The 7th ACM International Conference on Supercomputing
pp. 67 - 76, July 19 - 23, 1993, Tokyo, Japan.

Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache

Tse-Yu Yeh, Deborah T. Marr, Yale N. Patt
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109

Abstract

High performance computer implementation today is increasingly directed toward parallelism in the hardware. Superscalar machines, where the hardware can issue more than one instruction each cycle, are being adopted by more implementations. As the trend toward wider issue rates continues, so too must the ability to fetch more instructions each cycle. Although compilers can improve the situation by increasing the size of basic blocks, hardware mechanisms to fetch multiple possibly non-consecutive basic blocks are also needed. Viable mechanisms for fetching multiple non-consecutive basic blocks have not been previously investigated.

We present a mechanism for predicting multiple branches and fetching multiple non-consecutive basic blocks each cycle which is both viable and effective. We measured the effectiveness of the mechanism in terms of the IPC_f, the number of instructions fetched per clock for a machine front-end. For one, two, and three basic blocks, the IPC_f of integer benchmarks went from 3.0 to 4.2 and 4.9, respectively. For floating point benchmarks, the IPC_f went from from 6.6 to 7.1 and 8.9.

1 Introduction

Recent advances in computer architecture have focused on increasing the parallelism of the hardware in order to exploit instruction-level parallelism in the dynamic instruction stream. As these architectures become increasingly parallel, it is important to fetch more and more instructions each cycle. This can be done either by increasing basic block size and fetching the entire block in a single cycle, or by fetching multiple basic blocks per cycle. The optimal solution may be to combine both. The first alternative is being researched and implemented in today's advanced compilers. One approach is to enlarge the basic block into traces [3] or into superblocks [11]. Another approach is to exploit predicate execution to schedule instruction execution along multiple execution paths [4]. The disadvantage of predicate execution is that execution bandwidth is wasted on instructions whose results are discarded, and instruction fetch bandwidth is wasted on instructions which will not be executed.

0

Here we propose a scheme which allows us to fully utilize the fetch and execution bandwidth with useful instructions from a dynamically predicted path.

There are three essential components to providing the ability to fetch multiple basic blocks each cycle:

- Predicting the branch paths of multiple branches each cycle.
- Generating fetch addresses for multiple and possibly non-consecutive basic blocks each cycle.
- Designing an instruction cache with enough bandwidth to supply a large number of instructions from multiple, possibly non-consecutive basic blocks.

This paper provides an integrated solution for these problems. We introduce a highly accurate branch prediction algorithm capable of making predictions for multiple branches in a single cycle, a branch address cache to provide the addresses of the basic blocks to which the branches direct the instruction flow, and an instruction cache configuration with a suitably high bandwidth. Although hardware intensive, these solutions are not excessively so for the coming generation of processor implementations.

If we can correctly predict two to three branch paths every cycle and if the average basic block size is five instructions, then the average fetch size is 10 to 15 instructions. Many non-numeric applications today have an average basic block size of 5 instructions, and floating point applications tend to be much larger. The ability to fetch multiple basic blocks per cycle coupled with compiler technology to increase basic block size can result in significant performance gains. This paper shows that just providing the ability to fetch multiple instructions without specific compiler optimizations already increases the useful instruction fetch capacity of a machine by 40% and 63% when 2 and 3 basic blocks can be fetched each cycle, respectively, for integer benchmarks. For floating point benchmarks, the improvement is 8% and 35%.

This paper is organized in 6 sections. Section 2 summarizes some related work. Our multiple branch prediction algorithm is based on the Two-level Adaptive Branch Predictor [6, 7, 10]. The Two-level Adaptive

Branch Predictor achieves an average of 97% accuracy for a single branch prediction. Also, an instruction supply mechanism [9] to do back-to-back branch predictions and supply up to one basic block per instruction cache fetch is briefly reviewed.

Section 3 provides an overview of the multiple basic block supply mechanism. Section 3.1 describes the multiple branch prediction algorithm. Section 3.2 presents the structure and operation of the branch address cache. Section 3.3 discusses the instruction cache design issues. Section 4 describes the simulation model, and the benchmarks used, and Section 5 shows our simulation results. Finally, Section 6 concludes the paper.

2 Related Work

2.1 Two-level Adaptive Branch Predictor

Yeh and Patt [6, 7, 10] introduced several implementations of the Two-level Adaptive Branch Predictor, each with somewhat different cost vs. prediction accuracies. The average prediction accuracy on the SPEC89 benchmarks was shown to be 97 percent. One important result was that each of the different Two-level Adaptive Branch Prediction schemes can achieve the same accuracy by varying its configuration. The following is a brief overview of a few schemes. The interested reader is referred to [6, 7, 10] for more details.

The Two-level Adaptive Branch Predictor uses two structures, a *Branch History Register* (BHR) and a *Pattern History Table* (PHT), as shown in Figure 1. The Branch History Register is used to record the history of taken and not taken branches. For example, if the recent history of the branch behavior is: taken twice, not taken, and taken again, the BHR would contain the *pattern* 1101, where 1 indicates taken, and 0 indicates not taken.

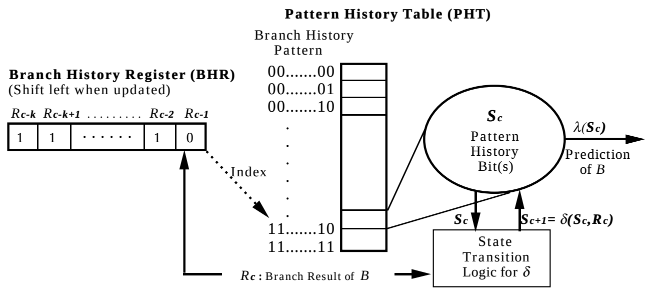
Pattern History Table (PHT)
Branch History Pattern
Branch History Register (BHR)
(Shift left when updated)
R_{c-k} R_{c-k+1} .... R_{c-2} R_{c-1}
1 1 ······· 1 0
Index
00.......00
00.......01
00.......10
11.......10
11.......11
S_c
Pattern History Bit(s)
$\lambda(S_c)$
Prediction of B
S_c
$S_{c+1} = \delta(S_c, R_c)$
State Transition Logic for δ
R_c : Branch Result of B

Figure 1: Basic structure of Two-level Adaptive Branch Prediction.

In addition, for each possible pattern in the BHR, a pattern history is recorded in the PHT. If the BHR contains k bits to record the history of the last k branches, then there are 2^k possible patterns in the BHR. Therefore the PHT has 2^k entries, each entry containing a 2-bit up-down saturating counter to record the execution history of the last several times the corresponding pattern occurred in the BHR. [7] showed that the 2-bit up-down saturating counter was sufficient in keeping pattern history to give highly accurate branch predictions. Prediction decision logic interprets the two pattern history bits to make a branch prediction. When the 2-bit up-down saturating counter is used, the prediction is based on the high-order bit of the counter value.

For example, if the BHR were 4 bits wide, the Pattern History Table would have $2^4 = 16$ entries. Suppose that each entry in the Pattern History Table contains 2 bits with initial value of 01, and that the last two times the pattern 1101 showed up the BHR, the branch was taken. Then the 1101_2-th entry of the Pattern History Table will contain 11 and the next prediction when the BHR has the pattern 1101 will be predicted as taken.

Based on the source of the first-level branch history, Two-level Adaptive Branch Prediction has three classes of variations: global history schemes, per-address history schemes, and per-set history schemes. Global history schemes (also called Correlation Branch Prediction [8]) use a single Global BHR to record the history of all branches. The pattern in the Global BHR is used to index into the PHTs. The prediction of a conditional branch is influenced by the history of other branches. Per-address history schemes use one BHR per static branch; therefore, multiple BHRs are used in the scheme. The prediction of a conditional branch is influenced by the history of the branch itself. Per-set history schemes use one BHR to record the history of a set of adjacent static branches. The prediction of a conditional branch is influenced by the history of the branches in the same set, not just the branch itself.

2.2 Instruction Supply

In [9] an instruction supply mechanism was introduced where up to one basic block per cycle can be fetched by predicting branch targets in back-to-back cycles. We summarize a few details of the mechanism in this section, but the interested reader is referred to [9] for more details.

We will use the term *fetch address* to be the address used to fetch a sequence of instructions from the instruction cache. Three things are done at the same time: the instruction cache access, the branch address cache access, and the branch path prediction. The fetch address is used for both the instruction cache access and the branch address cache access from which a fall-through address, target address, and branch type are retrieved.

If the instructions fetched include a branch, those instructions up to and including the branch instruction comprise one basic block. Instructions after the branch are not issued to the processor until the next branch prediction is made.

If the fetch address misses in the branch address cache, then either there is no branch in the sequence of instructions fetched, or the sequence is being fetched for the first time. In either case, the fetch address is incremented by the fetch size, and the hardware continues fetching the next sequential block of instructions. In the event that a branch instruction is discovered after the instructions are decoded, the fall-through address, target address, size, type of branch, and branch path are recorded in the branch address cache.

If the fetch address hits in the branch address cache, then we know that there is a branch somewhere in the sequence of instructions just fetched. Since the information from the branch address cache is available at the same time that the instructions are fetched from the instruction cache, a new fetch address (either the fall-through address or the taken address) can be determined immediately. The next instruction cache and branch address cache accesses begin on the next cycle.

3 Fetching Multiple Basic Blocks Each Cycle

The performance of the mechanism described in [9] and summarized in Section 2.2 limited the fetch capacity to one basic block per cycle. Since only one branch path prediction and only one set of consecutive instructions could be fetched from the instruction cache per cycle, instruction fetch stopped when a branch was encountered. This was due to the limitation of a single prediction per cycle and limitations in the instruction cache configuration.

Fetching multiple basic blocks each cycle requires more a multiple branch prediction algorithm. At the same time that multiple branch paths are being predicted, the addresses of the basic blocks following those branches must be determined. In addition, the instruction cache must be able to supply multiple nonconsecutive blocks of instructions in a single cycle. Our solutions to these issues are:

- The Multiple Branch Two-level Adaptive Branch Predictor which provides highly accurate predictions for multiple branch paths.
- The Branch Address Cache (BAC) which is a hardware structure to provide multiple fetch addresses of the basic blocks following each branch.
- An instruction cache with enough bandwidth to supply a large number of instructions from nonconsecutive basic blocks.

For this paper we will describe the mechanisms for fetching two and three basic blocks each cycle. The mechanisms described can be easily extended to more than three branches, but the hardware cost increases exponentially with each additional basic block.

3.1 The Multiple Branch Two-Level Adaptive Branch Predictor

The prediction algorithm for a single branch per cycle described in Section 2.1 can be extended to two branch predictions per cycle. We will henceforth identify the first branch as the *primary branch*, the second branch as the *secondary branch*, and the third branch as the *tertiary branch*.

The *primary basic block* is the basic block dynamically following the primary branch (i.e. the basic block containing a secondary branch). There are two possibilities for the primary basic block: the target and the fall-through basic blocks of the primary branch. These will be denoted as T or N, depending on whether the primary branch was taken or not taken. The *secondary basic block* is the basic block following the secondary branch. The secondary basic block can be one of up to 4 different blocks depending on the direction of the primary and the secondary branches. These will be denoted TT, TN, NT, or NN, depending on whether the primary and secondary branches were taken-taken, taken-not taken, not taken-taken, or not taken-not taken, respectively. Finally, the *tertiary basic block* is the one following the tertiary branch. The tertiary basic block can be one of 8 different blocks depending on the outcome of the primary, secondary, and tertiary branch paths, and its denotations are TTT, TTN, TNT, etc.

Figure 2 shows the primary and secondary branches and the primary and secondary basic blocks for the case when two predictions are made per cycle. If the darker branch paths are predicted, the darker basic block boxes are fetched.

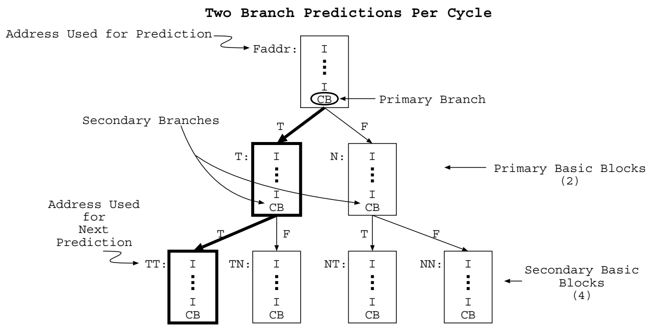
Two Branch Predictions Per Cycle
Address Used for Prediction
Faddr:
I
I
CB
Primary Branch
Secondary Branches
T
F
T:
N:
Primary Basic Blocks (2)
Address Used for Next Prediction
TT:
TN:
NT:
NN:
Secondary Basic Blocks (4)

Figure 2: Identification of the primary and secondary branches, and the primary and secondary basic blocks.

The multiple branch prediction algorithm introduced in this paper is modified from the global history schemes of Two-Level Adaptive Branch Prediction described in [10] and summarized in Section 2.1. The modified global history schemes not only make the prediction of the immediately following branch but also extrapolate the predictions of subsequent branches. The per-address history and per-set history schemes of Two-Level Adaptive Branch Prediction, on the other hand, require more complicated BHT accessing logic for making multiple branch predictions in each cycle because they use may different branch history to make predictions for different branches. In order to simplify the BHT design, we consider only the global history schemes in this paper.

The first multiple branch prediction variation is called *Multiple Branch Global Two-Level Adaptive Branch Prediction using a Global Pattern History Table* (MGAg). This scheme uses a global history register of k bits and a global pattern history table of 2^k entries, each entry containing 2 bits. The k bits in the history register record the outcome of the last k branches. The history register is updated speculatively with the predicted branch outcomes and corrected later in the event of an incorrect prediction because the prediction accuracy is expected to be high. The right-most bit corresponds to the prediction of the most recent branch, and the leftmost bit corresponds to the prediction of

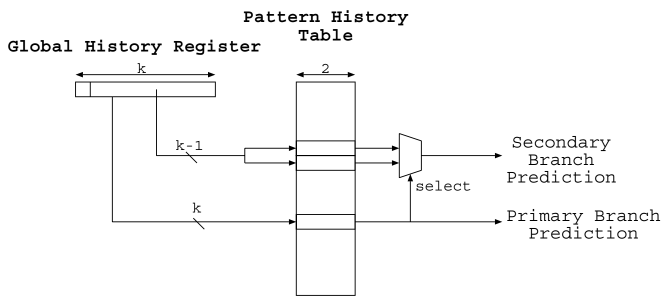

Figure 3: Algorithm to make 2 branch predictions from a single branch history register.

the oldest branch. As shown in Figure 3, all k bits in the history register are used to index into the pattern history table to make a primary branch prediction. The 2-bit counter value read from the pattern history table entry is used to make the prediction.

To predict the secondary branch, the right-most $k-1$ branch history bits are used to index into the pattern history table. $k-1$ bits address 2 adjacent entries, $BHR_{k-1..0}0$ and $BHR_{k-1..0}1$, in the pattern history table. The primary branch prediction is used to select one of the entries to make the secondary branch prediction. Finally, the tertiary prediction uses the right-most $k-2$ history register bits to address the pattern history table and access 4 adjacent entries. The primary and secondary predictions are used to select one of the 4 entries for the tertiary branch path prediction. This algorithm allows each of the multiple branch path predictions to take full advantage of k bits of branch history. Longer history registers increase the prediction accuracy, and as multiple branches are predicted, the accuracy becomes increasingly important.

The second multiple branch prediction variation is called *Multiple Branch Global Two-Level Adaptive Branch Prediction using Per-set Pattern History Tables* (MGAs). It differs from the previous scheme in that there are multiple pattern history tables. The pattern history tables are associated with the primary branches. Similar to MGAg, all k bits are used to index into a pattern history table to make a prediction for the primary branch. The pattern history table is selected based on the fetch address corresponding to the primary branch. The second prediction is made from the same pattern table since the address of the secondary branch is not known at the time of the prediction. This scheme attempts to limit the amount of pollution in the pattern history tables by different branches, but may result in less accurate secondary and tertiary branch predictions.

The extreme case of the MGAs scheme is when there is a separate pattern history table associated with each branch. This scheme is called *Multiple Branch Global Two-Level Adaptive Branch Prediction using Per-address Pattern History Tables* (MGAp).

The pattern table entries are updated after the branch instructions are resolved, which could take several cycles. Therefore the pattern table entries are always somewhat out-of-date. This is likely to degrade the accuracy of the multiple branch prediction algorithm more than the accuracy of a single branch prediction algorithm. The reason the branch may take several cycles to resolve is that it may have to wait for a condition to be evaluated or an address to be computed which may take several cycles due to data dependencies.

Since the branch predictions are done at the same time the instructions are fetched, the determination of whether there is a branch in a fetch sequence is done through the Branch Address Cache which is described in detail in the next section. If the fetch address hits in the Branch Address Cache, then there is a branch in the sequence being fetched. Otherwise no branch is assumed and the instruction fetch mechanism fetches down the sequential stream.

The branch path predictions made with the Multiple Branch Two-level Adaptive Branch Predictor are done at the same time the Branch Address Cache and instruction cache are accessed. These branch path prediction bits are used to select the fetch addresses that are needed for the next cycle from the possible fetch addresses provided by the Branch Address Cache. For now, we will merely state that if two predictions are made, then two fetch addresses are selected. If three predictions are made, then three fetch addresses are selected.

Multiple predictions might not be made every cycle for several reasons. The first case is when a basic block is very large, so the entire instruction cache bandwidth may be devoted to fetching the basic block. Fetching the primary basic block has higher priority than fetching secondary or tertiary basic blocks. Therefore if we cannot fetch one basic block in its entirety with its instruction cache bandwidth quota, then we allow it to usurp the bandwidth quota from a subsequent block.

If a secondary or tertiary basic block's bandwidth is usurped, the prediction of the branch in that basic block is delayed until the cycle when it is actually being fetched. At that point it becomes the primary branch and a (different) secondary and tertiary branch may be predicted along with it.

The other case when multiple branch path predictions are not made is when the branch is a return instruction. The return instruction's predicted target address is obtained from the return address stack. The next branch is difficult to predict because the return may direct the instruction stream to any number of locations.

3.2 The Branch Address Cache (BAC) Design

With each of the MGAg, MGAs, and MGAp algorithms, we use a Branch Address Cache (BAC) to store the addresses to which the branches may direct the instruction flow. Recall that with the single basic block instruction supply algorithm summarized in Section 2.2 the branch address cache is indexed by the fetch address, from which two potential fetch addresses are obtained (one for the target block and one for the fall-through block). The branch prediction chooses between the two addresses.

The *multiple* basic block supply algorithms use a similar BAC. The fetch address is used to access the BAC. This is done in parallel to the instruction cache access.

Although there may be two or three fetch addresses accessing the instruction cache simultaneously, only *a single fetch address is used to accessed the BAC*. If only one basic block is being fetched, that fetch address is used. If two basic blocks are being fetched simultaneously, the second fetch address is used to access the BAC. If three basic blocks are fetched, the third fetch address is used.

If the fetch address hits in the BAC, there is a branch in the sequence of instructions just fetched. The BAC entry records the branch type (conditional, unconditional, or return) and the target and fall-through basic block starting addresses of the primary branch. The same entry also contains the branch type and fetch addresses of basic blocks for each of the expected number of branches we will make predictions for, and all the known potential fetch addresses of their targets. If the number of basic blocks predicted and fetched per cycle is limited to 2, we get 6 fetch addresses; 2 for the two primary basic block addresses, and 4 for the four possible secondary basic blocks. If the basic block prediction and fetch limit is 3, we get 14 possible fetch addresses; 2 for the primary basic blocks, 4 for the secondary, and 8 for the tertiary basic blocks.

Each entry in a 512-entry, 4-way set associative Branch Address Cache which supports two branch predictions per cycle has the following fields: TAG, P_valid, P_type, Taddr, Naddr, ST_valid, ST_type, TTaddr, TNaddr, SN_valid, SN_type, NTaddr, NNaddr where each field contains:

- TAG field — The 23 high-order bits of the primary fetch address. A "BAC Hit" occurs if the tag matches with the upper address bits of the current fetch address and the primary branch is valid.
- valid bits — The valid bits for the corresponding branch entries. P refers to the primary branch, ST refers to the secondary branch if the primary branch is taken, and SN is the secondary branch if the primary branch is not taken.
- type fields — The branch type of the corresponding branch. The type can be conditional, unconditional, or return. Each type field consists of 2 bits.
- addr fields — The address of the corresponding basic block. Each address field consists of 30 bits.

A BAC supporting two branch predictions per cycle would have a total of 212 bits per entry. A BAC supporting three branches would have an additional eight address fields and four additional valid bits for the four possible tertiary branches, making each entry 464 bits wide. We are investigating several possible ways of reducing the number of fields needed per entry, such as storing only the addresses of more likely-taken path(s).

When a fetch address misses in the BAC, a large basic block is assumed and the entire instruction cache bandwidth is devoted to fetching sequential instructions. If a branch is discovered once the instructions are decoded and the branch is predicted taken or is an unconditional branch, the prefetched instructions after the branch are discarded. The address of the fall-through and target addresses are calculated in the cycle after decode. The branch is then allocated a primary branch entry in the BAC. The higher order bits of the fetch address are entered in the tag field, the primary branch valid bit is set, the secondary (and tertiary) branch valid bits are cleared, and primary fall-through and target addresses are entered. If the branch is an indirect branch, however, the target address is not calculated until the operands are ready, and the valid bit is not set until that time.

The branch will also be entered as a secondary branch in the BAC entry of the previous branch if:

- the previous fetch address had a valid primary branch entry in the BAC but did not predict a secondary branch *and*
- the basic block of the previous fetch address was not oversized (i.e. there was enough instruction cache bandwidth for another basic block fetch) *and*
- the previous branch was not a return.

3.3 The Instruction Cache

The ability of the instruction cache to provide enough instructions becomes critical when multiple possibly non-consecutive basic blocks are fetched each cycle. The instruction cache must have high bandwidth, low miss rate, and the ability to fetch from multiple addresses in parallel.

To satisfy the high bandwidth requirement, the cache must either have a large number of banks, or have wide banks. Also, due to off-chip bandwidth and pin limitations, the instruction cache should be on-chip.

The ability to fetch from multiple addresses in parallel implies a cache with either interleaved or multi-ported banks, or both. With interleaved banks, each independently addressable, multiple fetch addresses can access the instruction cache simultaneously provided that their accesses are not to the same bank. If there is a bank conflict, priority is given to the earlier (relative to the dynamic instruction stream) fetch address. Therefore it is important to have enough banks to make the probability of bank conflicts low.

A multi-ported cache eliminates the bank conflict problem. For example, a dual-ported cache allows the simultaneous access of two fetch addresses, and a tri-ported cache allows the simultaneous access of three fetch addresses. Unfortunately, multi-ported memories are expensive in terms of semiconductor chip area.

It is critical for the instruction cache miss rate to be low. Each instruction cache miss stalls the fetch sequence. Since multiple basic blocks can be fetched each cycle, the opportunity cost can be (up to) the number of cycles it takes to service the miss multiplied by the number of instructions we could have fetched during those idle fetch cycles. Also, since more instructions are fetched each cycle, there are fewer cycles between instruction cache misses. Therefore more time is spent waiting for instruction cache misses to be satisfied. Commonly used ways to minimize instruction cache miss rates are to increase the associativity, to increase the size of the cache, and to prefetch instructions.

We chose several cache configurations which gave us reasonably high bandwidth, the ability to fetch multiple addresses in parallel, and a relatively low miss rate. Most of our simulations were done with a 32K cache which was 2-way set associative with 8 interleaved single-ported banks, each bank having a line size of 16 bytes. Each fetch address can access two banks so that we guarantee between 5 and 8 instructions per fetch address (due to basic block alignment). This configuration and several others were compared in Section 5.

4 Simulation Methodology

4.1 Simulation Environment

We used a trace-driven simulator to evaluate the performance of a machine front-end which implements the Multiple Branch Two-level Adaptive Branch Predictor, a 512-entry 4-way set associative Branch Address Cache (BAC), and a high-bandwidth instruction cache. Unless otherwise specified, the instruction cache configuration used was 32K bytes, 2-way set associative, 8-way interleaved, single-ported, and with a line size of 16 bytes (4 instructions).

For the multiple basic block mechanisms, we can fetch two cache lines (a maximum of 8 instructions) per basic block fetch address because most basic blocks contain 4 to 8 instructions. In order to do a fair comparison, we allow the single basic block prediction and fetch algorithm to fetch up to 4 cache lines. The maximum number of instructions issued, passed to the back-end of the machine, is constrained to 16 instructions per cycle.

The benchmarks written in C were compiled with the Motorola Apogee C compiler for the Motorola 88100 instruction set and the ones written in Fortran where compiled with the Green Hill Fortran compiler. A Motorola 88100 instruction level simulator generated the instruction traces. The first 50 million instructions from each trace were used rather than the entire trace due to simulation time constraints.

Nine benchmarks were selected from the SPEC89 benchmark suite. These included 4 integer and 5 floating point benchmarks. The integer benchmarks are *li*, *gcc*, *eqntott*, and *espresso*. The floating point benchmarks are *doduc*, *fpppp*, *matrix300*, *spice2g6*, and *tomcatv*. The figures included in the result section have the abbreviations listed in Table 1 for the various benchmarks. Table 1 also shows the average basic block size of the first 50 million instructions of each benchmark.

Benchmark	Abbreviation	Average Basic Block Size
eqntott	eq	4.76
espresso	es	3.41
gcc	gc	4.94
li	li	4.14
doduc	dd	10.46
fpppp	fp	57.01
matrix300	mt	28.20
spice2g6	sp	5.36
tomcatv	tc	26.33

Table 1: Benchmark list and their average basic block size.

The MGAg, MGAs, and MGAp are parameterized according to the history register length and the number of Pattern History Tables. These parameters will be given as: HhPp, where h is the number of bits in the Global History Register, and p is the number of pattern history tables.

4.2 Performance Metric

Since the simulator only models the front end of a machine, we use a new metric, IPC_f, to evaluate the performance of an instruction fetch mechanism. IPC_f stands for the number of effective instructions fetched per cycle by an instruction fetch mechanism. To derive IPC_f, we assume the machine stalls or wastes cycles for various reasons from the instruction fetch mechanism but not from the rest of the machine, so the instructions issued can be executed without stalling the machine front end. Moreover, only effective instructions are counted; instructions fetched down the incorrectly predicted paths are not counted. The machine front end could waste cycles due to the following reasons:

- Instruction cache misses
- Incorrect branch predictions which include incorrect branch path predictions and incorrect fetch address predictions
- Branch Address Cache misses on taken branches

Since we do not simulate the rest of the machine, the exact mispredicted branch penalty is approximated. A 6 cycle mispredicted branch penalty is assumed; therefore, the instructions following an incorrectly-predicted branch will not be fetched until 6 cycles after the branch is fetched. The I-cache miss penalty is assumed to be 10 cycles. We also show how the machine performance changes as the branch misprediction penalty and I-cache miss penalty are varied.

5 Simulation Results

5.1 Effect on Prediction Accuracy and IPC_f of History Register Length

Figure 4 shows how the prediction accuracy changes as we increase the number of bits in the global history register of the MGAg scheme for two branch predictions per cycle. The prediction accuracy is the number of correctly predicted branches over the total number of branches in the dynamic instruction stream. Longer branch histories give better prediction accuracy which is reflected in the rising curves. The hardware cost goes up exponentially with the number of history bits due to the number of pattern history table (PHT) entries required.

The prediction accuracies varied between 91.5 and 98.4% for a branch history register (BHR) length of 14 bits, and between 93.5 and 98.7% for a history register length of 16 bits. The knees of the curves for most benchmarks are reached at a BHR length of 14-bits. We

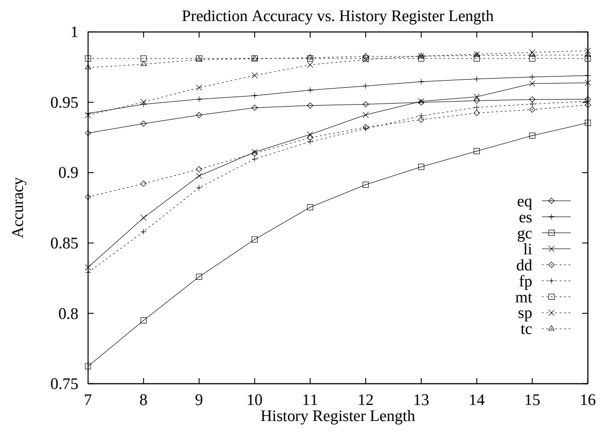

Figure 4: Variation of the size of the global branch history register.

used a 14-bit BHR length for the other experiments reported in this paper. A 14-bit BHR length means that a PHT has $2^{14} \times 2$ bits, or 32K bits.

5.2 Tradeoff between the Number of Pattern History Tables and History Register Length

We simulated several MGAg, MGAs, and MGAp configurations to determine how the performance accuracy changes with the number of PHTs for two branch predictions per cycle. Figure 5 for integer benchmarks and Figure 6 for floating point benchmarks show the IPC_f for 1 to 512 PHTs. Each configuration shown has the same hardware cost, which was achieved by decreasing the number of entries in each PHT as the number of PHTs is increased. Since the entries in the PHTs are addressed by the BHR, the BHR length is reduced when we decrease the number of entries in each PHT.

The PHT used to make the predictions is determined by the primary branch address. The experiments shown in Figures 5 and 6 used the branch address starting at bit 10 to select a PHT. This allows branches within the same 256-instruction block in the static code to map to the same PHT.

The prediction accuracies shown in Figures 5 and 6 tend to be higher for configurations with one to eight pattern history tables, then decreases when the number of pattern history tables is increased beyond 8. Longer branch history helps to increase the prediction accuracy. Increasing the number of PHTs reduces the interference between branches, but since the second branch is predicted using the PHT of the first branch, the probability of mapping two branches predicted together into different PHTs is higher when more PHTs are used.

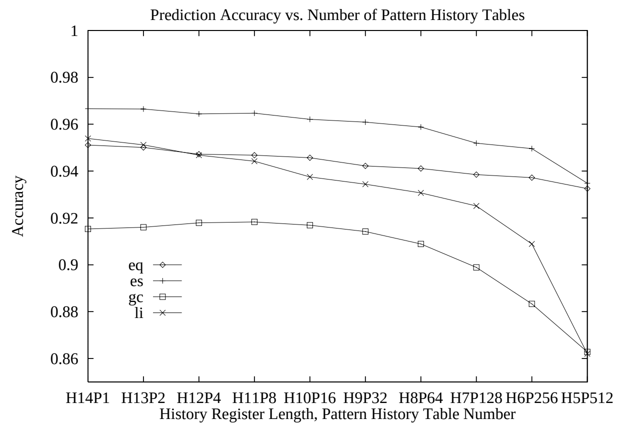

Figure 5: Variation of the number of the PHTs with the hardware cost held constant, for *integer* benchmarks.

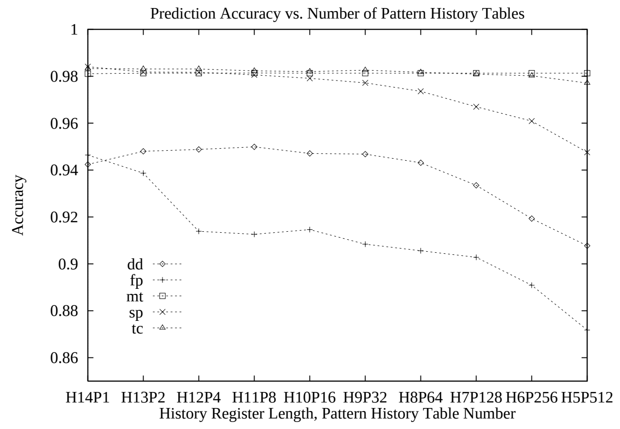

Figure 6: Variation of the number of the PHTs with the hardware cost held constant, for *floating point* benchmarks.

5.3 Number of Branch Predictions per Cycle

Figure 7 shows the IPC_f increase with the number of branch predictions per cycle. The number of opportunities for multiple branch prediction is quite high despite the greater likelihood of bank conflicts in the instruction cache when three basic blocks are fetched.

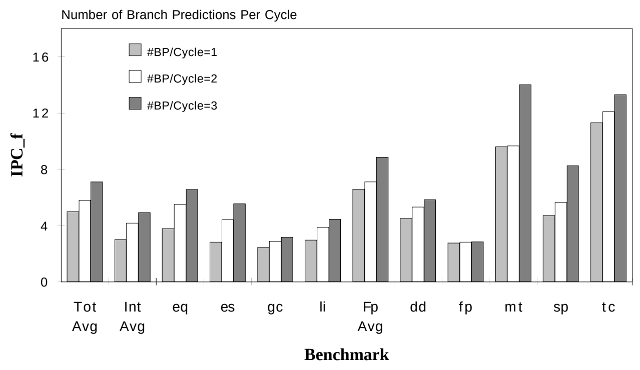

Figure 7: Instructions per cycle when 1, 2, and 3 branches are predicted each cycle.

The average IPC_f when one basic block is predicted and fetched per cycle is 3.0, and 6.6 for integer and floating point benchmarks, respectively. Two predictions per cycle increases this to 4.2 for integer and 7.1 for floating point. Three predictions per cycle increases IPC_f further to 4.9 for integer and 8.9 for floating point.

For the one and two predictions per cycle experiments we allowed a maximum of 16 instructions to be fetched from the instruction cache per cycle. For the three predictions per cycle experiments we increased the instruction cache bandwidth to 24 instruction in order to accommodate the 3 fetch addresses. To cap the number of instructions issued, we constrained the issue width to 16 instructions for all three cases. The larger instruction cache bandwidth allows more instructions to be fetched per cycle, which affects the performance of floating point programs more than integer programs because of the high branch prediction accuracy and large basic block size of floating point benchmarks. This effect results in the significant floating point performance increase when going from two to three predictions per cycle.

fpppp does not show significant performance increase when going from one to two to three predictions per cycle due to the repeated execution of an extremely long sequential code segment which causes the instruction cache to thrash. The instruction cache miss penalty dominates its performance.

Integer programs show noticeable performance increase except for *gcc* which is dominated by incorrect branch predictions.

Figure 8 shows the IPF, instructions per fetch, for the benchmarks as the number of branch predictions and basic block fetches of 1, 2, and 3 per cycle. An efficient instruction fetch mechanism should attain an IPC_f as close to the IPF as possible. The discrepancy between IPF and IPC_f is due to the branch misprediction penalty, BAC misses, and instruction cache miss penalty.

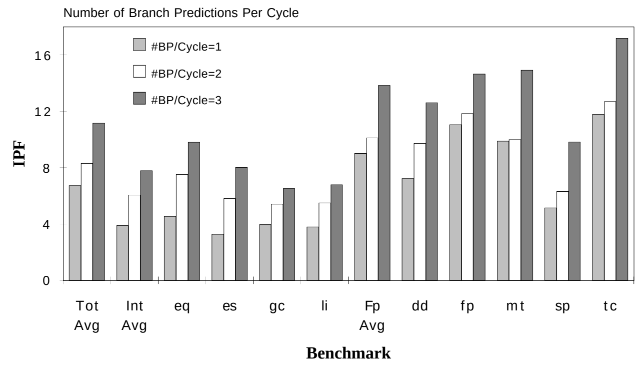

Figure 8: Instructions per fetch when 1, 2, and 3 branches are predicted each cycle.

5.4 Branch Prediction Efficiency

The multiple branch prediction utilization when 2 basic blocks can be predicted and fetched each cycle is shown in Table 2, where we counted the percentage of cycles when zero, one, and two branches were predicted. Zero branches are predicted if we are fetching a long sequential segment of code, or if the fetch address misses in the Branch Address Cache, *and* a branch is found in the sequence of instructions after the instructions are decoded. Fpppp has a high percentage of cycles with no predictions due to the extremely long sequential code segment which is repeatedly executed. The percentage of cycles when zero predictions were done per cycle is 10% per cycle for integer and 44% for floating point.

Benchmark	No Prediction	One Prediction		Two Predictions
		Oversized	Return	
eq	0.0839	0.1231	0.0272	0.7528
es	0.0364	0.1125	0.0145	0.8317
gc	0.1843	0.3634	0.0522	0.3840
li	0.0939	0.2518	0.1213	0.5244
dd	0.3335	0.2580	0.0602	0.3364
fp	0.7415	0.1909	0.0120	0.0550
mt	0.3386	0.3335	0.0042	0.3236
sp	0.2145	0.2142	0.1613	0.4081
tc	0.5893	0.3337	0.0006	0.0751

Table 2: Branch prediction utilization of an instruction fetch mechanism which is able to provide fetch addresses of two basic blocks in each cycle.

Only a single branch is predicted when the primary branch is a return, or the primary basic block is large (oversized) in which case the instruction fetch bandwidth of the secondary basic block is usurped. About 24% of the single basic block fetches are due to oversized basic blocks, and about 5% are due to the primary branch being a return. Two branch predictions are made and two basic blocks are fetched 62% of the time for integer and 24% of the time for floating point benchmarks.

Table 3 shows the percentage of fetches that cause the machine front-end to stall. The machine front-end stalls only due to instruction cache misses, mispredicted branches, and branch decode penalties.

Benchmark	No Delay	Decode Delay	Incorrect Branch Prediction	I-cache Miss	Bank Conflict
eq	0.8924	0.0004	0.0679	0.0001	0.0392
es	0.9207	0.0063	0.0565	0.0001	0.0164
gc	0.7674	0.0708	0.0980	0.0288	0.0351
li	0.8753	0.0202	0.0645	0.0060	0.0340
dd	0.8678	0.0110	0.0452	0.0632	0.0128
fp	0.6357	0.0003	0.0091	0.3508	0.0041
mt	0.6805	0.0000	0.0065	0.0001	0.3129
sp	0.9706	0.0068	0.0150	0.0034	0.0042
tc	0.9905	0.0006	0.0085	0.0001	0.0003

Table 3: Percentage of fetches causing the instruction fetch mechanism to stall.

No_Delay cause no stalls in instruction fetching. Bank_Conflicts to the same cache line do not stall instruction fetch, but conflicts to different cache lines within the same bank do stall instruction fetch. Therefore 84 to 90% of the fetches do not cause any instruction fetch stall. If a taken branch is not detected in a fetched instruction sequence (via a Branch Address

Configuration No.	Number of Interleaved Banks	Number of Read Ports	Set Associativity	Line Size	Fetch Size
0	8	1	2	16	2
1	8	1	1	16	2
2	4	1	2	16	2
3	8	1	4	16	2
4	8	1	2	32	1
5	8	2	2	16	2

Table 4: Instruction cache configurations.

Cache miss), a branch decode penalty is taken. Branch Decode penalties occur in approximately 2.4% and 0.4% of the fetch cycles for integer and floating point benchmarks, respectively. An incorrect branch path prediction requires a full branch penalty to be incurred. This happens about 7.2% and 1.7% of the time for integer and floating point.

5.5 Instruction Cache Configuration

We simulated six instruction cache configurations with various numbers of read ports, degrees of interleaving, set associativities, and line sizes. These configurations are listed in Table 4. Configuration 0 was used for most of our experiments. Fetch size refers to the number of cache lines each fetch address can access.

Figures 9 and 10 show the performance with the various instruction cache configurations. *Gcc* and *fpppp* were chosen because they have more significant instruction cache miss rates. Each curve represents a different cache size. More read ports and more banks reduce bank conflicts but result in only a minimal performance increase. Higher set associativity significantly improves performance. However, *fpppp* actually has better performance with either direct-mapped or 4-way set associative caches due to the large sequential code segment. 32-byte line size degrades the performance a little because some bandwidth is wasted due to basic block alignment.

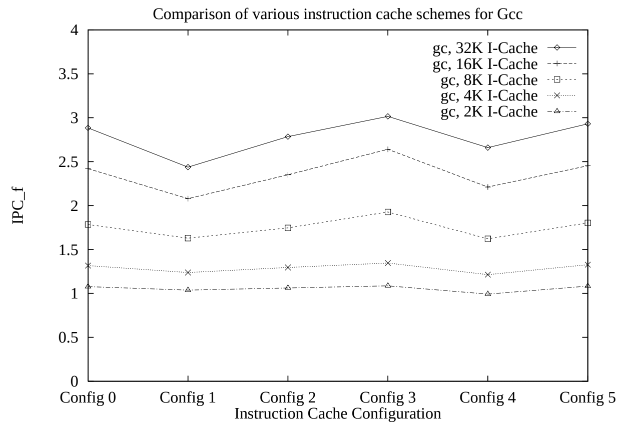

Figure 9: Machine performance of various instruction cache configurations on *gcc*.

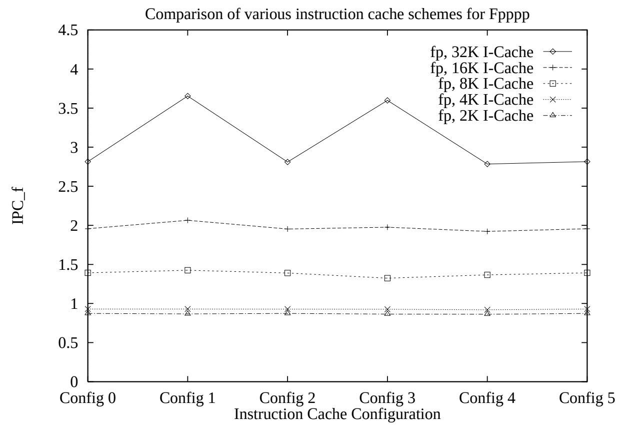

Figure 10: Machine performance of various instruction cache configurations on *fpppp*.

5.6 Effect of Branch Misprediction Penalty

To investigate the effect of branch misprediction penalty on machine performance, we varied the time to resolve a branch from 4 cycles to 12 cycles. Floating point programs have flatter curves because they contain fewer branches and the prediction accuracy of those branches is higher. The performance degradation when the branch resolution time is increased from 4 cycles to 12 cycles is less than 10%. Integer programs have about 20% to 30% performance degradation.

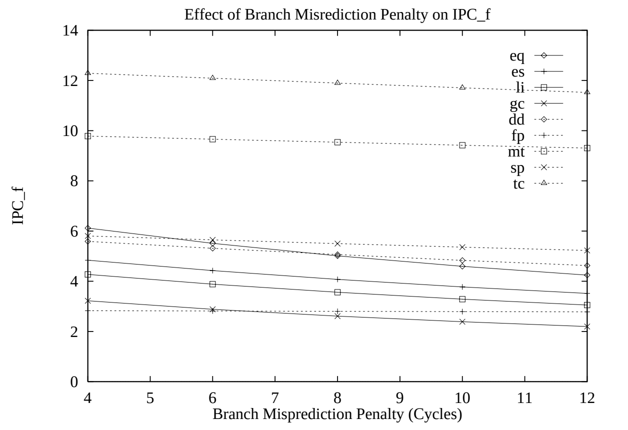

Figure 11: Effect of branch misprediction penalty on machine performance.

5.7 Effect of Instruction Cache Miss Penalty

We varied the instruction cache miss penalty from 4 cycles to 12 cycles. Configuration 0 of Table 4 is used. Among the nine benchmarks, *fpppp*, *doduc*, and *gcc* have lower cache hit rates, as listed in the legend of Figure 12. When the instruction cache miss penalty is increased from 4 cycles to 12 cycles, *Doduc*'s performance degrades by about 20%. *Fpppp*'s performance degrades by about 50%. The other benchmarks showed minimal

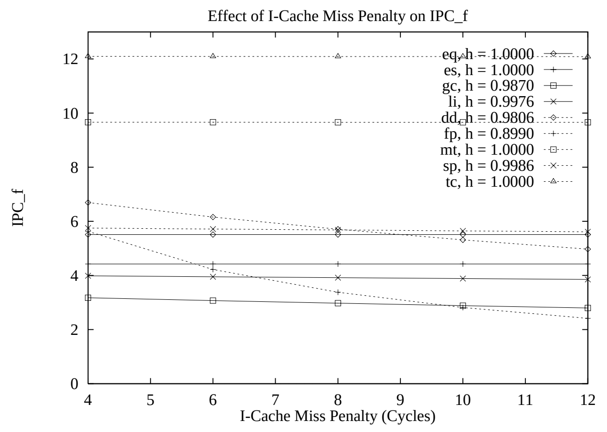

Figure 12: Effect of instruction cache miss penalty on machine performance.

performance degradation due to their low instruction cache miss rates.

6 Conclusion

The trend towards increasingly complex and parallel hardware mechanisms to extract instruction level parallelism from sequential code is advancing at an accelerated rate. Much research has gone into compiler technology to increase basic block size in order to fetch more and more instructions at a time. Increasing basic block size is not enough, however. We propose in this paper a hardware mechanism to fetch multiple basic blocks simultaneously.

We demonstrate in this paper the viability of such schemes by identifying the three essential problems and presenting solutions to each of these. The Multiple Branch Two-level Adaptive Branch Predictor provides the capability of predicting multiple branches each cycle, the Branch Address Cache supplies the starting addresses of basic blocks following the multiple predicted branches, and an instruction cache with interleaved banks provides sufficient bandwidth for fetching multiple non-consecutive basic blocks without the hardware cost of multiple read ports.

In addition, we have presented simulation results indicating that significant performance improvements can be achieved even without specific compiler optimizations. When going from one to two to three branch predictions and basic block fetches per cycle, we saw the IPC_f (instructions fetched per cycle for a machine front-end) improve from 3.0 to 4.2 and 4.9, respectively for integer benchmarks. For floating point benchmarks, the IPC_f went from 6.6 to 7.1 and 8.9. These improvements were achieved by providing the hardware mechanisms to predict and fetch multiple basic blocks without specific compiler optimizations.

Acknowledgement

This paper is one result of our ongoing research in high performance computer implementation at the University of Michigan. The support of our industrial partners: Intel, Motorola, NCR, HaL, Hewlett-Packard, and Scientific and Engineering Software is greatly appreciated. In addition, we wish to gratefully acknowledge the other members of our HPS research group for the stimulating environment they provide, and in particular, for their comments and suggestions on this work. We are particularly grateful to Intel and Motorola for technical and financial support, and to NCR for the gift of an NCR 3550, which is a useful compute server in much of our work.

References

[1] J.E. Smith, "A Study of Branch Prediction Strategies", *Proceedings of the 8th International Symposium on Computer Architecture*, (May 1981), pp.135-148.

[2] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", *IEEE Computer*, (Jan. 1984), pp.6-22.

[3] R. Colwell, R. Nix, J. O'Donnell, D. Papworth, and P. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Proc of the 2nd Intl Conf on Architectural Support for Programming Languages and Operating Systems*, (Oct. 1987), pp. 180-192.

[4] B.R. Rau, D. Yen, W. Yen, and R. Towle, "The Cydra 5 Departmental Supercomputer - Design Philosophies, Decisions, and Trade-offs," *IEEE Computer*, (Jan. 1989), pp. 12-35.

[5] M. Butler, T-Y Yeh, Y.N. Patt, M. Alsup, H. Scales, and M. Shebanow, "Instruction Level Parallelism is Greater Than Two", *Proceedings of the 18th International Symposium on Computer Architecture*, (May 1991), pp. 276-286.

[6] T-Y Yeh and Y.N. Patt, "Two-Level Adaptive Branch Prediction", *The 24th ACM/IEEE Intl Sym and Workshop on Microarchitecture* , (Nov. 1991), pp. 51-61.

[7] T-Y Yeh and Y.N. Patt "Alternative Implementations of Two-Level Adaptive Branch Prediction," *Proceedings of the 19th International Symposium on Computer Architecture*, (May 1992), pp. 124-134.

[8] S-T Pan, K. So, and J.T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, (Oct. 1992), pp. 76-84.

[9] T-Y Yeh and Y.N. Patt "A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution," *Proc of the 25th International Symposium on Microarchitecture*, (Dec. 1992), pp. 129-139.

[10] T-Y Yeh and Y.N. Patt "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," *Proceedings of the 20th International Symposium on Computer Architecture*, (May 1993).

[11] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellete, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, January 1993.