A Scalable Front-End Architecture for Fast Instruction Delivery

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

[†]Department of Computer Science and Engineering, University of California, San Diego
[‡]Microcomputer Research Labs, Intel Corporation

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 further complicate matters, a VLSI interconnect scaling trend is materializing that further limits the performance of front-end designs in future generation process technologies.

To counter these challenges, we present a fetch architecture that permits a faster cycle time than previous designs and scales better with future process technologies. Our design, called the Fetch Target Buffer, *is a multi-level fetch block-oriented predictor. We decouple the FTB from the instruction fetch and decode pipelines to afford it the fastest clock possible. Through cycle-based simulation and circuit-level delay analysis, we find that our multi-level FTB design is capable of delivering instructions 25% faster than the best single-level BTB-based pipeline configuration. Moreover, we show that our design scales better to future process technologies than traditional single-level designs.*

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. 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.

Unfortunately, scaling the performance of the front-end is no easy task. Three primary detractors work to make this a very challenging endeavor. First, instruction cache misses stall instruction delivery until instructions are returned from the next level of the instruction memory hierarchy. Second, the misprediction of the address or direction of a branch forces a pipeline flush, resulting in wasted fetch bandwidth between the time the branch was mispredicted and the time the misprediction was detected. Third, in modern front-end designs, resolving the target of a taken branch requires an access to the branch predictor and branch target buffer (BTB). As a result, the rate at which these devices can be cycled times the average basic block size places an upper limit on instruction delivery rates.

To further compound the challenge of front-end design, a process technology trend is materializing that will make it more difficult to design fast front-ends, *i.e.*, front-ends with low cycle times. Looking ahead a few process technology generations (*e.g.*, $0.18\mu m$ and $0.10\mu m$)[1] it becomes apparent that the performance (latency) of wires is not scaling as well as the performance of transistors [2, 3]. Wire performance may not scale at all and may even deteriorate in a few process generations. The problem is worse for large memories, like those typically found in front-end designs, because they are composed of significantly more interconnect. Large front-end designs may see little improvement and possibly even a reduction in the *rate* at which the processor can deliver instructions to the execution core.

As a result of this trend, architects must start concerning themselves less with the *amount of logic* in the critical path of a design and instead focus on the *amount of wire* in the critical path. Designs with less wire will naturally scale better because their latency is more a function of transistor latency which scales with process feature size. A *scalable*

[1] The notation $0.18\mu m$ indicates a process fabrication technology with a 0.18 micrometer minimum feature size.

design is one that can perform well in the face of process technology trends - ideally we would like to see performance increases commencerate with the feature size scaling factor. We predict, given the interconnect scaling bottleneck, a scalable design will be one with minimal wire lengths on the critical path of the design.

In this paper, we present a new scalable front-end design. Our design decouples the branch predictors and branch target buffers from the I-cache, to allow maximum performance for each. We call this new design a *Fetch Target Buffer* (FTB). The FTB organization was chosen to (1) maximize the number of instructions fetched for each prediction, and (2) perform a useful prediction every cycle. Each FTB entry represents a large variable length sequential fetch block up until the next taken branch. In an effort to provide fast cycle times, while allowing sufficient capacity to maintain history and targets for a large number of branches, the FTB uses a multi-level memory hierarchy. The top-level (L1) FTB is able to store its targets and predictor history into a larger second-level (L2) FTB. Through cycle-based simulation and circuit-level timing analysis, we show that this multi-level design performs better than traditional single-level designs, and is scalable to future process generations.

The remainder of this paper is organized as follows. In Section 2 we detail the interconnect scaling bottleneck and its impact on front-end design. In Section 3 we present a new scalable front-end architecture, and in Section 4 we detail the organization and operation of the fetch target buffer. In Section 5 we describe the methodology used to gather our results. In Section 6 we evaluate the scalability of the new designs, comparing the performance of single and two-level FTB designs with traditional BTB-based designs, both in the cycle and time domains. Section 7 presents related work. Finally, Section 8 provides a summary and concludes with future directions.

2 How Poor Interconnect Scaling Affects Front-End Performance

The interconnect scaling bottleneck is as follows: As process technology feature size scales by a factor S, the performance (*i.e.*, delay) of transistors scales linearly at roughly a factor S. Wire latency, on the other hand, scales at a rate less than S due to parasitic capacitance effects. There are three important results of this trend:

i. memory structures experience the full extent of this trend because they are composed of significant amounts of closely packed interconnect,

ii. larger memory performance scales worse than small memory because they are composed of significantly more interconnect, and

iii. interconnect scaling degrades as process feature size decreases due to increasing parasitic capacitance effects; if current trends continue, wire latency will no longer scale and may increase in future process generations.

There has been a significant amount of analytical [16] and empirical [2, 18] analyses of this trend in the process technology literature. Recently, these analyses have carried over into the computer architecture literature where their effects on the execution core have been examined [19]. In this section we provide a brief introduction to the problem, readers are referred to [2] or [19] for a more in-depth analysis of this bottleneck.

To better understand why on-chip memory performance scales poorly with process feature size, we need to examine more closely their structure. On-chip memory devices are composed of large two-dimensional arrays of memory cells. Connecting these memory cells to other parts of the chip is a tapestry of wire that forms two buses. The *wordline* bus runs the rows of the array, bringing signals to the cells that indicate if the cells are being accessed. The *bitline* bus runs the columns of the array, providing access to memory cell contents. To access the memory, a decoder “turns on” a row of the memory array by asserting a single wordline, this results in the contents of every cell in the row being asserted on the bitline bus. A MUX at the end of the bitlines is used to select the accessed data.

The latency of a memory device, to a first order, is the latency to exercise the logic in the decoder, assert the wordline wire, read the memory cell logic, assert the bitline wire, and finally exercise the logic in the bitline MUX to select the accessed data. As the process feature size is scaled, the latency of the transistors is scaled proportional to their size, thus the latency of the logic scales linearly with feature size reductions.

The latency of the wordlines and bitlines, on the other hand, does not scale as well due to *parasitic capacitance* effects that occur between the closely packed wires that form these buses. As the technology is scaled to smaller feature sizes, the thickness of the wires does not scale. [2] As a result, the parasitic capacitance formed between wires remains fixed in the new process technology (assuming wire length and spacing are scaled similarly). Since wire delay is proportional to its capacitance, signal propagation delay over the scaled wire remains fixed even as its length and width are scaled. This effect is what creates the *interconnect scaling bottleneck*.

Recently, some process technologies have begun employing copper interconnect and *low-k* dielectrics as a way to reduce the impact of poor interconnect scaling [14, 15].

[2] The reasons behind poor interconnect thickness scaling are numerous and complex. Two main reasons include 1) it is difficult to manufacture thinner wires, and 2) keeping wires thick increases their cross-sectional area, which in turn reduces current densities and reliability problems associated with metal electromigration.

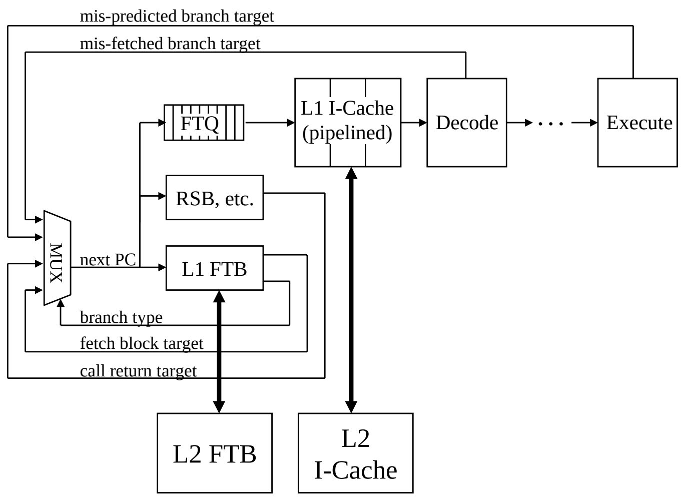

Figure 1: A Decoupled Multi-level Front-End.

These material lower the resistance and capacitance of wires, respectively, thereby improving signal propagation performance. However, these techniques only offer a one time reprieve for the first process generation that employs them. Poor interconnect scaling trends continue. It has been shown that splitting long wires with buffers can reduce their propagation delay [1]. However, this approach cannot be applied to the the densely packed interconnect of memory arrays without significantly increasing their area (due to many buffers).

Front-end designs tend to contain a significant amount of on-chip memory - in the branch predictors, BTBs, and I-caches. As a result, future generation front-end designs will scale poorly unless architects strive to limit the amount of wire on the critical paths of their designs. One effective approach to reduce wire lengths is to decrease the size of memory structures in the critical path of the front-end design. In the following section, we present a new scalable front-end design that attains this goal while at the same time providing competitive cycle times and prediction rates compared to traditional front-end designs.

3 A Scalable Front-End Architecture

In this section we describe our scalable front-end architecture illustrated in Figure 1. To create a scalable design, we decouple the I-cache from branch predictor, thereby eliminating this large and slow memory from the front-end critical path. Note that this implies that the instruction cache has its own local fetch address to control the cache fetching, and the branch predictor has its own local PC to control the branch predictions. The PC used for the current cycle's branch prediction, will be used in a subsequent cycle for the cache fetch address.

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 be eventually consumed by the instruction cache. The FTQ provides the buffering necessary to permit the branch predictor and I-cache to operate autonomously; the branch predictor can miss and stall while the I-cache continues fetching blocks. In contrast, the FTQ allows the branch predictor to work ahead of the I-cache when it is stalled due to a cache miss or a full instruction buffer. If the I-cache is multi-ported, multiple valid FTQ entries can be consumed in a single cycle (possibly out-of-order) until ports are exhausted.

Recall from Section 2, the interconnect scaling bottleneck only allows low access latency and good scalability for small memory arrays. As a result, large instruction caches will have to be pipelined to accommodate future clock rates. Fortunately, the decoupled design only exposes this additional I-cache latency during branch mispredictions. As an added benefit, pipelining the I-cache makes it easier to increase the cache size or associativity without impacting front-end critical path lengths.

To maintain good branch throughput and scalability it is important to make the branch predictors and branch target buffers as small as possible. At the same time however, a large branch predictor is desirable as this will ensure that we have sufficient capacity to predict the direction and targets of most branches, thereby eliminating most branch misprediction latencies. To solve this conundrum we turn to the time-tested solution of multi-level memory hierarchies, and use a multi-level branch prediction architecture called the *Fetch Target Buffer* (FTB).

To further improve instruction delivery throughput, the FTB is crafted to return information about the dynamic instruction stream each cycle it is accessed. It does this by predicting the address and size of *fetch blocks*. A fetch block is a sequence of instructions starting at a branch target, and ending with a strongly biased taken or unbiased branch. Branches which are biased and not taken may be embedded within fetch blocks. This optimization permits fetch block sizes to increase without cost. Since a strongly biased not taken branch does not change the flow of control, we can predict this branch by simply ignoring it. Our predictor allocation policy (described later) ensures that strongly biased not taken branches are embedded within fetch blocks.

During operation, the FTB provides branch address and target predictions. It is tagged and split into multiple levels. Predictor history and branch target data is demand fetched (or prefetched) from the L2 FTB into the L1 FTB. To minimize FTB access latency, only the information necessary to cycle the *next PC* computation is stored within it. Each cycle, the FTB produces a starting address for the next fetch block, the address where the fetch block ends, and the predicted target address (fall-through or taken) to be used for the prediction in the next cycle. These addresses are stored in an FTQ entry after each prediction, and are consumed in subsequent cycles by the instruction cache.

4 Fetch Prediction Architectures

In this section we describe prior branch target buffer architectures. We then describe our multi-level fetch target buffer design to provide fetch prediction for our decoupled front-end.

4.1 Branch Target Buffers

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 [21] 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, they found that the "additional complexity of the multi-level BTB is not cost effective" [21]. Technology has changed since their study, 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) [30, 31]. 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. In comparison, our decoupled front-end and FTQ allow our FTB predictor to speed ahead of the I-cache, potentially performing a useful prediction every cycle.

4.2 Fetch Target Buffer

The branch prediction architecture we model in this paper is an extension of the BBTB design by Yeh and Patt [30, 31], with two changes to their design. The first change is that we do not store basic blocks in our fetch target buffer that are fall-through basic blocks or basic blocks with branches that are seldom taken [6]. The BBTB design stores an entry for *all* basic blocks. Storing non-taken basic blocks wastes BBTB entries, and decreases the size of fetch blocks, which requires additional predictions to traverse what could have been one larger fetch block.

The second change we made to the BBTB design is that we do not store the full fall-through address in our FTB. Instead, we store only the pre-computed lower bits of the fall-through address along with a carry bit used to calculate the rest of the fall-through address [6]. This helps reduce the amount of storage for each BBTB entry, since the typical distance between the current fetch address and the BBTB's fall-through address is not large.

Our Fetch Target Buffer (FTB) design is shown in Figure 2. The FTB table is accessed with the start address of a fetch target block. Each entry in the FTB contains a tag, taken address, partial fall-through address, fall-through carry bit, branch type, oversize bit, and conditional branch prediction information. The FTB entry represents the *start* of a fetch block. The fall-through address minus 4 represents the location of a 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 is used as the next cycle's prediction address.

As described earlier, the fall-through address is not stored in its entirety in the FTB entry. Only the N low order bits of the fall-through address are stored along with a carry bit. If the carry bit is not set, the complete fall-through address is calculated by concatenating the upper $address\_size - N$ bits of the current fetch address with the N fall-through address bits stored in the FTB entry. If the carry bit is set, the complete fall-through address is calculated by adding one to the upper $address\_size - N$ bits of the current fetch address, and then concatenating this with the N fall-through address bits stored in the FTB entry. The calculation of adding the carry bit to the upper bits of the PC is done in parallel with the FTB lookup. Then if the branch is predicted as not-taken, the carry bit chooses between the two possible values for the upper bits of the fall-through address, and then performs the concatenation.

The size of the N partial fall-through bit field determines the size of the fetch blocks that can be represented in the fetch target buffer. If the fall-through 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.

An oversize bit is used to represent whether or not a fetch block spans a cache block [30]. 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

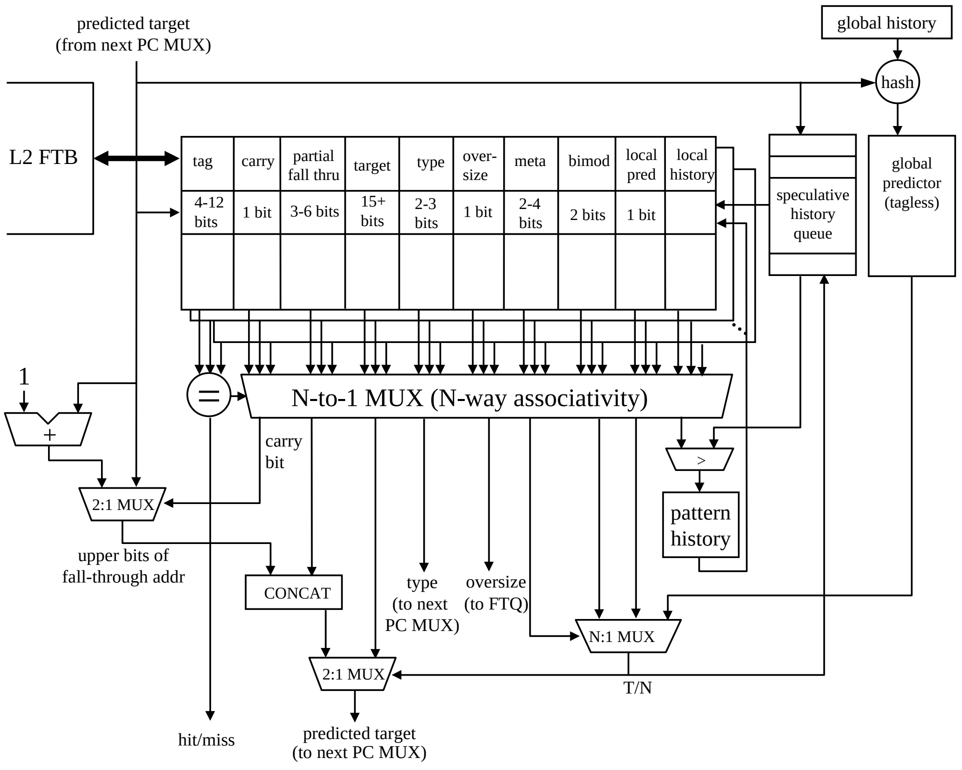

Figure 2: The Fetch Target Buffer.

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.

The branch direction predictor shown in the FTB in Figure 2 is a hybrid predictor with a meta-predictor that can select between a local history-based predictor, a global history predictor, and a bimodal predictor. Other combinations are possible, as well as non-hybrid predictors. The local history is composed of the last N branch directions for the branch at the end of the fetch block. The local branch history is used to index the pattern history table, returning a pattern prediction. 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. 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.

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 several predictions ahead of the instruction cache fetch. To facilitate the recovery of branch history, a small *Speculative History Queue* (SHQ) holds the speculative history of branches. When branches are predicted their updated local or global history is inserted into the SHQ. When predictions are made, the SHQ is searched in parallel with the L1 FTB, if a newer history is detected in the SHQ, it takes precedence over the history in the L1 FTB. Entries are only allocated in the SHQ when the history changes; this reduces capacity requirements in the SHQ. When the branch at the end of a fetch block retires, its speculative history is written into the FTB. When a misprediction is detected, the point in the SHQ of the mispredicted branch and later allocated entries are released. The SHQ is kept small to keep it off the critical path of the L1 FTB. If the speculative history queue becomes full, the oldest entry is written into the FTB. Skandron *et al.*, independently developed a similar approach for recovering branch history, and they provide detailed analysis of their design in [26].

The meta predictor, bimodal, and 2-bit pattern history table values are not updated speculatively. The front-end can only assume it made the correct prediction and thus reinforce bimodal or pattern history predictions. It has been shown in [13] that better performance results when the meta predictor and 2-bit PHT updates are delayed until the result of the branch outcome is known, *i.e.*, at execute or retirement.

Since the FTB 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 [25], as the predictor may encounter several returns or calls down a misspeculated path that will affect more than just the top of stack. We use two return address stacks to solve this problem. One is speculative (S-RAS) and is updated by the FTB during prediction. The other is nonspeculative (N-RAS) and is updated during writeback. When a misprediction is detected, the S-RAS will likely be polluted and can be recovered from the N-RAS. Then prediction can restart as normal, using the S-RAS. This provides accurate return address prediction. Additional analysis of our RAS recovery mechanism and our SHQ design can be found in [22].

4.3 Functionality of the 2-Level FTB

The L1 FTB is accessed each cycle using the predicted fetch block target of the previous cycle. At the same time, the speculative history queue, the return address stack, 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. In our models, we achieved good performance with L2 FTBs that have shorter latencies than one would use for a first level instruction cache, so this was not a problem. 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 misprediction is detected in the decode or writeback stage of the processor. Once a 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. By injecting fetch blocks sequentially into the machine, it's possible to partially overlap the generation of FTB entries with their execution.

Branch Misprediction Recovery In the decode stage, the predicted direction of unconditional branches, *e.g.*, jumps, calls and returns, and the targets of direct branches, *e.g.*, PC relative and absolute, are validated. In the writeback stage, the targets of indirect branches and the direction of conditional branches are validated. Fetch block targets and sizes are propagated down the pipeline with instructions. During validation, if a branch target does not match the accompanying fetch block, a branch misprediction recovery sequence is initiated. The FTB entry is updated with the correct fetch block information, misspeculated entries in the speculative history queue are released, and the pipeline is flushed behind the misspeculated branch. In any event, the prediction history of branches is updated. To facilitate the embedding of strongly biased not-taken branches within fetch blocks, not taken branches do not update history or create FTB entries unless they are already contained in the FTB and at the tail of a fetch block. In addition, new FTB entries are only allocated when branches are taken.

5 Methodology

The simulators used in this study are derived from the SimpleScalar/Alpha 3.0 tool set [5], 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 mis-prediction.

To perform our evaluation, we collected results for six of the SPEC95 C benchmarks plus 2 C++ programs. `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

Program	Input	# instr fwd (M)	# instr exec (M)	% br exe
compress	ref	0	93	13.9
deltablue	ref	0	96	17.0
gcc	1cp-decl	400	1041	17.3
groff	someman	0	52	17.3
go	5stone21	2000	32699	14.0
ijpeg	specmun	2000	34716	10.5
li	ref	2000	18089	19.1
m88ksim	ref	2000	76271	14.8
perl	scrabbl	2000	28243	16.1
vortex	vortex	2000	90882	14.7

Table 1: Program statistics for the baseline architecture.

we used in gathering results for each program, the number of instructions executed in the program to completion (in millions), and the percent of executed branches in each program. Also shown is the number of instructions that were executed (fast forwarded) before actual simulation. Results are then reported for simulating each program for up to 100 million instructions.

5.1 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 and issue up to 16 instructions per cycle. It has a 128 entry reorder 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 64k 2-way set-associative direct-mapped instruction cache and a 64k 4-way set-associative data cache. Both caches have block sizes of 32 bytes. The data cache is write-back, write-allocate, and is non-blocking with 2 ports. The data cache is pipelined to allow up to 2 new requests each cycle. There is a unified second-level 1 MB 4-way set-associative cache with 64 byte blocks, with a 10 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. We model the bus latency to main memory with a 10 cycle bus occupancy per request. 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 gshare predictor [17] for our conditional branch predictor. The predictor has a 2-bit meta-chooser and a 2-bit bimodal predictor, both stored in the FTB (or BBTB) entry with the branch. In addition, a tagless Gshare predictor is also available, accessed in parallel with the L1 FTB. 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.

5.2 Timing Model

We report our results using device timing metrics for four process technologies gathered using a modified version of the *Cacti* cache compiler [29]. *Cacti* contains a detailed model of the wire and transistor structure of on-chip memories. We modified *Cacti* to model a BBTB, FTB, and tagless branch predictors, and extended the 0.80μm process model to include timings for 0.35μm, 0.18μm, and 0.10μm processes. The 0.80μm process is a previous generation process with 0.80 micron minimum feature sizes. The 0.35μm process is a current generation process, and the 0.18μm and 0.10μm processes represent future generation technologies. The 0.80μm, 0.35μm, and 0.18μm process parameters are from [19]; the 0.10μm process parameters are expected values based on empirical analysis of experimental fabrication processes, taken from [2]. The specifics of the *Cacti* on-chip memory model and process models used are detailed in [22].

Table 2 lists the front-end architectures analyzed and their timing parameters for the four process technologies. The table lists the FTB sizes (in number of entries), and FTB, branch predictor, and cache latencies (in clock cycles). The latencies are shown for each process technology in the following order: 0.80μm, 0.35μm, 0.18μm, and 0.10μm technologies. The latencies for the L2 FTB, I-cache, and D-cache were selected by dividing the access times for these devices by the access latency of the L1 FTB. Since these devices have a multiple cycle latency, we simulate them as fully pipelined memories. We scaled the branch predictors to the maximum size that could be accessed in less time than the L1 FTB. The timing of the devices sometimes changed between process technologies due to varied interconnect scaling effects. All FTB organizations are 4-way set-associative. L1 FTB access latencies (in nanoseconds) are listed in the column labeled t_{clk} *min*. The size of L1 FTB entries varies depending on the size of the tag used, although most experiments have about 8 bytes of data per entry.

Config	L1 FTB/BTB		Predictor		L2 FTB		I-cache	D-cache	t_{clk} min
	size	latency	size	latency	size	latency	latency	latency	(ns)
F64	64	1,1,1,1	4k	1,1,1,1	n/a	n/a	2,2,3,4	2,2,3,4	6.05,2.03,1.03,0.76
F256	256	1,1,1,1	8k	1,1,1,1	n/a	n/a	2,2,2,3	2,2,3,3	6.33,2.21,1.16,0.95
F1k	1k	1,1,1,1	16k	1,1,1,1	n/a	n/a	2,2,2,2	2,2,2,2	7.06,2.55,1.44,1.30
F4k	4k	1,1,1,1	16k	1,1,1,1	n/a	n/a	2,2,2,2	2,2,2,2	8.30,3.40,2.07,2.03
F8k	8k	1,1,1,1	16k	1,1,1,1	n/a	n/a	1,1,1,1	1,1,1,1	9.16,3.91,2.57,2.68
F64x1k	64	1,1,1,1	4k	1,1,1,1	1k	2,2,2,2	2,2,3,4	2,2,3,4	6.05,2.03,1.03,0.76
F64x4k	64	1,1,1,1	4k	1,1,1,1	4k	2,2,2,3	2,2,3,4	2,2,3,4	6.05,2.03,1.03,0.76
F256x1k	256	1,1,1,1	8k	1,1,1,1	1k	2,2,2,2	2,2,2,3	2,2,3,3	6.33,2.21,1.16,0.95
F256x4k	256	1,1,1,1	8k	1,1,1,1	4k	2,2,2,2	2,2,2,3	2,2,3,3	6.33,2.21,1.16,0.95

Table 2: Analyzed Configurations. L1 and L2 FTB sizes are listed in terms of total number of entries. I-cache and D-cache sizes are all 64k bytes. F64x1k stands for a 64 entry L1 FTB with a 1k entry 2nd level FTB. n/a indicates the field is not applicable to the listed configuration.

6 Results

First we present a comparison of the BBTB design and the single-level FTB design. Next, we compare our single level FTB to a two level FTB, presenting results in Instructions Per nanoSecond (IPS). We then investigate how the performance of the FTB tolerates changes in factors such as the number of bits allocated to the fetch distance and variance in the size of the FTQ. For all of the FTB results we used a fetch distance of size 16 instructions (4 bits for the partial fall-through address), which we found to be sufficient as described in Section 6.3. Finally, the scalability of the FTB across different feature sizes is considered.

6.1 BBTB Comparison

Figure 3 shows IPC results comparing the BBTB design our single-level FTB configurations. Both architectures were simulated with a coupled front-end (no FTQ) to provide a fair comparison with non-pipelined caches. Overall, the FTB designs provide slightly better fetch bandwidth than the BBTB designs, since the FTB does not need to store every encountered basic block (branch) - only those that have been taken in the past.

Figure 3 shows that a small 64 entry FTB can hold the majority of branches executed by `compress`, `ijpeg`, and `m88ksim` whereas the rest of the programs benefit from having a large FTB. For all programs, little performance gain is seen when increasing the predictor size beyond 4K entries. The results show that the FTB design consistently outperformed or performed as well as a comparably sized BBTB design. The remainder of the results compare the performance of the two-level FTB with the single level FTB.

6.2 Two-level FTB Performance

Table 3 shows the average fetch block size in instructions and the percent of correct predictions provided by a 64 entry single-level predictor, and a 64 entry first level FTB with a 1K entry second-level FTB (64-1K). The average fetch block size is the dynamic size obtained for predictions on the non-speculative path. The correct prediction rates show the percent of time the FTB provided a correct prediction from each FTB level, and the percent of time predicting a fixed fall-through fetch distance was correct for an FTB hierarchy miss. For a miss, a fall-through fetch block of size 16 instructions is predicted. Once an FTB entry has been brought into the first level from the second level, it is counted as a L1 correct prediction for all subsequent predictions - until it is again swapped out of the first level. The average fetch block size of the single level FTB configuration is higher due to increased FTB misses. On FTB misses, the large fixed fetch distance (16 instructions) is predicted which makes the average fetch block size larger, but also increases the chance of including a taken branch. The two-level FTB structure provides a total of 83.3% correct fetch block predictions on

	1 Level FTB 64 entry			2 Level FTB 64-1K entry			
program	fetch size	%cor pred	%cor miss	fetch size	%cor L1	%cor L2	%cor miss
compress	5.7	70.4	13.7	5.7	72.3	0.3	12.7
deltablue	6.6	56.7	4.9	5.9	64.2	9.6	3.2
gcc	7.7	35.3	10.3	6.5	49.3	17.1	7.7
go	7.7	55.1	12.5	7.0	63.2	5.3	9.9
groff	8.0	31.1	8.5	6.5	47.7	23.9	6.8
ijpeg	7.5	84.6	12.8	7.5	84.7	0.0	12.8
li	6.2	62.7	3.6	5.5	70.4	9.1	2.0
m88ksim	7.4	84.0	7.9	6.5	85.6	0.0	3.7
perl	8.5	18.7	11.3	6.3	44.3	32.5	7.8
vortex	9.7	29.1	17.7	8.3	49.9	19.9	16.6
average	7.5	52.8	10.3	6.6	63.2	11.8	8.3

Table 3: FTB Performance. Average fetch block size provided by the FTB, along with the percent of fetch block predictions that were correct when hitting in the first (L1) or second (L2) level FTB, and the percent of correct predictions that occurred when missing in the FTB.

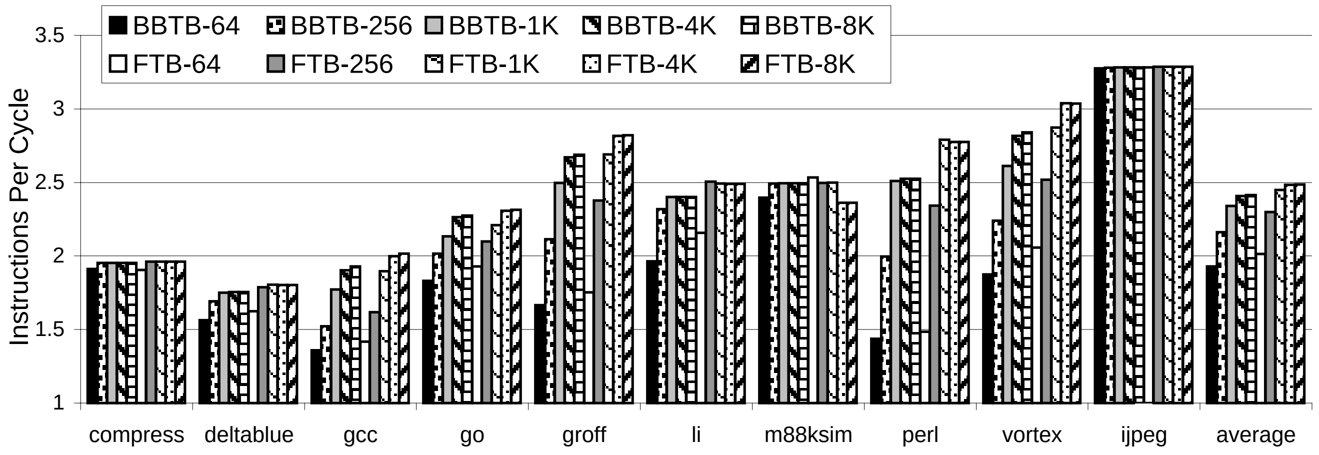

Figure 3: Instruction Per Cycle for BBTB and single-level FTB configurations.

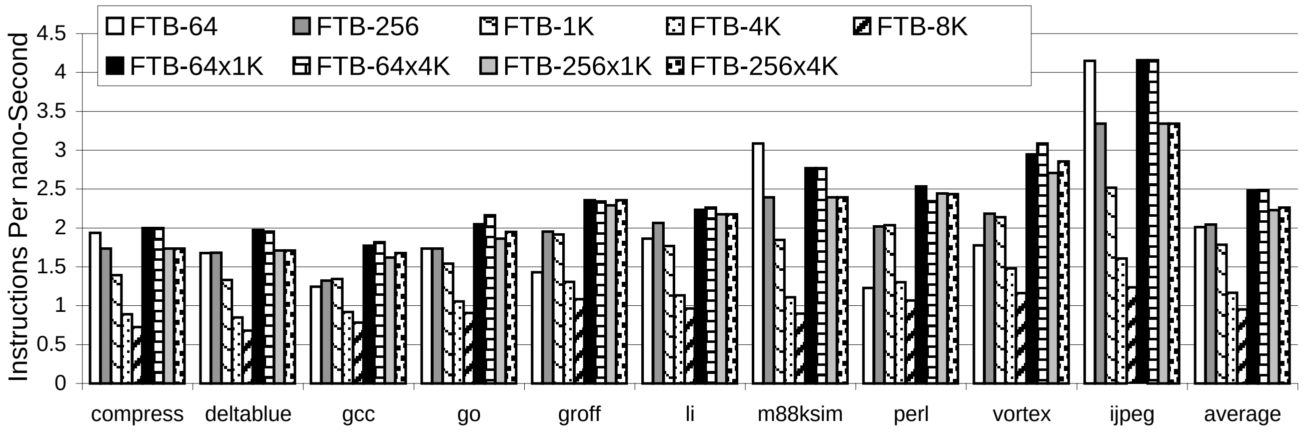

Figure 4: Fetch Target Buffer performance for 0.10μm feature size. Simulation results are shown in terms of the number of committed instructions per nano-second, which is computed from the IPC and cycle time used to simulate each configuration.

average, with an average fetch size of 6.6 instructions.

To fully assess the tradeoffs between different table sizes and the multi-level designs, we need to also look at their impacts on circuit performance. By combining the IPCs measured by the simulator with the front-end cycle time computed by our *Cacti* memory models shown in Table 2, we can compute the instruction delivery rates for each of the designs. [3] We report results in terms of of the number of *Instructions Per nanoSecond* (IPS), which is the number of instructions committed each nano-second. To calculate this number we divide the IPC by the cycle time of the front end (FTB). Remember that the second-level FTB, I-cache, and D-cache have additional latency for accessing them since they are pipelined for these simulations based on the latencies shown in Table 2.

Figure 4 shows the instruction delivery rates in IPS for all benchmarks using the 0.10μm timing and simulation models. The results show an average speedup of 25% using a 64 entry L1 FTB with a 1K entry L2 FTB (`F64x1k`) in comparison to the best performing single level FTB design (`F256`). The results show that the larger designs have fallen behind in performance. The `F1k`, `F4k`, and `F8k` designs, while having very good IPCs, also have long access latencies which works to reduce their front-end cycle times. The increases in IPC do not offset the impacts to cycle time and all these designs perform poorly. The two-level FTB designs perform best in almost all cases. These designs have fast cycle times and at the same time the L2 FTB provides additional resources in which to store prediction and target information.

The results for `m88ksim` show that the `F64` FTB single-level design outperforms the `F64x1K` FTB. This is due to fetch block fragmentation. If a branch inside a fetch block is infrequently taken, it will still cause that fetch block to be broken into two entries in the FTB. This effect causes the fetch block to require two predictions in order to fetch

[3]Technically, these rates represent the *maximum* instruction fetch rates possible with the proposed designs. A longer critical path elsewhere in the machine could reduce the front-end cycle time and instruction delivery rate.

it. Since the F64x1K case has more capacity, it holds on to the fragmented fetch blocks, whereas the F64 case may replace the fragmented blocks with other entries. If the entry is replaced and then brought back in again, the branch that caused the fragmentation may not be encountered as taken again, which will allow larger fetch blocks to be delivered into the FTQ with only one prediction. This is confirmed by Table 3, which shows that the single-level FTB has only a 84% correct prediction rate for m88ksim, which is lower than the 85.6% correct prediction rate of L1 FTB hits for the two-level design. The additional performance for single-level FTB comes from having 7.9% correct predictions from missing in the FTB, whereas the two-level FTB only gets 3.7% correct predictions from missing in the FTB.

6.3 FTB Design Parameters

Figure 5 shows how many bits are used to represent fetch distances over all nonspeculative FTB predictions for a given benchmark. The size of the fetch distance represents the number of bits needed to represent the partial fall-through address in Figure 2. The categories are disjoint and results are shown for the F64+1K configuration. For example, on average, 18.9% of predictions used only 1 bit for their fetch distance (corresponding to a distance of 1 instruction), 23.7% used exactly 2 bits (corresponding to a distance of 2-4 instructions), and 91.2% of all predictions could have been covered with 4 bits (a maximum distance of 16 instructions). This demonstrates that increasing the number of bits allocated to fetch distance in a FTB entry beyond 4 or 5 bits will not result in improvement. To test this, we simulated results with larger branch distance fields in the FTB entry. Our confirmed that fetch distances past 16 instructions do not appreciably improve either performance or prediction accuracy for the programs examined.

Figure 6 shows the percent of cycles in which there were a given number of occupied FTQ entries. These results show how far the predictor was able to run ahead of the fetch unit. On average, the FTQ is empty 21.1% of the time, and it is completely full 10.7% of the time. Some programs, such as ijpeg, fill up the FTQ a lot faster than they consume entries, which indicates that instruction cache stalls and resource contention (*i.e.* a full reorder/instruction buffer) are preventing the fetch unit from consuming FTQ entries as rapidly. m88ksim has an empty FTQ a larger proportion of the time than other programs, which indicates that the FTB is not keeping up with the speed of the fetch unit. Again, this can be traced back to the fragmentation of fetch blocks in the FTB for m88ksim.

6.4 Scalability

When designing a high-performance front-end architecture, the ideal design is one that is both fast and scalable. Fast, so

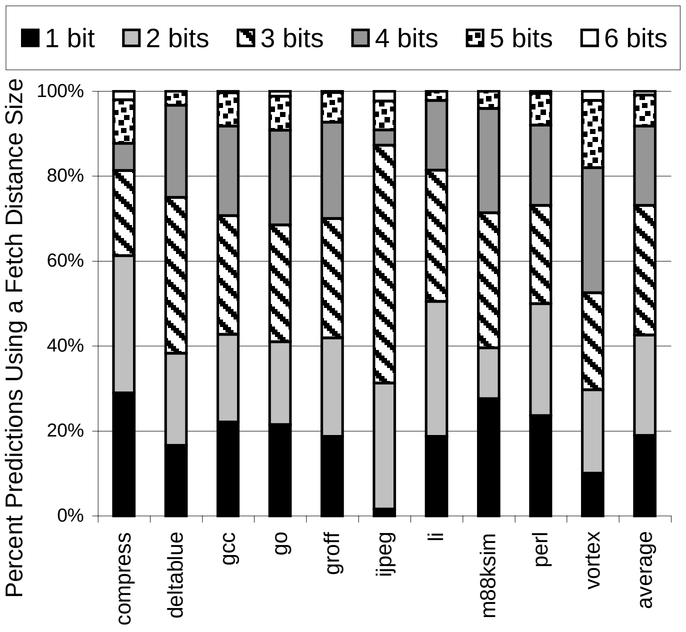

Figure 5: Percent of FTB predictions requiring a given number of bits to represent the fall-through fetch distance.

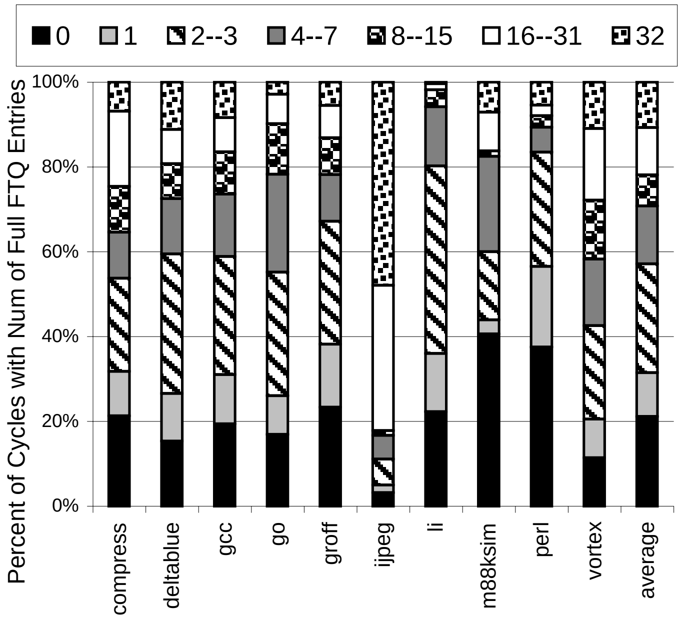

Figure 6: Percent of cycles with a given number of occupied FTQ entries during execution.

that it will deliver good performance in a particular implementation, and scalable, so that the investment in designing the initial implementation and later improving it can be carried forward into future process generations. Figure 7 shows the performance of the analyzed workload in IPS across four process technologies shown in Table 2. Each data point represents average performance across the entire benchmark set.

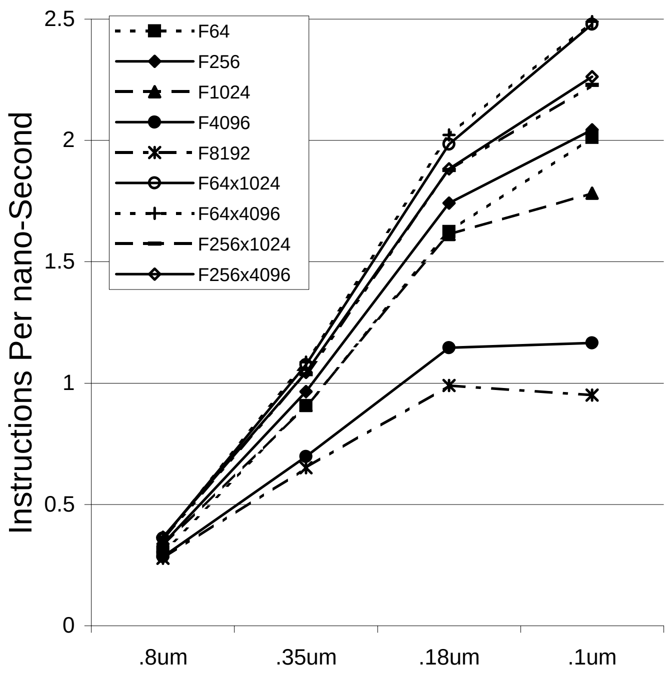

Figure 7: Impact of Process Technology on Performance.

The graph is drawn on a linear scale to highlight the scaling effects, if a device's latency scales with process feature size it will be drawn as a straight line with unit slope; less steep lines indicate poorer scaling properties.

Figure 7 shows that the multi-level FTB designs, especially the `F64x1k` and `F64x4k`, scale across the process generations. The `F64` and `F256` designs also scale well, but lack the IPC gains afforded by the L2 FTBs, making their design less attractive. The `F1K`, `F4K`, and `F8K` designs do not scale as well as the `F64` based design due to larger on-chip prediction memories and thus more interconnect in the critical paths of these designs. The `F4k` and `F8k` designs have significantly larger memories on the front-end critical path, thus they scale poorly in the future generation processes. The `F8k` design even experiences a slight reduction in performance in the $0.10\mu m$ process technology.

7 Related Fetch Bandwidth 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 [12] and instruction prefetch [28]. Techniques to reduce the impact of branch mispredictions include hybrid [17] and indirect [8] branch predictors, and recovery miss caches to reduce misprediction latencies [4]. A number of compiler-based techniques work to improve instruction delivery performance. They include branch alignment [7], trace scheduling [10], and block-structured ISAs [11].

Stark *et. al.*, [27] proposed an out-of-order fetch mechanism that features a decoupled branch target buffer that can continue cycling independent of instruction cache misses, to provide non-blocking I-cache fetch addresses. Their idea is similar to our decoupled front-end design, except there is no FTQ to allow the predictor to run ahead of the fetch unit.

Several architectures have been examined for efficient instruction throughput including the two-block ahead predictor [24], the collapsing buffer [9], and the trace cache [23]. Seznec *et. al.*, [24] proposed a high-bandwidth design based on two-block ahead prediction. By predicting not the target of a branch but rather the target of the basic block the branch will enter permits pipelining of the critical *next PC* computation. Conte *et. al.*, [9] proposed the collapsing buffer as a mechanism to fetch two basic blocks simultaneously. The design features a multiported instruction cache and instruction alignment network capable of replicating and aligning instructions for the processor core. Rotenberg *et. al.*, [23] 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.*, [20] extended the organization of the trace cache to include associativity, partial matching of trace cache lines, and path associativity.

8 Conclusions

A scalable front-end architecture was presented and evaluated. The design features the fetch target buffer (FTB), a multi-level fetch block-oriented target predictor. Simulation-based evaluations indicate the design is more capable than traditional BTB designs and single-level FTB designs. Circuit-level analyses show that the design also features a higher instruction delivery rate, measured in instructions per nano-second (IPS). For a $0.10\mu m$ technology, a two-level FTB design with a 64-entry first level and a 1k-entry second level provides a 25% improvement in IPS over the best performing single-level designs. When the performance of the various designs is examined across multiple process generations, the multi-level FTB designs exhibit the best performance and scalability of all the designs investigated.

We feel our approach is quite promising since it focuses on simplicity and raw speed. Unlike techniques that work to increase the number of instruction delivered per cycle, we were able to gain marked increases in performance while being able to sidestep the very difficult problems of multiple branch and target prediction.

We are currently extending this research in several directions. First, we are examining other FTB designs that may provide increased fetch block sizes. Second, we are evaluating using the FTQ to provide streaming of data from second-level cache to the first level. If the FTQ is full, because of I-cache misses or a backed up pipeline, the FTQ entries can be used to stream in cache blocks from the L2 cache into a stream buffer, eliminating L1 I-cache misses. Third, we are extending the multi-level FTB design to provide multiple branch prediction, which will produce multiple FTQ entries per cycle. Finally, we are comparing the performance of the FTB to other promising high-fetch bandwidth architectures like the trace cache. A complete evaluation of some of the above ideas along with a more detailed description of the Cacti timing models used in this paper can be found in [22].

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, a grant from Compaq Computer Corporation, and an equipment grant from Intel.

References

[1] H. Bakoglu and J. Meindl. Optimal interconnect circuits for VLSI. *IEEE Transactions on Computers*, 32(5):903–909, May 1985.

[2] M. Bohr. Interconnect scaling - the real limiter to high-performance ulsi. In *Tech. Dig. of the International Electron Devices Meeting*, pages 241–244, December 1995.

[3] M. Bohr. Silicon trends and limits for advanced microprocessors. *Communications of the ACM*, 41(3):80–87, March 1998.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] D. Lammers. IBM's copper interconnects hit the market. EETimes, 9/3 issue, September 1998.

[15] D. Lammers. TI's 0.13-micron process speeds system-on-a-chip designs. EETimes, 10/23 issue, October 1998.

[16] G. McFarland and M. Flynn. Limits of scaling mosfets. CSL TR-95-62, Stanford University, November 1995.

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

[18] S. Oh, K. Rahmat, O. Nakagawa, and J. Moll. A scaling scheme and optimization methodology for deep sub-micron interconnect. In *IEEE International Conference on Computer Design*, pages 320–325, October 1996.

[19] 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.

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

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

[22] G. Reinman, B. Calder, and T. Austin. Scalable multi-level instruction fetch prediction. Technical Report UCSD-CS99-613, University of California, San Diego, March 1999.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Compaq WRL TR-93-5, July 1994.

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

[31] 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.