Optimizations Enabled by a Decoupled Front-End Architecture

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

In the pursuit of instruction-level parallelism, significant demands are placed on a processor's instruction delivery mechanism. Delivering the performance necessary to meet future processor execution targets requires that the performance of the instruction delivery mechanism scale with the execution core. Attaining these targets is a challenging task due to I-cache misses, branch mispredictions, and taken branches in the instruction stream.

To counter these challenges, we present a fetch architecture that decouples the branch predictor from the instruction fetch unit. A Fetch Target Queue *(FTQ) is inserted between the branch predictor and instruction cache. This allows the branch predictor to run far in advance of the address currently being fetched by the cache. The decoupling enables a number of architecture optimizations including multi-level branch predictor design, fetch-directed instruction prefetching, and easier pipelining of the instruction cache. For the multi-level predictor, we show that it performs better than a single-level predictor, even when ignoring the effects of cycle-timing issues. We also examine the performance of fetch-directed instruction prefetching using a multi-level branch predictor and show that an average 19% speedup is achieved. In addition, we examine pipelining the instruction cache to achieve a faster cycle time for the processor pipeline, and show that pipelining provides an average 27% speedup over not pipelining the instruction cache for the programs examined.*

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

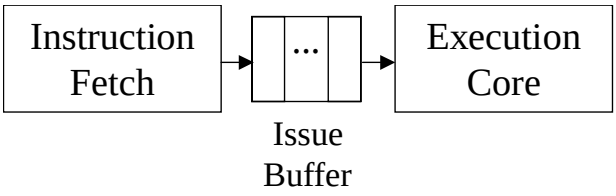

Figure 1: Two processing engines: the processor pipeline at a high level. The instruction fetch unit prepares and decodes instructions and supplies them to the issue buffer. The execution core consumes instructions from the issue buffer and then orchestrates their execution and retirement. The instruction fetch unit is a fundamental bottleneck in the pipeline: the execution core can only execute instructions as fast as the instruction fetch unit can prepare them.

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 is shown in Figure 1.

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. The trend towards exploiting more ILP in execution cores works to place further demands on the rate of instruction delivery from the front-end. Without complementary increases in front-end delivery performance, more exploitation of ILP will only decrease functional unit utilization with little or no increase in overall performance.

In this paper, we focus on improving the scalability and performance of the front-end by decoupling the branch predictor from the instruction cache. A *Fetch Target Queue* (FTQ) is inserted between the branch predictor and instruction cache, as seen in Figure 2. The FTQ stores predicted fetch addresses from the branch predictor, later to be consumed by the instruction cache. As we will show, the FTQ enables two primary classes of optimizations that can provide significant speedup in the processor.

First, the FTQ enables the branch predictor to run ahead of the instruction cache and provide a glimpse at the future stream of instruction fetch addresses. These addresses can then be used to guide a variety of PC-based predictors, such as instruction and data cache prefetchers, value predictors, and instruction reuse tables. Typically, these structures are accessed after the decode stage, but by bringing the predictor access before even the instruction cache is accessed, we are able to tolerate longer latency predictors. Furthermore, the fetch address stream made available is no longer constrained by the number of ports on the instruction cache. The rate at which predictions can be made determines the rate at which instructions are delivered to the FTQ, which can then be consumed by prediction mechanisms. A single ported instruction cache can only provide a cache block every cycle, but a high bandwidth branch predictor can provide several fetch blocks addresses each cycle. We investigate an instruction cache prefetching scheme in Section 4. In this scheme, we use the FTQ to guide prefetching into a fully-associative prefetch buffer.

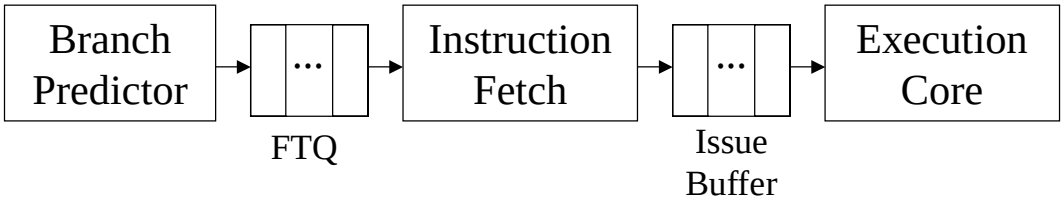

Figure 2: The decoupled front-end design at a high level. The fetch target queue (FTQ) buffers fetch addresses provided from the branch predictor. They are queued in the FTQ until they are consumed by the instruction fetch unit, which in turn produces instructions as in an ordinary pipeline. The FTQ allows the branch predictor to continue predicting in the face of an instruction cache miss. It also provides the opportunity for a number of optimizations, including multi-level branch predictor designs and fetch directed cache prefetching and predictor lookup.

We also investigate the use of different filtration methods to reduce the bus utilization of the prefetcher.

Secondly, the FTQ provides prediction buffering that can enable a multi-level branch predictor hierarchy. A multi-level branch predictor can provide highly accurate predictions while keeping the structure's access time low, much like a mult-level cache hierarchy. Entries stored in the FTQ can be consumed by the instruction cache while the branch predictor accesses higher levels of its hierarchy. This technique can be used with virtually any branch predictor, and may become even more useful as process technology sizes shrink and the access time for large structures grows [1, 23]. We discuss the benefits and implementation of a multi-level branch predictor in Section 5.

The remainder of this paper is organized as follows. Section 2 reviews some of the prior work in this area. In Section 3 we present the FTQ and the decoupled front-end it enables. In Section 4, we investigate the optimizations possible using the stream of fetch addresses stored in the FTQ. In Section 5 we examine the multi-level branch predictor design. In Section 6 we describe the methodology used to gather the results shown in Section 7. Section 8 provides a summary and concludes with future directions.

2 Related Research

Much work has been put into the front-end architecture in an effort to improve the rate of instruction delivery to the execution core. Techniques to reduce the impact of I-cache misses include multi-level instruction memory hierarchies [17] and instruction prefetch [40]. Techniques to reduce the impact of branch mispredictions include hybrid [21] and indirect [6] branch predictors, and recovery miss caches to reduce misprediction latencies [2]. A number of compiler-based techniques work to improve instruction delivery performance. They include branch alignment [5], trace scheduling [12], and block-structured ISAs [15].

We will now examine some of the more relevant prior work in detail.

2.1 Future Technology Scaling

Palacharla et al. [23] examined the effects of process technology scaling on the performance of the execution core. They found that the performance of operand bypass logic and datapaths scales poorly due to the large amount of interconnect in their datapaths.

Agarwal et al. [1] also examined the effects of process technology scaling, and found that current techniques of adding more transistors to a chip and superlinear clock scaling will provide diminishing returns to processor performance. Both studies show that larger memory performance scales worse than small memory because they are composed of significantly more interconnect. We examine the effect of this in Section 7.3.2.

2.2 Out of Order Instruction Fetching

Stark et. al. [39] proposed an out-of-order fetch mechanism that allows instruction cache blocks to be fetched in the presence of instruction cache misses. On 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.

The main difference between our approach and the one by Stark et. al. [39] is that our decoupled architecture can expose a larger address stream in front of the fetch unit enabling additional optimizations. A branch predictor can run farther ahead of the instruction cache, since it can produce fetch addresses for several cache blocks per prediction, which is more than the instruction cache can consume each cycle. These addresses can then be used for optimizations like prefetching, building a multi-level branch predictor, or making more scalable predictors (e.g., value prediction). In their approach [39], the branch predictor is tied to the lockup-free instruction cache, and produces addresses that are directly consumed by the instruction cache. This keeps the instruction cache busy and provides potential fetch bandwidth in the presence of instruction cache misses, but does not expose the large number of fetch addresses ahead of the fetch unit as in our decoupled design.

We call the number of instruction addresses stored in the FTQ at a given time the occupancy of the FTQ. Our results in Section 7 show that we can have a large FTQ occupancy even when the instruction window doesn't fill up and stall, since the branch predictor can predict more fetch blocks than the instruction cache can consume each cycle.

2.3 Fetch Guided Cache Prefetch

A form of fetch guided prefetching was first proposed by Chen and Baer [9] for data prefetching. 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 [9].

Chen, Lee and Mudge [7] 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 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 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 completely decouple the branch predictor from the instruction cache fetch via the FTQ. This is explained more thoroughly in Section 4.

2.4 Branch Prediction

Branch Target Buffers (BTB) have been proposed and evaluated to provide branch and fetch prediction for wide issue architectures. A BTB entry holds the taken target address for a branch along with other information, such as the type of the branch, conditional branch prediction information, and possibly the fall-through address of the branch.

Perleberg and Smith [26] conducted a detailed study into BTB design for single issue processors. They even looked at using a multi-level BTB design, where each level contains different amounts of prediction information. Because of the cycle time, area costs, and branch miss penalties they were considering at the time of their study, they found that the “additional complexity of the multi-level BTB is not cost effective” [26]. Technology has changed

since then, and as we show in this paper, a multi-level branch prediction design is advantageous.

Yeh and Patt proposed using a *Basic Block Target Buffer* (BBTB) [42, 43]. The BBTB is indexed by the starting address of the basic block. Each entry contains a tag, type information, the taken target address of the basic block, and the fall-through address of the basic block. If the branch ending the basic block is predicted as taken, the taken address is used for the next cycle's fetch. If the branch is predicted as not-taken, the fall-through address is used for the next cycle's fetch. If there is a BBTB miss, then the current fetch address plus a fixed offset is fetched in the next cycle. In their design, the BBTB is coupled with the instruction cache, so there is no fetch target queue. If the current fetch basic block spans several cache blocks, the BBTB will not be used and will sit idle until the current basic block has finished being fetched.

Several architectures have been examined for efficient instruction throughput including the two-block ahead predictor [33], the collapsing buffer [10], and the trace cache [31]. Seznec et. al., [33] proposed a high-bandwidth design based on two-block ahead prediction. Rather than predicting the target of a branch, they predict the target of the basic block the branch will enter, which allows the critical *next PC* computation to be pipelined. Conte et. al., [10] proposed the collapsing buffer as a mechanism to fetch two basic blocks simultaneously. The design features a multi-ported instruction cache and instruction alignment network capable of replicating and aligning instructions for the processor core. Rotenberg et. al., [31] proposed the use of a trace cache to improve instruction fetch throughput. The trace cache holds traces of possibly non-contiguous basic blocks within a single trace cache line. A start trace address plus multiple branch predictions are used to access the trace cache. If the trace cache holds the trace of instructions, all instructions are delivered aligned to the processor core in a single access. Patel et. al. [25], extended the organization of the trace cache to include associativity, partial matching of trace cache lines, and path associativity.

3 The Decoupled Front-end

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. The FTQ stores predictions from the branch prediction architecture, until these predictions are consumed by the instruction cache or are squashed. In this section we examine some of the issues related to the operation of the FTQ, including the occupancy of the FTQ (number of fetch addresses stored in the FTQ) and the speculative recovery mechanisms that we use in coordination with the FTQ.

3.1 FTQ Occupancy

The occupancy of the FTQ significantly contributes to the amount of benefit obtained from the decoupled front-end architecture. We define the occupancy of the FTQ to be the total number of branch predictions contained in the FTQ. If there are a large number of cache blocks represented by the predictions in the FTQ, the instruction cache will have plenty of predictions to consume, and we will have more flexibility in the branch predictor implementation (whether it be a multi-level design or a multi-cycle access design). Moreover, the higher the FTQ occupancy, the further ahead cache prefetching mechanisms and PC-based predictors can look into the future fetch stream of the processor.

High levels of occupancy can be obtained through the prediction of large fetch blocks, multiple predictions per cycle, instruction cache misses, and as a result of full instruction window. Larger fetch blocks mean that each prediction will carry more instructions, fundamentally increasing FTQ occupancy. Instruction cache misses delay consumption of FTQ entries, but the decoupled branch predictor will continue to produce predictions and fill the FTQ. An instruction window that is full due to data dependencies or even limited resources can slow the instruction fetch unit as well, thereby allowing the FTQ to fill. While these two latter situations are by no means desirable, they still can be taken advantage of to provide more FTQ occupancy.

In our study, we look at an FTQ that stores *fetch blocks*. A fetch block is a sequence of instructions starting at a branch target, and ending with a branch that has been taken in the past. Branches which are strongly biased not-taken may be embedded within fetch blocks. This type of block prediction was also examine by Michaud et. al. [22]. An FTQ could conceivably hold any form of branch prediction – though it is most useful when looking at larger prediction blocks, such as fetch blocks, as these will increase the occupancy of the FTQ.

3.2 Speculative Recovery Structures

For good predictor performance, especially for machines with deep speculation and large instruction windows, it becomes beneficial to recover branch history in the event the processor detects a mispredicted branch. This is even more important in our scalable front-end architecture design, because the branch predictor can get many predictions ahead of the instruction cache fetch. To facilitate the recovery of branch history, we proposed using a small *Speculative History Queue* (SHQ) in [27] to hold the speculative history of branches. When branches are predicted their updated global or local history is inserted into the SHQ. When predictions are made, the SHQ is searched in parallel with the L1 branch predictor — if a newer history is detected in the SHQ, it takes precedence over the current global history or the local history in the L1 branch predictor. When the branch at the end of a fetch block retires, its branch history is written into the global history register or the branch predictor for local history and its corresponding SHQ entry is removed. When a misprediction is detected, the entry in the SHQ at which the

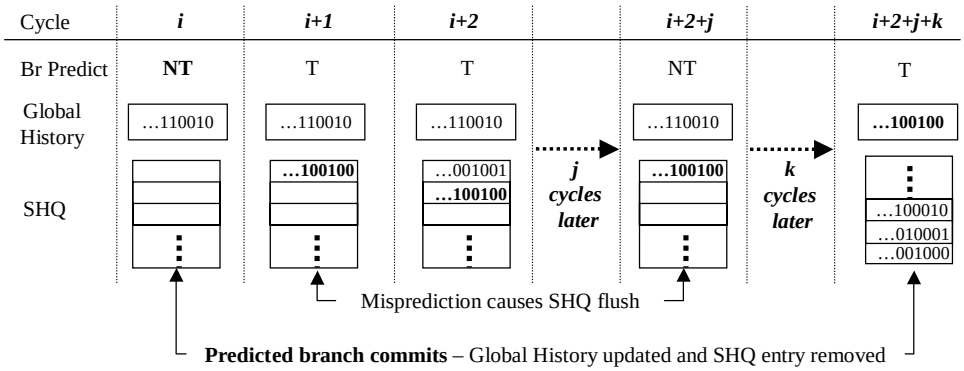

Figure 3: Speculative History Queue (SHQ) example for global history branch predictors. This figure shows the state of the non-speculative global history register (GHR) and the SHQ over several cycles of program execution. Cycles are delineated by vertical dotted lines and are labeled at the top of the figure. The current branch prediction is noted next, followed by the current state of the non-speculative global history register, followed by the current state of the SHQ. In this example, the SHQ contains the speculative state of the global history register. The trace begins at cycle *i*, where the SHQ is currently empty. Here, the current branch prediction is not taken (NT) and the global history register contains the current branch history up until the current prediction. In cycle *i+1*, the next prediction is taken (T) and the non-speculative global history has remained the same as no new branches have committed. However, a new history entry has been added to the SHQ to reflect the speculatively modified global history (from the prediction in cycle *i*). Similarly, the SHQ in cycle *i+2* contains a new entry that reflects the addition of branch prediction information from cycle *i+1*. In this example, the branch at cycle *i* is correctly predicted and the branch at cycle *i+1* is mispredicted. Therefore, at cycle *i+2+j*, all SHQ entries corresponding to predictions made after cycle *i* are flushed due to the misprediction at cycle *i+1*. As the mispredicted branch had never committed, the global history counter itself does not need recovery. Finally, in cycle *i+2+j+k*, once the correctly prediction branch in cycle *i* has committed, the global history register is updated with the information stored in the SHQ, and the entry corresponding to that branch is removed from the tail of the SHQ.

mispredicted branch occurred and all later allocated entries are released. The SHQ is kept small (32 entry) to keep it off the critical path of the L1 branch predictor. If the SHQ fills up, then the global history register and/or local history registers in the branch predictor are speculatively updated for the oldest entry in the SHQ. This allows new SHQ entries to be inserted, at the price of potentially updating the history incorrectly. Skadron *et al.*, independently developed a similar approach for recovering branch history, and they provide detailed analysis of their design in [36]. For our results, we found that there were around 8 entries in the SHQ on average. 94% of the time there were less than 32 entries in the SHQ on average for all benchmarks, and 74% of the time there were less than 16. Figure 3 gives an example of the operation of the SHQ for global history. The SHQ implementation, for other types of prediction history information (e.g., local history), is the same as what we describe in Figure 3 for global history.

Since the branch predictor can make predictions far beyond the current PC, it can pollute the return address stack if it predicts multiple calls and returns. It is necessary to use sophisticated recovery mechanisms to return the stack to the correct state. Simply keeping track of the top of stack is not sufficient [35], as the predictor may encounter several returns or calls down a mispeculated path that will affect more than just the top of stack. We

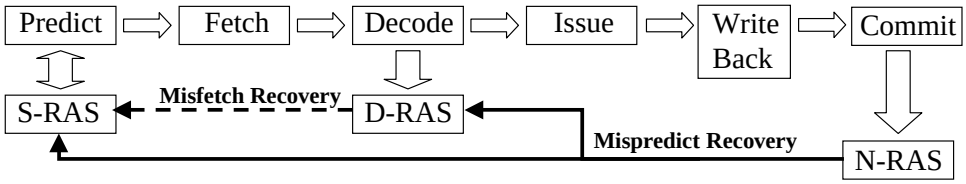

Figure 4: Simplified view of the processor pipeline showing the location of the S-RAS, D-RAS, and N-RAS. The S-RAS is updated during the branch prediction phase of the pipeline. The D-RAS is updated during the decode stage, and the N-RAS is updated during the commit stage. This means that the N-RAS will always have correct state, that the D-RAS could potentially be corrupted by a branch misprediction, and that the S-RAS could potentially be corrupted by a branch misfetch or misprediction. Therefore, the S-RAS can be recovered from the D-RAS in the case of a misfetch. And S-RAS and D-RAS can be recovered from the N-RAS in the case of a misprediction.

use three return address stacks to solve this problem. One is speculative (S-RAS) and is updated by the branch predictor during prediction. The next (D-RAS) is also speculative and is updated in the decode stage. The third is non-speculative (N-RAS) and is updated during the commit stage. The S-RAS can potentially be corrupted by branch misfetches and branch mispredictions. The D-RAS can potentially be corrupted by branch mispredictions - as misfetches will be detected in the decode stage. The N-RAS will not be corrupted by control hazards, as it is updated in commit. When a misfetch is detected, the S-RAS will likely be polluted and can be recovered from the D-RAS. When a misprediction is detected, the S-RAS and D-RAS will likely be polluted and can be recovered from the N-RAS. After either situation, prediction can restart as normal, using the S-RAS. This provides accurate return address prediction for a deep pipeline. This is summarized in the simplified pipeline in Figure 4.

4 FTQ Enabled PC-indexed Predictors

One of the main benefits of the FTQ is the ability to provide a glimpse into the future fetch stream of the processor. The stream of PCs contained in the FTQ can be used to access any PC-indexed predictor much earlier in the pipeline than in contemporary designs. The result of this prediction can be stored in the FTQ or in auxiliary structures until required.

In this section we give an example of using the stream of predicted addresses stored in the FTQ to improve performance. *Fetch-directed prefetching* (FDP) follows the predicted fetch stream, enqueuing cache prefetches from the FTQ. We now describe our FDP architecture and the heuristics that were used to better select which fetch blocks to prefetch.

4.1 A Fetch Directed Prefetching Architecture

Figure 5 shows the FDP architecture. As described in Section 3, 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

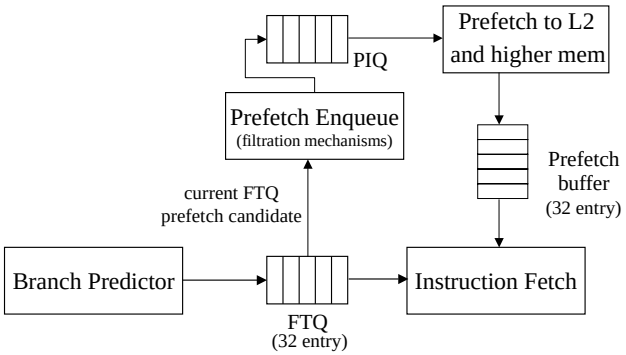

Figure 5: Fetch-directed prefetching architecture. The branch predictor provides a fetch stream to the FTQ, where it is consumed by the instruction fetch unit. While predictions are waiting in the FTQ, a prefetch enqueue mechanism can choose which cache blocks in the FTQ to predict. Cache blocks to be enqueued are stored in a prefetch instruction queue (PIQ) where they await prefetch from the L2 or main memory. Prefetches are performed into a 32 entry fully associative prefetch buffer. A variety of filtration techniques can be employed by the prefetch enqueue mechanism to limit the number of cache blocks prefetched. Both the instruction cache and prefetch buffer are searched in parallel for the cache block. An I-cache miss and prefetch hit, will move the cache block from the prefetch buffer into the I-cache.

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.

For this architecture, we assume the use of a high-bandwidth branch predictor that can provide large fetch blocks (as in our FTB design in Section 5). A fetch block from one prediction can potentially span 5 cache blocks in our design. Therefore, each FTQ entry contains five cache fetch block entries providing the fetch addresses for up to 5 cache blocks. Each fetch block entry in the FTQ contains a valid bit, a *candidate* prefetch bit, and an *enqueued* prefetch bit, as well as the cache block address. The candidate bit indicates that the cache block is a candidate for being prefetched. The 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 kept track of in a hardware-implemented pointer.

A fetch directed fully-associative prefetch buffer is added to the FDP architecture to hold the prefetched cache blocks. This is very similar to a predictor-directed stream buffer [34], except that it gets its prefetch addresses from the FTQ. Entries from the PIQ that are prefetched are allocated entries in the prefetch buffer. If the prefetch buffer is full, then no further cache blocks can be prefetched. When performing the instruction cache fetch, the prefetch buffer is searched in parallel with the instruction cache lookup for the cache block. If there is a hit in the prefetch buffer, it is removed from the buffer, and inserted into the instruction cache. On a branch misprediction, entries in the fully associative prefetch buffer are designated as replaceable, and removed only when no remaining free entries

exist in the FTQ. This way, prefetches down a mispredicted path can still be useful — especially in the case of short, forward mispredicted branches. We have also explored the use of a FIFO prefetch buffer that is flushed on a misprediction as a cheaper hardware alternative in [29].

We examined several approaches for deciding which FTQ entries to prefetch and insert into the PIQ, but in this paper we will only look at Cache Probe Filtering. Other approaches are discussed in [29].

4.2 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 canceled, 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, it may be beneficial to add an extra cache tag port for CPF, since this would only affect the timing of the tag array access, and not the data array.

An instruction cache port can become idle when (1) there is an instruction cache miss, (2) the current instruction window is full, (3) the decode width is exhausted and there are still available cache ports, or (4) there is insufficient fetch bandwidth. To use the idle cache ports to perform cache probe filtering during a cache miss, the cache needs to be lockup-free.

The first approach we examine, 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 prefetch 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* first checking the cache.

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.

4.3 Prior Hardware Prefetching Research

The following summarizes the prior hardware prefetching research, which we compare against our FDP architecture.

4.3.1 Tagged Next Line Prefetching

Smith [37] proposed tagging each cache block with a bit indicating when the next block should be prefetched. We call this *Next Line Prefetching* (NLP). 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 [38] 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. In our study, we examine using cache probe filtering with NLP. When the next sequential block is queued to be prefetched, we check whether the block is already in the cache - if an idle cache port is available. If no port is available, we initiate the prefetch anyway.

4.3.2 Stream Buffers

Jouppi proposed stream buffers to improve the performance of directed mapped caches [16]. If a cache miss occurs, sequential cache blocks, starting with the one that missed, are prefetched into a stream buffer, until the buffer is filled. The stream buffer is searched in parallel with the instruction cache when performing a lookup. He also examined using multiple stream buffers at the same time.

Palacharla and Kessler [24] improved on this design by adding a filtering predictor to the stream buffer. The filter only starts prefetching a stream if there are two sequential cache block misses in a row. This was shown to perform well for data prefetching. In addition, they examined using non-unit strides with the prefetch buffer.

Farkas et. al., [11] examined the performance of using a fully associative lookup on stream buffers. This was shown to be beneficial when using multiple stream buffers, so that each of the streams do not overlap, saving bus bandwidth.

We will present results for using the FTQ to guide instruction prefetching in Section 7. Next we will discuss other prediction techniques that could benefit from having the fetch PC stream in advance of instruction fetch.

4.4 Other PC-based Predictors

The stream of addresses stored in the FTQ could also be used to guide other PC-based predictors. Value and address prediction have been shown to be effective at reducing instruction latency in the processor pipeline [19, 20, 13, 14, 32, 41, 28].

A value predictor attempts to predict *the result of* or *the inputs to* a particular instruction. If the result of an operation is predicted, instructions that are currently waiting on the completion of the operation may speculatively execute. If the input to an instruction has predicted values, then the instruction can speculatively execute.

An address predictor attempts to predict the memory location that a load or store will access. The load or store

will speculatively execute with the predicted address.

The predictor tables used for these types of predictors can be quite complex and fairly large. The fetch addresses stored in the FTQ can allow the predictor access to initiate earlier in the pipeline, even before the corresponding instruction cache blocks are fetched, storing the result with the fetch block in the FTQ.

One example of this is context prediction [32]. In a context value predictor that predicts the results of an operation, the last *n* values of an instruction are stored in a first level table that is hashed by instruction PC. These values are used to create an index into a second level table that contains the actual value to be predicted. The table sizes used will likely require multiple cycles to access. But with the fetch addresses stored in the FTQ, we could conceivably initiate the predictor access earlier in the pipeline - storing the result in the FTQ itself if the prediction has not yet been consumed. This could allow even larger and more accurate predictors to be used, potentially even allowing these predictors to be located off-chip.

5 An FTQ Enabled Branch Predictor

The use of a decoupled design provides us with some flexibility in branch predictor design. Because the FTQ buffers predictions made by the branch predictor, it is possible to hide the latency of a multi-level branch predictor. Any branch predictor could take advantage of the multi-level approach, especially since future process technologies may favor smaller, faster structures [1, 23]. To fully take advantage of the FTQ, we want a branch predictor that is also capable of maintaining a high degree of FTQ occupancy. We chose to investigate a multi-level branch predictor hierarchy using our Fetch Target Buffer (FTB) [27].

Figure 6 shows the FTB design, which is an extension of the BBTB design by Yeh and Patt [42, 43] with three changes to their design. The first change is that we store fetch blocks rather than basic blocks. As mentioned in Section 3, fetch blocks may encapsulate one or more strongly biased not-taken branches, and so may contain multiple basic blocks. Therefore, an FTB entry is a form of a sequential trace, storing trace predictions with embedded fall-through branches.

The second change is that we do not store fetch blocks in our fetch target buffer that are fall-through fetch blocks, whereas the BBTB design stores an entry for *all* basic blocks, wasting some BBTB entries.

The third change we made to the BBTB design is that we do not store the full fall-through address in our FTB. The fall-through address is instead calculated using the *fetch distance* field in the FTB and the carry bit. The fetch distance really points to the instruction after the potential branch ending the FTB entry (fetch block). We store only the pre-computed lower bits of the fall-through address in the fetch distance along with a carry bit used to calculate the rest of the fall-through address in parallel with the FTB lookup [4]. This helps reduce the amount of storage

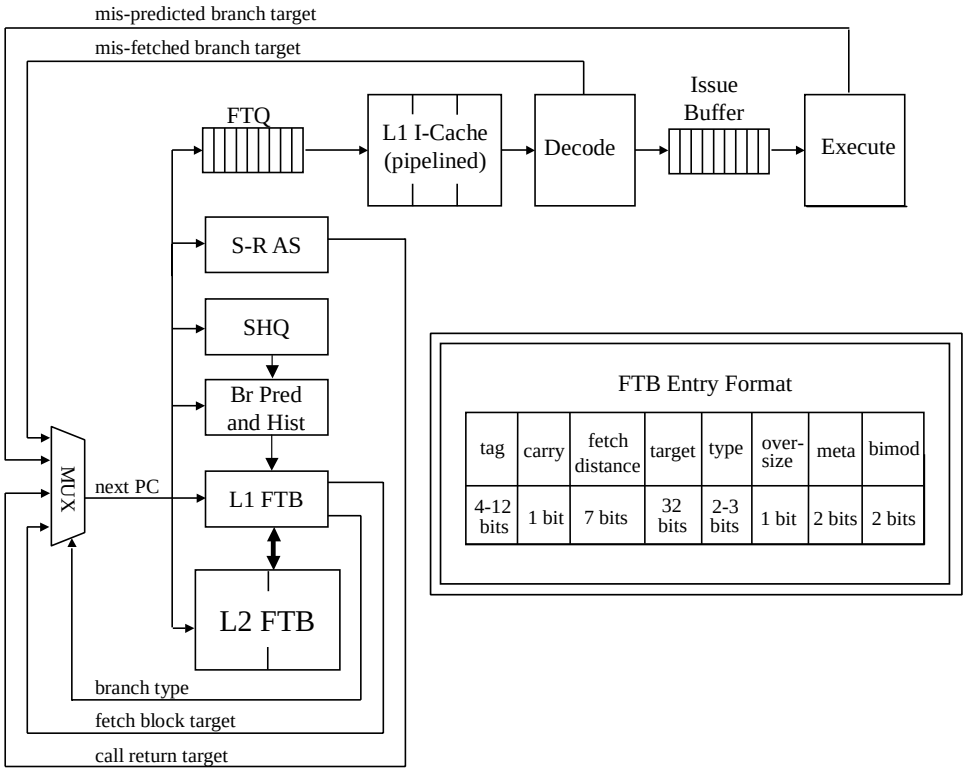

Figure 6: The decoupled front-end architecture with fetch target buffer. This figure elaborates upon the high-level pipeline from Section 1. The branch prediction unit, which feeds into the FTQ, is composed of a two-level fetch target buffer (*FTB*), a gshare predictor with global history (*Br Pred and Hist*), a speculative history queue (*SHQ*), and a speculative return address stack (*S-RAS*). The fetch target buffer predicts fetch addresses using prediction data from the gshare predictor and global history. The various fields of the FTB are shown in the diagram, and will be explained in detail in this section. Since the FTQ enables the FTB to predict far ahead of the current fetch stream, we use the SHQ and S-RAS to track and recover speculative state. Fetch addresses are stored in the FTQ, where they are then consumed by the instruction cache and decode hardware. Instructions are then supplied to the issue buffer, where they are executed and retired by the execution core.

for each FTB entry, since the typical distance between the current fetch address and the fetch block's fall-through address is not large.

5.1 FTB Structure

Our Fetch Target Buffer (FTB) design is shown in Figure 6. The FTB table is accessed with the start address of a fetch target block. Each entry in the FTB contains a tag, taken address, fetch distance, fall-through carry bit, branch type, oversize bit, and conditional branch prediction information. The FTB entry represents the *start* of a fetch block. The fetch distance represents the precomputed lower bits of the address for the instruction following the branch that ends the fetch block. The goal is for fetch blocks to end only with branches that have been taken during execution. If the FTB entry is predicted as taken, the taken address is used as the next cycle's prediction address. Otherwise,

the fall-through address (fetch distance and carry bit) are used as the next cycle's prediction address.

The size of the N bit fetch distance field determines the size of the sequential fetch blocks that can be represented in the fetch target buffer. If the fetch distance is farther than 2^N instructions away from the start address of the fetch block, the fetch block is broken into chunks of size 2^N, and only the last chunk is inserted into the FTB. The other chunks will miss in the FTB, predict not-taken, and set the next PC equal to the current PC plus 2^N, which is the max fetch distance. Smaller sizes of N mean that fetch blocks will be smaller - thereby increasing the number of predictions that must be made and potentially decreasing the FTQ occupancy. Larger sizes of N will mean less predictions and potentially higher FTQ occupancy, but will also mean that FTB misses will result in large misfetch blocks which can potentially pollute the instruction cache.

An oversize bit is used to represent whether or not a fetch block spans a cache block [42]. This is used by the instruction cache to determine how many predictions to consume from the FTQ in a given cycle. We simulated our results with two I-cache ports. The oversize bit is used to distinguish whether a prediction is contained within one cache block or if its fetch size spans two or more cache blocks. If the oversize bit is set, the predicted fetch block will span two cache blocks, and the cache will use its two ports to fetch the first two sequential cache blocks. If the bit is not set, the prediction only requires a single cache block, so the second port can be used to start fetching the target address of the next FTQ entry.

5.2 Branch Direction Predictor

The branch direction predictor shown in the FTB in Figure 6 is a hybrid predictor composed of a meta-predictor that can select between a global history predictor and a bimodal predictor. Other combinations are certainly possible, as well as non-hybrid predictors. The global history is XORed with the fetch block address and used as an index into a global pattern history table. The meta-prediction is used to select between the various predictions available, depending on the specifics of the design. The meta-predictor is typically implemented as a counter to select between two predictions or as a per-predictor confidence mechanism to select amongst three or more predictors [28]. The final prediction result is used to select either the target address of the branch at the end of the fetch block or the fetch block fall-through address.

The meta predictor and bimodal predictor values are not updated speculatively, since they are state machines and not history registers. The front-end can only assume it made the correct prediction and thus reinforce bimodal predictions. It has been shown in [18] that better performance results when such predictor updates are delayed until the result of the branch outcome is known, *i.e.*, at execution or retirement.

5.3 Functionality of the 2-Level FTB

The FTQ helps enable the building of a multi-level branch predictor, since the latency of the predictions from the L2 and higher predictors can be masked by the high occupancy of the FTQ. We will now describe the functionality of a 2-level FTB design.

The L1 FTB is accessed each cycle using the predicted fetch block target of the previous cycle. At the same time the speculative return address stack (S-RAS) and the global history prediction table are accessed. If there is an L1 FTB hit, then the fetch block address, the oversize bit, the last address of the fetch block, and the target address of the fetch block are inserted into the next free FTQ entry.

L1 FTB Miss and L2 FTB Hit If the L1 FTB misses, the L2 FTB needs to be probed for the referenced FTB entry. To speed this operation, the L2 FTB access begins in parallel with the L1 FTB access. If at the end of the L1 FTB access cycle a hit is detected, the L2 FTB access is ignored. If an L1 miss is detected, the L2 FTB information will return in $N - 1$ cycles, where N is the access latency of the L2 FTB (in L1 FTB access cycles). On an L1 FTB miss, the predictor has the target fetch block address, but doesn't know the size of the fetch block. To make use of the target address, the predictor injects fall-through fetch blocks starting at the miss fetch block address into the FTQ with a predetermined fixed length. Once the L2 FTB entry is returned, it is compared to the speculatively generated fetch blocks: if it is larger, another fetch block is generated and injected into the FTQ. If it is smaller, the L1 FTB initiates a pipeline squash at the end of the fetch block. If the fetch target has not made it out of the FTQ, then no penalty occurs. If the fetch target was being looked up in the instruction cache, those instructions are just ignored when the lookup finishes. The final step is to remove the LRU entry from the corresponding L1 FTB set, and insert the entry brought in from the L2 FTB. The entry removed from the L1 FTB, is then inserted into the L2 FTB also using LRU replacement.

L1 FTB Miss and L2 FTB Miss If the L2 FTB indicates the requested FTB entry is not in the L2 FTB, the L1 FTB enters a state where it continually injects sequential fetch blocks into the machine until a misfetch or misprediction is detected in the decode or writeback stage of the processor. Once a misfetch or misprediction is detected, the L1 FTB will be updated with the correct information regarding this new fetch block, and then the L1 FTB will once again begin normal operation.

6 Methodology

The simulators used in this study are derived from the SimpleScalar/Alpha 3.0 tool set [3], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, perform-

Program	Input	# fast fwd (M)	% br exe	2 Ports						1 Port	
				16K-2W		16K-4W		32K-2W		16K-2W	
				MR	IPC	MR	IPC	MR	IPC	MR	IPC
deltablue	ref	0	17.0	0.5	1.73	0.2	1.78	0.0	1.80	0.6	1.68
gcc	1cp-decl	400	17.4	4.6	1.25	3.8	1.28	1.8	1.48	4.6	1.18
groff	someman	0	17.3	4.4	1.78	3.0	1.91	1.6	2.21	4.4	1.65
gs	ref	0	7.1	0.1	4.98	0.0	5.00	0.0	5.02	0.1	4.54
go	5stone21	2000	13.4	2.6	1.34	1.4	1.43	0.4	1.55	2.7	1.28
ijpeg	specmun	2000	4.7	0.0	4.00	0.0	4.00	0.0	4.00	0.0	3.82
m88ksim	ref	2000	19.7	2.4	2.04	1.1	2.39	1.0	2.39	2.5	1.90
perl	scrabbl	2000	17.1	3.4	2.04	2.3	2.26	0.6	2.64	3.5	1.82
vortex	vortex	2000	14.7	9.4	1.44	6.9	1.54	3.6	1.95	9.5	1.36
average				3.0	2.29	2.1	2.40	1.0	2.56	3.1	2.14

Table 1: Program statistics for the baseline architecture. Each program is shown, along with the data set used, number of instructions fast forwarded (in millions) to avoid initialization effects, and the dynamic percentage of branches executed. The next eight columns show the miss rate and IPC for the different cache configurations we investigated. Columns 5 and 6 show the miss rate and IPC for our default configuration – a 16K 2-way associative instruction cache with 2 read/write ports. Columns 7 and 8 show the miss rate and IPC for a cache with a higher associativity – a 16K 4-way associative cache with 2 read/write ports. Columns 9 and 10 show the miss rate and IPC for a larger cache – a 32K 2-way associative cache with 2 read/write ports. The final two columns show the miss rate and IPC for a single ported instruction cache – a 16K 2-way associative cache with 1 read/write port.

ing 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 mis-prediction.

To perform our evaluation, we collected results for nine benchmarks. We examined six of the SPEC95 C benchmarks and three additional benchmarks. `Groff` is a text formatting program and `deltablue` is a constraint solving system. 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 data set we used in gathering results for each program, the number of instructions executed (fast forwarded) before actual simulation (in millions), and the percent of executed branches in each program. Also shown is the miss rate and IPC for three different cache configurations. Each program was simulated for up to 200 million instructions.

6.1 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 (from up to two cache blocks) and issue up to 16 instructions per cycle. The instruction cache is dual ported, and each port can fetch from a single cache block each cycle. The architecture has a 128 entry re-order buffer with a 32 entry load/store buffer. Loads can only execute when all prior store addresses are known. In addition, all stores are issued in-order with respect to prior stores. 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 mis-prediction 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.

The processor we simulated has a dual ported 16K 2-way set-associative direct-mapped instruction cache and a 32K 4-way set-associative data cache. Both caches have block sizes of 32 bytes, and have a 3 cycle hit latency. There is a unified second-level 1 MB 4-way set-associative cache with 64 byte blocks, with a 12 cycle cache hit latency. If there is a second-level cache miss it takes a total of 120 cycles to make the round trip access to main memory. The L2 cache has only 1 port. The L2 bus is shared between instruction cache block requests and data cache block requests. We modified SimpleScalar to accurately model L1 and L2 bus utilization. We model a pipelined memory/bus, where a new request can occur every 4 cycles, so each bus can transfer 8 bytes/cycle.

There is a 32 entry 8-way associative instruction TLB and a 32 entry 8-way associative data TLB, each with a 30 cycle miss penalty.

For this paper, we used the McFarling hybrid predictor [21] for our conditional branch predictor. The predictor has a 2-bit meta-chooser and a 2-bit bimodal predictor, both stored in the branch predictor entry with their corresponding branch. In addition, a tagless gshare predictor is also available, accessed in parallel with the branch predictor. The meta-chooser is incremented/decremented if the bimodal/gshare predictors are correct. The most significant bit of the meta-chooser selects between the bimodal and gshare predictions.

6.2 Timing Model

In order to get a complete picture of the impact of our front-end architecture, we also need to investigate the cycle time of the architecture. IPC results obtained through SimpleScalar can be combined with timing analysis to provide results in *Billion Instructions Per Second (BIPS)*. Architects need to consider the cycle time of the processor in addition to the number of instructions that can be performed in a single cycle (IPC). Since IPC is a relative measurement (as it is given in cycles), BIPS results are often more useful in determining the overall performance of an architecture – including the impact that large predictors and caches may have on the cycle time of the processor. A technique that provides a 10% improvement in IPC may not be worthwhile if it extends the processor cycle time and reduces the BIPS of the architecture. The BIPS results that we show are an upper bound on the instruction rate for the processor, as it is possible for other components of the pipeline that are not included in the front-end to influence the cycle time of the processor.

FTB num entries	t_{clk} (ns)
64	0.80
128	0.81
256	0.84
512	0.86
1024	0.90
2048	1.00
4096	1.11
8192	1.23

(a)

Cache Size	Assoc	t_{clk} (ns)
16K	2	1.20
16K	4	1.33
32K	2	1.34
32K	4	1.50

(b)

Table 2: Timing data from *CACTI* version 2.0. Table (a) shows timing data for various first level FTB configurations. For each FTB specification (shown as the number of entries in a 4-way associative FTB), the cycle time in nanoseconds computed with *CACTI* is shown. Table (b) shows the cycle times for a variety of cache configurations. The first column specifies the size of the cache. The second column specifies the associativity.

The timing data we need to generate results in BIPS is gathered using the *CACTI* cache compiler version 2.0 [30]. *CACTI* contains a detailed model of the wire and transistor structure of on-chip memories. We modified *CACTI* to model our FTB design. *CACTI* uses data from $0.80\mu m$ process technology and can then scale timing data by a constant factor to generate timings for other process technology sizes. We examine timings for the $0.10\mu m$ process technology size. However, this scaling assumes ideal interconnect scaling, unlike the model used in [27]. This provides a lower bound on the performance improvement of our multi-level predictor.

Table 2 contains the timing parameters for the FTB and cache configurations we examined for the $0.10\mu m$ technology size. Table 2(a) lists the FTB sizes (in number of entries) and the *CACTI* timing data in nanoseconds for each configuration, showing the access time in nano-seconds. All FTB organizations are 4-way set-associative. Table 2(b) lists the timing data for the instruction and data cache configurations we examined, showing the access time in nano-seconds.

In our study, we assume that the branch predictor would need to be accessed once per cycle. Therefore, we set the cycle time of the processor to the access time of the FTB. Then, the instruction and data caches are pipelined according to the cycle time of the processor. For example, a processor with a 1K entry FTB would have a cycle time around .9 ns at the $0.10\mu m$ feature size. A 16K 2-way associative dual-ported instruction cache would need to have two pipeline stages to accommodate this cycle time. To *calculate BIPS*, we simulate the particular architecture using SimpleScalar to provide an IPC value, and then divide this by the cycle time derived from *CACTI* data for the $0.10\mu m$ feature size.

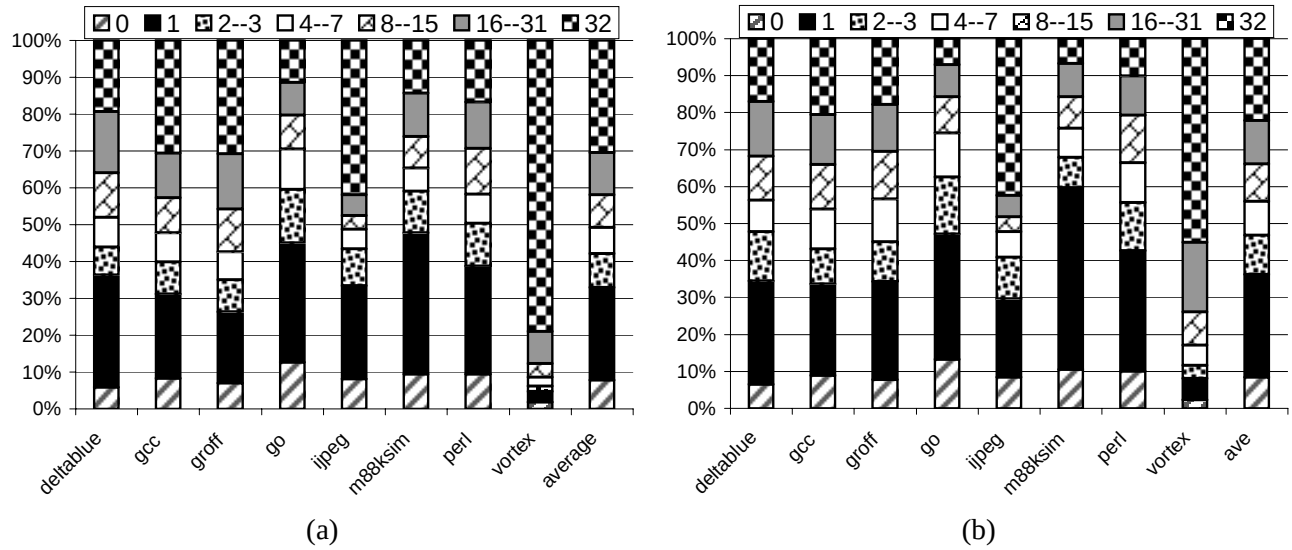

Figure 7: FTQ occupancy histograms. These graphs show the percent of executed cycles the 32 entry FTQ used in this study had a given number of predictions stored within it after instruction fetch. The disjoint categories of FTQ occupancy are shown in the legend at the top of the graphs. For example, the white section represents the percent of time that the FTQ held between 4 to 7 FTQ entries. Graph (a) corresponds to a single level branch predictor, and graph (b) corresponds to a two level branch predictor.

7 Results

In this section we examine results for the FTQ-enabled optimizations we described. First we examine the ability of the FTQ to fill up ahead of the fetch stream. Then we provide results for our fetch directed prefetching (FDP) architecture enabled by having an FTQ. Finally, we compare single-level and multi-level FTB predictors. The metrics used and base architecture are as described in Section 6.

7.1 FTQ Occupancy

As discussed in Section 3.1, the greater the occupancy of the FTQ, the greater the potential benefit provided by the FTQ. Figure 7 shows a breakdown of the occupancy of the FTQ for both a single level (a) and a two level (b) branch predictor. This figure demonstrates the percent of cycles that the FTQ held a given number of entries with predictions. For example, on average for the single level predictor, the FTQ only buffered a single prediction 25% of the time. It held a full 32 predictions 31% of the time. The two level results (b) show slightly less occupancy due to the fact that the second level occasionally corrects the first level and squashes some predictions in the FTQ. However, the two level predictor is more accurate, as will be shown in Section 7.3.2. Also, the FTQ is completely flushed on a branch misprediction.

7.2 Fetch Directed Prefetching Results

In [29], we investigated the performance of our prefetching architecture with an accurate, single level branch predictor. In this study, we show that we can achieve the same high levels of performance with a 2-level branch predictor. We chose to simulate our prefetching architecture with a 128 entry first level FTB and a 8192 entry second level FTB as will be examined in Section 7.3. We will show how this configuration was one of the best BIPS performers and had a high average prediction accuracy. Unless otherwise indicated, we used a 32 entry FTQ for results in this section. This gave us a large window to perform instruction prefetch.

Figure 8 shows the performance of our fetch-directed prefetching techniques, and prior Next-Line and Stream Buffer (MSB) prefetching techniques for an architecture with an 8 byte/cycle bus to the L2 cache. Unfiltered fetch-directed (NoFilt) prefetching provides a speedup of 13% due to contention for limited bandwidth. Tagged next line prefetching (NLP) with cache probe filtering performs slightly better. An architecture with one stream buffer (MSB1) using cache probe filtering provides a 14% speedup in IPC. The stream buffer architecture we implemented contained 4 entries, and prefetched sequential cache blocks after a cache miss. The results show that adding a second 8 entry stream buffer (MSB2), actually degraded performance in some instances.

Remove CPF performs the best on average with a speedup in IPC of 19%. Enqueue CPF achieves an IPC speedup of 18% on average. It would be expected that as bandwidth becomes more and more scarce, a conservative prefetching approach like enqueue CPF would begin to outperform a more aggressive technique like remove CPF, especially if data prefetching is performed at the same time. `Vortex` achieves a large speedup due to a relatively high instruction cache miss rate (more room for improvement), a large amount of FTQ occupancy, and highly accurate branch prediction. `Deltablue`, while having accurate branch prediction and high occupancy, does not have a high instruction cache miss rate. Moreover, it has a relatively high amount of bus utilization (as shown in table 3) due to a large number of data cache misses, so aggressive prefetching can degrade performance by interfering with demand data cache misses

This data is further explained by the results in table 3. This shows the amount of bus utilization that our prefetching techniques consume. FDP with remove CPF provides a small reduction in bus utilization over unfiltered fetch-directed prefetching, 69% on average down to 64%. Enqueue CPF provides a significant reduction, with an average 42% bus utilization, whereas an architecture with no prefetching has a bus utilization of 20%.

In Figure 9 we examine the performance of our prefetching architecture across three different cache configurations (16K 2-way, 16K 4-way, and 32K 2-way). The trends in performance are similar across all three configurations, but the magnitude of the benefit decreases for larger or more associative caches due to the decreased miss rate. Even so, for a 32K 2-way associative instruction cache with an 8 byte/cycle bus to the L2 cache, we are able to obtain

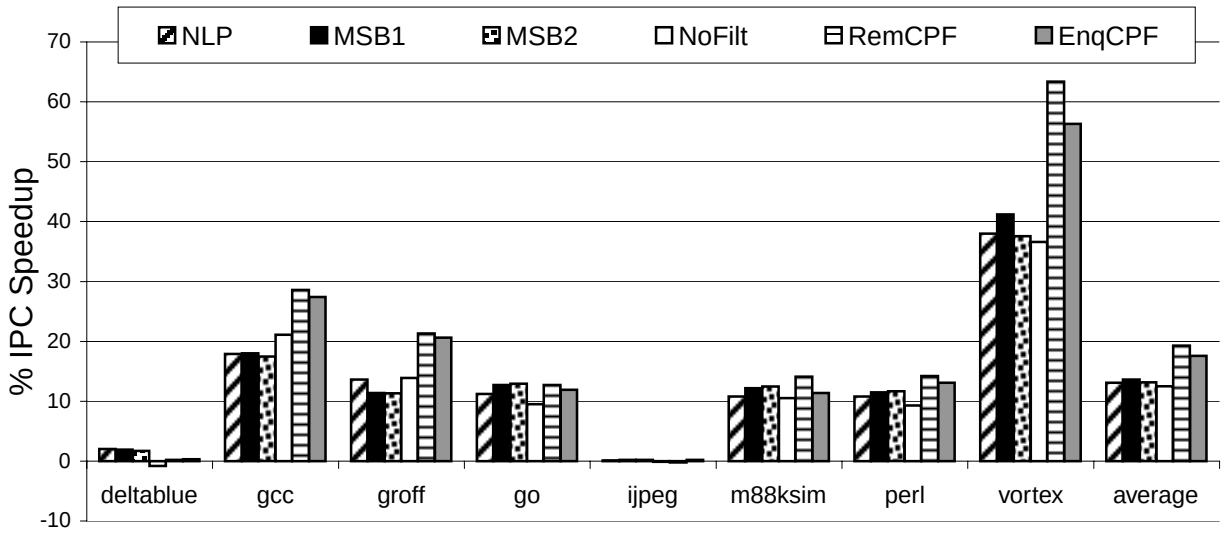

Figure 8: The performance of our prefetching techniques compared to prior work for an architecture with an 8 byte/cycle bus to the L2 cache. Results are shown as percent speedup in IPC over a baseline architecture. The first bar represents next line prefetching (NLP) with cache probe filtering. The second two bars represent multiple stream buffers (MSB1 and MSB2 - single and dual stream buffer configurations) with cache probe filtering. The fourth bar represents FDP with no cache probe filtering. The fifth bar represents FDP with remove CPF. The final bar represents FDP approach with enqueue CPF.

	Base			NoFilt				RemCPF				EnqCPF			
program	Bus Util	% i$	% d$	Bus Util	% pref	% i$	% d$	Bus Util	% pref	% i$	% d$	Bus Util	% pref	% i$	% d$
deltablue	45.5	2.4	43.2	80.0	31.5	1.8	46.7	73.9	25.8	1.3	46.9	63.2	15.4	1.5	46.2
gcc	21.5	17.5	4.0	69.5	52.4	11.4	5.7	64.5	49.3	9.3	5.9	47.9	33.1	9.2	5.5
groff	17.7	16.5	1.2	71.2	57.5	12.1	1.6	65.7	54.4	9.7	1.7	46.6	35.7	9.3	1.6
go	15.1	12.0	3.1	59.7	48.2	7.5	4.0	56.5	46.4	6.0	4.1	31.5	21.2	6.6	3.7
ijpeg	3.1	0.1	3.0	67.2	63.4	0.0	3.8	62.9	59.1	0.0	3.7	25.5	22.2	0.0	3.2
m88ksim	12.6	12.4	0.2	46.0	38.2	7.6	0.3	41.6	34.7	6.7	0.3	26.1	17.4	8.5	0.2
perl	15.2	14.0	1.2	66.1	54.0	10.6	1.5	60.5	50.5	8.5	1.5	33.4	23.3	8.7	1.4
vortex	27.2	24.8	2.4	92.9	68.0	20.9	4.0	84.9	64.9	15.2	4.8	60.7	42.2	14.2	4.3
ave-8	19.7	12.5	7.3	69.1	51.7	9.0	8.4	63.8	48.1	7.1	8.6	41.8	26.3	7.3	8.3

Table 3: This table shows bus utilization for the base configuration for fetch-directed prefetching. The first three columns show results for the base configuration. The next four columns show results for unfiltered fetch directed prefetching. The next four show results for remove CPF, and the final four for enqueue CPF. Each set of results shows the total percent bus utilization, the percent bus utilization by the instruction cache (i$), and the percent bus utilization by the data cache (d$). In addition, the prefetching techniques have the percent bus utilization used by the prefetching architecture (% pref). Results shown are for an 8 byte/cycle bus to the L2 cache.

10% speedup using remove CPF on average.

In table 4, we show the range of speedups obtainable through prefetching when we vary the size of the FTQ between 2 and 64 entries. The larger the FTQ, the further ahead a prefetch can potentially be made, but the more inaccurate the prefetch may potentially be. This tradeoff can be seen in the performance of the unfiltered fetch-directed prefetching approach. Performance improves when going from a 2 entry FTQ to a 16 entry FTQ, but drops

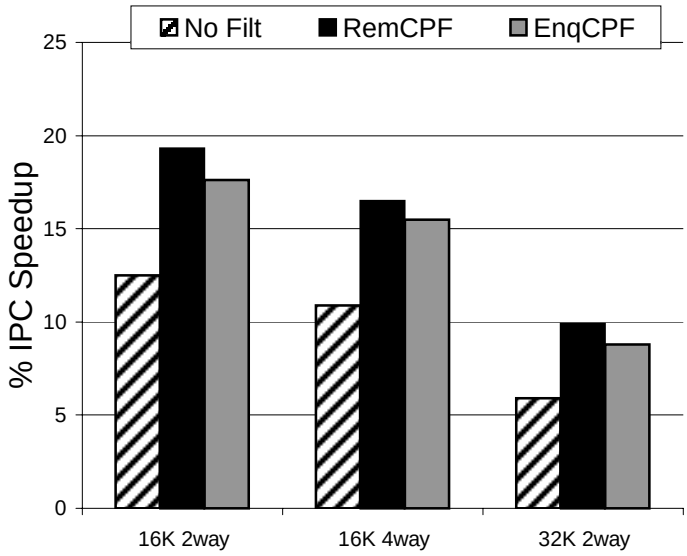

Figure 9: This figure depicts a comparison of three different dual ported cache configurations. The first bar in each cluster represents the average speedup obtained for fetch-directed prefetching without filtration. The second bar shows results for remove CPF. The third shows results for enqueue CPF.

	FTQ size																	
	2			4			8			16			32			64		
program	No Filt	Rem CPF	Enq CPF	No Filt	Rem CPF	Enq CPF	No Filt	Rem CPF	Enq CPF	No Filt	Rem CPF	Enq CPF	No Filt	Rem CPF	Enq CPF	No Filt	Rem CPF	Enq CPF
deltablue	0.0	1.0	1.0	-0.7	0.3	1.1	-0.5	0.6	1.1	-0.6	0.3	0.5	-0.8	0.2	0.3	-1.5	-0.1	-0.1
gcc	14.9	18.9	7.5	19.1	24.7	14.1	20.8	27.3	21.4	21.2	28.3	26.0	21.1	28.6	27.4	21.0	28.5	27.4
groff	8.1	14.2	4.4	13.7	19.5	10.1	14.3	22.3	16.0	15.6	23.8	21.3	13.9	21.3	20.6	14.0	22.0	20.9
go	7.7	9.7	5.2	8.7	11.2	8.2	9.2	12.2	10.7	9.5	12.6	11.7	9.5	12.7	11.9	9.5	12.7	11.8
ijpeg	-0.9	-0.9	-0.5	-0.6	-0.6	-0.2	-0.1	-0.1	0.3	-0.2	-0.2	0.2	-0.1	-0.2	0.2	-0.1	-0.2	0.2
m88ksim	5.9	8.3	3.2	8.9	12.3	7.2	8.8	14.4	10.1	9.8	13.3	11.2	10.5	14.1	11.4	10.4	14.2	11.2
perl	5.3	8.9	4.0	7.8	11.5	7.8	8.6	13.3	11.8	9.2	14.3	12.8	9.3	14.2	13.1	8.4	14.6	13.5
vortex	29.5	38.2	13.7	31.0	43.1	18.8	34.0	52.6	26.8	36.3	60.3	40.9	36.6	63.4	56.3	36.3	65.7	64.8
average	8.8	12.3	4.8	11.0	15.2	8.4	11.9	17.8	12.3	12.6	19.1	15.6	12.5	19.3	17.6	12.3	19.7	18.7

Table 4: This table shows the performance of the three FDP architectures for a variety of FTQ sizes. Results are given as the percent speedup in IPC over the base configuration. The first set of three columns shows the performance of unfiltered fetch directed prefetching, remove CPF, and enqueue CPF respectively - all for a 2 entry FTQ. The next set of three columns represents the same prefetching techniques for a 4 entry FTQ. The remaining clusters of four columns represent 8, 16, 32, and 64 entry FTQs respectively.

slightly when the FTQ continues to grow in size. The performance of both CPF techniques continue to improve with increasing FTQ size, but the rate of performance increase drops for larger sized FTQs. Going to a 64 entry FTQ from an FTQ of size 32, only provides a minimal speedup for remove CPF and enqueue CPF. The additional complexity and size of the FTQ at 64 entries did not seem to justify this amount of speedup. We decided to use the 32 entry FTQ for our prefetching results, as it provided a significant amount of speedup, but still had a small enough structure that would not be likely to affect cycle times.

Figure 7 is also useful to explain the performance of our prefetching scheme on certain benchmarks. `Vortex`, which enjoys a considerable speedup with our prefetching scheme has a significant amount of FTQ occupancy – it has a full FTQ around 55% of the time. This is due to both a high prediction accuracy and a large average fetch

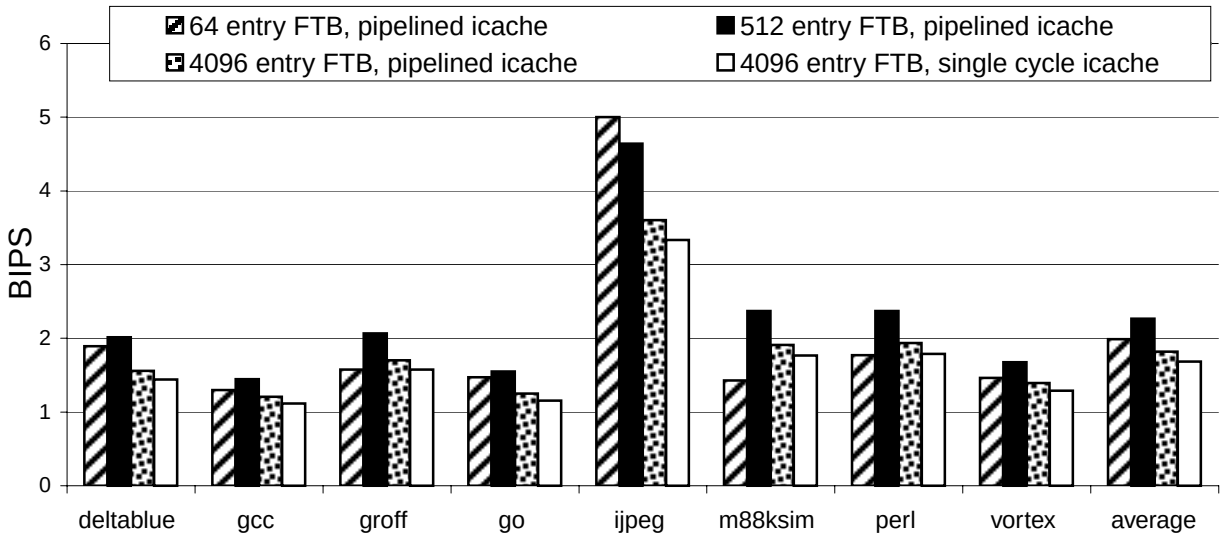

Figure 10: Billion Instructions per Second (BIPS) comparison across four FTB configurations. BIPS results were calculated using IPC values from SimpleScalar simulations and CACTI timing data. The first three bars for each benchmark represent single level FTB configurations with 64, 512, and 4096 entries respectively. These configurations have 2 cycle pipelined instruction cache, and the cycle time for each is set according to the cycle time of the FTB (see table 2 for timing data). The final bar represents an FTB with 4096 entries that does not have a pipelined instruction cache. In this case, the cycle time is set to the cycle time of the instruction cache.

distance size. Programs like `m88ksim` tend toward lower FTQ occupancies, due to smaller average fetch distance sizes.

7.3 Multi-level Predictor Results

We now examine the performance of a two-level FTB predictor, which benefits from having an FTQ.

7.3.1 Single-level Predictor Results

We first will motivate the potential benefit from pipelining the instruction cache for future processors. Figure 10 shows the Billion Instructions per Second (BIPS) results for three FTB configurations with pipelined (2 cycle) instruction cache, and an FTB configuration with a single cycle instruction cache. The first three FTB designs with a pipelined I-cache use a cycle time equal to the FTB access time, and the non-pipelined I-cache uses a cycle time equal to the I-cache access time as shown in Figure 2. The 512 entry FTB with the pipelined I-cache has an average BIPS of 0.28, and is the best single level performer, as it balances a moderate cycle time with a high IPC. The 4096 entry FTB with the single cycle instruction cache avoids lengthening the branch misprediction penalty by a cycle (pipelining the instruction cache in these experiments extends the pipeline by one stage). However, as can be seen, the increase in cycle time has more of an impact on BIPS, and the average BIPS for the single cycle instruction cache case is 0.22.

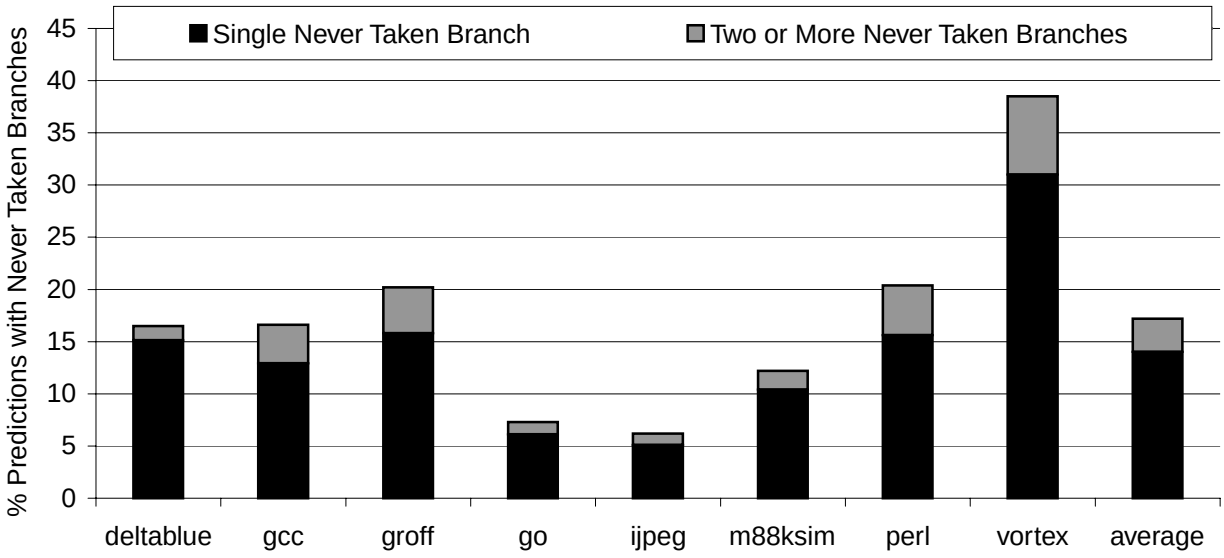

Figure 11: Percent of predictions from the FTB that span multiple basic blocks. The x-axis shows the benchmarks we examined, and the y-axis shows the percent of predictions that contain one or more never taken branches. The black bar shows the percent of predictions that contain a single never taken branch and the grey bar shows the percent of predictions that contain two or more never taken branches.

Next, we examine the ability of the FTB to encapsulate never taken branches. Figure 11 shows that on average, 14% of predictions include two basic blocks (i.e. include a single never taken branch) and an additional 3.2% of predictions include more than two basic blocks (i.e. include two or more never taken branches). Predictions in `vortex` span multiple basic blocks nearly 40% of the time. Michaud et al. [22] also examined predictors that are capable of bypassing a single not taken branch and found an average fetch rate of 1.3 basic blocks per cycle for an ordinary BTB.

7.3.2 Two-level Predictor Results

Figure 12 shows results in instructions per cycle (IPC) varying the size of the FTB for single and dual level FTBs, for the benchmarks we examined. For most benchmarks, the increase in predictor size results in increased performance. Benchmarks like `perl` and `m88ksim` are more affected by FTB size. For most, the difference between a 1K and a 4K entry FTB is minimal. The results for `ijpeg` show that this benchmark does not contain a significant amount of branch state. It has few taken branches encountered (only 4.7% of instructions are branches at all) and therefore requires very little space in the FTB. This, in addition to its low instruction cache miss rate, helps to explain the relatively high IPC obtained with this benchmark. The two level results demonstrate that the second level FTB is improving the prediction accuracy of the first level FTB.

Figure 13 shows results in Billion Instructions per Second (BIPS) for single level (64 entry to 4K entry) and two level FTB designs (64 to 512 entry first level table with an 8K entry 2nd level table) assuming perfect scaling. For

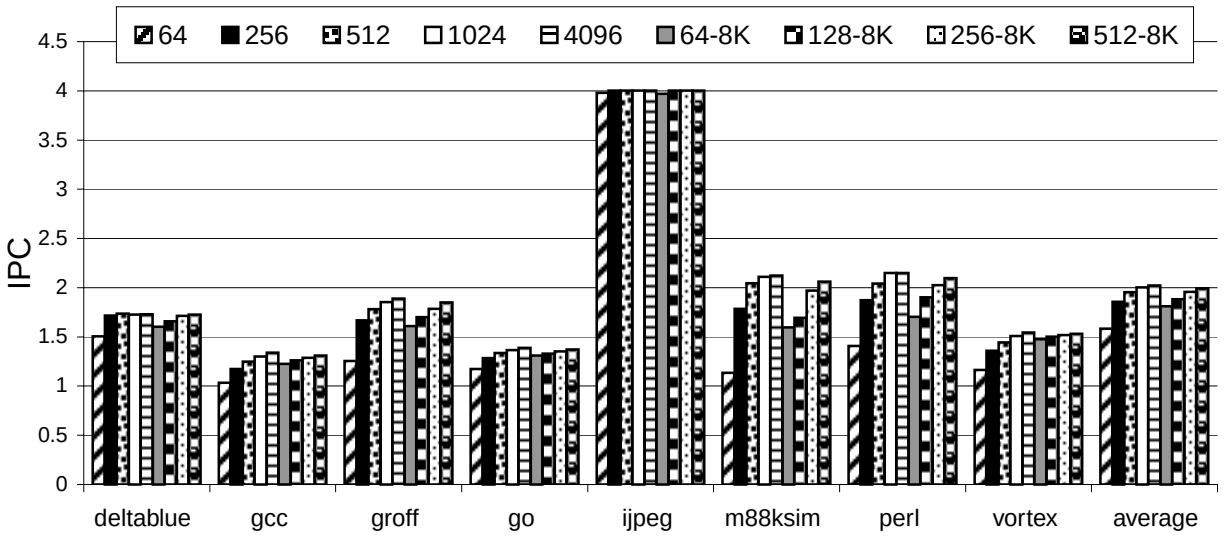

Figure 12: Instructions per cycle (IPC) comparison across a variety of FTB configurations. The first five bars represent single level FTB configurations: 64, 256, 512, 1024, and 4096 entry first level FTBs. The next four bars represent dual level FTB configurations: 64, 128, 256, and 512 entry first level FTBs, each with an 8192 entry second level FTB.

most benchmarks, the 512 entry FTB is the best single level performer, with a cycle time of 0.28ns. The exception is the performance of `ijpeg` for the above mentioned reasons. The second level results are very closely clustered on average. The best performer on average was the configuration with a 128 entry first level FTB and 8K entry second level FTB. But for some benchmarks, the best performance was seen with a 512 entry first level FTB. The most dramatic of these was `m88ksim`. This benchmark has a relatively low FTQ occupancy (due to a small average fetch distance), and therefore is not able to tolerate the latency to the second level FTB as well as other benchmarks. These results show that the two level FTB performs slightly better on average than a single level design in the absence of the interconnect scaling bottleneck (i.e. assuming ideal technology scaling).

When taking into consideration the interconnect scaling bottleneck, we found that the best performing two level FTB design provided a 14.6% improvement in BIPS over the best performing single level design. Figure 14 also shows results in Billion Instructions per Second (BIPS), but uses the technology scaling calculations from [27] as shown in Table 5. These results show the potential effects of the interconnect scaling bottleneck. This scaling assumes that wire latency scales at a rate less than the processor feature size due to the effects of parasitic capacitance. These results show an even more substantial difference in BIPS between single and two level FTB designs.

These results show that IPC does not provide the full picture of processor performance. The BIPS results in 10 and 13 show that if the FTB access time determines the cycle time of the processor, then a 512 entry FTB provides the best average performance of the single level FTB designs and the 128 entry first level and 8K entry second level FTB provides the best average performance of the 2-level FTB designs. If one were only to look at IPC, it would

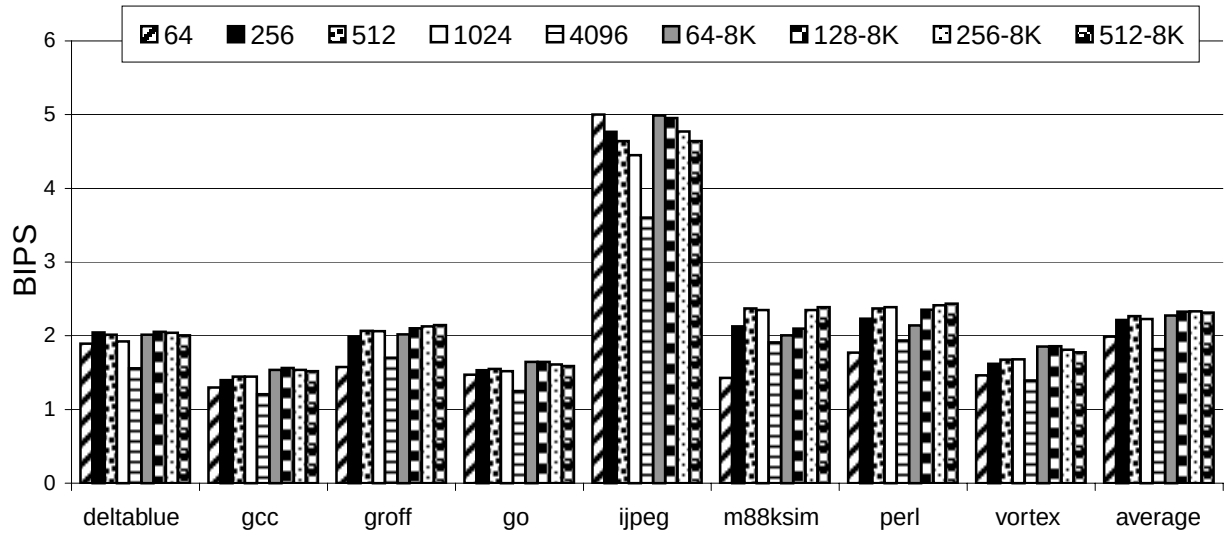

Figure 13: Billion Instructions per Second (BIPS) comparison across a variety of FTB configurations. BIPS results were calculated using IPC values from SimpleScalar simulations and CACTI timing data. The bars for each benchmark represent different FTB configurations. The first five bars represent single level FTB configurations: 64, 256, 512, 1024, and 4096 entry first level FTBs. The next four bars represent dual level FTB configurations: 64, 128, 256, and 512 entry first level FTBs, each with an 8192 entry second level FTB.

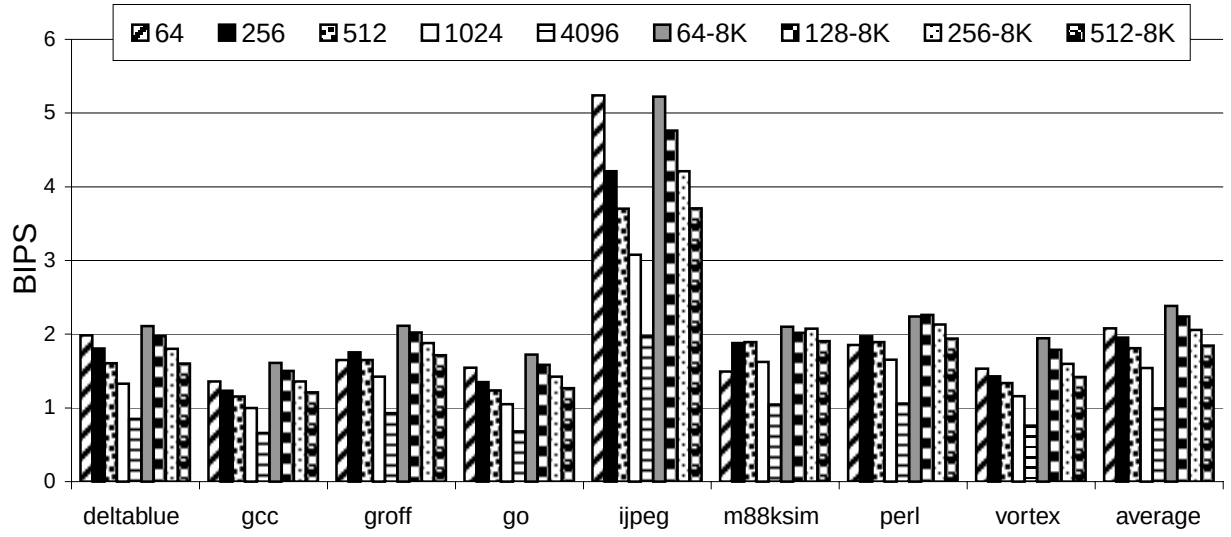

Figure 14: Fetch Target Buffer performance for $0.10\mu m$ feature size using technology scaling calculations from [27] modeling the potential effect of the interconnect scaling bottleneck.

appear that the bigger the table is, the better the performance looks, even though the access time for a given cycle would not be realistic.

Table 6 shows prediction accuracy data for the best single and dual level FTB configurations (from the BIPS data). On average, the results show that 11.2% of predictions from the 2nd level FTB are correctly predicted in the two-level configuration. `Ijpeg` again proves interesting as 31% of predictions in the single level FTB configuration

FTB num entries	t_{clk} (ns)
64	0.76
128	0.84
256	0.95
512	1.08
1k	1.30
2k	1.62
4k	2.03
8k	2.68

Table 5: Timing data from [27]. This data is only used for the BIPS results in Figure 14.

	512				128-8K					
program	% FTB acc	% from miss	ave pred size	preds in a row	% FTB acc	% from L1	% from L2	% from miss	ave pred size	preds in a row
deltablue	80.7	0.1	6.1	5.1	84.5	77.2	4.2	3.1	5.8	6.3
gcc	70.9	0.4	7.1	2.9	79.2	59.5	13.2	6.5	6.1	4.5
groff	81.4	0.4	7.3	4.8	84.2	60.2	18.0	6.0	6.4	6.0
go	67.1	1.5	7.8	2.7	72.4	56.1	8.2	8.1	6.8	3.5
ijpeg	88.5	31.0	15.4	8.6	88.6	39.4	0.0	49.2	11.4	8.5
m88ksim	88.0	0.2	4.9	8.0	83.9	60.8	19.0	4.2	4.8	6.0
perl	82.9	0.2	6.6	5.5	83.6	65.9	13.9	3.8	6.3	6.0
vortex	82.5	2.4	10.6	4.8	91.9	60.1	13.1	18.7	8.4	12.1
average	80.3	4.5	8.2	5.3	83.5	59.9	11.2	12.5	7.0	6.6

Table 6: This table examines two FTB configurations: a single level FTB with 512 entries, and a 128 entry FTB with a second level with 8192 entries. For both configurations, we list the percent of FTB predictions with valid fetch distances and correct branch predictions, shown in columns 2 and 6. Columns 3 and 9 show the percent of correct predictions that resulted from an FTB miss. In this case, the default fetch distance is predicted. In addition, for the two level configuration, we show the percent of correct predictions that were from the first level FTB (column 7) and the percent of correct predictions that were from the second level FTB (column 8). For both configurations, we show the average number of instructions in each fetch block that are predicted in a single cycle (columns 4 and 10). Also, we show the number of predictions in a row that were produced on average before a misprediction (columns 5 and 11). The product of these two numbers provides the average number of instructions between mispredictions.

are correctly predicted and come from misses in the FTB (which would use the default FTB fetch distance and a fall-through prediction). This is due to the nature of the benchmark which contains very few taken branches. Table 6 shows that the average FTB prediction size for the single level FTB is around 8 instructions, and 5 predictions occur in a row before reaching a misprediction. This means that on average, a single level FTB can supply around 43 instructions between mispredictions. The 2-level FTB is able to supply slightly more instructions between mispredictions - around 46 instructions on average. The exception is `m88ksim`, again due to the frequency of taken branches in this benchmark. Without sufficient FTQ occupancy, `m88ksim` is unable to tolerate second level FTB accesses and a single level configuration is able to outperform two-level configurations.

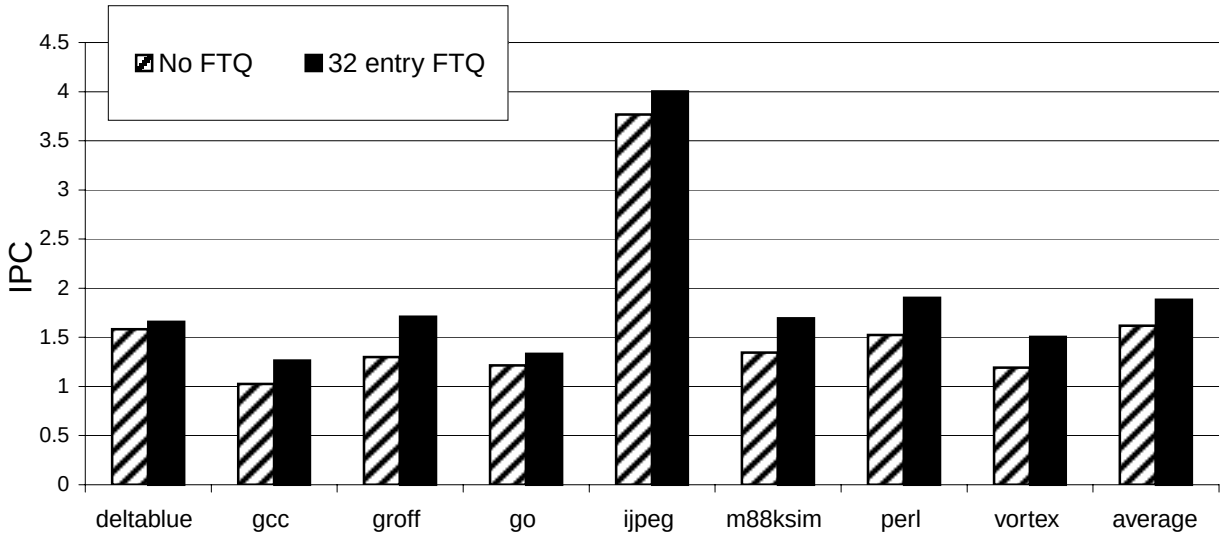

Figure 15: This figure shows IPC results for a two level FTB (128 entry first level, 8192 entry second level) without an FTQ and with a 32 entry FTQ. The x-axis shows the benchmarks we examined and an average.

Figure 15 shows the performance of a 2-level FTB with and without and FTQ, and demonstrates how the FTQ enables the 2-level FTB. Using a 32 entry FTQ provides a 16% improvement on average in IPC over a design without an FTQ. Results shown are for a 128 entry first level FTB with an 8192 entry second level FTB. Without an FTQ, there is limited opportunity for the second level FTB to correct a miss in the first level FTB. The occupancy in the FTQ helps tolerate the latency of the second level FTB access.

8 Conclusions

In this paper, we have investigated a decoupled front-end architecture that uses a fetch target queue (FTQ), which is a small FIFO queue used to store predicted fetch addresses from the branch prediction unit. The FTQ provides a decoupled front-end – allowing the branch predictor and instruction cache to run more independently. The FTQ enables a number of optimizations, and we demonstrated two of these in this study – effective instruction cache prefetching and the construction of a multi-level branch predictor. To better evaluate our architecture, we made use of results in both IPC and BIPS (billion instructions per nanosecond). While IPC provides some notion of performance, BIPS gives a complete picture as it takes into account the cycle time of the processor. It is essential for architects to study both the improvement in performance obtained through a particular technique and the impact the technique (table sizes) will have on the cycle time of the processor.

The FTQ enables a number of optimizations based on the predicted fetch stream it contains. We first examined using the FTQ to direct instruction cache prefetching. With this technique, we were able to provide a speedup in IPC of 19% for an 8 byte/cycle bus to the L2 cache. We showed how cache probe filtering could be an effective technique

for reducing bus utilization and attaining higher levels of performance. However, the performance of fetch-directed prefetching is essentially tied to the accuracy of the prediction stream and the occupancy of the FTQ.

Next, we showed that a multi-level branch predictor can be an effective technique to achieve high prediction accuracy while retaining small cycle times. In our results, we saw that the best performing two level branch predictor (a 128 entry first level fetch target buffer with an 8192 entry second level) achieved a 2.5% speedup on average in BIPS over the best performing single level FTB configuration (a 512 entry FTB). These results assume no interconnect scaling bottleneck. In the event of poor interconnect scaling, the gap in cycle times between the single level tables and the multi-level tables can be significant, as we showed in [27]. We found that the best performing two level FTB design provided a 14.6% improvement in BIPS over the best performing single level design, when taking into consideration the interconnect scaling bottleneck.

There are a number of other optimizations which could be enabled by the FTQ design. In addition to instruction cache prefetching, the FTQ could be used to direct data cache prefetch in a manner similar to that proposed in [8]. The FTQ could also be used to index into other PC-based predictors (such as value or address predictors) further ahead in the pipeline. This would allow these predictors to be large without compromising processor cycle times.

Additionally, the use of a multi-level branch predictor, like the FTB, provides us with a significant amount of instruction state. Extra bits could be stored in the branch predictor to help guide later stages of the pipeline. For example, we used extra state in the multi-level branch predictor to perform eviction prefetching in [29]. The state can be used to help guide cache replacement policies, and to track branch confidence, value predictability, or any other useful metric.

Acknowledgments

We would like to thank 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-9808697, and a grant from Compaq Computer Corporation.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th Annual International Symposium on Computer Architecture*, 2000.

[2] J. O. Bondi, A. K. Nanda, and S. Dutta. Integrating a misprediction recovery cache (MRC) into a superscalar pipeline. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 14–23, December 2–4, 1996.

[3] 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.

[4] B. Calder and D. Grunwald. Fast and accurate instruction fetch and branch prediction. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 2–11, April 1994.

[5] B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251, October 1994.

[6] P. Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 274–283, June 1997.

[7] 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.

[8] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[9] 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.

[10] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

[11] 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.

[12] J. A. Fisher. Trace scheduling : A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, 1981.

[13] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[14] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ilp. In *12th International Conference on Supercomputing*, 1998.

[15] E. Hao, P. Chang, M. Evers, and Y. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 191–200, December 1996.

[16] 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.

[17] N. P. Jouppi and S. J. E. Wilton. Tradeoffs in two-level on-chip caching. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 34–45, April 1994.

[18] S. Jourdan, T. Hsing, J. Stark, and Y. Patt. The effects of mispredicted-path execution on branch prediction structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1996.

[19] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *17th International Conference on Architectural Support for Programming Languages and operating Systems*, pages 138–147, October 1996.

[20] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

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

[22] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[23] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[24] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache replacement. In *21st Annual International Symposium on Computer Architecture*, April 1994.

[25] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. CSE-TR-335-97, University of Michigan, May 1997.

[26] C.H. Perleberg and A.J. Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, 1993.

[27] 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.

[28] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st International Symposium on Microarchitecture*, December 1998.

[29] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *32st International Symposium on Microarchitecture*, November 1999.

[30] G. Reinman and N. Jouppi. Cacti version 2.0. http://www.research.digital.com/wrl/people/jouppi/CACTI.html, June 1999.

[31] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, December 1996.

[32] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[33] 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.

[34] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, pages 42–53, December 2000.

[35] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 259–271, December 1998.

[36] K. Skadron, M. Martonosi, and D. Clark. Speculative updates of local and global branch history: A quantitative analysis. Technical Report TR-589-98, Princeton Dept. of Computer Science, December 1998.

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

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

[39] 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.

[40] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 345–356, June 1995.

[41] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.

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

[43] T. Yeh and Y. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 129–139, December 1992.