Improving Branch Predictors by Correlating on Data Values

Timothy H. Heil[1], Zak Smith[2], J. E. Smith[1]

[1]Electrical and Computer Engineering
University of Wisconsin-Madison
Madison, WI 53706
{heilt,jes}@ece.wisc.edu

[2]VLSI Technology Lab, Hewlett-Packard
3404 E. Harmony Rd.
Ft. Collins, CO 80528-9599
zak@fc.hp.com

Abstract

Branch predictors typically use combinations of branch PC bits and branch histories to make predictions. Recent improvements in branch predictors have come from reducing the effect of interference, i.e. multiple branches mapping to the same table entries. In contrast, the branch difference predictor (BDP) uses data values as additional information to improve the accuracy of conditional branch predictors. The BDP maintains a history of differences between branch source register operands, and feeds these into the prediction process.

An important component of the BDP is a rare event predictor (REP) which reduces learning time and table interference. An REP is a cache-like structure designed to store patterns whose predictions differ from the norm.

Initially, ideal interference-free predictors are evaluated to determine how data values improve correlation. Next, execution driven simulations of complete designs realize this potential. The BDP reduces the misprediction rate of five SPEC95 integer benchmarks by up to 33% compared to gshare and by up to 15% compared to Bi-Mode predictors.

1. Introduction

Conditional branch prediction is an important technique for improving processor performance. Correct branch predictions avoid control dependences and provide a smooth flow of instructions to be executed. On the other hand, branch mispredictions halt fetching of usable instructions until the branch outcome is known. In out-of-order processors this serializes program execution, dividing the out-of-order instruction window into sequentially executed segments. Since these problems become worse with increasing window size, improved branch prediction is considered a key hurdle for future microarchitectures. Consequently, there continues to be ongoing research (and progress) in branch prediction mechanisms.

The first dynamic branch predictors [23] primarily relied on local history information to make their predictions. Since that time, conditional branch predictors have undergone steady improvements. These improvements fall into three basic categories.

(1) adding global path and history information [19, 24, 27]
(2) refining the ways that global and local history are combined [3, 15]
(3) reducing table interference through more intelligent table indexing schemes [5, 10, 12, 16, 22]

Virtually all the conditional branch predictors proposed to date use control flow information as basic inputs either in the form of branch outcomes or branch PCs. In effect, proposed predictors combine the same type of information in increasingly clever ways. And, despite the steady improvements that have been made, many branches are still mispredicted, and branch prediction remains an impediment to performance.

To find new ways to improve branch prediction, we first took a microscopic view, examining the execution of individual branches to understand why some conditional branches are difficult to predict. For many important branches, PCs and branch outcomes alone do not contain the right information, or the information is not in the right form. However, many of these branches become easily predictable if data-value information can be used. This led us to develop a new class of branch predictors that use data values; a development culminating in the *branch difference predictor* (BDP) -- the topic of this paper.

A straightforward method of integrating data values into branch prediction is what we refer to as *speculative branch execution*, illustrated in Fig. 1. With this type of predictor, a conventional data-value predictor [13, 20, 21, 26] is first used to predict the input values for a branch instruction, then the branch instruction is evaluated using the predicted values. Because conventional branch predictors predict some branches very well, a chooser selects between a conventional predictor and prediction through speculative branch execution. We initially considered this mechanism [9], but ultimately chose a different approach.

We chose to pursue the method illustrated in Fig. 2, where data-value history information is fed directly into the branch predictor. In effect, the predictor correlates on data values similar to the way conventional predictors correlate on global branch history. This method has a couple of advantages. First, it leads to a lower latency predictor than speculative branch execution. As explained in Section 3.1, all predictor tables are accessed in parallel,

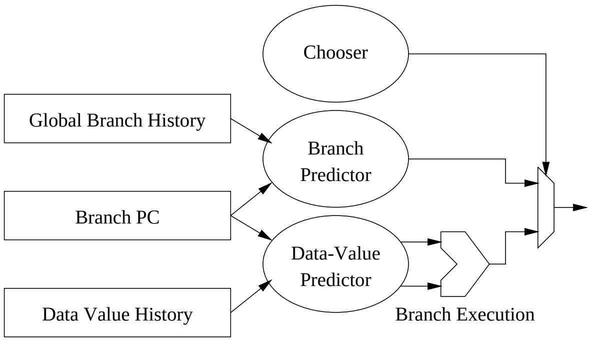

Fig. 1. Branch prediction via speculative execution.

while speculative branch execution requires one or two serial tables accesses (depending on the data-value predictor used), followed by execution of the branch instruction. Second, it allows some branches to be predicted using combined branch outcomes and data value information; speculative branch execution, as shown in Fig. 2, uses either branch history or data values, *but not both*. As a result we found the direct method (Fig. 2) provides better prediction accuracy, even when compared to speculative branch execution using a data-value predictor of unlimited size and an oracle chooser that selects the best prediction mechanism for each static branch [9].

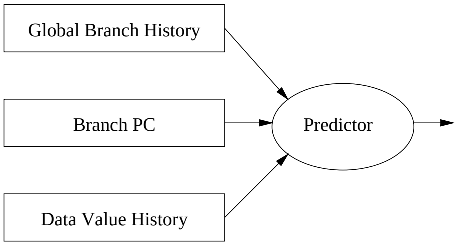

Fig. 2. Using data values for branch prediction directly.

Prediction through speculative branch execution was recently pursued independently by Gonzalez y Gonzalez [7], and a similar scheme was studied by Farcy et al. [6]. Gonzalez and Gonzalez also recognized the importance of using global history and data-value history together by developing a chooser based on path information.

Other related work uses compiler synthesized dynamic branch prediction [1] as a software-based approach for using data values. Compiler synthesized prediction uses an execution profile to guide a compiler in inserting instructions into the binary that compute branch-specific prediction functions. The results of these functions, which are based on register values, are supplied to hardware predictors at runtime through ISA extensions. This mechanism is evaluated for a class of prediction functions on selected branches [1].

The rest of this paper is organized as follows. Section 2 illustrates reasons for branch mispredictions with examples motivating the use of data values. Section 3 describes the branch difference predictor in detail. Section 4 reports the potential performance of the proposed predictor using trace-driven simulations of interference-free predictors. Section 5 reports results of execution-driven simulations using fixed-size predictors. Section 6 discusses future research and concludes the paper.

2. Why branches are mispredicted.

In order to improve branch prediction, we first set out to determine when and why branch history, especially global branch history, does not correlate well with branch outcomes. For this purpose we simulated SPEC benchmarks with an interference-free gselect predictor -- a separate 2-bit counter was allocated to every combination of branch PC and global history. Then, we located the static branches yielding the highest number of mispredictions. We manually inspected the code surrounding these difficult-to-predict branches, and noted the mechanisms at work. This process was repeated for a number of branches, until some basic patterns began to appear. This study was not intended to be comprehensive, but to provide insight for designing better predictors. The following example branches from *gcc* illustrate typical problems with global branch history and how data values help. mispredictions for this branch.

Example 1. One common problem is that the branch history register is too short to encode the last iteration of a loop. An example of this is shown in Fig. 3 below. This simple loop is part of the common memset() C library function. For the *gcc* benchmark, the loop runs for 21 iterations on average, which is too many to be recorded in a typical branch history register (either local or global). Moreover, the loop iterates a variable number of times. As a result, the loop terminating branch is mispredicted most of the time. However, since the loop decrements to

```
xlen = len / (OPSIZ * 8);
while (xlen > 0)
     {
          ((op_t *) dstp)[0] = cccc;
          ((op_t *) dstp)[1] = cccc;
          ((op_t *) dstp)[2] = cccc;
          ((op_t *) dstp)[3] = cccc;
          ((op_t *) dstp)[4] = cccc;
          ((op_t *) dstp)[5] = cccc;
          ((op_t *) dstp)[6] = cccc;
          ((op_t *) dstp)[7] = cccc;
          dstp += 8 * OPSIZ;
          xlen -= 1;
     }
```

Fig. 3. BDP removes 79% of the mispredictions of this difficult-to-predict branch in *gcc*.

0, using the loop counter value as an input to branch prediction permits this branch to be predicted very accurately. BDP, described in the next section, uses this value to eliminate 79% of the mispredictions for this branch.

We note that loop-closing branches of this type can also be successfully handled using a special architected count register, such as found in the PowerPC [14] and IA-64 [8] instruction sets. Nevertheless, we also found many examples, such as those below, where count registers will not work effectively. Using data values in the predictor provides benefits similar to a count register, but is more general, and does not require ISA extensions. □

Example 2. The code in Fig. 4 is part of the parser from *gcc*'s front-end. The parser is automatically generated by the *Bison* tool, which produces C code for a finite-state machine to parse LALR(1) grammars. The if-statement in bold tests whether or not the next parser action can be determine solely from the current state, without reference to a look-ahead token.

```
yyn = yypact[yystate];
if (yyn == YYFLAG) goto yydefault;
...
switch(yyn) {
     ... 281 cases ...
}
```

Fig. 4. BDP removes 75% of the mispredictions of this difficult-to-predict branch in *gcc*.

The heart of the parser is a switch function with 281 cases surrounded by complex control flow. This tends to confuse the global branch history, with no consistent correlation between specific static branches and bits in the history register. Global branch history does not correlate well to this branch; the interference-free gselect predictor used in Section 4 mispredicts this branch 19.5% of the time. However, since source code is highly regular, it is not surprising that parser states and actions are also regular. BDP capitalizes on the regularity of these data-values, reducing the misprediction rate to 4.8%.

Local history does not work well either. Using only local history results in a 34% misprediction rate for this branch. Predicting the branch with both global and local history produces an 8.0% misprediction rate, a significant improvement, though still not as good as BDP. Table 1 illustrates why. Table 1 shows a selection of the branch's executions. These instances have the same global history (*TTTN TNTN NTTT TNTN*) and local history (*TTTN*). The local history pattern maps to both taken and not-taken branches. Hence, the local-history based predictor misses frequently, even in combination with global history. However, the branch is never taken when the data-value history is 8126, and is always taken when the data-value history is 82cf. Data-value history contains the correct information to predict for these instances of the branch, while local branch history does not.□

Table 1. Branch executions from Example 2.

History		T/N	Prediction Result	
Local	Data Value		BDP	Local + Global
TTTN	82cf	T	H	H
TTTN	8126	N	H	M
TTTN	8126	N	H	H
TTTN	82cf	T	H	M
TTTN	82cf	T	H	M
TTTN	8126	N	H	M
TTTN	82cf	T	H	M
TTTN	8126	N	H	M

From studying these branches and others, we concluded that many hard-to-predict branches would be, in fact, relatively easy to predict if the predictor could be given the right information. In particular, these same branches correlate well with the values tested by the branch instruction. This conclusion led us to develop the branch difference predictor described next.

3. The branch difference predictor (BDP)

The overall design of the BDP is shown in Fig. 5. The basis of the proposed predictor is correlation on data values. To make this practical, the predictor overcomes two difficulties discussed below -- the large numbers of patterns that occur with data values, and the delay in updating the data values due to out-of-order execution and pipeline latencies.

3.1. Managing large numbers of data values.

Because many branch instructions test two register values, we reduce the amount of data history by using the *difference* of the two branch source register operands instead of the operands themselves. Since branch outcomes are based on subtracting the two inputs and looking at the sign and/or detecting a zero difference, these

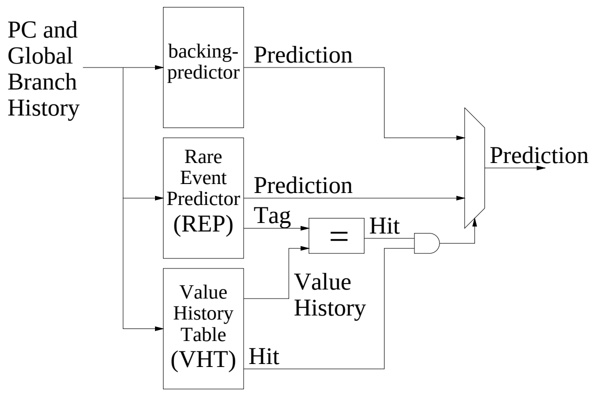

Fig. 5. The branch difference predictor (BDP).

branch differences correlate very well with branch outcomes. Consequently, the BDP collects these branch difference values in the *value history table* (VHT) in Fig. 5. Value history information, described below in Section 3.2, is maintained per static branch, and is retrieved using the branch PC.

Even when using branch differences, the number of patterns is still very large. A single static branch instruction may produce many values, but only a few of these branch differences will be significant. If all values are used for all branches, then either the storage space becomes excessive, or there will be excessive table interference.

In order to further reduce storage space, we use the observation that not all difference patterns need to be stored. The conventional *backing predictor* in Fig. 5 predicts most cases without using difference values. The small remaining subset of the patterns are predicted by a separate structure, a *rare event predictor* (REP), using both the data-value *and* global branch history.

An REP is a tagged cache-like structure which is used to predict only the difficult, exceptional cases. The replacement policy used by the REP explicitly searches for patterns that lead to correct branch predictions differing from the one made by the backing predictor. The REP derives from partial pattern matching [4, 17], and these structures have appeared in various kinds of predictors [11, 18, 25], most similarly the YAGS branch predictor [5].

Referring again to Fig. 5, while the VHT is being accessed, the REP is accessed in parallel with the PC and global branch history (GBH). The value history is used for the tag check. If the pattern is found, the REP provides the prediction. If the REP does not contain the pattern, the backing predictor provides the prediction. The tags in the REP serve as a built-in chooser to indicate when the REP should be used instead of the backing predictor, and have the additional benefit of eliminating interference within the REP.

Since the REP is intended only to correct the rare, exceptional cases, we place new patterns into the REP only when the backing predictor mispredicts. Since the branch outcome is known when a new entry is placed in the REP, the two bit counters for the entry are set to weakly taken or not-taken according to the outcome of the branch. The backing predictor is only updated when it provides the prediction. This is important because it allows the REP to reduce interference in the backing predictor, as well as improve correlation.

3.2. Managing delayed updates. The actual data values used by a branch are rarely available at the time the branch is to be predicted. In fact, because of loops the branch predictor may be several dynamic instances ahead of the computation of input data values. Using the most recent known-good (committed) data values leads to stale data and inaccurate predictions. As an alternative, the data values used can be predicted and updated speculatively, but this requires an explicit data-value predictor, as well as recovering and restoring data-value history when there is a misprediction.

As a compromise, for each static branch we keep the last known-good branch difference and a dynamic count of the occurrences of the branch fetched beyond this point. These two items together are nearly as good as using the speculatively updated data, *but* no data-value predictor is needed. Consider a predictable, repeating stream of branch differences. The last committed branch difference indicates where the last committed branch was in the stream. The number of outstanding branches indicates how much further down the stream the current branch is from the last committed branch. Together, they form a reliable indication of the current branch difference, and the outcome of the current branch.

The VHT collects branch differences and counts for use by the predictor in two tables (Fig. 6). The *branch difference cache* (BDC) stores the most recently committed branch difference, indexed and tagged by the branch PC. Misses in the BDC are simply handled by not using the REP for that prediction. Since the BDC only contains committed values, these values never need to be repaired after exceptions or branch mispredictions.

The *branch count table* (BCT) keeps track of the number of outstanding instances of each static branch, indicating how out-of-date the differences in the BDC are. When a branch is fetched, the corresponding counter in the BCT is incremented. When the branch is committed the counter is decremented and the BDC is updated with the new difference. After a branch misprediction, the count for each squashed branch is decremented. Note that the BCT is untagged; tagging makes it difficult to keep the counters synchronized across replacements.

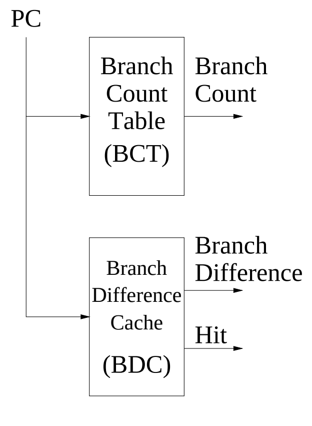

Fig. 6. The value history table.

3.3. A special case: set instructions. *Set* instructions in the MIPS-based SimpleScalar ISA perform a comparison and pass the result of the comparison to a later branch as a zero or one. In this case the branch difference contains no additional information beyond the branch outcome. To allow BDP to work well with *set* instructions, these instructions also produce the difference of their inputs. When a branch uses a value produced by a *set* instruction, this *set difference* is used to compute the branch difference in place of the zero/one branch input.

This requires an extra table containing a valid bit and a set difference for each architected register. The configuration studied here truncates differences to eight bits (with little loss in performance), and requires only a 36 byte table. When a *set* instruction retires, it stores its set difference in the table, and sets the valid bit for the output architectural register. Other instructions clear the valid bit when they retire. When a branch retires, it consults the valid bit for each operand in the table. If the valid bit is set, then that operand was produced by a *set* instruction. In this case the set difference from the table is used to compute the branch difference, in place of the zero/one operand. Note that this complication is on the update path, not on the critical prediction path.

4. Interference-free predictors.

Initially we use interference-free tables for all portions of the branch predictor so we can study the potential improvement of correlating on data values. We compare with an interference-free gselect predictor using 16 bits of global branch history, also used as the backing predictor for the REP. The VHT contains a unique difference and count for every static branch. Likewise the REP has a separate fully-tagged two-bit counter for every combination of PC, global history, branch difference, and count.

Since these tables are interference-free, we always update both the REP and the backing predictor, regardless of which table made the prediction. We also place all patterns in the REP, regardless of whether the prediction was a hit or miss.

Since this initial study was trace-driven, the timing of updates to the VHT are not accurate. The difference values are artificially "aged" by throwing away the n most recent values, which is equivalent to setting all BCT counts to (n+1), as the count of outstanding branches includes the branch being predicted. A count of one means the branch currently being predicted is the only instance of that static branch outstanding. Varying the count in this way allows us to measure how sensitive the correlation is to the age of the branch difference. To determine how much data-value history is helpful, the number of difference values used for correlation is also varied in this initial study. That is, we consider sequences of difference values ranging in length from 1 to 3, in a manner similar to context-based data predictors [21].

4.1. Simulation methodology.

We study five of the less predictable SPECint95 benchmarks, *compress*, *gcc*, *go*, *ijpeg*, and *li*, shown in Table 2. We simulate the first 200M instructions of each benchmark using a reduced input data set to exercise more benchmark code. *go* and *ijpeg* both complete before 200M instructions. The branch working set is the number of static branches that account for 90% of the dynamic branch executions.

The benchmarks are simulated with the SimpleScalar 2.0 suite [2], modified to simulate the Bi-Mode predictor, the value history table, and the REP. The initial studies use trace driven simulation. The final study uses *sim-outorder*, the SimpleScalar cycle-level superscalar timing simulator.

Table 2. Benchmark characteristics

Bench	Input	Instr. (1.0e6)	Dyn. Branch (1.0e6)	Branch Working Set
comp	400000 e	200	28.4	26
gcc	gcc.i	200	30.6	3126
go	9 9	133	16.0	1092
ijpeg	specmun	129	9.9	84
li	queen.lsp	200	30.9	63

4.2. Results.

Fig. 7 shows the branch misprediction rates for *gcc*, *go*, and the average over all five benchmarks. Cumulative misprediction rates are computed using a weighted average based on a constant workload

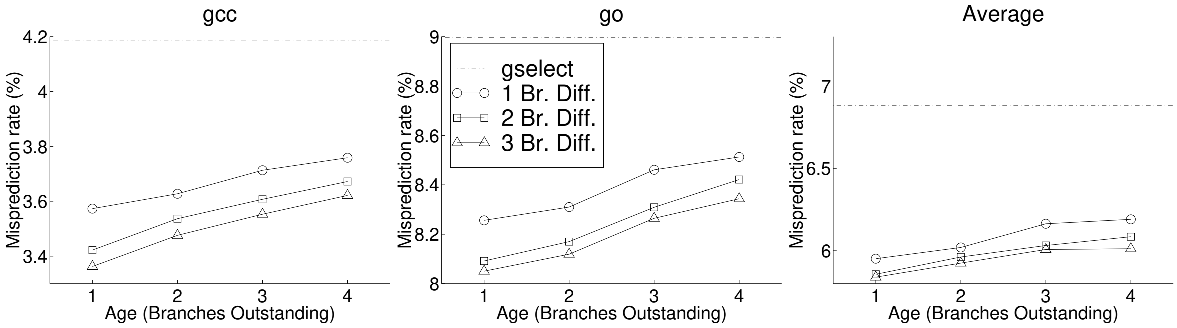

Fig. 7. Misprediction rates for interference-free predictors.

model in which each of the five benchmarks account for the same number of branches. The horizontal dashed line near the top of each graph is the gselect misprediction rate, and the three lower curves are the BDP misprediction rates using value histories one, two, and three branch differences long. Compared to gselect, using branch differences in addition to branch history further reduces the misprediction rate by 7% (*compress*) to 51% (*li*) with an average of 19%.

The majority of the improvement is gained by using a single branch difference. The average misprediction rates plotted in Fig. 7 show that adding a second or third branch difference to the value history results in little additional improvement. This means BDP can provide the bulk of the improvement with a single branch difference, and we use this simpler design throughout this paper.

The X axis is the age of branch differences, which is varied from one to four. The clock-cycle simulations in Section 5.4 will show that the count field is within this range 93% of the time. The positive slope of the curves in Fig. 9 indicate that older branch differences do not correlate with the branch outcome as well as more recent values, but the degradation is not severe. Using a linear least-squares fit, the misprediction rate increases by 0.09% per outstanding branch.

It is also interesting to look at the improvement of individual static branches in the program. Fig. 8 contains scatter plots of all the static branches executed in *compress* and *gcc*. For each branch the mispredictions made by the gselect predictor are compared to mispredictions made by BDP. The X axis shows the number of mispredictions caused by the static branch when predicted with gselect. The Y axis shows the number of mispredictions eliminated when using the branch difference predictor. Branches improved by BDP are plotted above zero on the Y axis; branches mispredicted more often by BDP are plotted below zero on the Y axis. A point on the *x=y* line indicates that all mispredictions of that branch were eliminated by the branch difference predictor. Looking at *compress*, four branches have mispredictions virtually eliminated by the branch difference predictor. All other branches are relatively unaffected. On the other extreme, *gcc* has a wide range of static branches helped by varying amounts; very few are hurt. The example branches from Fig. 3 and 4 have been marked on this plot. The other three benchmarks we studied look more like *gcc* than *compress* [9].

5. Fixed-size predictors

After verifying that branch differences have significant potential to improve branch prediction accuracy, we confirm this using detailed simulations of realistic predictors. Performance simulations are important to correctly determine how old the data being used for prediction is, which depends on when branches commit. Details of the simulated superscalar pipeline are in Table 3.

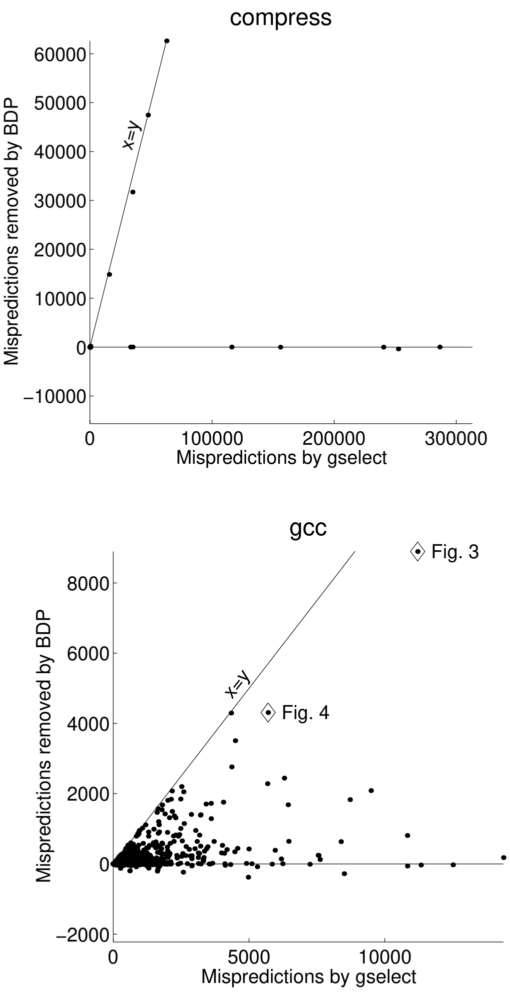

Fig. 8. Improvements for each static branch.

Table 3. Modeled superscalar pipeline parameters.

Fetch	8 instructions/cycle, 1 taken branch
Issue	8 instructions/cycle, 4 loads or stores/cycle
Branch penalty	8 cycle minimum (fetch to execute)
Instr. window	64 instructions, 32 loads and stores
L1 D-cache	64KB, 2-way set associative, 16 byte lines, 1 cycle latency
L1 I-cache	64KB, 2-way set associative, 16 byte lines, 1 cycle latency
L2 unified cache	1MB, 4-way set associative, 16 byte lines, 8 cycle latency
Memory	100 cycles, 16 byte bus

5.1. Predictor configurations. The branch difference predictor presents a large design space. The configuration used in this study, shown in Fig. 9a and b, was incrementally derived through many simulations [9], but is not presented as an optimal point.

The VHT (Fig. 9a) is only 164 bytes total. The count of outstanding branches is stored in 256 untagged two-bit counters. To reduce table cost, the branch difference values are truncated to the low-order eight bits of the difference and stored in 64 entries, indexed with the low order bits of the PC, and tagged with the next four bits of PC.

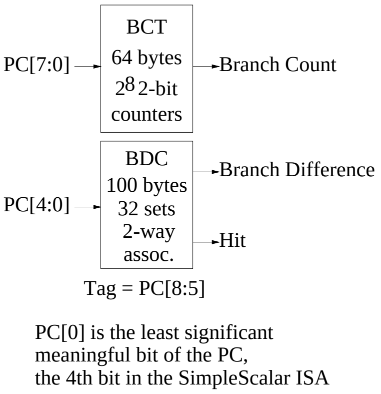

Fig. 9a. Value history table (VHT) configuration.

The REP (Fig. 9b) is a 6KB table containing 2048 entries of three bytes each. Each entry has a six bit replacement counter, whose purpose is explained in Section 5.1, four 2-bit counters for prediction, and a ten bit tag. The most recent bits of global branch history select among the four 2-bit counters per entry. Eight bits of PC along with four more bits of GBH select the set. The tag is a XOR-based hash of seven PC bits, four GBH bits, the eight low-order bits of the branch difference and the two bit count from the VHT.

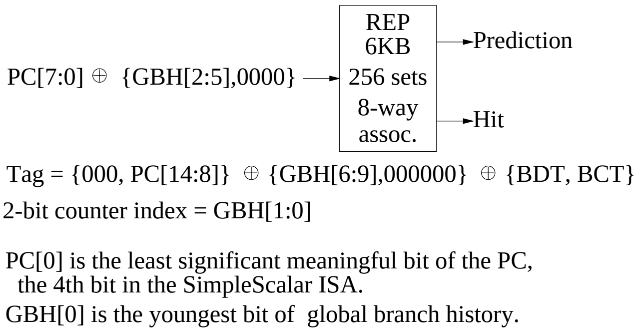

Fig. 9b. Rare event predictor configuration.

We use the gshare and Bi-Mode predictor configurations shown in Fig. 9c and d as both base-case predictors and as backing predictors for the REP.

It is important to avoid the trap of using too much global history in the gshare predictor [10, 15, 22]. Consequently, for a gshare predictor with a *pattern history table* (PHT) containing 2^N 2-bit saturating counters this study uses only N-4 bits of GBH, as shown in Fig. 9c. Aligning the GBH bits with the most-significant bits of the PC reduces interference.

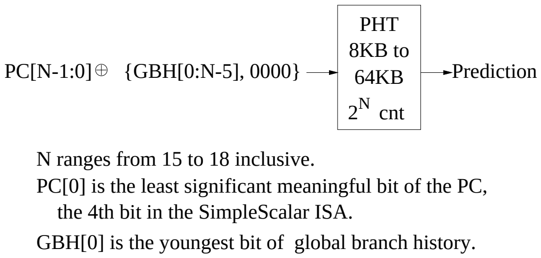

Fig. 9c. Gshare predictor configuration.

The Bi-Mode predictor [12] reduces interference over gshare by splitting the PHT into two *direction PHTs*. A *choice PHT* selects between the two tables. The choice PHT directs predominantly taken branches to use the "taken" direction PHT and predominantly not-taken branches to use the "not-taken" direction PHT, reducing destructive interference. Although the large gshare predictors studied here have very little interference, the Bi-Mode predictor still consistently improves prediction accuracy. The primary reason is that the choice PHT acts like local branch history, improving the correlation of the predictor, as well as reducing interference.

Similar to the gshare predictor, to further reduce interference the direction PHT is indexed with one less bit of global branch history than the maximum possible. Making the choice PHT half the size of each of the direction PHTs reduces the space-overhead of using Bi-Mode, without significantly hurting prediction accuracy.

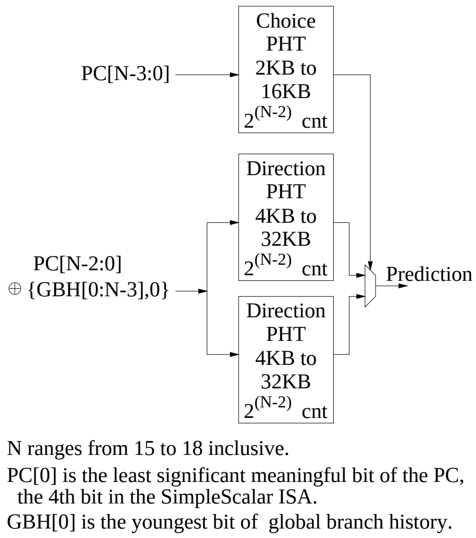

Fig. 9d. Bi-Mode predictor configuration.

Like Bi-Mode, YAGS [5] is another advanced predictor aimed at reducing interference in a global history based predictor. Because the YAGS branch predictor also uses an REP, we considered using it as a "state-of-the-art" comparison for BDP. However, in our simulations YAGS did not perform as well as the gshare or Bi-Mode predictors described above.

5.2. REP replacement policy.

The replacement policy used by the REP deserves special attention. Many more combinations of PC, global history, count value, and branch difference occur than can be held in the 2048 available REP entries. The conclusion is that the REP is a cache that must be managed very carefully to avoid destructive thrashing. Under these circumstances traditional replacement policies like random replacement, and least-recently-used do not work well [9].

The goal of the replacement policy is to find the most useful patterns and keep these, and only these, in the REP. An entry is useful when it provides *corrective predictions*, i.e. correct predictions where the backing predictor would have missed. When the REP makes the same prediction as the backing predictor, the prediction is *redundant*.

The REP replacement policy employs a six bit counter per entry to keep track of how useful an entry has been recently. The entry with the lowest counter value in the set is replaced. The counter is incremented by two when a corrective prediction is detected during branch commit. The replacement counter is decremented by two for every redundant prediction. When the counter saturates at 63, all the other counters in the same set are divided by two. To prevent a single entry from saturating repeatedly and zeroing all the other counters this is only done if the counter was not already at 63. This replacement policy directly measures the usefulness of a pattern over the recent past. Furthermore, the usefulness of the pattern is compared against the other competing patterns in the same set, as opposed to an arbitrary threshold.

Although the replacement policy is complex, replacements only occur for one quarter of mispredicted branches. Three quarters of the mispredictions are randomly ignored. This gives the replacement policy time to evaluate new patterns, improving branch prediction accuracy and reducing the replacement bandwidth. Furthermore, this complexity is off the critical prediction path.

5.3. Overall performance.

Fig. 10 shows the misprediction rates for four predictors, gshare, Bi-Mode, BDP backed by gshare (BDP/gshare) and BDP backed by Bi-Mode (BDP/Bi-Mode). Misprediction rates are plotted against the size of the predictor in kilobytes. On average, BDP removes 13% to 9% of the mispredictions over gshare, and 12% to 8% of the mispredictions over Bi-Mode. Greater improvement is generally seen for the smaller configurations.

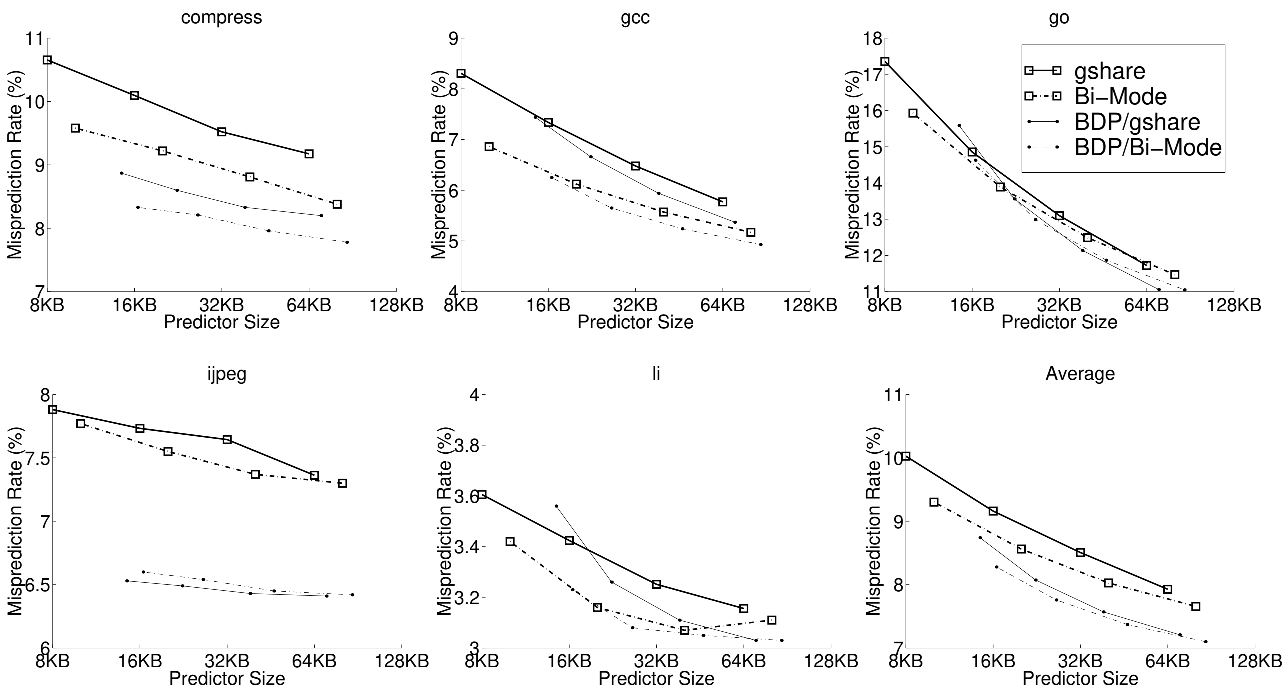

Fig. 10. Branch misprediction rates

Fig. 11 plots the average instructions-per-cycle (IPC) versus predictor size. BDP produces speedups of 2.4% to 1.2% over gshare, and speedups of 1.5% to 0.8% over Bi-Mode alone. Although these speedups are relatively small, there are several things to note. First, they are in line with speedups obtained for other recent innovations in branch predictors, such as the Bi-Mode predictor simulated here. (We note that most branch prediction studies are trace driven and do not provide the resulting speedup.) Secondly, speedups are model-dependent. More aggressive implementations may reduce other bottlenecks in the processor, making performance more sensitive to branch prediction. However, some preliminary investigations with 128 instruction windows showed little difference in either IPC or branch prediction rate.

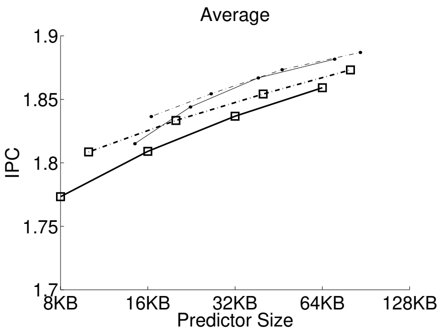

Fig. 11. Instructions-per-cycle.

5.4. REP effectiveness. Fig. 12 shows a histogram of the number of outstanding branches, as indicated by the BCT. 93% of the time only one to four branches are outstanding, and most of the time, the branch being predicted is the only instance. This is a favorable result, since Section 4.2 showed that branch differences provide helpful information for prediction over this range, and it allows two-bit wrapping counters to be used to store the count values. The counts stray above 8 a mere 1.2% of the time.

Since two-bit counters do sometimes wrap over, the predictor may not know exactly how many branches are outstanding. A outstanding count of 2 may mean 2, 6 or 10 branches are outstanding. We have found that this effect does not degrade accuracy significantly. Moving to three bits would increase the BCT from 64 bytes to 96 bytes.

Table 4 provides statistics for the 26KB BDP/Bi-Mode predictor configuration demonstrating the effectiveness of the REP. The backing predictor makes 87.0% of the predictions on average; the REP predicts a small minority of the branches. The REP has a *higher* misprediction rate (12.1% vs. 7.1%), even though it has very little interference and better correlation. The fact that the REP improves the overall prediction rate means that the REP is doing better on these predictions than the backing predictor does by itself. This indicates that the REP is doing what it is supposed to do: provide exceptionally good predictions for the exceptionally difficult cases. The final column shows that 51.5% of REP predictions correct the prediction made by the backing predictor.

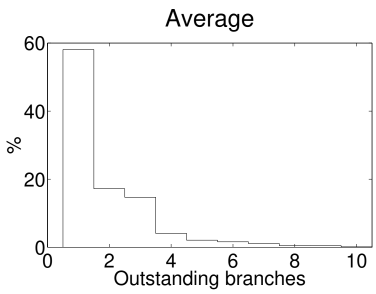

Fig. 12. Average counts of outstanding branches.

Table 4. REP performance statistics.

	Backing Pred.		REP	
Bench	Portion	Miss	Miss	Corrective
comp	87.1	8.0	9.7	53.8
gcc	91.6	5.0	12.5	53.0
go	91.8	12.6	17.8	40.9
ijpeg	84.1	5.8	10.4	54.4
li	81.6	1.8	8.9	59.1
Avg.	**87.0**	**7.1**	**12.1**	**51.5**

6. Conclusions and further research

We have demonstrated that general purpose register values provide useful information for predicting branches. Using data-values has produced a significant incremental improvement in branch prediction accuracy, an improvement as large as the previous improvements over gshare. The differences of branch source operands used by BDP are a particularly useful and readily available form of register value.

In combination with data-values, the rare event predictor (REP) improved correlation, reduced interference in the backing predictor, and handled the large number of patterns which data-values tend to cause. By using the REP, BDP improved on both gshare and the Bi-Mode predictors when used alone. Furthermore, BDP can be used with other predictors that have been or may be developed in the future.

Future research will focus on further extending the amount and type of information embedded in the REP. Since the REP tags are computed in parallel with the array access, adding or modifying the information used in the prediction is easy. For instance, further research may find more local history [27] and path history [7, 19, 24] are helpful.

The information used in the REP can even be adapted dynamically. Rather than a one-size-fits-all approach, predictor parameters can be adjusted to suit the program being run. For instance, this study correlated on a single single branch difference because larger data-value contexts increased the number of patterns, hurting most of our benchmarks. However, *li* is aided significantly by using two or three branch differences [9]. Similar to Dynamic History-Length Fitting [10] it should be possible to dynamically adapt the type and amount of information used by the REP.

Register data values are one example of the wealth of information processors provide beyond branch PCs and outcomes. Future progress will be made by intelligently assimilating this information into branch predictors.

Acknowledgments

This work was supported by NSF Grant MIP-9505853, by an IBM Partnership Award, and by SUN Microsystems. The authors would also like to thank Rushan Chen for her valuable contributions in the preliminary stages of this research.

References

[1] David L. August, Daniel A. Connors, John C. Gyllenhaal, Wen-mei W. Hwu, "Architectural Support for Compiler-Synthesized Dynamic Branch Prediction, Strategies: Rational and Initial Results," *3rd Intl. Symp. on High Performance Comp. Arch.*, pp. 84-93, Feb. 1997.

[2] Douglas C. Burger, Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0," *Univ. of Wisconsin - Madison Comp. Sci. Tech. Report #1342,* June 1997.

[3] Po-Yung Chang, Eric Hao, Yale N. Patt, "Alternative Implementations of Hybrid Branch Predictors," *28th Intl. Symp. on Microarchitecture*, pp. 252-257, Nov. 1995.

[4] I-Cheng K. Chen, John T. Coffey, Trevor N. Mudge, "Analysis of Branch Prediction via Data Compression," *7th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Sys.,* pp. 128-137, Oct. 1996.

[5] Avinoam Nomik Eden, Trevor Mudge, "The YAGS Branch Prediction Scheme," *31st Intl. Symp. on Microarchitecture,* pp. 69-77, Dec. 1998.

[6] Alexandre Farcy, Olivier Temam, Roger Espasa, Toni Juan, "Dataflow Analysis of Branch Mispredictions and Its Applications to Early Resolution of Branches," *31nd Intl. Symp. on Microarchitecture,* pp. 59-68, Nov. 1998.

[7] Jose Gonzalez, Antonio Gonzalez, "Control-Flow Speculation through Value Prediction for Superscalar Processors," *Intl. Conf. on Parallel Arch. and Comp. Tech.,* Oct. 1999.

[8] Linley Gwenmap, "Intel Discloses New IA-64 Features," *Microprocessor Report,* pp. 16-19, March 1999.

[9] Timothy H. Heil, "Branch Difference Prediction and the Rare Event Predictor," Univ. of Wisonsin - Madison Tech. Rep. #ECE-99-4.

[10] Toni Juan, Sanji Sanjeevan, Juan J. Navarro, "Dynamic History-Length Fitting: a third level of adaptivity for branch" prediction," *25th Intl. Symp. on Computer Architecture,* pp. 155-166, June 1998.

[11] John Kalamatianos, David R. Kaeli, "Predicting Indirect Branches via Data Compression," *31st Intl. Symp. on Microarchitecture,* pp. 272-280, Dec. 1998.

[12] Chih-Chieh Lee, I-Cheng K. Chen, Trevor N. Mudge, "The Bi-Mode Branch Predictor," *30th Intl. Symp. on Microarchitecture,* pp. 4-13, Dec. 1997.

[13] Mikko H. Lipasti, John Paul Shen, "Exceeding the Dataflow Limit via Value Prediction," *29th Intl. Symp. on Microarchitecture,* pp. 226-237, Dec. 1996.

[14] Cathy May, ed., *The PowerPC Architecture,* 2nd ed., Morgan Kaufmann, May 1994.

[15] Scott M. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[16] Pierre Michaud, Andre Seznec, Richard Uhlig, "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," *24th Intl. Symp. on Computer Architecture,* pp. 292-303, June 1997.

[17] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Trans. on Communications,* vol. 38, no. 11, pp. 1917-1921, Nov. 1990.

[18] Andreas Moshovos, Gurindar S. Sohi, "Streamlining Inter-operation Memory Communication via Data Dependence" Prediction," *30th Intl. Symp. on Microarchitecture,* pp. 235-245, Dec. 1997.

[19] Ravi Nair, "Dynamic path-based branch correlation," *28th Intl. Symp. on Microarchitecture,* pp. 15-23, Nov. 1995.

[20] Bohuslav Rychlik, John Faistl, Bryon Krug, John P. Shen, "Efficacy and Performance Impact of Value Prediction," *Intl. Conf. on Parallel Arch. and Comp. Tech.,* Oct. 1998.

[21] Yiannakis Sazeides, James E. Smith, "The Predictability of Data Values," *30th Intl. Symp. on Microarchitecture,* pp. 248-258, Dec. 1997.

[22] Stuart Sechrest, Chih-Chieh Lee, Trevor Mudge, "Correlation and Aliasing in Dynamic Branch Predictors," *23rd Intl. Symp. on Computer Architecture,* pp. 22-31, May 1996.

[23] James E. Smith, "A study of branch prediction strategies," *8th Intl. Symp. on Computer Architecture,* pp. 135-148, May 1981

[24] Jared Stark, Marius Evers, Yale N. Patt, "Variable Length Path Branch Prediction," *8th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Sys.,* pp. 170-179, Oct. 1998.

[25] Stefanos Kaxiras, James R. Goodman, "Improving CC-NUMA Performance Using Instruction-Based Prediction," To appear in *5th Intl. Symp. on High Performance Comp. Arch.,* 1999.

[26] Kai Wang, Manoj Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors," *30th Intl. Symp. on Microarchitecture,* pp. 281-290, Dec. 1997.

[27] Tse-Yu Yeh, Yale N. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," *20th Intl. Symp. on Computer Architecture*, pp. 257-266, May. 1993.