Target Prediction for Indirect Jumps

Po-Yung Chang Eric Hao Yale N. Patt

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
email: {pychang,ehao,patt}@eecs.umich.edu

Abstract

As the issue rate and pipeline depth of high performance superscalar processors increase, the amount of speculative work issued also increases. Because speculative work must be thrown away in the event of a branch misprediction, wide-issue, deeply pipelined processors must employ accurate branch predictors to effectively exploit their performance potential. Many existing branch prediction schemes are capable of accurately predicting the direction of conditional branches. However, these schemes are ineffective in predicting the targets of indirect jumps achieving, on average, a prediction accuracy rate of 51.8% for the SPECint95 benchmarks. In this paper, we propose a new prediction mechanism, the target cache, for predicting indirect jump targets. For the perl and gcc benchmarks, this mechanism reduces the indirect jump misprediction rate by 93.4% and 63.3% and the overall execution time by 14% and 5%.

1 Introduction

As the issue rate and pipeline depth of high performance superscalar processors increase, the amount of speculative work issued also increases. Because speculative work must be thrown away in the event of a branch misprediction, wide-issue, deeply pipelined processors must employ accurate branch predictors to effectively exploit their performance potential.

A program's branches can be categorized as conditional or unconditional and direct or indirect, resulting in four classes. A conditional branch conditionally redirects the instruction stream to its target whereas an unconditional branch always redirects the instruction stream to its target. A direct branch has a statically specified target which points to a single location in the program whereas an indirect branch has a dynamically specified target which may point to any number of locations in the program. Although these branch attributes can combine into four classes, only three of the classes – conditional direct, unconditional direct, and unconditional indirect – occur with significant frequency.

In the past, branch prediction research has focused on accurately predicting conditional and unconditional direct branches [11, 7, 15, 6, 2, 8]. To predict such branches, the prediction mechanism predicts the branch direction (for unconditional branches, this part is trivial) and then generates the target associated with that direction. To generate target addresses, a branch target buffer (BTB) is used. The BTB stores the fall-through and taken address for each branch. For indirect jumps, the taken address is the last computed target for the indirect jump. (For a jump-to-subroutine, the fall-through address is needed so that it can be pushed onto the return address stack, even though the jump-to-subroutine will never jump to the fall-through address.) Unfortunately, BTB-based prediction schemes perform poorly for indirect branches. Because the target of an indirect branch can change with every dynamic instance of that branch, predicting the target of an indirect branch as the last computed target for that branch will lead to poor prediction accuracy. Figures 1 through 8 show the number of different dynamic targets seen for each indirect branch in the SPECint95 benchmarks. Table 1 lists the indirect branch target misprediction counts achieved by a 1K-entry 4-way set-associative BTB for the SPECint95 benchmarks. For gcc and perl, the two benchmarks with significant numbers of indirect branches, the misprediction rates for indirect branches were 66.0% and 76.4%.

In this paper, we propose a new prediction mechanism, the target cache, for predicting the targets of indirect jumps[1]. The target cache uses a concept central to the 2-level branch predictor: branch history is used to distinguish different dynamic occurrences of each indirect branch. When fetching an indirect jump, the fetch address and the branch history are used to form an index (A) into the target cache. The target cache is then accessed for the indirect jump. Later, when the indirect branch retires, the target cache is accessed again using index A , and the computed target for the indirect jump is written into the target cache. As the program executes, the target cache records the target for each indirect jump target encountered. When fetching an indirect jump,

[1]Although return instructions technically are indirect jumps, they are not handled with the target cache because they are effectively handled with the return address stack [13, 4].

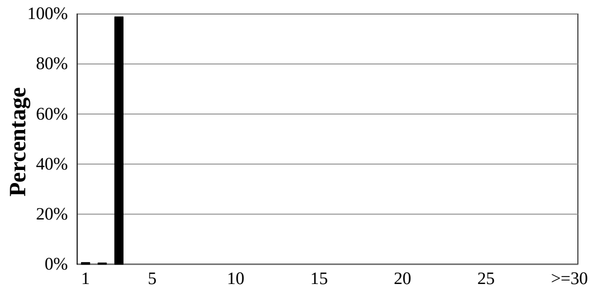

Figure 1: Number of Targets per Indirect Jump (compress)

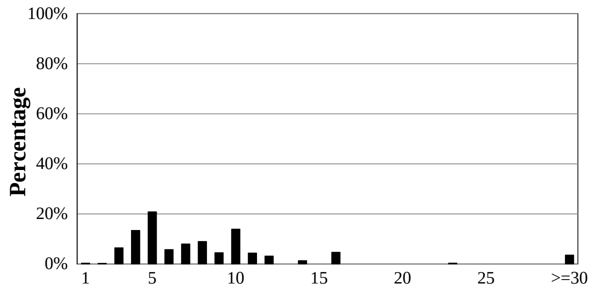

Figure 2: Number of Targets per Indirect Jump (gcc)

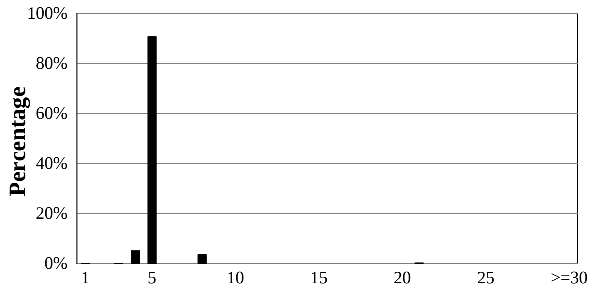

Figure 3: Number of Targets per Indirect Jump (go)

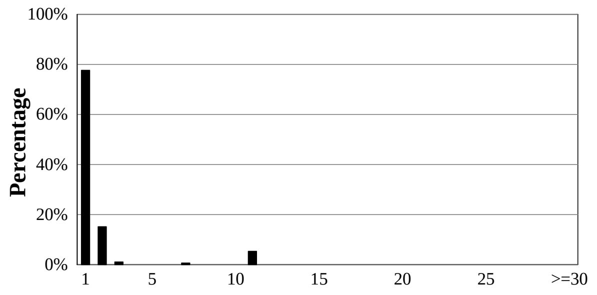

Figure 4: Number of Targets per Indirect Jump (ijpeg)

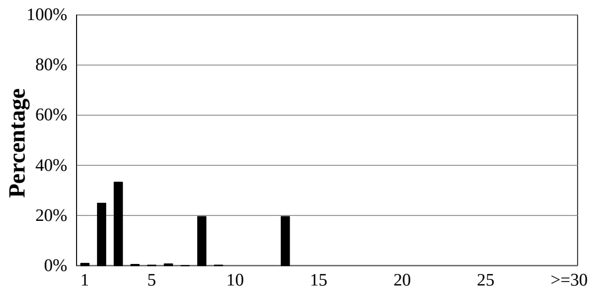

Figure 5: Number of Targets per Indirect Jump (m88ksim)

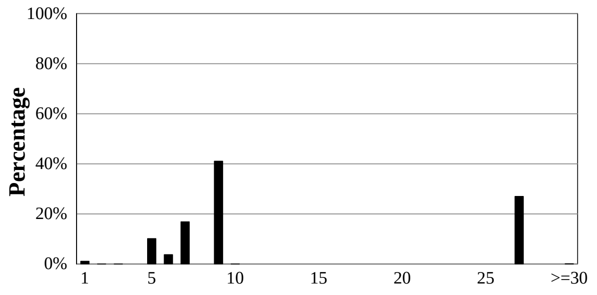

Figure 6: Number of Targets per Indirect Jump (perl)

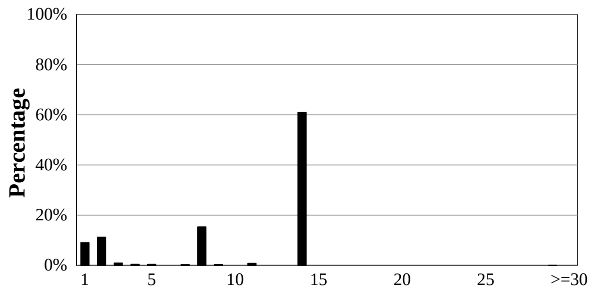

Figure 7: Number of Targets per Indirect Jump (vortex)

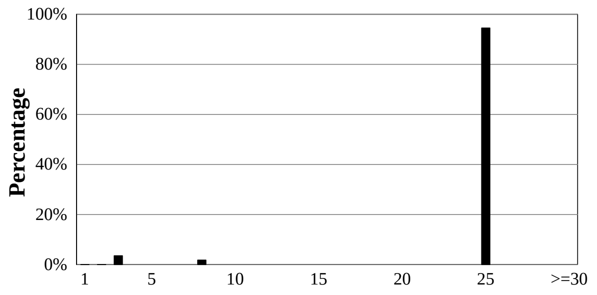

Figure 8: Number of Targets per Indirect Jump (xlisp)

Benchmark	Input	#Instructions	#Branches	#Indirect Jumps	Ind. Jump Mispred. Rate
compress	test.in[2]	125,162,687	17,460,753	590	61.7%
gcc	jump.i	172,328,834	35,979,748	939,417	66.0%
go	2stone9.in[3]	125,637,006	23,378,150	173,719	37.6%
ijpeg	specmun.ppm[4]	206,802,135	23,449,572	103,876	14.3%
m88ksim	dcrand.train.big	131,732,141	23,840,021	186,285	37.3%
perl	scrabbl.pl[5]	106,140,733	16,727,047	588,136	76.4%
vortex	vortex.in[6]	236,081,621	44,635,060	243,706	11.3%
xlisp	train.lsp	192,569,022	40,909,525	114,789	80.7%

Table 1: Misprediction counts for indirect jumps in the SPECint95 benchmarks

the target cache is accessed with the fetch address and the branch history to produce the predicted target address. The target cache improves on the prediction accuracy achieved by BTB-based schemes for indirect jumps by choosing its prediction from (usually) all the targets of the indirect jump that have already been encountered rather than just the target that was most recently encountered. We will show that for the perl and gcc benchmarks, a target cache can reduce the indirect jump misprediction rate by 93.4% and 63.3% and the overall execution time by 14% and 5%.

This paper is organized into five sections. Section 2 discusses related work. Section 3 introduces the concept of a target cache. Section 4 shows the performance benefits of target caches. Section 5 provides some concluding remarks.

2 Related Work

To address the problem of target prediction for indirect jumps in C++ programs, Calder and Grunwald proposed a new strategy, the 2-bit strategy, for updating BTB target addresses [1]. The default strategy is to update the BTB on every indirect jump misprediction, Calder and Grunwald's 2-bit strategy does not update a BTB entry's target address until two consecutive predictions with that target address are incorrect. This strategy was shown to achieve a higher target prediction accuracy than that achieved by the default strategy.

The 2-bit strategy is not very successful in predicting the targets of indirect branches in C programs such as the SPECint95 benchmarks. Table 2 compares the indirect branch target misprediction rate achieved by a 1K entry, 4-way set-associative BTB that uses the 2-bit strategy to that achieved by an identically configured BTB that uses the default target update strategy. The 2-bit strategy reduced the misprediction rates for the compress, gcc, ijpeg, and perl benchmarks, but increased the misprediction rates for the m88ksim, vortex, and xlisp benchmarks. The target prediction scheme proposed by this paper, the target cache, has significantly lower indirect jump misprediction rates than those achieved by the 2-bit strategy. For example, a 512-entry target cache achieve the misprediction rates of 30.4% and 30.9% for gcc and perl respectively.

Benchmark	Misprediction Rate	
	BTB	2-bit BTB
compress	61.7%	44.4%
gcc	66.0%	64.1%
go	37.6%	36.4%
ijpeg	14.3%	9.36%
m88ksim	37.3%	38.1%
perl	76.4%	67.5%
vortex	11.3%	14.6%
xlisp	80.7%	85.4%

Table 2: Performance of 2-bit BTB

Kaeli et al. proposed a hardware mechanism, the case block table (CBT), to speed up the execution of SWITCH/CASE statements [5]. Figure 9 gives an example of a SWITCH/CASE statement and the corresponding assembly code for that statement. The assembly code consists of a series of conditional branches that determine which case of the SWITCH/CASE statement is to be executed. Because a single variable, the case block variable, specifies the case to be executed, this series of conditional branches can be avoided if the instruction stream could be directly redirected to the appropriate case. The CBT enables this redirection by recording, for each value of the case block variable, the corresponding case address. This mapping of case block variable values to case addresses is dynamically created. When a SWITCH/CASE statement is encountered, the value of the case block variable is used to search the CBT for the next fetch address. In effect, the CBT is dynamically generating a jump table to replace the SWITCH/CASE statements.

The study done by Kaeli et al. showed that an oracle CBT, that is, a CBT which always selects the correct case to execute, can reduce significantly the number of conditional branches in the instruction stream. However, the CBT's usefulness is limited by two factors. First, modern day compilers are capable of directly generating jump tables for SWITCH/CASE statements at compile-time, eliminating the need for the CBT to generate the tables dynamically.

[2]Abbreviated version of the SPECint reference input set bigtest.in. The initial list consists of 30000 elements.

[3]Abbreviated version of the SPECint training input set 2stone9.in using 19 levels instead of 50 levels.

[4]Abbreviated version of the SPECint test input set specmun.ppm where the compression quality is 50 instead of 90.

[5]Abbreviated version of the SPECint reference input set.

[6]Abbreviated version of the SPECint test input set. PART_COUNT = 100 and LOOKUPS = DELETES = STUFF_PARTS = 10

```
High-Level
Construct             Assembly Code
----------            -------------

switch(v) {              r1 <- compare v, 1
 case 1:                 beq r1, L1
                          ; beq == branch if equal
   char = 'a';           r1 <- compare v, 2
   break;                beq r1, L2
 case 2:                 r1 <- compare v, 3
   char = 'b';           beq r1, L3
   break;                goto L4
 case 3:             L1: char <- 'a'
   char = 'c';           goto L5
   break;            L2: char <- 'b'
 default:                goto L5
   char = 'd';       L3: char <- 'c'
}                        goto L5
                     L4: char <- 'd'
                     L5:
```

Figure 9: An Example of a SWITCH/CASE Construct

Second, for processors with out-of-order execution, the value of the case block variable is often not yet known when the instruction corresponding to the SWITCH/CASE statement is fetched. As a result, the CBT cannot redirect the instruction stream to the appropriate case until that value is computed.

The target cache proposed in this paper has some similarities to the CBT. Like the CBT, the target cache dynamically records the case addresses for a program's SWITCH/CASE statements. However, the target cache differs from the CBT in two significant ways. First, the purpose of the target cache is to accurately predict the target of the indirect jumps used to implement the jump tables, not to dynamically construct jump tables. The target cache assumes that the compiler has generated jump tables for the SWITCH/CASE statements. Second, the target cache was designed for highly speculative machines, where the value of the case block variable is likely to be unknown when the prediction is made; therefore, a branch history is used to make the prediction instead.

3 Target Cache

The target cache improves on the prediction accuracy achieved by BTB-based schemes for indirect jumps by choosing its prediction from (usually) all the targets of the indirect jump that have already been encountered rather than just the target that was most recently encountered. When fetching an indirect jump, the target cache is accessed with the fetch address and other pieces of the machine state to produce the predicted target address. As the program executes, the target cache records the target for each indirect jump target encountered.

3.1 Accessing Target Cache

In our study, we consider using branch history along with the branch address to index into the target cache. Two types of branch history information are used to decide which target of the indirect jump will be predicted – pattern history and path history.

- **Branch History**

  It is now well known that the 2-level branch predictor improves prediction accuracy over previous single-level branch predictors [14]. The 2-level predictors attain high prediction accuracies by using pattern history to distinguish different dynamic occurrences of a conditional branch. To predict indirect jumps, the target cache is indexed using branch address and global pattern history. Global pattern history is a recording of the last n conditional branches. No extra hardware is required to maintain the branch history for the target cache if the branch prediction mechanism already contains this information. The target cache can use the branch predictor's branch history register.

- **Path History**

  Previous research [16, 8] has shown that path history can also provide useful correlation information to improve branch prediction accuracy. Path history consists of the target addresses of branches that lead to the current branch. This information is also useful in predicting indirect branch targets.

  In this study, two different types of path history can be associated with each indirect jump – global or per-address. In the per-address scheme, one path history register is associated with each distinct static indirect branch. Each n-bit path history register records the last k target addresses for the associated indirect jump. That is, when an indirect branch is resolved, n/k bits from its target address are shifted into the path history register.

  In the global scheme, one path history register is used for all indirect branches. Because the history register has a fixed length, it can record a limited number of branches in the past history. Thus, the history register may be better utilized by only recording a particular type of branch instruction. For example, if a sequence of conditional branches is sufficient to distinguish the different paths of each indirect jump, then there is no need to include other types of branches in the path history. Four variations of the global scheme were considered – `Control`, `Branch`, `Call/ret`, `Ind jmp`. The `Control` scheme records the target address of all instructions that can redirect the instruction stream. The `Branch` scheme only records the targets of conditional branches. The `Call/ret` scheme records only the targets of procedure calls and returns. The `Ind jmp` scheme records only the targets of indirect jumps.

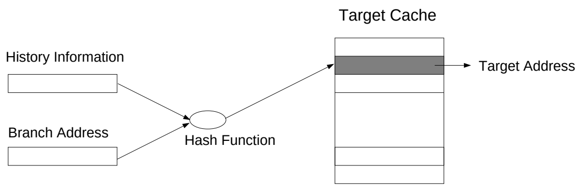

Figure 10: Structure of a Tagless Target Cache

3.2 Target Cache Structure

In addition to varying the type of information used to access the target cache, we also studied tagged and tagless target caches.

- **Tagless Target Cache**

  Figure 10 shows the structure of a tagless cache. The target cache is similar to the pattern history table of the 2-level branch predictor; the only difference is that a target cache's storage structure records branch targets while a 2-level branch predictor's pattern history table records branch directions.

  The target cache works as follows: during instruction fetch, the BTB and the target cache are examined concurrently. If the BTB detects an indirect branch, then the selected target cache entry is used for target prediction. When the indirect branch is resolved, the target cache entry is updated with its target address.

  Several variations of the tagless target cache can be implemented. They differ in the ways that branch address and history information are hashed into the target cache. For example, the branch address can be XORed with the history information for selecting the appropriate entry. Section 4 will describe the different hashing schemes considered.

- **Tagged Target Cache**

  Like the previous 2-level branch predictors, interference occurs in the target cache when a branch uses an entry that was last accessed by another branch. Interference is particularly detrimental to the target cache because each entry in the target cache stores the branch target address. Since the targets of two different indirect branches are usually different, interference will most likely cause a misprediction.

  To avoid predicting targets of indirect jumps based on the outcomes of other branches, we propose the tagged target cache where a tag is added to each target cache entry (see Figure 11). The branch address and/or the branch history are used for tag matching. When an indirect branch is fetched, its instruction address and the associated branch history are used to select the appropriate target cache entry. If an entry is found, the instruction stream is redirected to the associated target address.

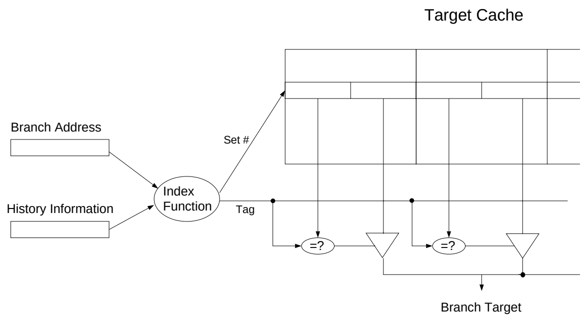

Figure 11: Structure of a Tagged Target Cache

4 Results

4.1 Experimental Methodology

The experimental results presented in this paper were measured by trace-driven simulations using an instruction level simulator. The benchmarks simulated were the SPECint95 benchmarks and they were all simulated to completion. Table 1 (see Section 1) lists the input data set used for each benchmark and the dynamic instruction counts. We will concentrate on the gcc and perl benchmarks, the two benchmarks with the largest number of indirect jumps.

The machine model simulated is the HPS microarchitecture [9] [10]. HPS is a wide-issue out-of-order execution machine, using Tomasulo algorithm for dynamic scheduling [12]. Checkpointing [3] is used to maintain precise exceptions. Checkpoints are established for each branch; thus, once a branch misprediction is determined, instructions from the correct path are fetched in the next cycle.

The HPS processor simulated in this paper supports 8 wide issue with a perfect instruction cache and a 16KB data cache. Latency for fetching data from memory is 10 cycles. Table 3 shows the instruction classes and their simulated execution latencies, along with a description of the instructions that belong to that class. In the processor simulated, each functional unit can execute instructions from any of the instruction classes. The maximum number of instructions that can exist in the machine at one time is 128. An instruction is considered in the machine from the time it is issued until it is retired.

Instruction Class	Exec. Lat.	Description
Integer	1	INT add, sub and logic OPs
FP Add	3	FP add, sub, and convert
FP/INT Mul	3	FP mul and INT mul
FP/INT Div	8	FP div and INT div
Load	2	Memory loads
Store	-	Memory stores
Bit Field	1	Shift, and bit testing
Branch	1	Control instructions

Table 3: Instruction classes and latencies

4.2 Tagless Target Cache

This section examines the performance of tagless target caches. The size of every target cache considered in this section is 512 entries. Since the BTB has 256 sets and is 4-way set-associative, the target cache increases the predictor hardware budget by 10 percent. The cost of the predictor is estimated using the following equations:

$$\begin{aligned}\text{BTB}^{\dagger} &= 77 \times 2048 \texttt{ bits}\\ \text{target cache}(n) &= 32 \times n \texttt{ bits}\\ \text{predictor budget} &= \text{BTB} + \text{target cache}(n) \texttt{ bits}\end{aligned}$$

where n is the number of target cache entries.

4.2.1 Hashing Function

With the tagless schemes, branch history information and address bits are hashed together to select the appropriate target cache entry. An effective hashing scheme must distribute the cache indexes as widely as possible to avoid interference between different branches.

Table 4 shows the performance benefit of tagless target caches using different history information for indexing into the storage structure. The GAg(9) scheme uses 9 bits of branch pattern history to select the appropriate target cache entry. In the GAs schemes, the target cache is conceptually partitioned into several tables. The address bits are used to select the appropriate table and the history bits are used to select the entry within the table. The GAs(8,1) scheme uses 8 history bits and 1 address bit while the GAs(7,2) scheme uses 7 history bits and 2 address bits. For the perl benchmark, GAg(9) outperforms GAs(8,1), showing that branch pattern history provides marginally more useful information than branch address. This is because the perl benchmark executes only 22 static indirect jumps. On the other hand, GAs(8,1) is competitive with GAg(9) for the gcc benchmark, a benchmark which executes a large number of static indirect jumps. In the gshare scheme, the branch address is XORed with the branch history to form the target cache index. Like the previous 2-level branch predictors, the gshare scheme outperforms the GAs scheme because it effectively utilizes more of the entries in the target cache. In the following sections, we will use the gshare scheme for tagless target caches.

[†] each BTB entry consists of 1 valid bit, 2 least-recently-used bits, 23 tag bits, 32 target address bits, 2 branch type bits, 4 fall-thru address bits, and 13 branch history bits.

4.2.2 Branch Path History: Tradeoffs

Path history consists of the target addresses of branches that lead to the current branch. Ideally, each path leading to a branch should have a unique representation in the path history register. However, since only a few bits from each target are recorded in the path history register, different targets may have the same representation in the path history. When this occurs, the path history may not be able to distinquish between different paths leading to a branch. Thus, the performance of a path based target cache depends on the address bits from each target used to form the path history. Table 5 shows that the lower address bits provide more information than the higher address bits. In the following experiments, the least significant bits from each target are recorded in the path history register; the 2 least significant bits from each address are ignored because instructions are aligned on word boundaries.

Because the length of the history register is fixed, there is also a tradeoff between identifying more branches in the past history and better identifying each branch in the past history. Increasing the number of bits recorded per address results in fewer branch targets being recorded in the history register. Table 6 shows the performance of target caches

Perl		
config	Misprediction Rate	Reduction in Exec. Time
gshare(9)	30.9%	8.92%
GAg(9)	31.2%	8.87%
GAs(8,1)	33.4%	8.36%
GAs(7,2)	48.1%	5.17%
Gcc		
config	Misprediction Rate	Reduction in Exec. Time
gshare(9)	30.4%	4.27%
GAg(9)	35.8%	3.61%
GAs(8,1)	35.7%	3.62%
GAs(7,2)	36.7%	3.48%

Table 4: Performance of Pattern History Tagless Target Caches

Perl					
addr bit no.	Reduction in Execution Time				
	Per-addr	Global			
		branch	control	ind jmp	call/ret
2	7.64%	7.41%	4.70%	12.36%	8.45%
3	10.09%	6.85%	3.41%	11.00%	9.75%
4	5.52%	5.97%	4.88%	10.50%	9.97%
5	7.34%	9.13%	4.31%	9.24%	10.82%
7	6.17%	6.89%	4.22%	11.34%	11.16%
10	6.19%	6.60%	3.94%	12.82%	10.42%
15	1.87%	2.98%	2.30%	10.97%	9.92%
20	0.00%	0.00%	0.00%	0.00%	0.00%
Gcc					
addr bit	Reduction in Execution Time				
	Per-addr	Global			
		branch	control	ind jmp	call/ret
2	2.57%	3.82%	3.75%	2.68%	2.52%
3	2.46%	3.71%	3.81%	2.63%	2.24%
4	2.67%	3.88%	3.63%	2.69%	2.80%
5	2.02%	3.75%	3.82%	2.37%	2.63%
7	2.60%	3.57%	3.76%	2.72%	2.98%
10	1.34%	3.24%	3.14%	1.83%	2.60%
15	0.01%	1.05%	0.78%	1.06%	1.56%
20	-0.02%	0.26%	0.14%	0.24%	0.78%

Table 5: Path History: Address Bit Selection

Perl					
bits per addr	Reduction in Execution Time				
	Per-addr	Global			
		branch	control	ind jmp	call/ret
1	7.64%	7.41%	4.70%	12.36%	8.45%
2	9.17%	3.12%	0.88%	11.81%	8.40%
3	7.19%	0.30%	0.74%	10.99%	6.24%

Gcc					
bits per addr	Reduction in Execution Time				
	Per-addr	Global			
		branch	control	ind jmp	call/ret
1	2.57%	3.82%	3.75%	2.68%	2.52%
2	2.96%	2.60%	2.21%	3.35%	2.66%
3	3.57%	2.02%	1.66%	3.83%	2.63%

Table 6: Path History: Address Bits per Branch

using different numbers of bits from each target in the path history. In general, with nine history bits, the performance benefit of the target cache decreases as the number of address bits recorded per target increases. This is especially true for the `Control` and `Branch` schemes. With the `Control` scheme, we record the target of all instructions that can redirect the instruction stream, which may include branches that provide little or no useful information, e.g. unconditional branches. Each of these uncorrelated branches takes up history bits, possibly displacing useful history. As a result, the performance benefit of the `Control` scheme drops even more significantly when the number of bits to be recorded for each target increases from 1 to 2; for perl, the reduction in execution time dropped from 4.7% to 0.9%.

4.2.3 Pattern History vs. Path History

Table 4 and Table 6 show that using pattern history results in better performance for gcc and using global path history results in better performance for perl. Using the `Indirect Jmp` scheme, the execution time for perl was reduced by 12.34%, as compared to 8.92% for the best pattern history scheme. The branch path history was able to perform extremely well for the perl benchmark because it is an interpreter. The main loop of the interpreter parses the the perl script to be executed. This parser consists of a set of indirect jumps whose targets are decided by the tokens (i.e. components) which make up the current line of the perl script. The perl script used for our simulations contains a loop that executes for many iterations. As a result, when the interpreter executes this loop, the interpreter will process the same sequence of tokens for many iterations. By capturing the path history in this situation, the target cache is able to accurately predict the targets of the indirect jumps which process these tokens.

4.3 Tagged Target Caches

This section examines the performance of tagged target caches. While the degree of associativity was varied, the total size of each target cache simulated was kept constant at 256 entries. The tagged target caches have half the number of entries as that of tagless target caches to compensate for the hardware used to store tags.

4.3.1 Indexing Function

Since there can be several targets for each indirect branch and each target may be reached with a different history, a large number of target cache entries may be needed to store all the possible combinations. Thus, the indexing scheme into a target cache must be carefully selected to avoid unnecessary trashing of useful information.

Three different indexing schemes were studied – `Address`, `History Concatenate`, and `History Xor`. The `Address` scheme uses the lower address bits for set selection. The higher address bits and the global branch pattern history are XORED to form the tag. The `History Concatenate` scheme uses the lower bits of the history register for set selection. The higher bits of the history register are concatenated with the address bits to form the tag. The `History Xor` scheme XORs the branch address with the branch history; it uses the lower bits from the result of the XOR for set selection and the higher bits for tag comparison. Table 7 shows the performance of the different indexing schemes. Global pattern history is used in these experiments. The `Address` selection scheme results in a significant number of conflict misses in target caches with a low degree of set-associativity because all targets of an indirect jump are mapped to the

Perl			
set-assoc.	Reduction in Exec Time		
	Addr	History	
		Conc	Xor
1	0.00%	8.31%	7.42%
2	0.21%	8.30%	7.24%
4	2.76%	8.71%	7.76%
8	6.82%	9.11%	7.73%
16	8.67%	9.14%	8.17%
32	8.67%	9.14%	8.20%
64	9.10%	9.14%	8.10%
128	9.06%	9.14%	8.10%
256	8.10%	9.14%	8.10%

Gcc			
set-assoc.	Reduction in Exec Time		
	Addr	History	
		Conc	Xor
1	0.00%	3.51%	3.56%
2	0.30%	3.85%	3.92%
4	1.27%	4.01%	4.13%
8	2.54%	4.19%	4.24%
16	3.48%	4.30%	4.28%
32	4.22%	4.32%	4.30%
64	4.16%	4.38%	4.31%
128	4.26%	4.52%	4.31%
256	4.31%	4.59%	4.31%

Table 7: Performance of Tagged Target Cache using 9 pattern history bits

same set. Since there are several targets for each indirect jump for gcc and perl as shown in Section 1, a high degree of set-associativity is required to avoid trashing of useful information due to conflict misses. The `History Concatenate` and `History Xor` schemes suffer a much smaller number of conflict misses because they can map the targets of an indirect jump into any set in the target cache, removing the need for a high degree of associativity in the target cache.

In the following sections, we will use the `History Xor` scheme for tagged target caches.

4.3.2 Pattern History vs. Path History

Table 8 shows the performance of tagged target caches that use branch path histories. The path history schemes reported in this section record one bit from each target address into the 9-bit path history register. This design choice resulted in the best performance for most of the path history schemes. As in the tagless schemes, using pattern history results in better performance for gcc and using global path history results in better performance for perl.

Perl					
	Reduction in Execution Time				
set assoc	Per-addr	Global			
		branch	control	ind jmp	call/ret
1	7.13%	6.20%	2.98%	10.71%	7.59%
2	8.20%	6.34%	4.15%	11.62%	7.91%
4	9.15%	7.21%	4.30%	12.29%	7.91%
8	9.46%	7.21%	4.70%	13.38%	8.79%
16	9.69%	7.21%	4.70%	13.78%	8.69%
32	9.77%	7.21%	4.70%	13.97%	8.83%
64	9.77%	7.21%	4.70%	14.14%	8.78%
128	9.77%	7.21%	4.70%	14.15%	8.98%
256	9.77%	7.21%	4.70%	14.15%	8.66%

Gcc					
	Reduction in Execution Time				
set assoc	Per-addr	Global			
		branch	control	ind jmp	call/ret
1	2.45%	3.28%	2.88%	2.15%	2.07%
2	2.72%	3.35%	3.28%	2.46%	2.25%
4	2.90%	3.59%	3.58%	2.65%	2.42%
8	3.01%	3.73%	3.55%	2.76%	2.56%
16	3.05%	3.76%	3.58%	2.86%	2.58%
32	3.11%	3.76%	3.59%	2.90%	2.63%
64	3.12%	3.77%	3.59%	2.92%	2.64%
128	3.14%	3.79%	3.57%	2.92%	2.65%
256	3.14%	3.79%	3.57%	2.94%	2.65%

Table 8: Performance of Tagged Target Caches using 9 path history bits

4.3.3 Branch History Length

For tagged target caches, the number of branch history bits used is not limited to the size of the target cache because additional history bits can be stored in the tag fields. Using more history bits may enable the target cache to better identify each occurrence of an indirect jump in the instruction stream.

Table 9 compares the performance of 256-entry tagged target caches using different numbers of global pattern history bits. For caches with a high degree of set-associativity, using more history bits results in a significant performance improvement. For example, when using 16 history bits, an 8-way tagged target cache reduces the execution time of perl and gcc by 10.27% and 4.31% respectively, as compared to 7.73% and 4.28% when using 9 history bits. Greater improvements are seen for caches with higher set associativity. For a 16-way tagged target cache, using 16 history bits results in 12.66% and 4.74% reduction in execution time for perl and gcc while using 9 history bits results in 8.17% and 4.28% reduction in execution time.

For target caches with a small degree of set-associativity, using more history bits degrades performance. Using more history bits enables the target cache to better identify each occurrence of an indirect jump in the instruction stream; however, more entries are required to record the different occurrences of each indirect jump. The additional pressure on the target cache results in a significant number of conflict misses. The performance loss due to conflict misses outweighs the performance gain due to better identification of each indirect jump.

Perl		
	Reduction in Exec Time	
set-assoc.	9 bits	16 bits
1	7.42%	7.07%
2	7.24%	8.39%
4	7.76%	10.02%
8	7.73%	10.27%
16	8.17%	12.66%
32	8.20%	12.71%
64	8.10%	12.83%
128	8.10%	12.84%
256	8.10%	12.85%

Gcc		
	Reduction in Exec Time	
set-assoc.	9 bits	16 bits
1	3.56%	2.35%
2	3.92%	2.99%
4	4.13%	3.62%
8	4.24%	4.31%
16	4.28%	4.74%
32	4.30%	4.87%
64	4.31%	5.11%
128	4.31%	5.19%
256	4.31%	5.24%

Table 9: Tagged Target Cache: 9 vs 16 pattern history bits

4.4 Tagless Target Cache vs Tagged Target Cache

A tagged target cache requires tags in its storage structure for tag comparisons while a tagless target cache does

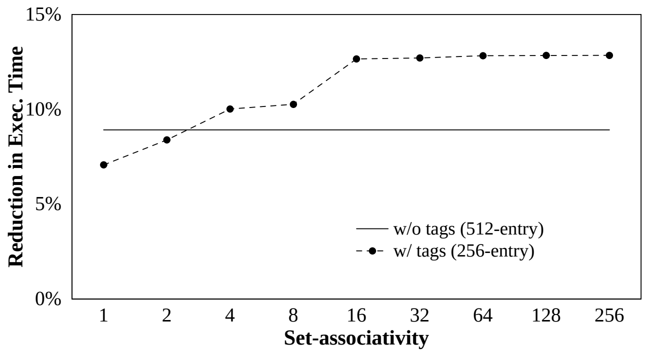

Figure 12: Tagged vs. tagless target cache – perl

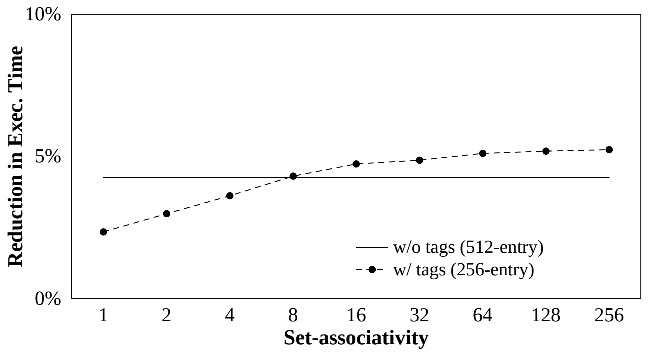

Figure 13: Tagged vs. tagless target cache – gcc

not. Thus, for a given implementation cost, a tagless target cache can have more entries than a tagged target cache. On the other hand, interference in the storage structure can degrade the performance of a tagless target cache.

Figures 12 and 13 compare the performance of a 512-entry tagless target cache to that of several 256-entry target caches, with various degrees of set-associativity. The tagless target cache outperforms tagged target caches with a small degree of set-associativity. On the other hand, a tagged target cache with 8 or more entries per set outperforms the tagless target cache.

5 Conclusions

Current BTB-based branch prediction schemes are not effective in predicting the targets of indirect jumps. These schemes have a 48% misprediction rate for the indirect jumps in the SPEC95 integer benchmarks. To address this problem, this paper proposes the target cache, a prediction mechanism that significantly improves the accuracy of predicting indirect jump targets. The target cache uses concepts embodied in the 2-level branch predictor.

By using branch history to distinguish different dynamic occurrences of indirect branches, the target cache was able to reduce the total execution time of perl and gcc by 8.92% and 4.27% respectively.

Like the pattern history tables of the 2-level branch predictors, interference can significantly degrade the performance of a target cache. To avoid predicting targets of indirect jumps based on the outcomes of uncorrelated branches, tags are added to the target cache. However, with the tagged target cache, useful information can be displaced if different branches are mapped to the same set. Thus, set-associative caches may be required to avoid mispredictions due to conflict misses.

Our experiments showed that a tagless target cache outperforms a tagged target cache with a small degree of set-associativity. On the other hand, a tagged target cache with 8 or more entries per set outperforms a tagless target cache. For example, a 512-entry tagless target cache can reduce the execution time of perl and gcc by 8.92% and 4.27% respectively, as compared to 7.42% and 3.56% for a direct mapped 256-entry tagged target cache. A 16-way set-associative tagged target cache can reduce the execution of gcc and perl by 12.66% and 4.74% respectively.

We examined the SPEC95 integer benchmarks where only a small fraction of instructions are indirect branches; e.g. 0.5% in gcc and 0.6% in perl. For object oriented program where more indirect branches may be executed, tagged caches should provide even greater performance benefits. In the future, we will evaluate the performance benefit of target caches for C++ benchmarks.

6 Acknowledgments

We gratefully acknowledge the support of our industrial partners, without which it would not have been possible to undertake this work, in particular Intel Corporation. In addition, we wish to gratefully acknowledge the other members of our HPS research group for the stimulating environment they provide, in particular, Marius Evers, Sanjay Patel and Jared Stark for their comments and suggestions on this work.

References

[1] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in c++ programs. In *21st Symposium on Principles of Programming Languages*, pages 397–408, 1994.

[2] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale N. Patt. Branch classification: A new mechanism for improving branch predictor performance. In *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 22–31, 1994.

[3] W. W. Hwu and Yale N. Patt. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 18–26, 1987.

[4] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–41, 1991.

[5] David R. Kaeli and Philip G. Emma. Improving the accuracy of history-based branch prediction. IEEE Transactions on Computers, April 1994.

[6] Scott McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[7] Scott McFarling and John Hennessy. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, 1986.

[8] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 15–23, 1995.

[9] Yale Patt, W. Hwu, and Michael Shebanow. HPS, a new microarchitecture: Rationale and introduction. In *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 103–107, 1985.

[10] Yale N. Patt, Steven W. Melvin, W. Hwu, and Michael C. Shebanow. Critical issues regarding HPS, a high performance microarchitecture. In *Proceedings of the 18th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 109–116, 1985.

[11] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, 1981.

[12] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33, January 1967.

[13] C. F. Webb. Subroutine call/return stack. *IBM Technical Disclosure Bulletin*, 30(11), April 1988.

[14] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, 1991.

[15] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, 1993.

[16] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, 1994.