Multiple Branch and Block Prediction

Steven Wallace and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92697
swallace@ece.uci.edu, nader@ece.uci.edu

Multiple Branch and Block Prediction

Steven Wallace and Nader Bagherzadeh
Department of Electrical and Computer Engineering
University of California, Irvine
Irvine, CA 92697
swallace@ece.uci.edu, nader@ece.uci.edu

Abstract

Accurate branch prediction and instruction fetch prediction of a microprocessor are critical to achieve high performance. For a processor which fetches and executes multiple instructions per cycle, an accurate and high bandwidth instruction fetching mechanism becomes increasingly important to performance. Unfortunately, the relatively small basic block size exhibited in many general-purpose applications severely limits instruction fetching. In order to achieve a high fetching rate for wide-issue superscalars, a scalable method to predict multiple branches per block of sequential instructions is presented. Its accuracy is equivalent to a scalar two-level adaptive prediction. Also, to overcome the limitation imposed by control transfers, a scalable method to predict multiple blocks is presented. As a result, a two block, multiple branch prediction mechanism for a block width of 8 instructions achieves an effective fetching rate of 8 instructions per cycle on the SPEC95 benchmark suite.

1 Introduction

The goal of a superscalar microprocessor is to execute multiple instructions per cycle. Instruction-level parallelism (ILP) available in programs can be exploited to realize this goal [4]. Unfortunately, this potential parallelism will never be utilized if the instructions are not delivered for decoding and execution at a sufficient rate [9]. A high performance fetching mechanism is required.

Conditional branches create uncertainty in fetching instructions, which can cause severe performance penalties if not accurately predicted. Also, when a control transfer is detected, its target address must be predicted in order to avoid a stall. Even with perfect dynamic branch prediction, predicting one branch per cycle drastically limits performance, due to small basic block sizes. Multiple branch predictions and multiple target addresses need to be predicted in a single cycle in order to overcome this limitation and achieve a high fetching rate [10].

Researchers have shown how to accurately predict conditional branches. Yeh and Patt introduced a two-level adaptive branch prediction. It uses previous branches' history to index into a Pattern History Table (PHT). They report a 97% branch prediction accuracy [14]. Calder and Grunwald proposed the Next Line Set (NLS), which predicts the next instruction cache line and set to fetch [1]. Both the PHT and NLS were designed for a *scalar* processor and only attempt to fetch one instruction per cycle. Yeh also showed how to perform multiple branch prediction using the PHT and a branch address cache [11]. Unfortunately, the cost of this implementation grew exponentially. In this paper, however, we present a *scalable* mechanism to perform multiple branch and multiple block prediction using the PHT and NLS concepts.

Seznec et. al. [8] recently introduced an innovative way to fetch multiple (two) *basic* blocks. Their idea is to always use the current instruction block information to predict the block following the next instruction block. Its accuracy is as good as a single block fetching and requires little additional storage cost. The major drawback, as the authors explain, is that the prediction for the second block is dependent on the prediction from the first block (the tag-matching is serialized). Our scheme, however, is able to predict multiple blocks in parallel without such a dependency.

A *basic* block is defined to be instructions between branches, whether they are taken or not taken. We refer to a block simply as a group of sequential instructions up to a predefined limit, n, or up to the end of a line. Instructions after the first control transfer in a block are not used. A *line* of instructions refers to the group of instructions physically accessed in the instruction cache. The size of a line may be greater than or equal to the block width n.

We first present a method to predict multiple branches in a single block of instructions. Then we present a method to predict the addresses of two blocks in a single cycle. Next, we evaluate the performance of predicting multiple branches and blocks. Finally, we give cost estimates.

2 Multiple Branch Prediction

Yeh and Patt introduced a two-level correlated branch prediction for conditional branches [12] capable of predicting one branch per cycle. They also proposed a method for multiple branch prediction [11]. Multiple branches can be predicted in a single cycle by looking up an entry in the PHT using the global history register and also looking up the entries for the two possible outcomes (branch taken or branch not taken) for the first prediction. If three predictions are required, then four additional entries are looked up. This process grows exponentially based on the number of conditional branches predicted.

Although this method retains the accuracy of the original scalar prediction, we have found that this exponential lookup is not necessary and is wasteful. Our solution is a scalable expansion of Yeh's original two-level adaptive branch prediction. All of his schemes involve finding pattern history information to predict a single branch using a 2-bit up/down saturating counter. We expand this pattern history to contain information not for one branch instruction, but for an entire *block* of potential branch instructions. For example, if eight instructions per block are being fetched, a PHT entry will contain eight 2-bit counters, one for each position in a block. One important difference is updating the global history register (GHR) or branch history register (BHR). Instead of being updated after the prediction of each individual branch, it is updated after the prediction for the entire block. For example, if three branches are predicted not taken, not taken, taken, then the GHR/BHR is shifted to the left three bits and a "001" inserted. All of Yeh's original variations may be expanded in this manner, except his per-addr variation now becomes a per-block variation.

Figure 1 is a block diagram of a multiple branch prediction fetching mechanism. While the instruction cache is reading the current block of instructions, the instruction fetch mechanism at a *minimum* must predict the index of the next line to retrieve from the instruction cache. The complete address may be determined later. Therefore, an efficient method to predict target addresses is to use an NLS table. We modify and expand it to be indexed by the instruction *block* address and contain target lines for an entire block of instructions. Alternatively, a Branch Target Buffer (BTB) may be used [6]. The BTB, however, is also modified to be indexed and checked against the instruction block address and contain target addresses for an entire block of instructions. The NLS or BTB may be viewed as n separate tables accessed in parallel, which predict the target address for each of the n possible branch exit positions. The actual target address, if any, is selected at a later time. We call an NLS or BTB which predicts targets for a whole block a *target array*.

In addition, the *branch type* information is no longer contained in the NLS table, but in a separate block instruction type (BIT) table. We have discovered that in superscalar fetch prediction, knowing what type of instructions are in a block is the most critical piece of information. Each BIT entry contains two bits of information for each instruction in a cache line. This BIT information may be pre-decoded and contained in the instruction cache itself. Depending on implementation considerations, a separate array with a faster access time may be required. If a separate table is used, the BIT table may be smaller than the number of lines in the instruction cache at the expense of performance.

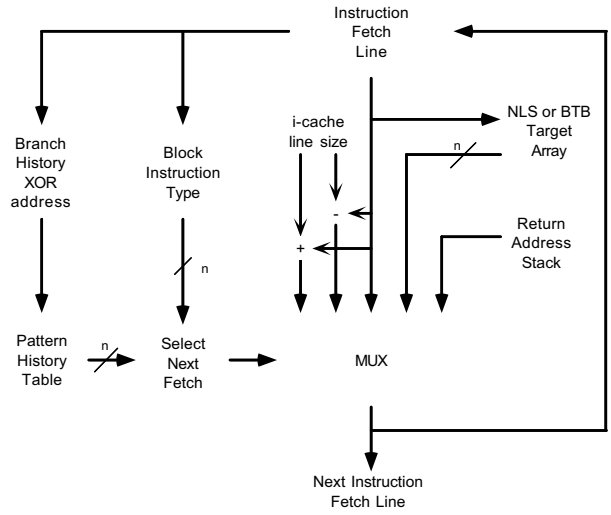

Figure 1. Block diagram of a multiple branch prediction fetching mechanism

At a minimum, the BIT information for each instruction in a fetch block must contain at least two bits to represent that an instruction is either not a branch, a return instruction, a conditional branch, or other types of branches. If we expand this to three bits per instruction, it can contain additional information about conditional branches with targets adjacent to the current line, referred to as *near-block* targets. The offset into the line may be quickly added with a $lg(n)$-bit adder as soon as the branch offset is ready. As a result, near-block target addresses do not need to be stored in the target array, and the size of the target array can be reduced.

Given the starting position in the line fetched, BIT and PHT block information, the instruction fetch control logic uses the instruction type information to find the first unconditional branch or conditional branch predicted to be taken based on its pattern history. The next line to be fetched is then selected from a multiplexer whose input contains the current line, previous line, following line, two lines after the current line, the top of the return address stack (RAS) [5],

and the n possible targets from branches in a block. The BIT types and resulting prediction sources are summarized in Table 1.

Table 1. BIT Types and Prediction Sources

			Instruction Type	Prediction Source
0	0	0	Non-branch	Fall-through PC
0	0	1	Return	Return Stack
0	1	0	Other branches	Always use Target Array
0	1	1	Conditional branch, long target	Target Array entry or Fall-through, depending on PHT
1	0	0	Cond. branch, prev line	Current line - line size
1	0	1	Cond. branch, same line	Current line
1	1	0	Cond. branch, next line	Current line + line size
1	1	1	Cond. branch, next line+1	Current line + 2 * line size

The processor should keep track of the target address of each conditional branch that is predicted not taken. In the case it was mispredicted, the correct block may be immediately fetched the following cycle after branch resolution. Otherwise, an additional cycle is required to read the target address from the target array.

Table 2 is an example showing a line of instructions and the result of prediction. The type of instruction, BIT information code, and PHT entry values are given. The starting position corresponds to the beginning of a block. The exit position is where an instruction transfers control. For each *possible starting position*, the exit position, next line select prediction, target used for a misprediction, and the new prediction used after a misprediction are shown. NLS(x) indicates that the target address for the exit position x is selected from the NLS target array. For instance, if the starting position is 4, the exit position is 5 where a conditional branch is predicted to be taken and the NLS at position 5 is used for the target address. If the branch is mispredicted, the return address stack is used as the target for the next block. Since the pattern history indicates a "second chance" bit, the prediction will not change the next time the branch is encountered.

3 Multiple Block Prediction

Once an instruction which transfers control is encountered, no more instructions in a block may be used. Another cycle is required to fetch from a different line in the instruction cache. This is a barrier to fetching a large number of instructions in a single cycle. Hence, what is needed is the capability to fetch multiple blocks at the same cycle. The problem is determining which blocks to fetch each cycle.

Fetching two blocks per cycle requires predicting two lines per cycle. In order to accomplish this prediction completely in parallel, *only* the address of the two lines currently being fetched may be used as a basis for prediction. Using the PC from the last block currently being fetched, the first line can be predicted using methods from the previous sections. The difficulty arises in predicting the following (second) line. Yeh and Patt used a branch address cache to give all possible starting basic block addresses based on the current PC [11]. Depending on the branch prediction, the appropriate addresses were selected. The drawback again is the branch address cache grows exponentially with the number of branch predictions.

The underlying problem with predicting two lines to fetch is that the prediction for the second line is *dependent* on the first. Hence, the PHT and BIT information for the second line cannot be fetched until the first line has been predicted and the new PC and GHR have been determined. The solution to this problem is essentially to predict our prediction. The end result of using the BIT and PHT for prediction is a multiplexer selector. Therefore, because the BIT and PHT information for the second block prediction are not available, we store the multiplexer selection bits of a previous prediction for that block into a *select table* (ST). The select table is indexed by the exclusive-or of the GHR and the current PC block address [7]. This index is the same as the index into the PHT for the prediction of the first block. The select value read from the select table is used to directly control the multiplexer for the second block prediction. A 3-bit selector can be used with a block width of four ($n = 4$). Four bits are required for $n = 8$.

3.1 Single Selection

Figure 2 is a block diagram of a dual block prediction fetching mechanism. It has two multiplexors to select the next two lines to fetch. The first selection is calculated from the PHT and BIT information. The second selection comes from the select table. To accurately predict target addresses, a *dual* target array is used. It provides n target addresses for the first target and n target addresses for the second target. The address of the second block currently being fetched is used as the index into both target arrays. Although the NLS must have two target arrays, a BTB may use its tag to indicate the target number (block one or two).

Undesirable duplication of target addresses is inherent to the dual target array. A branch's target address could be stored in both target arrays. Also, it may be represented in the second target array multiple times, since a branch may have multiple predecessor blocks. This duplication, however, does not significantly reduce its accuracy compared to a single target array.

The second multiplexer shown in Figure 2 is dependent on the output of the first multiplexer. An addition to determine the fall-through address of the first prediction or other near-block targets is required. Although the addition of a line index is relatively small, if timing is critical, each of the n targets from the first target array and the RAS can cal-

Table 2. Next line prediction example based on starting position

Position in block	0	1	2	3	4	5	6	7
instruction type	shift	branch	add	jump	sub	branch	move	return
BIT value	000	100	000	010	000	011	000	001
PHT value	XX	10	XX	XX	XX	11	XX	XX
exit position	1	1	3	3	5	5	7	7
select prediction	line--	line--	NLS(3)	NLS(3)	NLS(5)	NLS(5)	RAS	RAS
target on misprediction	NLS(3)	NLS(3)	N/A	N/A	RAS	RAS	N/A	N/A
select replacement	NLS(3)	NLS(3)	N/A	N/A	NLS(5)	NLS(5)	N/A	N/A

culate the fall-through (and possibly near target(s)) indexes *before* the first block selector is ready. The fall-through adder used as input for the second multiplexer can now be replaced with a multiplexer which selects the correct precomputed fall-through address from the first target.

The RAS sends the top of its stack to the input of the first multiplexer. For the second multiplexer, if the first block performs a call, the RAS input is bypassed with the address after the exit address of the first block. If the first block performs a return, the RAS sends the second address off the stack. Otherwise, the top of the stack is sent to the second multiplexer. In addition, the target array should encode whether or not its target is a result of a call, so that proper return bypassing can take place.

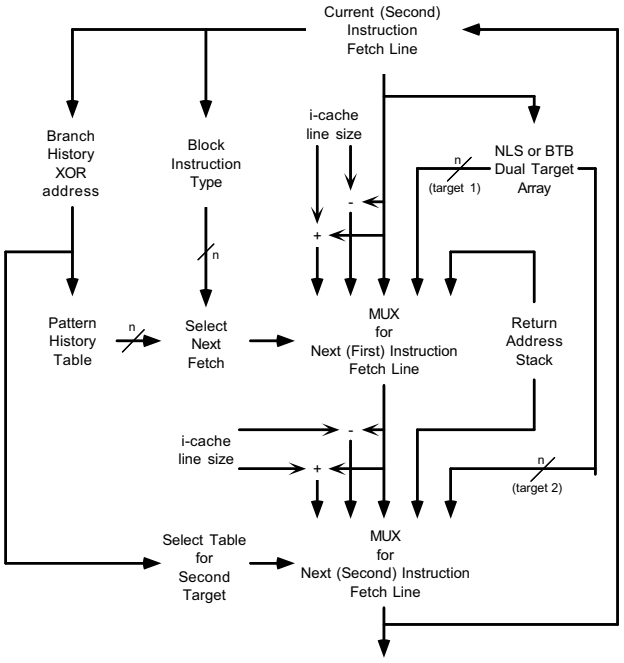

Figure 2. Block diagram for dual block prediction

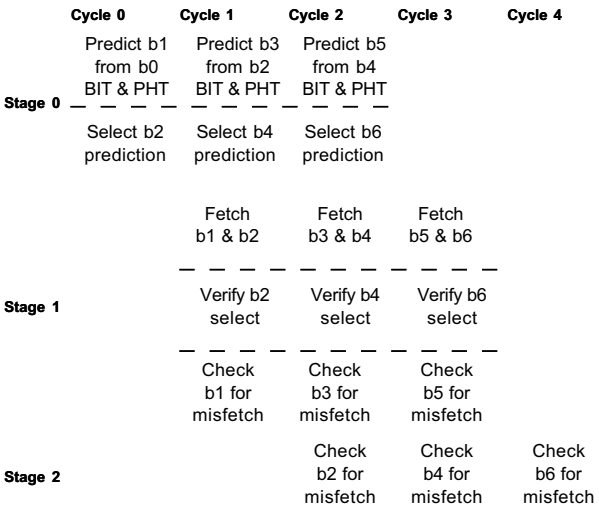

Figure 3. Pipeline stage diagram for dual block prediction

Figure 3 displays the pipeline stages involved in the dual block prediction. The first stage is the prediction of the next two blocks (bX denoted block # X). The selector for the first predicted block is computed from BIT and PHT information. The second block is predicted by reading the select table. The second stage fetches the two blocks. It also verifies the select prediction in the previous stage against the PHT and BIT information which is now available. If the prediction is different, then a *misselect* has occurred. The previous prediction is replaced with the new prediction in the select table, and the new block is fetched. Also during the second stage, the predicted target address of the first block is checked against the calculated branch offset or immediate branch from the previous block (misfetch). The third stage checks for a misfetch of the second block.

From the pipeline diagram, we observe two problems. One problem is with the updating of the GHR. The GHR can reflect the outcome of the first block prediction, but for the second block prediction, there is no information about the number of conditional branches predicted or their outcome. Therefore, a select table entry needs to contain pre-

diction information to update the GHR. This can be accomplished by using $lg(n)$ bits to represent the number of not taken branches and one bit to represent either a fall-through case or a taken branch. The second problem is with near-block select prediction of the second block. It does not give information about the offset into the line. As a result, up to $lg(n)$ extra bits are needed to provide this information, or there may be enough time to calculate the line offset after its source block has been read. Alternatively, one could choose not to use near-block targets to avoid this problem. The GHR and position prediction (if any) are verified at the same time as the select prediction.

3.2 Double Selection

The selection prediction can be used on the first block as well as the second block. We refer to selection prediction of both blocks as *double selection*. Figure 4 is a block diagram of multiple block prediction using double prediction. Double prediction increases the misselect penalty. However, the benefit is the removal of BIT storage altogether. The instruction type is decoded after the line has been fetched. The select table is still indexed by the GHR XOR starting address, but it is now a *dual* select table, providing selectors for both multiplexors. Timing concerns regarding the calculation of the selector for the first target no longer exist. The potential for timing problems from the adders between the multiplexers is significantly reduced. Selector and GHR prediction bits for both blocks are required, although the starting position prediction for the second block is no longer needed.

Figure 5 is a pipeline diagram using double selection. The first stage predicts the next two blocks from the dual select table. The second stage fetches the two blocks, and verifies the first block's select prediction and target address. The third stage verifies the second block's select prediction and target address.

3.3 Misprediction

The penalties for the different types of possible mispredictions are listed in Table 3. It is assumed that it takes four cycles to resolve a branch after it has been fetched. For the first block, if there are remaining instructions required to be re-fetched after a conditional branch was mispredicted taken, then it will take an additional cycle. A misprediction on the second block always requires another cycle. There is a one cycle misselect or GHR mispredict penalty using a single select on the second block.

With a double selection prediction, the first block has a one cycle penalty while the second block takes two cycles. Since a misselect is detected during or immediately after the instructions have been fetched, instructions that would

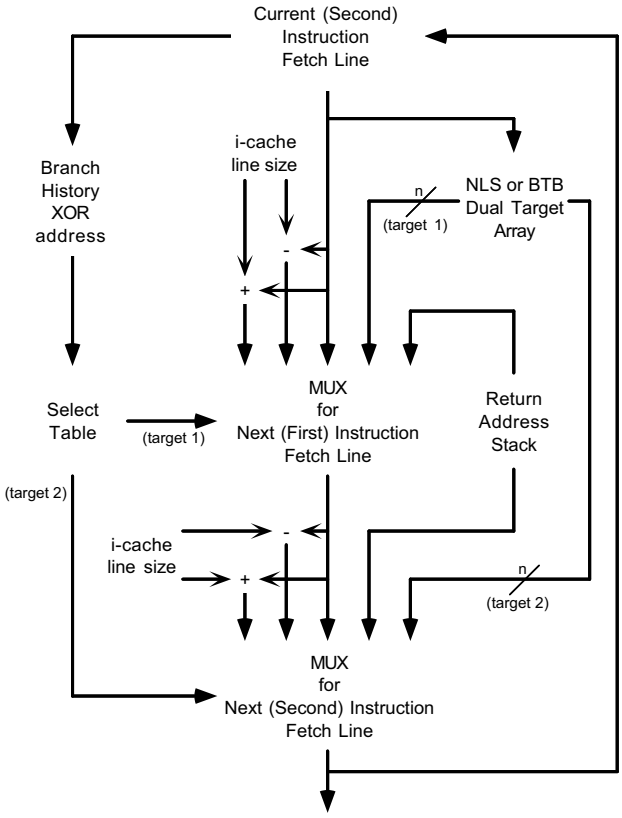

Figure 4. Block diagram for dual block prediction using double selection

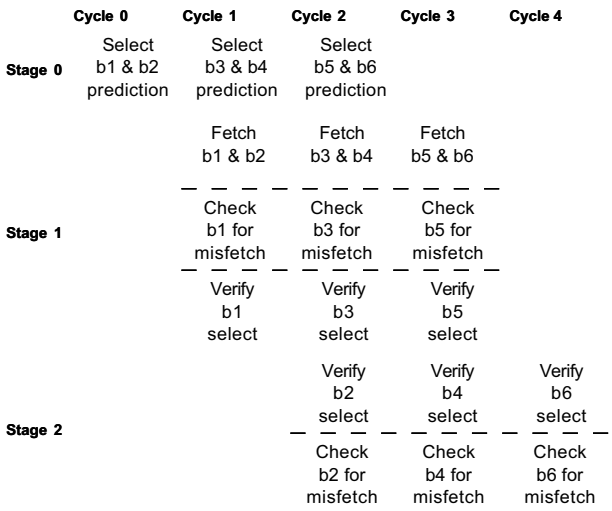

Figure 5. Pipeline stage diagram for dual block prediction using double selection

have been discarded on a taken branch become valid, and no re-fetch cycle is needed. A misfetch takes one cycle for the first block and two cycles for the second block to detect.

Since multiple blocks are being fetched using different cache lines, a multiple banked instruction cache is required. Since two lines are fetched simultaneously, they may map into the same cache bank. Should a conflict arise, the second line is read the next cycle.

Table 3. Misprediction Penalties

Misprediction	Single Select		Double Select	
	1^{st} blk	2^{nd} blk	1^{st} blk	2^{nd} blk
Conditional branch	4*	5	4*	5
Return	4	5	4	5
Misfetch indirect	4	5	4	5
Misfetch immediate	1	2	1	2
Misselect	N/A	1	1	2
GHR	N/A	1	1	2
BIT	1	1	N/A	N/A
I-cache bank conflict	0	1	0	1
* Add one cycle if instructions remain and need to be re-fetched.				

In order to facilitate recovery from a mispredicted branch, each conditional branch is assigned a bad branch recovery (BBR) entry, which provides information on how to update branch prediction tables and provide a new target. The processor must create this entry and keep track of it as the branch moves down the pipeline. Table 4 lists a description and sizes of the fields in a recovery entry. A recovery entry is created when the block which contains a conditional branch is predicted using BIT and PHT information. When a prediction is made for a conditional branch, another prediction is made for that block assuming its original prediction is incorrect. If a branch is predicted not taken, then the alternate target address is the branch's target address. If it is predicted taken, then the alternate address is the next control transfer or fall-through address in its block (see the example in Table 2). The alternate target address is entered into the recovery entry. In addition, a replacement selector and new GHR are generated.

Table 4. Bad Branch Recovery Entry

Bits	Description
1	Block 1 or 2
1	Predicted taken or not taken.
1	Second chance.
8-12	PHT index.
2n	PHT block (optional).
8-12	Corrected GHR.
8-11	Replacement selector.
10/30	Corrected i-cache index or full addr.

The recovery entry may also contain the entire PHT block that reflects a successful pattern history update for each branch in the block up to the current branch. After the last branch in a block has been successfully resolved, it uses this field to update the PHT to reflect a *correct* prediction. When a branch is mispredicted, it is modified to reflect an *incorrect* prediction (by using the original second chance information) and written to the PHT. If the PHT is not updated by using a PHT block field, then the pattern history for a branch has to be updated using a read/modify/write cycle to the PHT block for each individual branch when it is resolved.

If the branch does not have a "second chance" when it is mispredicted, then the pre-computed selector from the bad branch recovery entry is written into the select table.

If a misprediction occurs for the second block, then any remaining instructions from the first block are fetched along with a new second block target retrieved from the recovery entry. On the other hand, if the misprediction occurs for the first block, an extra cycle may be required to fetch any remaining instructions from the previous block.

4 Performance

The performance of different variations of multiple branch and block prediction was determined by running the SPEC95 benchmark suite on the SPARC architecture. The suite was compiled using the SunPro compiler with standard optimizations (-O). Programs were simulated using the Shade instruction-set simulator [2]. Each program ran for the first one billion instructions.

All the results presented use a block width of eight ($n = 8$). Single selection is used for dual block prediction unless otherwise noted. The results presented only use a global adaptive branch prediction scheme using one global blocked pattern history table. The default size of a select table is 1024 entries, which corresponds to a GHR length of 10 bits. The size of the RAS is 32 entries. It was assumed the processor would always have enough bad branch recovery entries available. Instruction cache misses were not simulated, i.e., a perfect instruction cache was assumed. The only consideration for the instruction cache were the line size and bank conflicts. A line size equal to the block width was used, and the instruction cache was split into eight banks. Also, by default, near-block prediction is not used.

The default target array is a 256-entry NLS array. The set prediction was not simulated. Therefore, the results presented for the NLS configuration are really a direct-mapped tag-less BTB. The performance of a real NLS is affected by the instruction cache configuration. For a performance and cost comparison of an NLS verses a BTB, please refer to [1].

We compare the performance of different types of multiple branch and block architectures using two metrics, as used by Yeh and Patt [13]. The first is the branch execution

penalty,

$$BEP = \frac{Total\ Penalty\ Cycles}{\#Branches}$$

If we assume the other parts of the processor are ideal, we can compute the effective instruction fetch rate,

$$IPC\_f = \#Valid\ instructions / \#Fetch\ cycles$$

where the number of fetch cycles is equal to

$$\#Total\ Penalty\ Cycles + \frac{\#Blocks\ fetched}{Maximum\ blocks\ per\ cycle}$$

The BEP gives information regarding performance and the interaction between the many different types of penalties as listed in Table 3. Nevertheless, all the types of penalties are recorded. Overall performance is best understood from the effective instruction fetch rate. One cannot directly compare a scalar BEP with a superscalar BEP or a multi-block BEP since higher penalties are overcome by increased number of instructions per successful fetch block.

Also, when fetching two blocks per cycle of potentially eight instructions each, up to sixteen instructions may be returned in one cycle. Consequently, the effective instruction fetching rate, IPC_f can be greater than n. If an eight issue processor is used, then extra instructions returned can be buffered [10]. When the raw two block rate is greater than n, the issue unit will usually receive, and average close to, n instructions per request. Of course, a simpler configuration to satisfy issue unit constraints in such a situation would be to use two blocks of four instructions each. This would still yield an excellent fetching rate.

4.1 Conditional Branch Accuracy

To begin with, the conditional branch accuracy of a blocked PHT for multiple branch prediction was evaluated. The branch history length varied from 6 to 12, and the results were compared to a scalar PHT. The scalar scheme used a per-addr PHT with 8 PHTs to give it equal size of a blocked PHT for $n = 8$. Figure 6 displays the branch misprediction rates (*line*) and the improvement over a scalar PHT (*bar*). The difference in accuracy between the scalar and blocked schemes across all variations were small, and the accuracy favored the blocked PHT scheme for most programs. The accuracy of SPECint95 averaged 91.5% while the accuracy of the SPECfp95 averaged 97.3%, using a GHR length of 10. In this case, the blocked PHT had a better accuracy by a few hundredths of a percent for SPECfp95 and a few tenths of a percent for SPECint95.

4.2 BIT

Correct instruction type information for a block is critical to making accurate predictions. Incorrect BIT information can still result in a correct prediction, but this possibility is reduced with larger block sizes. Different BIT table sizes were simulated to evaluate its impact. Using single block fetching, Figure 7 shows the BEP contribution from inaccurate BIT information (*bar*). Also shown is the IPC_f (*line*). Small sized BIT tables result in poor performance. Only until about 2048 entries does the percentage of BEP drop below 5%. Therefore, for smaller sized instruction caches, it may be more beneficial to store the BIT information inside the instruction cache. Conversely, a separate BIT table would be more cost effective because the one cycle miss penalty of the BIT is much lower than an instruction cache miss. The rest of the results presented use two blocks and assume BIT information is stored in the instruction cache.

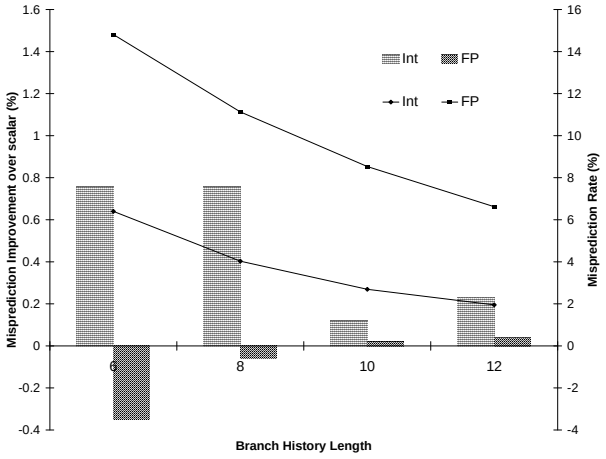

Figure 6. Branch Misprediction Rate and Improvement

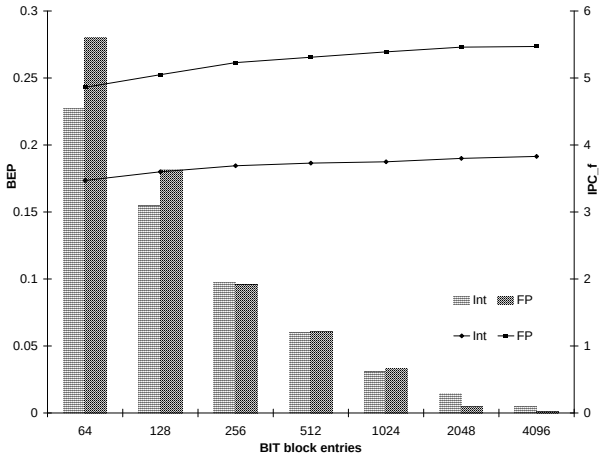

Figure 7. BIT Penalty and Performance

4.3 Single vs. Double Selection

The performance of the select table depends on the branch history length and the number of select tables used. Multiple select tables are indexed by the starting position from the current address. The correct target depends on the entering position in a block, so multiple select tables help identify which target should be selected. The least significant bits of the starting address determine which select table is used. Figure 8 shows the performance of dual block prediction for single and double selection. The global history register length varies from 9 to 12. There can be 1, 2, 4, or 8 STs. However, there are not multiple PHTs. The results demonstrate that increasing the number of STs improves performance as well as increasing the branch history length. The extra penalties from using double selection significantly reduced performance, roughly 10% for most cases. Hence, single selection should be used if implementation considerations permit. Double selection significantly improves, though, with more STs.

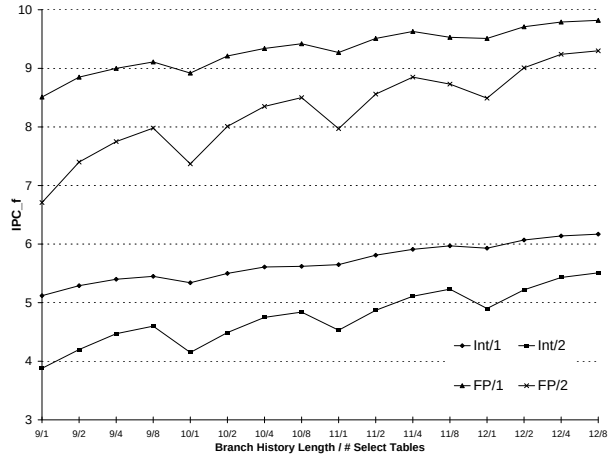

Figure 8. Integer and Floating Point performance using Single and Double Selection

4.4 Target Arrays

Target arrays can use a BTB or NLS. In addition, if a near-block target is used, this will reduce the number of immediate targets used in the target array. Table 5 shows the percentage of BEP due to indirect and immediate misfetches for SPECint95. The total BEP and IPC_f are also reported. The number of *block* entries is varied for both NLS and a 4-way BTB using LRU replacement algorithm. A BTB entry can be for the first or second target, while an NLS entry has two separate targets. The data indicates that eight NLS block entries are needed for comparable performance of one 4-way BTB entry. About 70% of the conditional branches are near-block targets. As a result of using near-block encoding, the number of BTB or NLS entries can be reduced in half for about the same performance.

Table 5. Indirect and Immediate Misfetch Penalty Comparison for Different Target Array Configurations

Target Type	# blk entries	near-block?	%BEP misfetch imm.	%BEP misfetch ind.	BEP	IPC_f
BTB	8	no	19.2	18.7	0.603	5.02
BTB	8	yes	10.6	16.3	0.520	5.40
BTB	16	no	12.6	15.1	0.523	5.32
BTB	16	yes	6.5	12.6	0.476	5.57
BTB	32	no	7.4	11.6	0.473	5.58
BTB	32	yes	3.6	9.6	0.446	5.73
BTB	64	no	4.0	9.6	0.447	5.72
BTB	64	yes	1.9	7.9	0.431	5.80
NLS	64	no	12.0	14.7	0.516	5.41
NLS	64	yes	6.7	13.1	0.480	5.54
NLS	128	no	8.3	12.3	0.481	5.53
NLS	128	yes	4.2	10.8	0.454	5.67
NLS	256	no	5.5	10.1	0.457	5.66
NLS	256	yes	2.7	8.7	0.438	5.77
NLS	512	no	3.8	9.2	0.444	5.74
NLS	512	yes	1.6	7.9	0.429	5.81

4.5 Instruction Cache Configurations

The performance can be dramatically improved if a different type of instruction cache configuration is used. Using the same line size and block width of eight instructions, the number of valid instructions in a block has been limited due to misalignment. The target address of a control transfer can be in the middle of a line, thus reducing the size of a block. To increase the number of instructions per block (IPB), the cache line size can be extended to 16 instructions [10]. Only up to 8 instructions are returned as a block, but the probability less than 8 instructions are found has been reduced. To solve this problem completely, a self-aligned cache can combine two consecutive lines to form a block [3, 10]. If a self-aligned cache is used though, the number of banks should be doubled to offset the increase in bank conflicts, since up to four lines are being simultaneously accessed to return two blocks. Although there are no bank conflicts with single block fetching, the extended and self-aligned caches improve the instructions fetched per block (IFB) and overall fetching performance.

With the extended and self-aligned caches, when branch prediction is performed using the PHTs, the values wrap around the PHT block. Also the target arrays must be correspondingly extended or self-aligned. The performance of these three cache types are compared using one and two block fetching with single selection. The results are shown

in Table 6, using 8 STs and a branch history length of 10. Outstandingly, the self-aligned cache achieves 10.9 IPC_f for the floating point benchmarks. It averages over 8 IPC_f for the entire SPEC95 suite. The high performance is primarily due to the increase in IFB. Also, the starting address becomes more random which helps associate a select table and use it efficiently. The performance of the extended cache type is between a normal and self-aligned cache. Compared to single block prediction, dual block prediction results in an effective fetching rate approximately 40% higher for integer programs and 70% higher for floating point programs.

Table 6. Instructions per block (IPB) and IPC_f for different cache types

			SPECint95			SPECfp95		
cache	line		IPB	IPC_f		IPB	IPC_f	
type	size	bnks		1 blk	2 blk		1 blk	2 blk
normal	8	8	5.01	3.96	5.66	5.81	5.48	9.43
extend	16	8	5.30	4.12	5.87	6.03	5.65	9.80
align	8	16	5.99	4.53	6.42	6.76	6.33	10.88

Using a self-aligned cache, 8 STs, and a branch history length of 10, Figure 9 shows the BEP of each program and the contribution of BEP by each type of misprediction as described in Section 3.3. The effective instruction fetching rate is inversely proportional to BEP. The most significant BEP contribution is from misprediction of conditional branches. Misselection is the next most significant contribution. Target array mispredictions are also a significant factor in BEP. Some of the floating point programs performed exceedingly well. On the other hand, some integer programs had a high BEP because of poor conditional branch prediction.

5 Cost Estimates

Table 7. Simplified hardware cost estimates

Symbol	Description
n	block width
k	history register length
p	number of PHTs
s	number of Select Tables
t	number of NLS block entries
l	size of line index
a	cache associativity
b	number of BBR entries
i	number of BIT block entries

Table	Simple hardware cost estimate
PHT	$p \times 2^k \times n \times 2$
ST	$s \times 2^k \times 2 \times (lg(n) + 1)$
NLS	$t \times n \times (l + lg(a))$
BIT	$i \times n \times 2$
BBR	$b \times (2k + l + lg(a) + 2lg(n) + 5)$

Table 7 lists a simplified hardware cost estimates for the PHT, ST, NLS, and BIT tables. If we use a block width of 8, a 32 KByte direct-mapped instruction cache, a 10-bit history register, 1 PHT, 1 ST, 256 NLS entries, 1024 BIT entries, and 8 BBR entries, the cost estimates evaluate to:

- **PHT:** 16 Kbits
- **ST:** 8 Kbits
- **NLS:** 20 Kbits
- **BIT:** 16 Kbits
- **BBR:** .3 Kbits
- **single block total:** 52 Kbits
- **dual block, single select total:** 80 Kbits
- **dual block, double select total:** 72 Kbits

As the number of instructions that can be predicted in a block increase, the cost increases proportionally. In addition, it is possible to predict more than two blocks per cycle. In that case, the cost grows proportionally to the number of blocks predicted. Another block prediction basically requires another select table and target array, and another read/write port to the PHT and BIT tables.

6 Conclusion

A scalable mechanism to predict multiple branches in a single block was presented. Its conditional branch accuracy is essentially the same as a scalar two-level adaptive branch prediction of equal size. By recording previous predictions in a select table, two blocks can be fetched in a single cycle. Dual block prediction uses either a single or double select table. Double selection may be used at the expense of a slower fetching rate, but may be desirable in processors with deep pipelines and extremely fast cycle times. An extremely high rate can be achieved with a multiple branch and block prediction mechanism.

References

[1] B. Calder and D. Grunwald. Next cache line and set prediction. In *22nd Annual International Symposium on Computer Architecture*, pages 287–296, June 1995.

[2] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS*, 1994.

[3] 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.

[4] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, 1991.

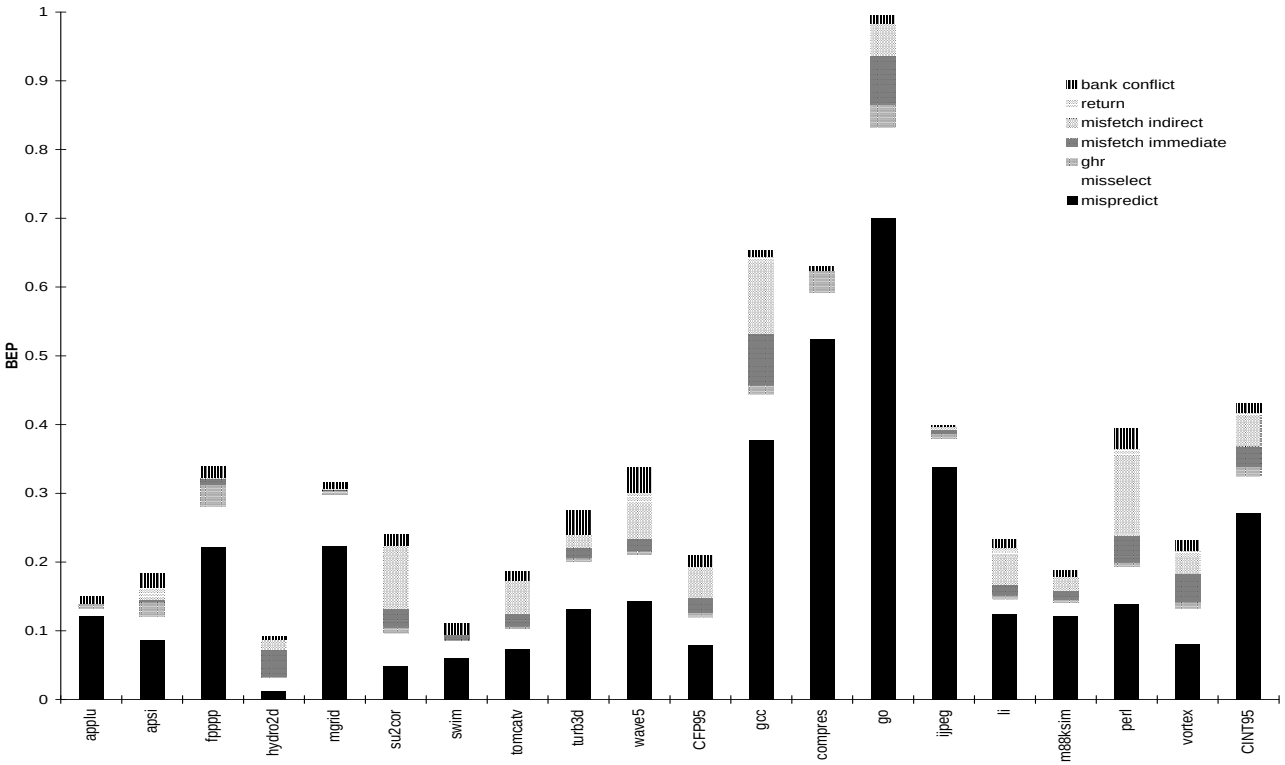

Figure 9. Branch Execution Penalties for two block, single selection

[5] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th Annual International Symposium on Computer Architecture*, pages 34–42, May 1991.

[6] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, 1984.

[7] S. McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

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

[9] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Proceedings of the Third International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 290–302, April 1989.

[10] S. Wallace and N. Bagherzadeh. Instruction fetching mechanisms for superscalar microprocessors. In *Euro-Par '96*, August 1996.

[11] T.-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *7th ACM International Conference on Supercomputing*, pages 67–76, Tokyo, Japan, July 1993.

[12] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *19th Annual International Symposium on Computer Architecture*, pages 124–134, Gold Cost, Australia, May 1992.

[13] T.-Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *25th Annual ACM/IEEE International Symposium on Computer Microarchitecture*, pages 129–139, Dec. 1992.

[14] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, pages 257–266, San Diego, California, May 1993.