, 21], and this is the focus of our paper.

We recently proposed a scalable fetch architecture to relieve the fetch bottleneck [14]. One aspect of that architecture was to decouple the branch predictor from the instruction cache. The branch predictor produces fetch blocks into a *Fetch Target Queue* (FTQ), where they are eventually consumed by the instruction cache. This decoupling allows the branch predictor to run ahead of the instruction fetch. This can be beneficial when the instruction cache has a miss, or when the execution core backs up, thereby filling up the connecting buffers and causing the instruction cache to stall. This second case can occur because of data cache misses or long latency instructions in the pipeline. In this paper we examine using the fetch block addresses in the FTQ to provide *Fetch Directed Prefetching* (FDP) for the instruction cache. We use the fetch block addresses stored in the FTQ to guide instruction prefetching, masking stall cycles due to instruction cache misses.

Future branch prediction architectures may be able to hold more state than the instruction cache, especially multi-level branch predictors and those that are able to predict large fetch blocks or traces. When a cache block is evicted from the instruction cache, we examine marking the entry that corresponds to this cache block in the Fetch Target Buffer (FTB) [14], which is similar to a branch target buffer, but can predict larger fetch blocks. If a branch predicted FTB entry is marked as being evicted, we can then prefetch the predicted fetch block using our fetch directed prefetching architecture.

We also examine taking advantage of idle ports on the instruction cache to check if potential prefetch addresses are in the cache before using them to perform a prefetch. This will filter prefetch requests and save bus bandwidth to the unified L2 cache. We use a lockup free instruction cache, so that the I-cache port(s) can be used for prefetch filtering even during an instruction cache miss.

The remainder of this paper is organized as follows. In Section 2 we describe the methodology used to gather our results. Section 3 describes using idle cache ports to filter instruction prefetch requests, and Section 4 describes our previous work on decoupled front-end branch predictor. Section 5 proposes and provides results for our fetch directed prefetching architecture, and Section 6 describes and provides results for prior prefetching architectures. Section 7 compares fetch directed prefetching to prior prefetching architectures for different cache sizes and associativities and when using one or two ports for the cache. Finally, Section 9 provides a summary and discusses future directions.

program	# inst fwd	16K-2W		16K-4W		32K-2W	
		MR	IPC	MR	IPC	MR	IPC
groff	0	5.9	1.90	4.7	2.05	2.6	2.43
gcc	400	7.1	1.43	6.4	1.48	3.4	1.77
go	1000	3.4	1.48	2.1	1.61	0.9	1.77
m88ksim	1000	2.8	2.08	1.2	2.49	1.2	2.43
perl	2000	3.9	2.18	2.7	2.43	0.6	2.91
vortex	1000	12.8	1.50	10.8	1.61	6.5	2.09

Table 1: Program statistics for the baseline architecture.

2 Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

To perform our evaluation, we collected results for 5 of the SPEC95 C benchmarks plus `Groff` a C++ text formatting program using the reference inputs. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Table 1 shows the number of instructions (in millions) that were executed (fast forwarded) before simulation began. These fast forward numbers were taken from a study by Sherwood and Calder [16], which shows the time varying behavior (IPC, cache miss rates, branch misprediction, etc.) at 100 million instruction intervals for all the SPEC 95 programs. We report results for simulating each program for up to 100 million instructions after fast forwarding.

2.1 Metrics

To evaluate instruction prefetching we provide results in terms of (1) percent speedup in IPC achieved from using prefetching over the base IPCs shown in Table 1, (2) percent L2 bus bandwidth used, and (3) instruction cache miss rates. When providing the results for bus utilization, the utilization is the percent of cycles the bus is busy servicing L1 instruction and data cache misses, and any instruction prefetches. The instruction cache miss rates are in terms of the percent of *fetched cache blocks* that missed in the instruction cache. This is calculated by taking the number of cache blocks fetched that missed in the instruction cache, and dividing this by the number attempted cache blocks fetched during execution.

2.2 Baseline Architecture

Our baseline simulation configuration models a future generation out-of-order processor microarchitecture. We've selected the parameters to capture underlying trends in microarchitecture design. The processor has a large window of execution; it can fetch up to 8 instructions per cycle. It has a 128 entry reorder buffer with a 32 entry load/store buffer. To compensate for the added complexity of disambiguating loads and stores in a large execution window, we increased the store forward latency to 3 cycles.

There is an 8 cycle minimum branch misprediction penalty. The processor has 8 integer ALU units, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, and 2-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles, Integer DIV 12 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle.

2.3 Memory Hierarchy

We completely rewrote the memory hierarchy in SimpleScalar to better model bus occupancy, bandwidth, and pipelining of the second level cache and main memory.

We examine the performance of instruction prefetching for a 16K 2-way and 4-way associative cache, and a 32K 2-way associative cache, each with 32-byte lines. Table 1 shows the percent of fetched cache blocks that missed (MR) for each of these cache configurations, along with their corresponding IPCs. The data cache for each configuration is a 4-way associative 32K cache with 32 byte lines.

The second level cache is a unified 1 Meg 4-way associative pipelined L2 cache with 64-byte lines. The L2 hit latency is 12 cycles, and the round-trip cost to memory is 100 cycles. The L2 cache has only 1 port. We examine two different L2 caches. The first configuration corresponds to pipelining the L2 cache to allow a new request each cycle, so the L2 bus can transfer 32 bytes/cycle. The second configuration corresponds to allowing a new request every 4 cycles, so the L2 bus can transfer 8 bytes/cycle. The L2 bus is shared between instruction cache block requests, data cache block requests, and prefetch requests.

3 Using Idle Cache Ports to Filter Instruction Prefetch Requests

Lockup-free caches were originally proposed to increase the performance for a unified instruction and data cache [8], and have been shown to improve the performance of data caches by allowing multiple outstanding loads [4]. Lockup free caches are inexpensive to build, since they only require a few Miss Status Holding Registers (MSHRs) to hold the information for an outstanding miss. When the missed block has been completely fetched from memory it is then inserted into the cache. In this paper we make use of a lockup-free instruction cache to implement a form of prefetch filtering during an I-cache miss.

3.1 Cache Probe Filtering

Prefetching blocks that are already contained in the instruction cache results in wasted bus bandwidth. When the instruction cache has an idle port, the port can be used to check whether or not a potential prefetch address is already present in the cache. We call this technique *Cache Probe Filtering* (CPF). If the address is found in the cache, the prefetch request can be cancelled, thereby saving bandwidth. If the address is not found in the cache, then in the next cycle the block can be prefetched if the L2 bus is free. Cache probe filtering only needs to access the instruction cache's tag array. Therefore, if it is found to be beneficial to add an extra port for CPF, it would only effect the timing of the tag access, and not the data array.

An instruction cache (port) may be idle and go unused because of (1) an instruction cache miss, (2) a full instruction window, or (3) insufficient predicted fetch blocks. If an instruction cache miss occurs, then the fetch engine will stall until the miss is resolved. When the instruction window becomes full because of a slow instruction in the execution core, the instruction cache has to stall since the fetch buffer is full. In addition, a dual ported or banked instruction cache can potentially have idle ports if the branch predictor does not provide enough cache block addresses to fetch in a given cycle.

To use the idle cache ports to perform cache probe filtering during a cache miss, the cache needs to be lockup-free. To benefit from cache probe filtering in the presence of (2) and (3) above, the instruction cache does not have to be lockup-free.

3.2 Prior Cache Probe Filtering Research

Prior instruction cache prefetch studies have examined filtering instruction prefetches based on whether or not the address was already in the instruction cache [2, 9]. For every instruction prefetch, these studies first check the instruction cache and only perform the prefetch if the block is not in the instruction cache. This assumes extra instruction cache ports to perform the cache probe check.

In our simulations we examine the performance with and without cache probe filtering. We model cache port usage in our simulations, and only allow cache probe filtering to occur when there is an idle cache port.

4 Decoupled Front-End

A traditional fetch engine couples its branch prediction architecture with the instruction cache. Each cycle the branch prediction architecture is used to produce the fetch address(es) to be used in the next cycle's cache fetch. If the instruction cache stalls, then instruction fetch stops and the branch predictor and instruction cache sit idle until the cache miss is resolved.

In [14], we examined decoupling the branch predictor from the instruction cache. To provide a decoupled front-end, a *Fetch Target Queue* (FTQ) is used to bridge the gap between the branch predictor and the instruction cache. Every cycle, the branch predictor will produce a fetch target block prediction and store it in the FTQ, where it will eventually be consumed by the instruction cache. The FTQ provides the buffering necessary to permit the branch predictor and instruction cache to operate autonomously. The FTQ allows the branch predictor to work ahead of the instruction cache when it is stalled due to a cache miss or a full instruction buffer. If the instruction cache is multi-ported, multiple valid FTQ entries can be consumed in a single cycle until ports are exhausted.

Any branch predictor (Basic Block Target Buffer [22], Two-Block Ahead Predictor [15], or Fetch Target Buffer [14]) can be put in front of the FTQ and produce fetch block addresses to be consumed by the instruction cache. It is the FTQ that allows the branch predictor to run ahead of the instruction cache.

4.1 Baseline Branch Prediction Architecture

In prior work, we showed that a 2-level Fetch Target Buffer (FTB) can provide scalable branch prediction [14]. In this paper we use a single level 4K entry 4-way associative FTB, which feeds a 32 entry FTQ, instead of a 2-level FTB table, in order to concentrate on fetch direct prefetching. A 4K entry FTB is large, but is implementable using the 2-level FTB approach described in [14]. We are currently evaluating using FDP with a 2-level FTB design. For conditional branch prediction, we use the McFarling bi-modal gshare predictor [11], with an 8K entry gshare table and a 64 entry return address stack in combination with the FTB.

Table 2 shows the effects of using the FTB with a dual ported instruction cache. The results show that 7% speedup is achieved on average when using 2 ports instead of 1 port for the instruction cache. The last 3 columns in table 2 show the percent of executed cycles when (1) both of the ports were idle, (2) when only one of the ports was idle, and (3) when none of the ports were idle. The first case occurs because of a stalled fetch engine. The second case occurs when the FTQ can only provide enough fetch bandwidth to fetch from one cache block. The final column shows that on average the FTQ can provide enough fetch bandwidth to keep both ports on the instruction cache busy 20% of the time. Cache probe filtering, as described in section 3, can take advantage of these idle cache ports to filter prefetches the remaining 80% of the time. In our baseline architec-

program	% speedup with 2 ports	% port availability 2 idle	1 idle	0 idle
groff	8.2	35.1	45.6	19.3
gcc	6.9	36.8	46.1	17.1
go	5.4	32.2	44.0	23.9
m88ksim	5.6	31.0	46.5	22.5
perl	11.2	32.3	42.5	25.3
vortex	6.9	41.2	47.7	11.1
average	7.4	34.8	45.4	19.9

Table 2: Baseline statistics for the dual-ported instruction cache. The first column shows the percent speedup obtained when using a dual-ported instruction cache over a single ported cache for the baseline architecture. The last three columns represent the percent of executed cycles when both ports on the cache are idle, when one port is idle, and when both ports are busy.

ture we use 2 ports for the instruction cache. In section 7, we compare the prefetching performance of single and dual ported L1 instruction caches.

5 Fetch Directed Prefetching

Fetch Directed Prefetching (FDP) follows the predicted stream, enqueuing prefetches from the FTQ. This is made possible if the branch prediction architecture can run ahead of the instruction fetch, which is what the FTQ based branch predictor provides [14]. One advantage of this design, is that FDP can continue to prefetch down the predicted stream even when the instruction cache is stalled. We now describe our Fetch Directed Prefetching architecture, describe the heuristics that were used to better select which fetch blocks to prefetch, and evaluate their performance.

5.1 Fetch Directed Prefetching Architecture

Figure 1 shows the FDP architecture. As described in section 4, we use a decoupled branch predictor and instruction cache, where the FTQ contains the fetch blocks to be fetched from the instruction cache. The FDP architecture uses a *Prefetch Instruction Queue* (PIQ), which is a queue of prefetch addresses waiting to be prefetched. A prefetch from the PIQ will start when the L2 bus is free, after first giving priority to data cache and instruction cache misses.

One of the benefits of the FTB branch predictor design is that it can provide large fetch blocks [14]. A fetch block from one prediction can potentially span 3 cache blocks. Therefore, each FTQ entry contains three cache fetch block entries providing the fetch addresses for up to 3 cache blocks. Each fetch block entry contains a valid bit, a *candidate* prefetch bit, and an *enqueued* prefetched bit, as well as the cache block address. The candidate bit indicates that the cache block is a candidate for being prefetched. The

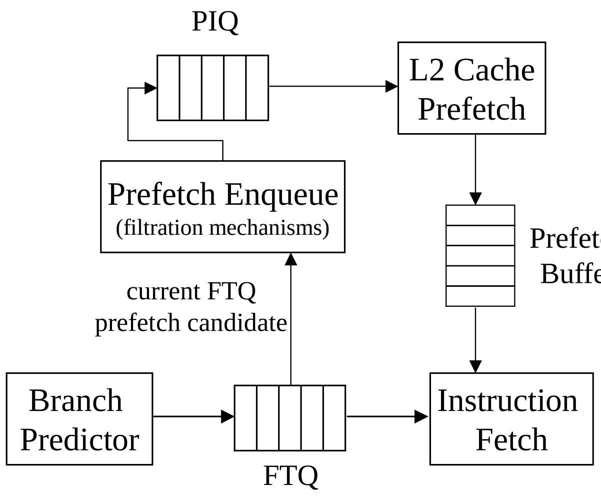


Figure 1: Fetch Directed Prefetching Architecture.

bit is set using filtration heuristics described below. The enqueued bit indicates that the cache block has already been enqueued to be prefetched in the PIQ. Candidate prefetches from FTQ entries are considered in FIFO order from the FTQ, and are inserted into the PIQ when there is an available entry. The current FTQ entry, under consideration for inserting prefetch requests into the PIQ, is tracked via a hardware-implemented pointer.

A fetch directed FIFO prefetch buffer is added to the FDP architecture to hold the prefetched cache blocks. This is very similar to a streaming buffer [7], except that it gets its prefetch addresses from the FTQ. Each time a cache block is inserted into the PIQ for prefetching, an entry is allocated for that cache block in the prefetch buffer. If the prefetch buffer is full, then no further cache blocks are prefetched. When performing the instruction cache fetch, the FIFO prefetch buffer is searched in parallel with the instruction cache lookup for the cache block. If there is a hit in the *oldest entry* in the FIFO prefetch buffer, it is removed from the buffer, and inserted into the instruction cache. The prefetch buffer is flushed on a branch misprediction to keep the FIFO prefetch buffer in-sync with the FTQ which is also flushed on a misprediction.

An alternative approach would be to use a fully associative prefetch buffer with a sophisticated LRU replacement strategy, adding a *replaceable* bit to each prefetch buffer entry. A prefetch would be initiated (allocated in the prefetch buffer) only if there was an entry marked as replaceable in the prefetch buffer, choosing the least recently used replaceable entry. On a branch misprediction, all entries would be designated as replaceable, rather than flushing the whole prefetch buffer as in the FIFO prefetch buffer. They would also be marked as replaceable in the prefetch buffer when the matching cache blocks are fetched during the instruction cache fetch. Using a fully associative prefetch buffer like this would allow prefetches down a mispredicted path to still be useful — especially in the case of short, forward branches. A buffer with this design would be expected to perform better than a FIFO design, but at the cost of more complex hardware. We are currently investigating the performance of using a fully associative prefetch buffer with replaceable bits. In this paper we only report results for the FIFO prefetch buffer.

We examined several approaches for deciding which FTQ entries to prefetch and insert into the PIQ, which we describe in the following sections.

5.2 Filter Based on Number of FTQ Entries

The earlier the prefetch can be initiated before the fetch block reaches the instruction cache, the greater the potential to hide the miss latency. At the same time, the farther the FTQ entry is ahead of the cache, the more likely that it will be on a mispredicted path, and the more likely a prefetch from the FTQ entry might result in a wasted prefetch, since the prefetch buffer is flushed on a branch misprediction.

We examined filtering the number of FTQ entries to be considered for prefetching based on the position of the fetch block entry in the FTQ. Our prior results showed that when using an FTQ, its occupancy can be quite high [14]. The FTQ contains 16 or more fetch blocks for 30% of the executed cycles on average, and 4 or more fetch blocks for 60% of the executed cycles. We found that starting at the second entry from the front of the FTQ and going up to 10 entries in the FTQ provided the best performance. We skip the first entry in the FTQ, because there would be little to no benefit from starting it as a prefetch when it is that close to being fetched from the instruction cache. The FTQ we implemented can hold up to 32 entries, but stopping prefetching at 10 entries provided good performance, since prefetching farther down the FTQ resulted in decreased probability that the prefetch would be useful, and potentially wastes memory bandwidth.

5.3 Cache Probe Filtering

Cache probe filtering uses idle cache ports to check if a potential prefetch request is in the cache. We examined two approaches to CPF.

The first approach, called cache probe enqueuing (Enqueue CPF), will *only* enqueue a prefetch into the PIQ from the FTQ if it can first probe the instruction cache using an idle cache port to verify that the cache block does not exist in the first level cache. This is a very conservative form of prefetching.

The second approach, called remove cache probe filtering (Remove CPF), enqueues all cache blocks into the PIQ by default, but if there is an idle first level cache port, it will check the cache tags to see if the address is in the cache. If the address is in the cache, the prefetch entry will be removed from the list of potential prefetch addresses. If there are no idle cache ports, then the request will be prefetched without checking the cache tags.

5.4 Cache Miss Filtering

It is desirable to concentrate on prefetching only those fetch blocks that will most likely miss in the instruction cache in order to reduce bus utilization. If a given cache *set* has a lot of conflict misses, then it can be beneficial to prefetch all blocks that map to that high conflict cache set. To capture this behavior we examine using a confidence counter associated with each instruction cache set to indicate which cache sets miss most frequently. Fetch blocks that access these cache sets will have a greater chance of missing in the instruction cache and will therefore be worth prefetching.

We examined adding a cache miss buffer that contains a 2-bit saturating counter for each instruction cache set. We index the cache miss buffer in parallel with the FTB. If the fetch block being predicted maps to a cache set that has missed frequently in the past, that fetch block is marked to be prefetched, otherwise it is not a candidate for being prefetched.

We found the following finite state machine to work well for the miss counters. When a cache miss occurs, its corresponding set counter in the miss buffer is incremented. A cache hit does not change the cache set miss counters. Instead, the confidence counters are cleared every million cycles to prevent extraneous prefetching. A predicated cache block from the FTB whose set counter is saturated (has a value of 3) is marked as candidate for being prefetched, otherwise it is not.

5.5 Fetch Block Evicted Prefetching

Another approach for finding fetch blocks to prefetch that may have been evicted from the instruction cache due to cache conflicts is to keep track of these cache blocks in the branch prediction architecture. To achieve this, we store an *evicted* bit in each FTB entry and it is set when the cache block of the corresponding branch is evicted from the instruction cache. The implementation we examined stores N bits with each cache block to identify the FTB entry that last caused the cache block to be brought into the cache. For the implementation we simulated, 12 bits are used to map the block to the FTB entry – 10 bits to index into the set, and 2 bits for the way (4-way associative FTB). Since the bits stored in the cache are a direct index into the FTB, there is no guarantee that the entry that loaded the cache block is still in the FTB, but for an FTB which can hold more state and have a larger associativity than the instruction cache, the mapping will likely be correct.

When a cache block is evicted from the cache, the N bit index is used to access the FTB. The evicted bit in the

FTB entry corresponding to that index is set, indicating that the cache block will be prefetched the next time it is used as a branch prediction. An eviction can cause at most one prefetch, since the evicted bit is cleared for an FTB entry when it is inserted into the FTQ.

5.6 Fetch Directed Prefetching Results

Figure 2 shows results for fetch directed prefetching without any filtering (NoFilt), for remove cache probe filtering (Remove CPF), remove CPF along with using the 2-bit Cache Miss table for filtering, for cache probe enqueuing (Enqueue CPF), using only evicted bits (Evict) to guide prefetching, and when using both evicted bits and enqueue CPF (Enq CPF + Evict). Results are shown for bus bandwidths of 32 bytes/cycle and 8 bytes/cycle for a *shared* instruction and data bus to the pipelined L2 cache, which has 1 port. Figure 3 shows the average percent L2 bus utilization for all of these configurations.

Fetch directed prefetching, even without any filtering techniques, provides substantial benefits (38% on average for a bandwidth of 32 bytes/cycle). As can be seen in figure 3, this technique uses a great deal of bus bandwidth (67% bus utilization). For an 8 byte/cycle bandwidth, utilization reaches 90% on average, and the IPC speedup obtained drops to 19% on average.

Using cache probe filtering to remove prefetches (Remove CPF) reduces bus utilization (56% utilization for a bandwidth of 32 bytes/cycle and 80% for a bandwidth of 8 bytes/cycle) by reducing the number of the useless prefetches. This allows more useful prefetches to be enqueued sooner, resulting in an average speedup of 40% and 26% in the high and low bandwidth cases respectively. Cache probe enqueuing (Enqueue CPF) only prefetches a cache block if it can first verify that it is not in the cache, and this eliminates useless prefetches and brings the bus utilization down to 17% for the high bandwidth case, and down to 37% for the low bandwidth case. Its performance is 20% on average for high bandwidth, and 16% on average for low bus bandwidth.

Using the 2-bit Cache Miss Table with Remove CPF, only marks predicted cache blocks from the FTB as candidates for prefetching if its corresponding cache set counter is saturated. These candidates are then filtered using Remove CPF if an idle cache port is available. Adding the Cache Miss Table to Remove CPF provides about the same percent speedup as applying Remove CPF on all predicted cache blocks. The big difference is the drop in bus utilization, which drops from 56% without cache miss filtering to 44% with cache miss filtering.

Using only the Evicted bit to determine which predicted cache blocks to prefetch provides an average 11% speedup for the high bandwidth case, while only using 6% of the bus bandwidth. Combining the use of the evicted bit with cache probe enqueuing (Evict + Enqueue CPF) provides a large increase in speedup for some programs (27% for the high bandwidth case), but still only uses a small fraction (17%) of the memory bandwidth. For this combination, predicated cache blocks are marked as candidates for prefetching if either their evicted bit is set *or* the cache block is found not to be in the instruction cache using CPF.

6 Next Line Prefetching and Streaming Buffers

We now describe the prior instruction prefetching research we implemented, and compare their relative performance. We also investigate augmenting streaming buffers with cache probe filtering.

6.1 Tagged Next Line Prefetching

Smith [17] proposed tagging each cache block with a bit indicating when the next block should be prefetched. When a block is prefetched its tag bit is set to zero. When the block is accessed during a fetch and the bit is zero, a prefetch of the next sequential block is initiated and the bit is set to one. Smith and Hsu [18] studied the effects of tagged next line prefetching and the benefits seen based upon how much of the cache line is used before initiating the prefetch request.

6.2 Target and Wrong Path Prefetching

Smith and Hsu [18] also examined the benefits of combining next-line prefetching with target prefetching. For target prefetching, they used a table of branch target addresses, which was indexed in parallel with the instruction cache lookup. If there was a hit, then the target address was prefetched. For a given branch, they examined prefetching both the fall through and the target address.

Pierce and Mudge [13] examined what they called Wrong Path Prefetching, where they examine prefetching both paths of a branch. There are two major differences in their approach and the approach suggested by Smith and Hsu [18]. They only prefetch the taken target if the branch was not-taken, and they only do this after the taken address is calculated in the decode stage. The address is not derived from the branch target buffer. This has the advantage of being able to prefetch branch targets not in the BTB. Their results showed that target prefetching provided only a small improvement over next-line prefetching.

6.3 Streaming Buffers

Jouppi proposed streaming buffers to improve the performance of directed mapped caches [7]. If a cache miss occurs, sequential cache blocks, starting with the one that missed, are prefetched into a streaming buffer, until the buffer is filled. A streaming buffer is implemented as a FIFO queue. The streaming buffer is searched in parallel with the

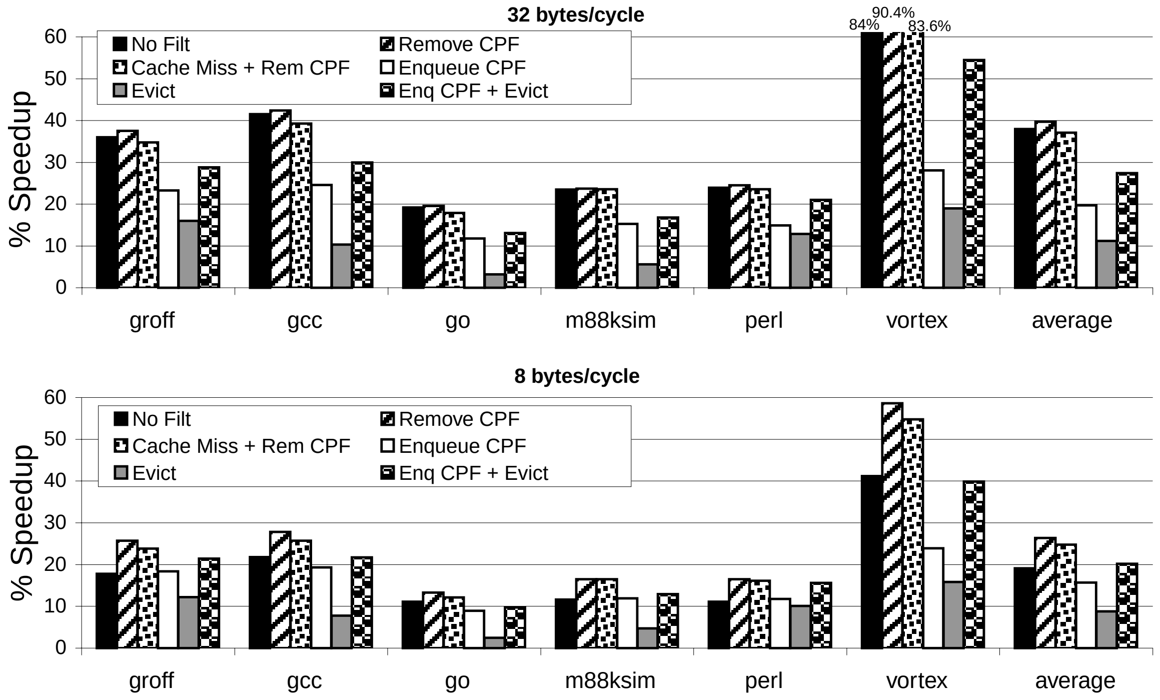


Figure 2: Percent IPC speedup for fetch directed prefetching using a 16K 2-way instruction cache. Results are shown for both high and low bus bandwidths. The first bar shows results for unfiltered fetch directed prefetching. The next two bars show results for fetch directed prefetching with Remove Cache Probe Filtering: first alone, and then in conjunction with cache miss filtering. The fourth bar shows results for Enqueue Cache Probe Filtering. The fifth bar shows results when using the evicted bit alone, and the sixth bar shows results when using the evicted bit and Enqueue CPF together.

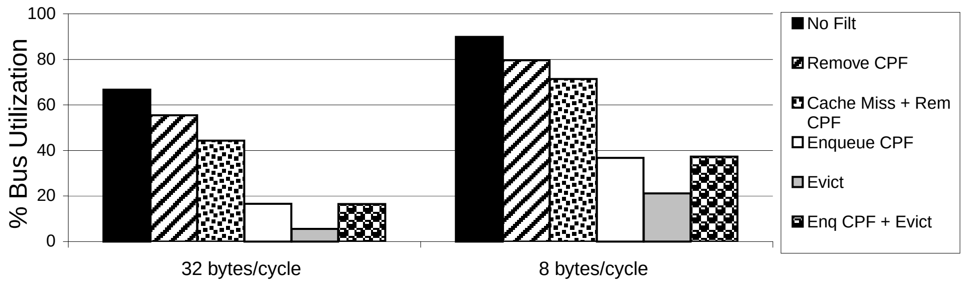


Figure 3: Percent bus utilization for fetch directed prefetching results.

instruction cache when performing a lookup. He also examined using multiple streaming buffers at the same time.

Farkas et. al., [5] examined the performance of using a fully associative lookup on streaming buffers and a uniqueness filter. These were shown to be beneficial when using multiple streaming buffers, so that each of the streams did not overlap, saving bus bandwidth.

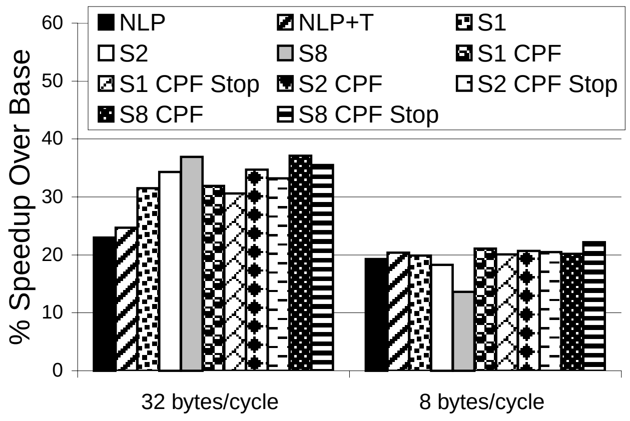
6.4 Performance of Prior Work

We now examine the performance of next-line prefetching, next-line with target prefetching, and streaming buffers. We then compare the performance of these prefetching architectures with fetch directed prefetching in section 7.

6.4.1 Next-line and Target Prefetching

Figure 4 shows the performance and bus utilization results for prior instruction prefetching work. We implemented tagged Next-Line Prefetching (NLP) as described in [17], except we prefetch the cache blocks into a 4-way associative 32 entry prefetch NLP buffer with LRU replacement. This buffer is queried in parallel with the cache during a lookup to find a potential hit. We then implemented target prefetching and combined that with NLP as described in [18], called NLP+T in figure 4.

Next line prefetching provides large speedups for high and low bandwidth (23% and 19% respectively), while using very little bandwidth. In fact, for our low bandwidth



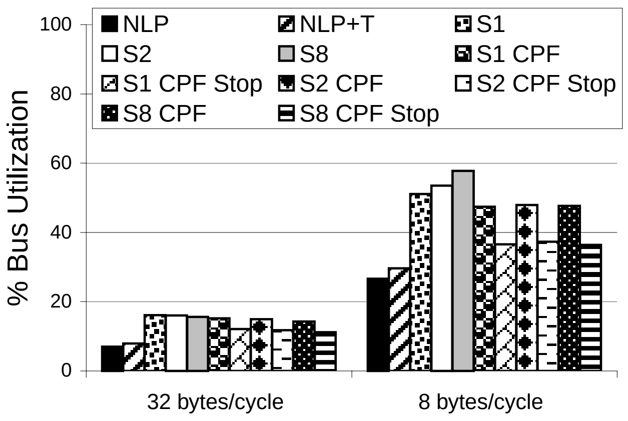


Figure 4: Percent IPC speedup and percent bus utilization comparisons for prior work using a 16K 2-way associative I-cache. Results shown are the average of all benchmarks, and are shown for high (32 bytes/cycle) and low (8 bytes/cycle) bus bandwidths. The first two bars in each set correspond to results obtained through next line prefetching with and without target prefetching. The remaining bars represent streaming buffer results. Results are shown for streaming buffers for 1, 2, and 8 buffers, and with cache probe filtering alone and with cache probe filtering that stops when a cache hit is seen.

architecture, NLP performed almost as well as streaming buffers, while using the least amount of bandwidth. Adding target prefetching to NLP provided minimal improvement.

6.4.2 Streaming Buffers

We implemented stream buffers as described earlier in section 6.3. In addition, we used the uniqueness filter proposed by Farkas et. al, [5]. Therefore, a given cache block will only appear in one stream at a time, and all valid entries in all streaming buffers are checked for a hit in parallel with the instruction cache lookup. We provide results using a single four entry streaming buffer (S1), an architecture with 2 four entry streaming buffers (S2), and 8 four entry streaming buffers (S8) in figure 4. When using multiple streaming buffers a streaming buffer is allocated on a cache miss using a modified least recently used algorithm. A use of a streaming buffer is considered to be a streaming buffer allocation or a cache block hit in the streaming buffer.

We found that four entry streaming buffers provided good performance for our pipeline architecture. Streaming buffers performed well (ranging from 32% speedup for 1 streaming buffer to 37% speedup on average for 8 streaming buffers), but performance degraded when used at a lower bus bandwidth. Single streaming buffers performed better at the lower bandwidth (20% in the best case), while the 8 streaming buffers provided 14% performance improvement on average.

We also investigated using cache probe filtering with streaming buffers. We used the remove CPF heuristics described in section 5.3. If a free port is available on the instruction cache, it can be used to filter out streaming buffer prefetches that are already in the cache. This filter could be used in one of two ways. If a cache block is found in the instruction cache, the prefetch can be skipped, and the streaming buffer can continue as usual and attempt to prefetch the next cache block. This is labeled as (Sx CPF) in figure 4. Alternatively, once a cache block is found in the instruction cache, the prefetch can be skipped, and the streaming buffer can be prevented from producing further prefetches until it is reallocated. We refer to this latter technique as the stop filter (Sx CPF Stop). When using cache probe filtering, about the same performance is achieved for high bus bandwidth, and larger speedups are seen for the low bus bandwidth. For low bus bandwidth, the stop filter provides the same performance, with a 10% decrease in bus utilization.

We also examined combining tagged next-line prefetching with streaming buffers (results not shown), but this provided only slight improvements. This is because the streaming buffers follow the sequential path after a cache miss, which overlaps the prefetch blocks that next-line prefetching captures.

7 Performance Tradeoffs

We now compare fetch directed prefetching with prior instruction prefetching work, and the performance from combining these techniques. Figures 5, 6, and 7 show results for next-line prefetching (NLP), streaming buffers with stop cache probe filtering (SBx CPF Stop), fetch directed prefetching without any filtering (No Filt), and fetch directed prefetching with a variety of filtration techniques. These techniques are described in detail in sections 5 and 6.

7.1 Combination Approaches

Fetch Directed Prefetching (FDP) concentrates on the predicted path of execution taken by the processor. There

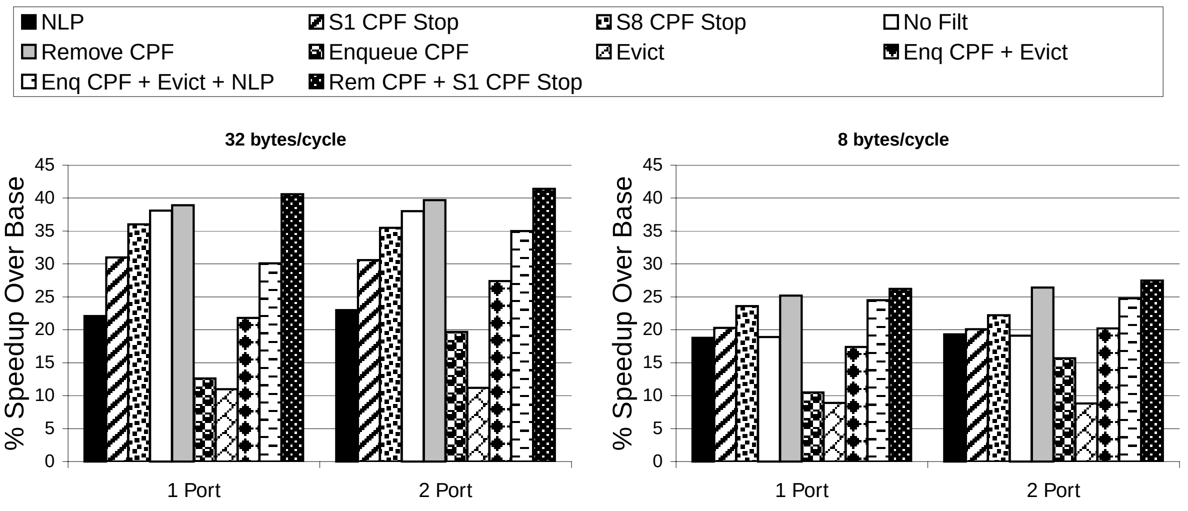


Figure 5: Average percent IPC speedups for single and dual-ported 16K 2-way associative instruction cache. Results are shown for high (32 bytes/cycle) and low (8 bytes/cycle) bandwidth architectures.

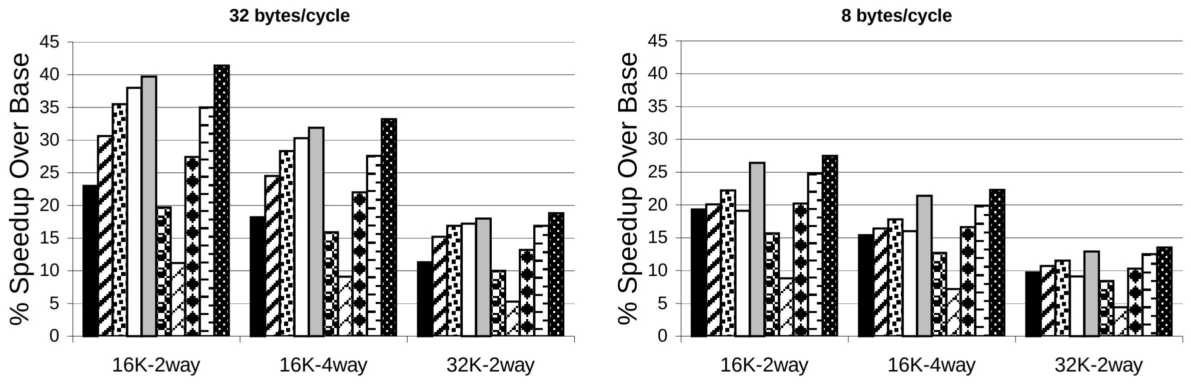


Figure 6: Average percent IPC speedups for 16K 2-way associative instruction cache, 16K 4-way associative instruction cache, and 32K 2-way associative instruction cache.

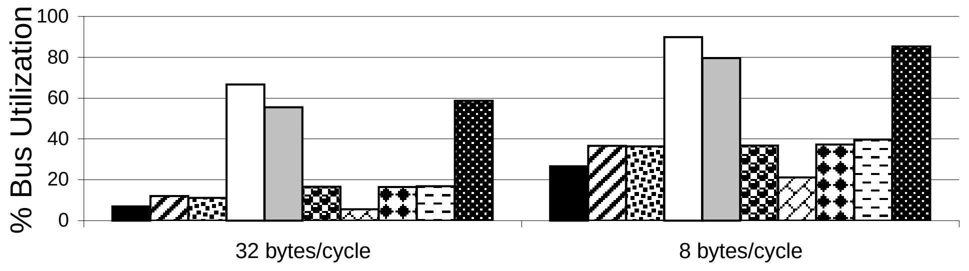
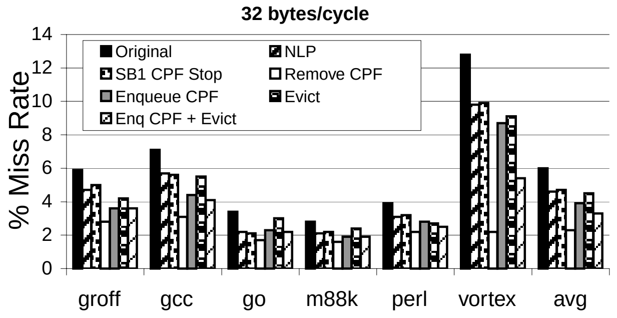


Figure 7: Average percent bus utilization for the seven prefetching techniques. Results shown for 16K 2-way associative instruction cache, for both high and low bandwidth buses.

are two benefits from combining next-line prefetching or streaming buffers with FDP. The first benefit is that sequentially prefetching down a miss stream could prefetch the wrong path into a prefetch buffer, which can be beneficial if the branch was mispredicted. The second benefit, is that they can potentially prefetch the not-taken path earlier than the fetch directed prefetch architecture, masking more of the cache miss latency. We tried several combinations of fetch directed prefetching and prior work, and present two combinations – one that works well for a low bandwidth bus



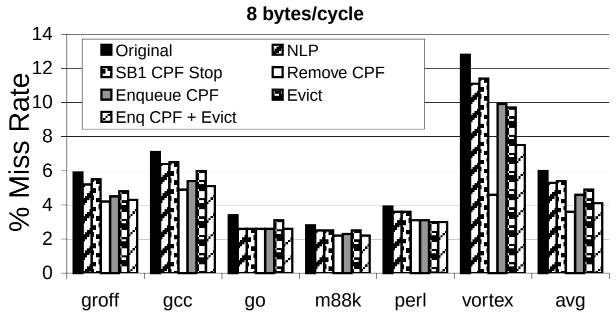


Figure 8: Instruction cache fetch block miss rates for the different prefetching architectures for a 16K 2-way I-cache.

architecture and one that works well for a high bandwidth bus.

The low bandwidth combination combines next-line prefetching with eviction-based prefetching and Enqueue Cache Probe Filtering (Enq CPF + Evict + NLP). Prefetch priority was given first to next line prefetching, then to eviction-based prefetching, and finally, to Enqueue CPF.

The second approach we looked at involves combining fetch directed prefetching with streaming buffers. This technique is aimed at exploiting more available bandwidth. We used fetch directed prefetching in combination with a single streaming buffer, with cache probe filtering on both mechanisms (Rem CPF + S1 CPF Stop). The stop filter was also employed with the streaming buffer.

7.2 Port Comparison

To determine the effect that the number of instruction cache ports would have on the performance of our prefetching schemes, we compare the results for instruction prefetching with both a single and a dual ported 16K 2-way associative instruction cache in figure 5.

A single ported instruction cache would mean lower instruction throughput from the fetch unit, but it would also mean higher levels of FTQ occupancy. This would in turn imply that fetch directed prefetching would be able to initiate prefetches farther ahead of a dual ported configuration (the longer an entry sits in the FTQ, the more the latency of the memory access can be hidden). However, less ports also means less opportunities to perform cache probe filtering. As can be seen in figure 5, most schemes had similar speedups on a high bandwidth architecture over their baseline results. The notable exceptions are those schemes with Enqueue CPF where performance goes down because of the lack of a free port to perform cache probe filtering.

Overall the single and dual ported configurations achieve similar speedups (relative to different IPCs). The dual ported configuration does a better job at cache probe filtering and the rate in which fetch blocks are processed is higher during normal instruction fetching than the single ported architecture. The single ported architecture still does very well, since the cache fetch blocks are processed slower, which results in a higher FTQ occupancy and more of the miss penalty can be masked when using fetch directed prefetching.

7.3 Cache Configurations

We now investigate the effect of different cache configurations on the various prefetching schemes. Figure 6 shows performance across three different cache configurations, for both high and low bandwidth buses. We examined the performance seen when using a 16K 2-way associative, a 16K 4-way associative, and a 32K 2-way associative instruction cache. Increasing the associativity of a cache should reduce the conflict misses, while increasing the size of a cache should reduce the capacity misses.

In figure 6, when adding more associativity we see a decline in the speedup obtained with all techniques, although the general trend of the results remains similar. For example, fetch directed prefetching with Remove Cache Probe Filtering dropped from a speedup of 40% to a speedup of 32% for the high bandwidth architecture, while the combination of next line prefetching, eviction-based prefetching, and Enqueue CPF dropped from a speedup of 35% to a speedup of 28%.

When increasing the size of the cache, the speedup obtained with fetch directed prefetching with Remove CPF falls to 18%. Although not shown, the cache miss filter performs well here, as it concentrates on conflict misses.

7.4 Cache Miss Rates

Figure 8 shows the percent of instruction cache fetch block misses for the original architecture without prefetching and for several of the prefetching architectures examined. Results are shown for a 16K 2-way associative cache. There is a definite correlation between the cache miss rate and IPC speedup. However other factors affect IPC speedup, such as data cache stalls due to low L2 cache bandwidth and the relative importance of instruction cache predictions.

Remove CPF consistently produces the lowest miss rate across all programs for the high bandwidth cases (2.3% on average) - and for most of the low bandwidth ones (3.6% on average). The bandwidth for Remove CPF could potentially be decreased significantly if an extra port was added to the tag array to further filter prefetch requests.

8 Related Work

In this section we describe related work to instruction cache prefetching not covered in section 6.

8.1 Fetch Directed Prefetching Research

Fetch directed prefetching was first proposed by Chen and Baer [3]. In their prefetching architecture they created a second PC called the *Look-Ahead PC*, which runs ahead of the normal instruction fetch engine. This LA-PC was guided by a branch prediction architecture, and used to index into a reference prediction table to predict data addresses in order to perform data cache prefetching. Since the LA-PC provided the address stream farther in advance of the normal fetch engine, they were able to initiate data cache prefetches farther in advance than if they had used the normal PC to do the address prediction. This allowed them to mask more of the data cache miss penalty. Chen and Baer only looked at using the LA-PC for data prefetching [3].

Chen, Lee and Mudge [2] examined applying the approach of Chen and Baer to instruction prefetching. They examined adding a separate branch predictor to the normal processor; so the processor would have 2 branch predictors, one to guide prefetching and one to guide the fetch engine. The separate branch predictor uses a LA-PC to try and speed ahead of the processor, producing potential fetch addresses on the predicted path. This separate branch predictor was designed to minimize any extra cost to the architecture. It only included (1) a global history register, (2) return address stack, and (3) an adder to calculate the target address.

In their design, each cycle the cache block pointed to by the LA-PC is fetched from the instruction cache in parallel with the normal cache fetch. If it is not a miss, the cache block is decoded to find the branch instructions and the target addresses are also calculated. When a branch instruction is found in the cache block it is predicted using the separate branch prediction structures, the LA-PC is updated, and the process is repeated. This whole process is supposed to speed ahead of the normal instruction fetch, but it is limited as to how far it can speed ahead because (1) the prefetch engine uses the instruction cache to find the branches to predict and to calculate their target addresses, and (2) their fetch directed prefetch engine has to stop following the predicted stream whenever the LA-PC gets a cache miss. When the LA-PC gets a cache miss, their prefetcher continues prefetching sequentially after the cache line that missed. In contrast, our fetch directed prefetching architecture follows the fetch stream prefetching *past* cache blocks that miss in the cache and does not need to access the instruction cache to provide predicted branch target and prefetch addresses since we have a decoupled front-end.

8.2 Out of Order Instruction Fetching

Stark et. al., [19] examined using a lockup-free instruction cache to fetch instructions, even in the presence of an instruction cache miss. The branch predictor would continue to produce one prediction per cycle, fetch the instructions, and put them into a result fetch queue out of order. Their idea is similar to our decoupled front-end design [14], except that there is no FTQ to allow the predictor to run ahead of the fetch unit, and they do not examine instruction prefetching. Therefore, they will not see any benefit from fetching out of order in the presence of cache stalls once the instruction window becomes full. In our approach, fetch directed prefetching can continue to prefetch until the prefetch buffer is full.

8.3 Code Placement

Many software techniques have been developed for improving instruction cache performance. Techniques such as basic block re-ordering and procedure placement [12], and reordering based on control structure [10] have all been shown to significantly improve instruction cache performance.

One goal of basic block reordering is to place basic blocks sequentially for the most likely path through a procedure. This can allow next-line and streaming buffer prefetching architectures to achieve better performance than what we saw in our results [21]. Examining the effects of code placement and software guided prefetching were beyond the scope of what we were able to investigate in this paper. Using these techniques in combination with fetch directed prefetching, our filtering techniques, and prior prefetching architectures is a topic for future research.

9 Summary

Decoupling the front-end of the processor with an FTQ between the branch predictor and instruction cache was shown in prior work to provide a scalable branch predictor design [14]. As we have shown in this paper, this same decoupling with the FTQ can allow efficient fetch directed instruction cache prefetching. We compared fetch directed prefetching with next-line prefetching and streaming buffers for different bus bandwidths, number of cache ports, and cache sizes and associativities.

The results show that NLP is still an attractive prefetching architecture achieving an average speedup of 20% while using only a small amount of bus bandwidth. Using one streaming buffer for prefetching provided 21% to 32%

speedup, while using only slightly more bus bandwidth than NLP. The streaming buffer performs better since it can continue to sequentially prefetch past a cache miss, which masks more of the miss penalty than the NLP architecture.

Fetch directed prefetching was shown to be a potential alternative to streaming buffers achieving the highest speedups of 25% to 40% on average, but using the highest amount of bus bandwidth. We examined using cache probe filtering to reduce the amount of wasted prefetches, and to save bus bandwidth. To reduce bus bandwidth further it may be beneficial to add an additional port to the instruction cache's tag array to provide additional cache probe filtering.

Another filtering approach we examined was the effect of marking evicted cache blocks in the branch prediction architecture, and to only prefetch those predicated cache blocks that are marked as evicted. Guiding fetch directed prefetching with only the evicted bit used the least amount of bandwidth and provided 11% speedup on average. If a processor's branch predictor can hold state for more branches than are held in the instruction cache, then implementing evicted prefetching can be a win for a fetch directed prefetching architecture.

The fetch directed prefetching architecture we examined in this paper uses a very simple FIFO prefetch buffer design. We are currently examining using a fully associative prefetch buffer with replaceable bits (as described in section 5). This will allow the prefetch buffer to hold prefetched cache blocks down mispredicted paths. In addition, we are currently investigating using fetch directed prefetching and the decoupled FTQ architecture to help provide fetch throughput for Simultaneous Multithreading. In this design, a separate FTQ is maintained for each hardware thread, allowing fetch directed prefetching to run ahead of each thread, which can potentially eliminate a significant number of instruction cache misses.

Acknowledgments

We would like to thank Norm Jouppi and the anonymous reviewers for providing useful comments on this paper. This work was funded in part by NSF CAREER grant No. CCR-9733278, NSF grant No. CCR-9711714, and a grant from Compaq Computer Corporation.

References

[1] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] I.K. Chen, C.C. Lee, and T.N. Mudge. Instruction prefetching using branch prediction information. In *International Conference on Computer Design*, pages 593–601, October 1997.

[3] T.F. Chen and J.L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 51–61, October 1992.

[4] K. Farkas and N. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[5] 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 *Proceedings of the First International Symposium on High-Performance Computer Architecture*, January 1995.

[6] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[7] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[8] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium of Computer Architecture*, pages 81–87, May 1981.

[9] C.-K. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *31st International Symposium on Microarchitecture*, December 1998.

[10] S. McFarling. Procedure merging with instruction caches. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6):71–79, June 1991.

[11] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[12] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):16–27, June 1990.

[13] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *29th International Symposium on Microarchitecture*, pages 165–175, December 1996.

[14] G. Reinman, T. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[15] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, October 1996.

[16] T. Sherwood and B. Calder. The time varying behavior of programs. Technical Report UCSD-CS99-630, University of California, San Diego, August 1999.

[17] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[18] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing*, November 1992.

[19] J. Stark, P. Racunas, and Y. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 34–45, December 1997.

[20] A. Veidenbaum, Q. Zhao, and A. Shameer. Non-sequential instruction cache prefetching for multiple-issue processors. *International Journal of High-Speed Computing*, 10(1):115–140, 1999.

[21] C. Xia and J. Torrellas. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In *23rd Annual International Symposium on Computer Architecture*, June 1996.

[22] T. Yeh. Two-level adpative branch prediction and instruction fetch mechanisms for high performance superscalar processors. Ph.D. Dissertation, University of Michigan, 1993.