The Fetch Block Predictor: Implementing a Higher-Bandwidth Fetch

Kenneth Barr Albert Hou Jennifer Kiessel Eric Marsman
Computer Architecture, University of Michigan
{kbarr, bertwho, jkiessel, emarsman}@engin.umich.edu

1 Introduction

In the past several years, CPU speed has complied with Moore's law, doubling every 18 months. Meanwhile, memory access time has improved by only a tenth per year. This has caused instruction fetch to become an increasingly significant bottleneck in pipelined processors. No matter how many execution resources are introduced, they cannot improve performance unless the fetch stage provides sufficient instructions on which to operate.

There are three main impediments to a high-bandwidth fetch: branch mispredictions, I-cache misses, and the current limitation of branch predictors that make only one prediction at a time. We choose to focus on the latter impediment and propose a Fetch Block Predictor (FBP). When teamed with an instruction cache that can provide multiple, randomly accessed blocks of instructions and an instruction window to hold this stream of instructions, the FBP will enable higher fetch bandwidth by enabling the fetching of basic blocks of instructions on every cycle instead of just one instruction per cycle. After implementing a fetch unit which can fetch a "basic block" of instructions rather than just one, we extend our design to take advantage of easy-to-predict biased branches which make up more than half of all branches. The FBP can now essentially predict several branches per cycle, effectively addressing the fetch bottleneck problem at a very small hardware cost.

2 Design

Our design is implemented and tested by modifying the contents and the role of the Branch Target Buffer (BTB) on the SimpleScalar simulator [1]. Care was taken to code only mechanisms that could be implemented in hardware.

In each of the project's four phases (sections 2.1 - 2.4), we measure the fetched instructions per cycle (fetchIPC) of the specINT95 benchmarks running on the SimpleScalar simulator. Table 1 shows the data set used for each benchmark. Each was run for 100 million instructions.

We assume a five-stage pipeline, and we assign representative penalties to mispredictions from the BTB (a one-cycle penalty) and direction predictor (a two-cycle penalty). Direction predictions are provided by a gshare predictor with 15 bits of global history; branch targets are predicted by a 4-way set associative BTB with 512 sets.

Table 1: Inputs to specINT95 benchmarks

Program	Input
compress	ref
gcc	jump
go	9stone21
ijpeg	penguin.ppm
li	ref
m88ksim	ref
perl	scrabbl
vortex	ref

2.1 Baseline measurements

As expected, the baseline fetchIPC is less than one due to mispredicted branches:

Table 2: Baseline fetchIPC

Benchmark	FetchIPC
compress	0.97
gcc	0.97
go	0.97
ijpeg	0.99
li	0.98
m88ksim	1.00
perl	0.99
vortex	0.99
specINT95 average	**0.98**

2.2 Fetching Basic Blocks

The next step was to transform the BTB so that it may fetch a "basic block" of instructions in one cycle to improve the default implementation that fetches just one instruction per cycle. A basic block is defined as a sequence of instructions between an entry point (i.e. the target of a branch) and a control instruction. We allow a basic block of up to 16 instructions to be brought into a buffer from instruction memory. As branch instructions tend to be located after every four or five instructions, 16 instructions is a reasonable selection to accommodate most blocks. The BTB is now indexed by the entry point of the block, instead of the branch instruction's address. Our mechanism uses the same 15-bit history gshare predictor to offer a direction prediction for the branch that terminates the block. Four bits have been added to the BTB to store the predicted next block size -- up to 16 instructions. The BTB continues to store the target address, which now refers to the branch terminating the block.

The BTB has been further modified to look up predicted block sizes even in cases where the direction and/or target prediction is ignored. A good example is the treatment of jump register (JR) instructions, which are often used as returns from subroutines. The JR instruction gets its target from the Return Address Stack (RAS) by virtue of the push/pop behavior exhibited by function calls. However, even though the BTB's target is being ignored, the predicted size of the next block to fetch is still an important piece of information to return. With the predicted next block size, the processor can reasonably know how many instructions reside in the next basic block.

Table 3 shows the results of fetching a basic block of multiple instructions in single cycle. The average fetchIPC is dramatically increased!

Table 3: Fetching basic blocks

Benchmark	**FetchIPC**
compress	3.01
gcc	3.07
go	3.69
ijpeg	3.60
li	2.85
m88ksim	2.80
perl	3.41
vortex	4.07
specINT95 average	**3.31**

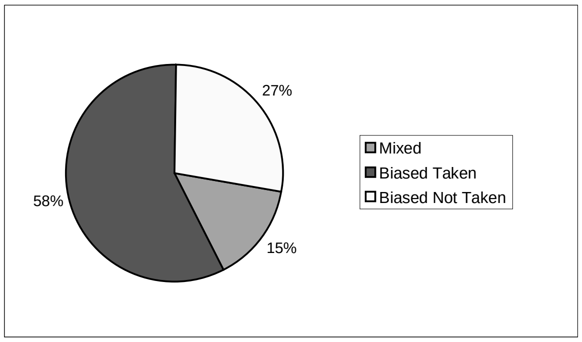
2.3 Fetching Past Not-Taken Branches

Table 4 and Figure 1 show the classification of branches in the specINT95 benchmarks. Branches are classified with the simple method presented in [2]. Branches that are taken <=10% of the times they appear are labeled biased not-taken. Branches which are taken on >=90% of their occurrences are called biased taken. The rest are considered unbiased. This method reveals that 85% of branches are biased. This large percentage is more believable when one notes that many of the specINT95 benchmark programs (especially compress and lisp) are dominated by very few distinct branches that get executed many times [4].

Table 4: Branch classification in specINT95

% Taken	**Occurrences**	**% of all branches**
≤ 5%	11807	26.29%
≤ 10%	534	1.19%
≤ 50%	3303	7.36%
≤ 90%	3313	7.38%
≤ 95%	628	1.40%
≤ 100%	25320	56.39%

Figure 1: Biased branches in specINT95



Furthermore, depending on test methodology, one sees discrepancies in results (e.g., 60% of specINT92 branches are labeled as biased by [4] while [2] reports only 50% biased). We also note that neither [2] nor [4] make allowances for branches that are executed only once. By the definition in [2], they are classified as biased, but are never seen again. Thus they do lend themselves to repeated behavior or easy prediction and should not be considered truly biased.

To complicate the issue further, our own run-length algorithm for classifying branches suggests that just 38% of branches can be called strongly biased. This approach keeps a running record of the streak of taken/not-taken for a particular branch. After execution finishes, an average run-length (either taken/not-taken) is calculated for each branch. We consider branches with average run length of at least four to be biased.

Nevertheless, no matter which figure is most representative of real-life workloads (in the worst case 38%), it is still clear that biased branches are a common case. Given that Amdahl's Law demands we speed up the common case, these biased branches are well worth our attention.

A simple modification can be made to our initial design in order to take advantage of the 27% of branches that are biased not-taken (Table 4). In this phase of our design, a two-bit saturating counter (initialized to biased not-taken) is added to each BTB entry to predict whether the block-terminating branch is a not-taken branch that can be treated as a NOP. This technique was inspired by the Fetch Target Buffer in [3]. Our version of the FTB will allow the fetch cycle to bring in multiple contiguous basic blocks in one cycle if they are separated by not-taken branches. Since approximately 27% of all branches are biased not-taken, the number of instructions that can be fetched per cycle can be increased 27% of the time.

Note that the maximum block size in this design phase is increased to 32 instructions (five bits of storage per BTB entry). This increase is warranted because the modified BTB now has the capability to fetch multiple blocks around not-taken branches.

Table 5 shows the resulting increased fetchIPCs after implementation of fetching past not-taken biased branches.

Table 5: Fetching past NT biased branches

Benchmark	**FetchIPC**
compress	3.5454
gcc	2.9967
go	4.1926
ijpeg	3.7751
li	3.0619
m88ksim	3.0727
perl	3.7221
vortex	4.7380
specINT95 average	3.6381

2.4 Fetching Around Taken Branches

The final step was to modify the design to take advantage of biased taken branches as well – an idea not yet explored in literature. This was accomplished by adding a second number-of-instructions field, a second bias counter, and a second target address to the BTB.

An additional complication that arose from predicting the results of taken branches is saving data for additional cycles. When connecting two blocks joined by a biased not-taken branch, we simply increment the number of instructions to fetch, because the two blocks will appear consecutively in memory. However, when linking two basic blocks over a biased taken branch, we must save the information from the first block until after the second block is fetched on the next cycle. We can update the BTB (indexed by the entry point of the first block) with the information we acquire once we have executed the second block.

Table 6 shows our results in fetching through biased-taken branches.

Table 6: Fetching through biased-taken branches

Benchmark	**FetchIPC**
compress	3.6119
gcc	3.0346
go	4.2506
ijpeg	3.7750
li	3.0807
m88ksim	3.0153
perl	3.6724
vortex	4.7193
specINT95 average	3.6450

2.5 Design Tradeoffs and Concerns

In all stages of design, we had to be conscious to implement in SimpleScalar only such things as could be realized in hardware. Specifically, it was important to 1) limit the amount of information stored in the BTB and 2) discipline our BTB access patterns. The BTB can strictly perform one update for the block that was just executed, in addition to one lookup for the block that will be executed next, per single clock cycle.

There is obviously an increase in the size of the BTB. In the baseline SimpleScalar implementation a BTB entry contained a 10-bit tag, 29-bit branch target (32-bits minus 3-bits for alignment of an 8-bit instruction), 3-bit opcode, and 5-bits of LRU information (implemented as two LRU chaining pointers in SimpleScalar), for a total of 47-bits per entry. For the modifications added in phase 1 of our design to fetch basic blocks, we add an additional 4-bit “block size” field to the BTB for a total of 51-bits. For part 2, we add an additional bit to “block size” and added a 2-bit bias counter, for a total of 54-bits per entry. Finally, in part 3 of the design, the additional 29-bit branch taken block target, 5-bit size, and 2-bit bias combine for a total of 90-bits per BTB entry. However, if bandwidth (rather than die-size) is a driving force, the 372% increase in fetchIPC that we gain more than makes up for the 191% increase in BTB area.

A final concern is that our simulation is relying on information that a real hardware implementation would not have. For example, SimpleScalar can execute a control instruction and immediately determine whether it is taken/not-taken, and more importantly if the prediction from the direction predictor was correct. SimpleScalar can then use this information to correctly fetch the next instruction/block without paying penalties. However, to account for this fact, we assign penalties when appropriate in calculating the IPC. The SimpleScalar tool set has always operated in this manner; the modified BTB is not abusing the data provided by the simulations.

3 Verification

Testing of the modified BTB was based principally on three methods – execution traces, ad hoc testing of block sizes, and stress testing. The combination and thoroughness of testing involved in the development of the FBP ensures the accuracy of fetchIPC results and precise evaluation of its design tradeoffs.

3.1 Execution Traces

To verify that implementation changes did not break the SimpleScalar simulator, traces of the execution paths through each of the benchmarks were used. A trace execution is constructed by running through a benchmark and recording for each cycle:

- cycle number
- PC
- index into fetched instruction block
- opcode

First, a trace standard was constructed by running the baseline SimpleScalar simulator and recording data during each cycle. Next, following each design phase, we ran the benchmarks for roughly 100,000 instructions and recorded the same data. This new trace was compared with the established trace standard. If no discrepancies were found (using UNIX utilities such as diff and sdiff) then the new implementation was deemed verified – that is it did not violate program semantics.

3.2 Ad hoc Testing

Ad hoc testing methods were employed to verify that the modified BTB worked according to the design specifications. These included print statements to 1) mark control instructions that were ignored because they were strongly biased branches, 2) output the value of the 2-bit bias counters, and 3) inform when the BTB had mispredicted a target or block size. This also involved making sure that corner cases worked such as misses in the BTB (e.g. compulsory misses), reaching the end of a block without encountering a branch, reaching the limits of the fixed instruction window, and accounting for unusual sequences of branches and targets. Ad hoc testing is by its nature hard to document; however it is certainly extremely useful for verifying compliance to design specifications.

3.3 Stress Testing

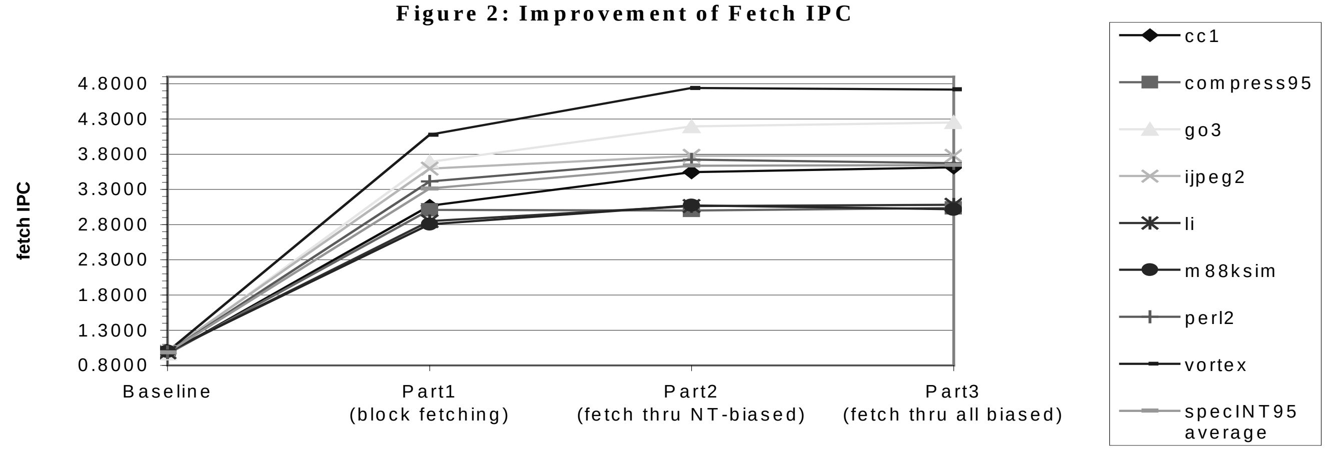
As a final test measure, the modified BTB was run with 100 million instruction test cases. At the end of these simulations, the results of the statistical analysis were checked for consistency and sanity (e.g. % of control instructions, % branches predicted correctly, etc). Furthermore, if the new design did not break after 100 million instructions, then that was seen as a good sign of robustness of the implementation.

4 Evaluation of Design

The performance improvement offered by the various types of FBPs is shown in Figure 2. From the graph, it is clear that the proposed fetch block predictor improves the fetched instructions per cycle. As the FBP is enhanced to take advantage of the presence of all biased branches, the improvement grows to 3.645 fetchIPC on average. This is a 372% improvement in fetch bandwidth over the baseline average.

The performance gained by allowing fetching through taken branches was surprisingly small. This can most likely be attributed to the FBP’s

Figure 2: Improvement of Fetch IPC



ability to fetch through just one taken branch. In situations with two biased taken branches in a row, such as the common loop, we are limited to extending the block to the second taken branch. In addition, it is possible that the "intelligence" of the taken-branch unit helps it fetch instructions that are actually executed by the program whereas the not-taken unit could extend the blocks with instructions that never were used.

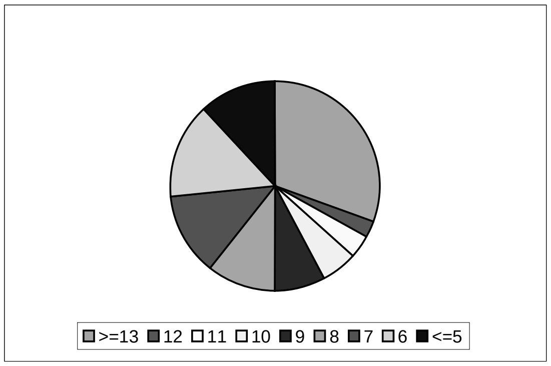
When evaluating our design, we must also consider that to truly be implemented, we need a sophisticated memory that can return non-consecutive blocks in a single cycle. (Recall that this is done when fetching around biased-taken branches.) A scheme to accomplish this is the striped I-cache, which could allow for multiple simultaneous accesses into the instruction memory. For the purposes of this experiment however, we assume that such hardware exists and is readily available.

To reduce the resources necessary to build the FBP, one could try to store target offsets rather than entire 29-bit addresses. Figure 3 shows the number of bits needed to encode the offset of a branch target in the specINT95 suite. Eight bits are adequate to encode the target offset of half the specINT95 branches. Twelve bits handle about 70%. Either choice seems preferable to storing the full 29-bit target in the BTB. Obviously this would require logic in the fetch unit to perform the address computation. One would have to weigh the size of this logic against the size it removes from the BTB; since it liberates up to 3KB (2048*12 bits) from the BTB (13.3% of the current size), the use of offsets seems like a wise decision.

5 Conclusion and Suggested Research

Since the implementation of the FBP is simple, one should consider using these concepts in place of or in addition to the trace cache, another scheme proposed to improve fetch bandwidth.

Figure 3: Bits needed to present target offset



In the process of our design we experimented with "smarter" FBPs that could remember block sizes through a single deviation in branch direction (e.g., a loop). Surprisingly, the fetch bandwidth of our preliminary designs never exceeded the "stupid" predictor that overwrites its information when a branch deviates from its bias. Still, we believe that an FBP that was tolerant of loop behavior would further improve fetchIPC. Perhaps using a confidence mechanism would insure that the work of stitching two blocks together only occurred if the device was convinced the separating branch was truly biased. The smart FBP must also deal with the address of one entry point referring to several different control instructions in the predictors. This could be fixed by indexing the direction predictor with the address of the specific control instruction while continuing to index the BTB with the address of the block's entry point.

In conclusion, it is apparent from our findings that the fetchIPC is improved by a straightforward transformation of the BTB into an

FBP. The higher bandwidth of the fetch stage will reduce the effects of costly memory accesses and create a faster pipeline. Further research will help to improve the FBP mechanism so that it may have an even greater effect on fetch bandwidth.

References

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

[2] P-Y. Chang, E. Hao, T-Y. Yeh, and Yale Patt. Branch Classification: A New Mechanism for Improving Branch Predictor Performance. In *Proceedings of the 27th International Symposium on Microarchitecture*, November, 1994.

[3] G. Reinman, T. Austin, and B. Calder. A Scalable Front-End Architecture for Fast Instruction Delivery. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[4] S. Sechrest, C-C. Lee, and T. Mudge. The Role of Adaptivity in Two-Level Adaptive Branch Prediction. 28th Annual IEEE/ACM Symp. Microarchitecture (MICRO-28), December, 1995.

Code

All tables, figures, and performance results included in this paper were generated by code produced by the authors. The code, modifications to SimpleScalar's sim-bpred.c, bpred.c, and bpred.h, may be found in /afs/engin.umich.edu/courses/f99/eecs470/fbp. Refer to the following table for details.

Table 7: Locations of project code

Project	**Directory**
Branch classification	kbarr/src/runlength_simple
Branch Target Distance Distribution	kbarr/src/distance
fetchIPC for baseline implementation	kbarr/src/fetchipc1122
fetching basic blocks	emarsman/golden/part1
fetching basic blocks through not-taken branches	emarsman/golden/part2
fetching basic blocks through taken branches as well	emarsman/golden/part3