Creating Artificial Global History to Improve Branch Prediction Accuracy

Leo Porter and Dean M. Tullsen
University of California, San Diego
La Jolla, California 92093

ABSTRACT

Modern processors require highly accurate branch prediction for good performance. As such, a number of branch predictors have been proposed with varying size and complexity. This work identifies techniques to improve the accuracy of most predictors. It is especially effective with smaller, simpler predictors, allowing those predictors to be competitive with more expensive and complex variants.

Modern branch predictors rely heavily on global history to produce accurate branch predictions. However, not all regions of control flow are correlated with recently executed branches. For these regions, the extra information encoded in the global history does more harm than good. This work performs artificial modifications to the global history register to improve branch prediction accuracy, targeting regions with limited branch correlation. This approach is applied to a number of realistic modern branch predictors of varying size.

The total number of mispredicts in select SPEC2000 benchmarks and the Championship Branch Prediction traces can be reduced by 12% overall for a 32 Kb alloyed perceptron predictor. 2Bc-gskew, A21264, filter, and gshare predictors also benefit from these techniques.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiprocessors

General Terms

Design, Performance

Keywords

Branch Prediction

1. INTRODUCTION

The importance of accurate branch prediction has been well documented in the literature. Modern processors rely on highly accurate branch prediction for good performance. In this work, we propose a simple technique based on heuristics that improves branch prediction for a number of branch predictors. Smaller, less complex branch predictors benefit the most from our technique.

In the uniprocessor era, it made sense to pursue increasingly larger, more complex predictors to squeeze out even small margins of performance. In the multi-core era, every transistor we do not use on one core can be put to other use – allowing more cores, faster cores, more cache, better interconnects, etc. This is even more true of the power envelope, as future processors will be designed under very tight power constraints. Therefore, the highest performance processor is composed not from the highest performance building blocks, but rather from the most area-efficient and power-efficient building blocks. Thus even the branch predictor must carefully justify the use of its transistor budget, and smaller predictors may provide higher processor-wide performance by better utilizing those resources elsewhere. This research demonstrates techniques that improve the branch prediction accuracy of most modern predictors. By requiring no additional storage and minimal logic, it improves overall processor performance with no power or real estate cost. Moreover, as it is most effective on small predictors, allowing those to become more competitive with larger, more complex predictors, it potentially enables a reduction in predictor size with no cost in per-core performance.

To produce highly accurate predictions, modern branch predictors use global history to index prediction tables [27, 1, 10, 13, 19, 20], as input to a neural network [8], or as tags in table lookup [4, 17]. Global history is successfully used to produce accurate branch predictions because branches often correlate with previously executed branches (other nearby branches and themselves). Longer branch histories enable predictors to view a larger window of previously executed branches and learn based on correlations with those branches.

Evers, et al. [5] demonstrate that the amount of correlation with prior branches varies per branch. For branches highly correlated with recent history, global history can provide key prediction information. However, we show that for a branch that is not highly correlated, that history is mostly noise and does more harm than good. It increases the time to train the predictor and it significantly expands the level of aliasing in the prediction tables, reducing the accuracy of prediction on this and other branches.

The technique we propose directly modifies global history based on simple code heuristics. These heuristics identify

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

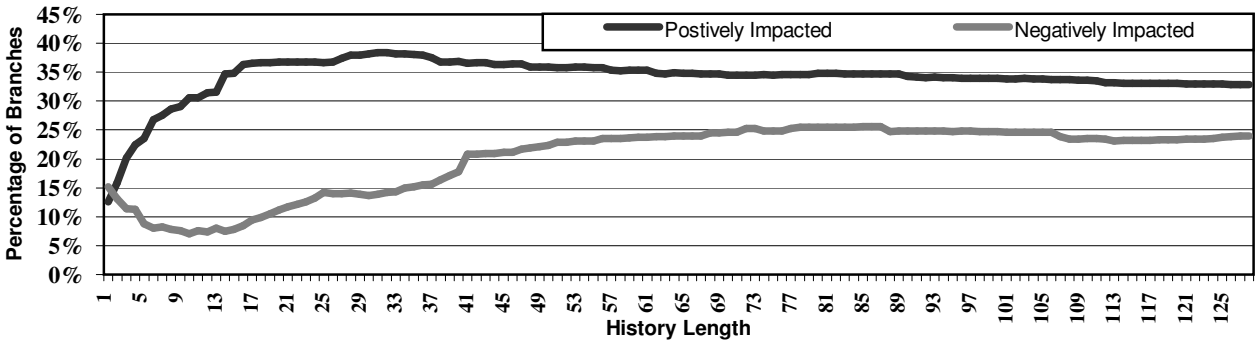

Figure 1: Percentage of dynamic branches positively or negatively impacted by history.

sections of code where branch correlation is likely to be low, and modifies the GHR to reduce or eliminate unnecessary noise. Specifically, these heuristics target backward branches, function calls, and returns.

We evaluate these techniques using select SPEC2000 benchmarks and the Championship Branch Prediction (CBP) traces from The Journal of Instruction Level Parallelism. These techniques provide gains for a number of branch predictors. For a set of 32Kb predictors, these techniques improve each of the A21264, gshare, and alloyed perceptron predictors. The latter reduces mispredicts per thousand instructions (MISP/KI) by 12% overall. A 416Kb implementation of 2Bc-gskew has a 9% reduction in MISP/KI for the CBP traces. Smaller predictors benefit more from these techniques - an 8Kb implementation of gshare achieves a 18% overall reduction in MISP/KI for our select SPEC benchmarks.

This work provides the following contributions: (1) We provide a technique of GHR modification that has exceptionally low hardware cost and demonstrate that branch prediction can benefit from its use. (2) The benefit of this technique is shown for a number of branch predictors of low to medium complexity and for a number of branch prediction sizes. These gains are shown both on a selection of SPECint benchmarks and the Championship Branch Prediction Competition traces.

This paper is organized as follows: Section 2 provides the motivation and basic architectural approach. Section 3 discusses related work. Section 4 discusses our hardware techniques based on code heuristics, and also discusses some potential hardware/software techniques. Section 5 provides the methodology. Section 6 provides results and Section 7 concludes.

2. MOTIVATION

The global history register (GHR) allows a predictor to exploit correlations with recently executed branches. A longer GHR enables correlation with more distant branches, but also increases the number of uncorrelated branches that are included in the history. Those uncorrelated branches can create significant noise. Consider a 15-bit GHR. A branch that is highly correlated with 3 prior branches will make good use of a correlating predictor; but even in this positive scenario, the history contains 12 bits of useless noise. This means that (worst case), we could have to use 2^{12} times more entries to predict this branch than we need, greatly increasing the training period and the aliasing with other branches. For a branch uncorrelated with prior branches, the entire 15 bits are noise, only serving to confuse the predictor and pollute the tables.

In our simulations of select SPEC benchmarks, we found that the average branch, using a 16-bit GHR, observes over 100 different branch histories; with 20 bits, it more than doubles and we see over 200 histories per branch. Some of those histories will be useful and indicate useful correlations, but most will not [5]. Thus, it is reasonable to surmise that on average we are using dozens of times more table entries than desired.

Figure 1 shows the percentage of branches that achieve either loss or benefit from correlation history. It compares the accuracy of a correlating predictor (a unique 2-bit predictor for every possible history) with the actual bias of the branch. For example, if the correlating predictor achieved 78% accuracy, but the branch was 90% biased (toward taken or not-taken), we would say this branch was negatively impacted by history. The remaining branches not indicated on this graph had less than a 1% difference between the two.

In this graph, we see (1) that in all cases there are a number of branches that degrade because of branch history, and (2) that for larger histories, the number of degraded branches can be quite large. The effect we are observing is that with large histories, the noise begins to dominate. This leads to the somewhat counter-intuitive result that the longer the history, the fewer branches actually make effective use of that history.

Clearly, some branches do gain from large history, as prior work has shown. What we want, then, is an architecture that allows some branches to benefit from large histories, but eliminates or reduces the history noise in those regions where the noise is not useful. While this prior research examined a number of techniques that target the optimal history for individual branches, this paper focuses on a surprisingly effective (yet simple) heuristic that only requires that we identify a place in the execution stream where we transition from one region (with potentially high correlation) to another, but where there is not correlation between the regions. It turns out that this captures the most significant correlation gaps.

Not surprisingly, the control flow instructions themselves are the most useful clues to these transition regions. The

```
static void sched_analyze_2 (x, insn)
     rtx x;
     rtx insn;
{
  register int i;
  register int j;
  register enum rtx_code code;
  register char *fmt;

  if (x == 0)             // Branch A
    return;

  ...
}
```

Figure 2: Example function in sched.c from gcc.

transitions we found to be most interesting are the transitions between procedures (indicated by call and return instructions), and loop exits (often indicated by not taken backwards branches). Upon identifying one of these transitions, we do better if we ignore all prior history (thus collapsing all possible paths into one). By setting the GHR to a single known value, we begin training the predictor for the subsequent branches quickly, and greatly reduce pollution and aliasing.

2.1 Source Code Example

Gcc, a SPEC2000 integer benchmark, benefits from our techniques. An example from that benchmark which serves to illustrate the advantage of our technique appears in Figure 2. Branch A is the first branch instruction in a function call which performs a null pointer check. The branch executes 2455 times during the execution of the 100M instruction simpoint and is never taken. The branch encounters 208 unique 16-bit histories. As a result, it is only predicted correctly 91% of the time using non-aliasing 2-bit predictors per history. If you were to set the ghr to a fixed value when the function call is made, the branch would be predicted correctly 99% of the time, again assuming no aliasing. In this particular case the Filter predictor would also solve this problem, but in the more general case (e.g., if Branch A were not completely biased, but did follow a pattern), it would not.

3. RELATED WORK

The Global History Register, first proposed by Yeh and Patt [27], is a special case of their two-level adaptive branch predictor, with per-branch pattern history being collapsed into one global history. The benefits of the GHR were further demonstrated by McFarling [13] with his *gselect* and *gshare* predictors. Branch prediction research has continued to use global history for branch prediction accuracy.

One trend has been to exploit increasingly long global histories. Longer histories aid in establishing correlation with more distant branches as well as (in some cases) reducing aliasing in indexed predictors. The Alpha 21264 predictor [19] uses 12 bits of global history to index its global history and choice tables. The 2Bc-gskew predictor [20] uses 21 bits of global history to index its three prediction tables. The perceptron predictor [8] uses 34 bits of global history to train a simple neural network. More recent branch predictors use even longer histories to improve performance. The O-GEHL predictor [17] exploits history lengths ranging from 100-200 bits. The PPM predictor [2] detects closest matching patterns in very long history lengths. The L-Tage [18] predictor uses history lengths varying from 0 to 640 bits. However, the complexity of some of these predictors have deterred their adoption in modern processors [12]. L-Tage is of particular interest to our work in that its ability to dynamically use different history lengths likely benefits from the phenomenon we identify.

Another trend leverages the property of branch bias. The bimodal [22], bi-mode [10], YAGS [4], and Filter [1] predictors dynamically exploit the bias of a large percentage of branches. The bi-mode predictor separates branches, predicting those with a taken bias using a different table than those with a not-taken bias. The YAGS predictor maintains a similar table of biases indexed by the pc and two gshare tagged caches which store the branches whose behavior is contrary to the bias. The Filter predictor [1] uses the BTB to identify highly biased branches. Those branches are then excluded from the dynamic gshare prediction thus reducing the amount of aliasing. One benefit of our predictor is that, like these techniques, it allows the prediction of some biased branches with minimal resource utilization; however, this is only part of the benefit, as evidenced by the fact that we improve even the performance of the Filter predictor, which has already eliminated the biased branches from the tables.

The seminal work of Pan, et al. [14] investigates the benefits of global branch correlation. Evers, et al. [5] continue the investigation into branch correlation and recognize that while many branches are highly correlated with a small number of prior branches, some are not. This phenomenon - exploited by others in the work discussed in this section - is critical to this work. Additionally, the work of Thomas, et al. [25] similarly identifies branches which are correlated and those which are not. Their technique removes non-correlated branches on a branch-by-branch basis. Recent work by Sazeides, et al. [16] demonstrates that selecting the subset of history that is most highly correlated with a given branch can improve predictor accuracy.

Gao and Sair [6] target function entry and return as a point where correlation may diminish. A similar approach is taken in "Path-Based Next Trace Prediction" by Jacobson, et al. [7] of discarding some of the irrelevant history from within a subroutine and after a subroutine return by using a Return History Stack. Our work similarly addresses entry and return but does so with a different mechanism. Their work attempts to save the GHR at entry and restore at return whereas our work does not require saving any copies of the GHR. The Frankenpredictor [11] also targets call and return by shifting in masks depending on the instruction opcode. Their work likely benefits from the phenomenon we identify in this work.

The notion of removing useless bits from history is not entirely novel. The perceptron predictor, by the nature of its neural network, attempts to do exactly that. "Dynamic history-length fitting" [9] directly tries to cut history to the desired length based on trial and error rather than using heuristics as we recommend. The Elastic History Buffer [24] and Variable Length Path Branch Prediction [23] both propose allowing branches to specify how much history will be

used. Our proposal differs from this technique in that it works entirely using simple heuristics and because the modifications made to the ghr in our technique are not unique to a single branch but rather affect all subsequent branches.

Choi, et al. [3] propose modifying the GHR during thread migration between cores on a CMP featuring speculative multithreading (to provide a useful GHR to begin thread execution). For some benchmarks, they find that inserting the program counter of the thread spawning instruction during thread creation can provide slightly better accuracy than providing an oracle-based correct GHR. This effect is likely related to the regions of limited branch correlation targeted by our work. Our work differs in that it improves branch prediction even for single-threaded execution.

4. RESETTING THE GHR

In this section, we discuss the particular hardware techniques we examine in this paper. In addition, we will discuss some potential software/hardware techniques based on ISA modification and profile analysis. However, the latter is primarily only interesting in that it motivates the much simpler hardware-only techniques.

The goal of this section and the next is to identify points in the program control flow with little correlation to prior branches. These Regions of Limited Branch Correlation (RLBCs) may benefit from artificial modifications to the GHR.

4.1 Hardware RLBC Identification

Existing control flow constructs provide hints for finding these RLBCs. Loops and function calls often represent breaks in control flow. The not-taken path following a backward branch often indicates a loop exit, and we typically expect branches following the loop to be less correlated with branches inside the loop. Similarly, branches in a function call may not be correlated with the branches preceding the call. Finally, branches following a return from a procedure may lack correlation with the branches in the procedure. When these regions are detected - by the execution of the applicable instruction - we can perform modifications to the GHR to improve accuracy.

As mentioned before, when entering an RLBC, the GHR contains noise. To eliminate this noise, we could zero the GHR but this may cause potential problems for index-based predictors. By zeroing the GHR, index-based predictors like gshare would bias training toward one particular region of the predictor. Therefore, in resetting the GHR we use the same technique as [3]. They generate a GHR from the program counter, in their case to manufacture a GHR when forking a speculative multithreading thread, for which the correct GHR is unknowable. Using the PC provides a unique history for each point at which we reset the GHR (eliminating aliasing between the different resetting points), and ensures that when we return to this code, we will reset to the same value.

For not-taken backward branches, the value inserted into the GHR is the PC of the backward branch. For function calls, the PC of the calling instruction is inserted into the GHR. Finally, for function returns, the PC of the return instruction is inserted into the GHR.

For the hardware-only techniques, we assume a very simple decision – we either always reset the GHR, or not. But in which cases we want to reset the GHR (e.g., on backwards branches and calls, but not returns) depends on the specific branch prediction hardware (specifically, which branch predictor and what size) we are modifying. For example, the higher the incidence of branch aliasing, the more aggressive we will want to be. For a wide variety of predictors, we will find the best combination of these three resetting points.

Modifications to the GHR will typically take place in the fetch pipeline stage, just like any other modification. For example, if a backwards branch is predicted not taken, we will modify the GHR as described, but checkpoint the old GHR as on any other speculative branch. If the branch is mispredicted, we restore the GHR. If the branch was originally predicted taken, we update the GHR as normal, and only apply our technique if the branch is resolved mispredicted. In either case, the GHR is modified in the same places as other branches.

Results for these approaches are discussed in Section 6.

4.2 Static Analysis of RLBC Points

We also examined profile-based identification of the best points to modify the GHR (and even allow more flexible modification, such as only clearing regions of the GHR). However, the static techniques are somewhat problematic for several reasons, and ultimately provided little gain over our hardware-only techniques.

The issues include (1) it requires ISA modification to indicate the modification point and possibly how the GHR should be modified, (2) it requires expensive profiling, (3) it requires software that knows the exact details of the branch predictor, and some manufacturers have been extremely protective of those details. However, these results are still interesting as a comparison with the hardware-only techniques. But we omit many details of the profile-driven analysis, since the results are primarily useful as a point of comparison.

We created a branch trace of each benchmark, and for that trace recorded the expected result (branch predicted correctly or incorrectly) assuming a history length of any given value below a certain maximum. Prediction accuracy assumed a correlating predictor for each branch with no aliasing. Accounting accurately for aliasing at this stage in the profiler would have made the subsequent steps prohibitively expensive.

We could identify a "good" place to dynamically reset the GHR, by identifying a place in the trace where the next branch predicted well with zero bits of history, the subsequent branch with 1 bit of history, the next with 2 bits, etc. By calculating all such places (a single possible location would be following a possible static branch, in either the taken or not-taken case – we could make different decisions for each), we select the best. Because the different resetting points will interact, we need to start the analysis anew after one is chosen to select the next. We continue the process until we reach a minimum threshold of marginal improvement. Interestingly, the optimal minimum threshold was actually a negative improvement. This is because the effect of aliasing makes the gains higher than the trace predicts – so we need to make it more aggressive than it would otherwise be.

Of the four types of conditional branches (forward taken and not-taken, backwards taken and not-taken), we quickly learned that the first three are almost always better left alone, and the last are usually best modified. Therefore, our analysis technique worked best if we just forced it to

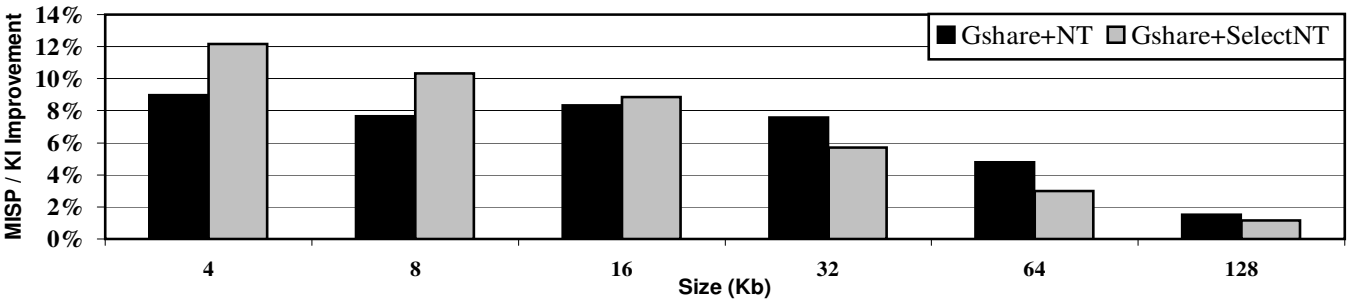

Figure 3: MISP/KI improvement from targeting all backward branches or those selected by profiling.

ignore all but backwards-NT, and just decide which of those to modify the GHR.

Figure 3 shows the best parameterized static result, compared to our simplest technique which blindly modifies the GHR at all backwards NT branches. These results are based on a gshare predictor using a maximum number of history bits executing select SPECint benchmarks. While we see that for small predictor sizes, the static analysis does indeed do a better job of selecting branches than the hardware-only technique, it does not seem to be enough of a gain to account for the concerns over this technique. But this does indicate there is room for future investigation in this direction.

5. METHODOLOGY

This section describes our simulation framework and benchmark selection. We modified a version of SMTSIM [26] to implement our array of branch predictors. SMTSIM executes unmodified Alpha ISA code and supports out-of-order SMT or CMP processors. In this study, SMTSIM was executing a single threaded binary on a single core.

We also modified the framework provided by the Championship Branch Prediction Competition to execute the array of branch predictors via trace-based simulation.

The SPEC results, then, come from detailed simulation of the predictors running on a modern core, and includes, for example, the effects of delayed updates to the predictor. The details of the simulated core are not particularly important, however, as we only produce branch mispredict rates in this paper. This is because the CBP results only allow trace-based simulation of mispredict rates and no direct performance results.

5.1 Branch Predictors

Gshare [13] is a standard implementation with varying size prediction tables. Filter [1] is implemented using a three bit saturating counter and one bias bit per BTB entry. After eight sequential executions with the same outcome, the counter becomes saturated. Predictions for branches with a BTB saturated counter are given as the bias. If the bias prediction is correct, no update is performed. All other branches are predicted using a gshare predictor. The BTB assumed has 512 entries with 4 way associativity. The additional BTB hardware required by filter is not included in the hardware budget for the filter predictor. Our default size for gshare and filter is 32Kb.

	BIM	G0	G1	META
prediction table	16K	64K	64K	64K
history length	4	13	21	15

Table 1: Characteristics of 2Bc-gskew predictor

The Alloyed (Global/Local) Perceptron is a 32Kb implementation [8]. The Alloyed Perceptron has a 91 entry table and uses 34 bits of global and 10 bits of local history.

Our 2Bc-gskew predictor [20] is a 416Kb implementation with four prediction tables. Three of these tables are indexed with the GHR. The fourth is a meta predictor which chooses between the results produced by the 2-gskew predictor (two of the GHR indexed tables) and the bimodal predictor. We provided each entry with its own hysteresis bit — we did not simulate the space optimization of shared hysteresis bits. More details of the 2Bc-gskew predictor are given in Table 1. We did not attempt to create a reduced version of this predictor (similar in size to our other predictors). That predictor was carefully tuned in [20] for this size; additionally, this allows us to demonstrate that our technique is effective even on a very large branch predictor.

Our implementation of the Alpha 21264 predictor [19] is 29Kb with a 4k entry choice prediction table, 4k entry global prediction table, a 1k entry local history table with 10 bits of history per entry, and a 1k entry local prediction table.

BTB misses are faithfully modeled and we assume branch instructions which miss in the BTB are repredicted when identified by decoding. Similar to [15] we use the number of conditional branch mispredictions per thousand instructions executed (MISP/KI) as our primary metric.

5.2 History Tuning

For each of our predictors, we evaluated all possible history lengths to ensure the strongest baselines performance. To determine the optimal history length we averaged the average MISP/KI for SPECint, SPECfp, and CBP. Most predictors benefited from the maximum available history lengths. However, some predictors, especially our smallest sizes of gshare and filter, benefited from low amounts of history. In fact - the 4Kb implementation of gshare uses only 2 bits of history.

5.3 Benchmark Selection

We choose eight benchmarks from the SPEC2000 suite. We intentionally select eight programs that are sensitive

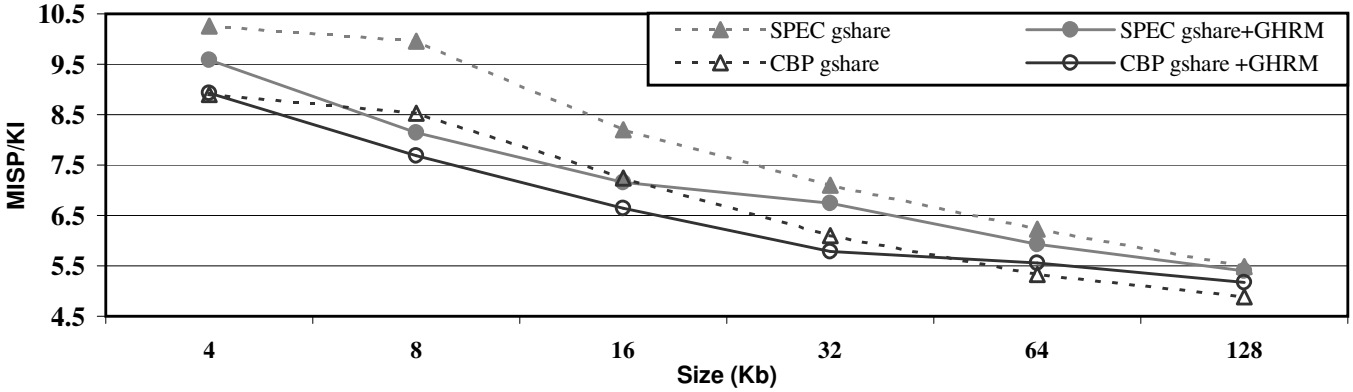

Figure 4: MISP/KI for different size gshare predictors for select SPEC2K benchmarks and CBP traces.

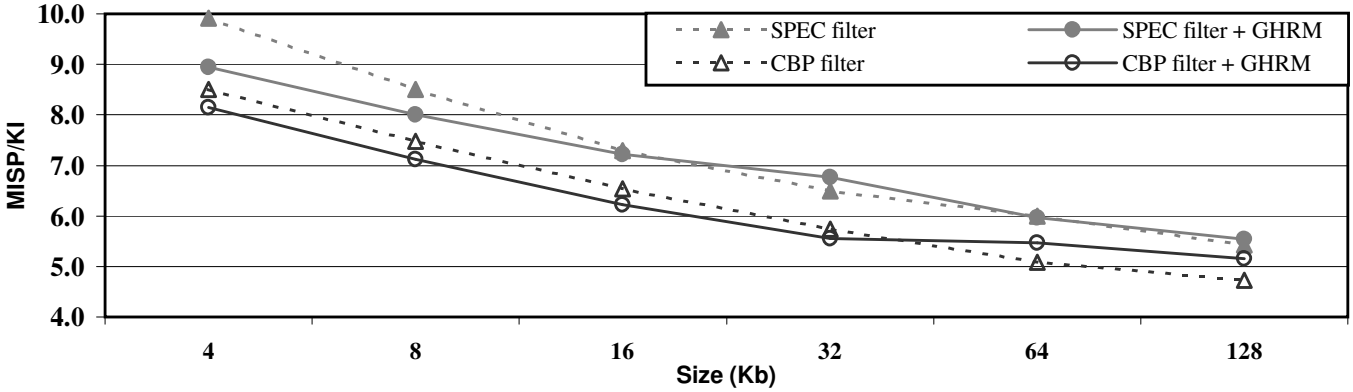

Figure 5: MISP/KI for different size filter predictors for select SPEC2K benchmarks and CBP traces.

to the (overall) branch prediction accuracy in our simulation framework. We do this by filtering out those programs whose performance improved by less than 3% when a perfect branch predictor was introduced. We simulate 100 million instructions starting at a single execution Simpoint [21].

We also use the traces provided for the Championship Branch Prediction Competition.

6. RESULTS

The techniques detailed in previous sections for GHR modification (GHRM) will likely have different impacts for different branch predictor sizes. For realistic predictor sizes, we compare the accuracy of GHRM when applied to simple predictors against the accuracy of more advanced and larger predictors. Lastly, we evaluate the effectiveness of GHRM when applied to these more advanced predictors.

6.1 Predictor Size

In general, the benefits of these techniques are most substantial when working with smaller predictors. This is because aliasing is most extensive in smaller predictors, and reducing aliasing is the most important result of our modifications. However, we also show that these approaches are still applicable to larger predictors. In particular, the 2Bcgskew predictor is discussed in section 6.3.

We start by examining the gshare and Filter predictors initially. We do this because these are simple, highly effective predictors, and they are easily parameterized by size. The results for varying gshare and Filter sizes are contained in Figure 4 and Figure 5 respectively. The MISP/KI for each baseline predictor is provided for both the select SPEC2K benchmarks and the CBP traces.

The percentage reductions in MISP/KI are provided in Table 2. The results are highest for the small predictors, and actually go negative when the tables get large. There are still gains for individual branches, but this is where our simpler hardware-only technique breaks down because it cannot distinguish between instances – in this region, approaches like our static technique may become more attractive.

Our technique (GHRM) uses the best configuration of heuristics (which specific points to modify the GHR on) for each predictor type and size. The actual configurations used are contained in Table 3. We see in this table that as predictors get larger, and the effect of aliasing is reduced, we tend to get less aggressive; for example, resetting the GHR on returns tends to only be beneficial with the smallest predictors.

It should be noted that the negative results are primarily the effects of the variety of compilation systems. For example, if we could optimize for SPEC alone, we get positive

	Size (Kb)					
Predictor	4	8	16	32	64	128
SPEC gshare+GHRM	6.5%	18.2%	12.7%	5.0%	4.8 %	1.5%
CBP gshare+GHRM	-0.5%	9.9%	8.1%	5.3%	-4.2%	-5.7%
SPEC filter+GHRM	9.8%	6.0%	1.1%	-4.3%	0.6%	-1.9%
CBP filter+GHRM	4.1%	4.9%	4.7%	3.1%	-7.4%	-8.8%

Table 2: MISP/KI reductions per size for gshare+GHRM when compared against standard gshare and filter+GHRM compared against standard filter for both the select SPEC2K benchmarks and CBP traces.

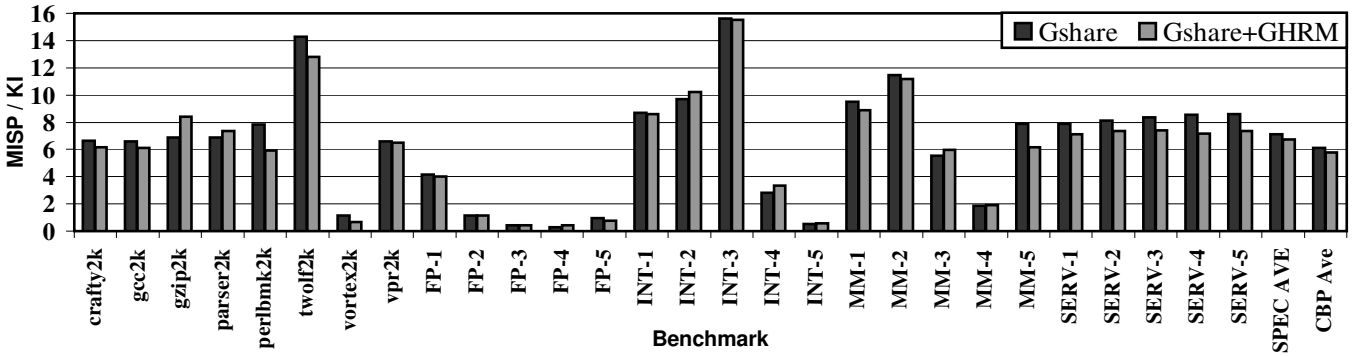

Figure 6: Baseline and improved gshare for each benchmark. (Lower bars indicate better accuracy.)

	Gshare			Filter		
Size (Kb)	FC	RTR	NT-BB	FC	RTR	NT-BB
4	T	T	T	T	F	T
8	T	F	T	T	F	F
16	T	F	T	T	F	F
32	T	F	F	T	F	F
64	F	F	T	F	F	T
128	F	F	T	F	F	T
256	F	F	T	F	F	T

Table 3: Gshare and filter heuristic configurations: either function calls (FC), return (RTR) instructions, and/or not-taken backward branches (NT-BB) can trigger GHR modifications.

results for this entire range. Similarly for the CBP results, which were generated differently than our SPEC binaries. But trying to find a single best configuration for both resulted in significantly lowered results in many cases. This implies that if the compilation systems are universally aware of what the hardware is doing, and at least does not generate code that is at cross purposes, the potential gain from these techniques can be much higher than shown here. It may also indicate that a simple dynamic technique that chose dynamically (but at a coarse granularity) which of our eight configuration combinations was most effective could also provide good results.

From Figure 4, we can see that for some of the smaller sizes of gshare, gshare+GHRM enables branch prediction accuracies similar to predictors of twice the size. Filter (Figure 5), because of its ability to filter out highly biased branches, benefits less from these techniques but still shows noticeable improvements at smaller sizes. These results validate the assertion that in some cases, our simple and nearly cost-free branch prediction modification can provide the same performance with a significant decrease in predictor size.

Figure 6 provides the MISP/KI for each of the selected benchmarks. For the select SPEC2K benchmarks, almost all benchmarks benefit from applying GHRM. For the CBP traces, applying GHRM has a positive benefit on the majority of traces, most notably the server traces.

6.2 Heuristic Configuration

In the previous section, different heuristics were said to be effective for the same predictors given different benchmark sets. The most notable difference is that the CBP traces benefit more from function call modification than the select SPEC2K benchmarks. This likely comes primarily from the difference in code generation.

Although one configuration may offer the best performance for each predictor of a given size, often other heuristics offer competitive performance. Figure 7 provides the MISP/KI improvement for both working sets given each configuration using different sizes of gshare. Different sizes of gshare are provided up to 64Kb where the utility of the configurations drops (as shown in Figure 4). This figure shows us that while some configurations may provide the best performance, many still provide reasonable benefits.

6.3 Other Predictors

The primary goal of these techniques is to aid simpler branch predictors. However, these techniques can benefit other predictors as well. Even in the absence of significant aliasing, this mechanism allows faster training (when the branch working set changes) and retraining (when branch

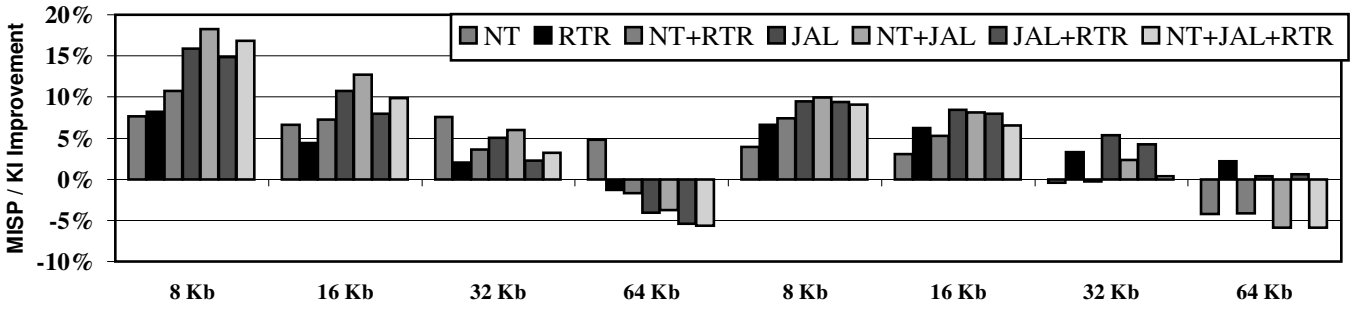

Figure 7: MISP/KI improvement for different heuristic configurations for varying sizes of gshare.

	Configuration		
Predictor	FC	RTR	NT-BB
2Bc-gskew	T	F	F
A21264	T	F	T
Filter	T	F	F
Gshare	T	F	F
Alloyed Perceptron	T	F	T

Table 4: The best heuristic configuration for each predictor.

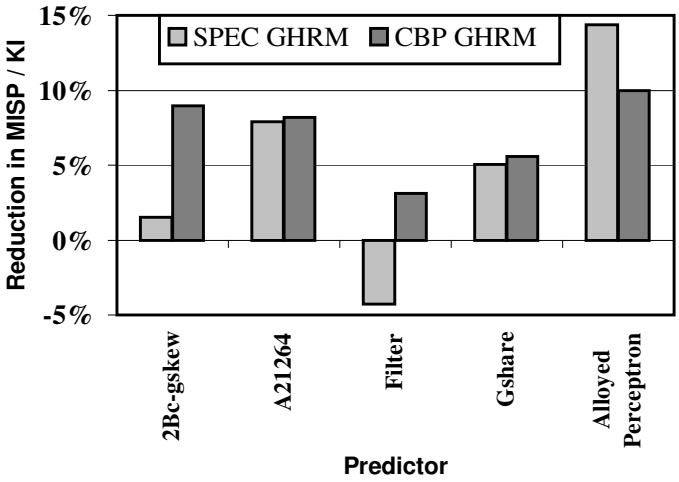

Figure 8: Reduction in MISP/KI for various predictors. Gshare and filter are 32Kb implementations.

bias or patterns change). Figure 8 provides the percentage reduction in MISP/KI for a larger set of predictors, some but not all of similar size. The GHRM configuration selected for each of these predictors is provided in Table 4.

All predictors benefit from these techniques. The Alloyed Perceptron has the largest MISP/KI reduction, 14.4% for the select SPEC2K benchmarks and 10.0% for the CBP traces. The Alloyed Perceptron predictor results are particularly interesting. Perceptron predictors are specifically targeted at reducing the impact of non-correlated branches, and are largely successful at doing so. But it removes the effect of the noise only probabilistically, and therefore some of the noise always remains. We find that even in the context of that predictor, we are able to further remove the impact of useless noise.

The other really interesting results are for the 2Bc-gskew predictor. This result is in contrast to some of our earlier results that might indicate that this technique is only useful for small predictors. This predictor, with multiple large tables, and significant features to tolerate aliasing, still takes significant advantage of our ability to identify non-correlated regions of code.

6.4 Non-Select Benchmarks

The prior results are shown for the select SPEC2000 benchmarks. When averaging the benefit of our technique across all, not just select, SPEC2000 we found that 8Kb, 16Kb, 32Kb implementations of gshare achieved 9%, 6%, and 1% MISP/KI improvements. Our techniques were not beneficial for larger sizes of gshare. All the other predictors except 2Bc-gskew and 16Kb or greater implementations of filter also saw a reduction in MISP/KI.

SPECfp is not included in the select benchmarks and has some interesting characteristics. The IPC performance of these benchmarks was not found to be highly tied to the branch prediction accuracy. In addition, nearly all our branch predictors were able to predict SPECfp with high accuracy. In fact - the average MISP/KI for SPECfp is less than 1 for our 32Kb gshare implementation.

One particularly interesting result relates to a methodology common in recent branch prediction research. Following this precedent, as described in Section 5, we tuned all of our baseline predictors to find the optimal amount of GHR history to use. This methodology, though, has a tendency to overtune the predictor for a small set of benchmarks (where a real predictor would be tuned for a much larger set). In fact, this methodology significantly reduced the magnitude of our overall gains, which were always higher when our predictor and the baseline used the same amount of history. In the case of the smallest predictor (4 Kb gshare), this overtuning becomes very apparent – although we used all of SPEC to find the optimal length, the greater importance of branch prediction in the integer benchmarks created an optimal history length (2 bits!) which was an extremely poor choice for the FP benchmarks. So when applying our techniques, we see tremendous improvements in those benchmarks – a 47% reduction in average MISP/KI for SPECfp.

But we did not choose to highlight these results in this paper, because we feel they are more an artifact of the standard methodology.

But this does highlight an important advantage of our branch predictor – it enables the use of longer histories by eliminating the artifacts that create the pressure to use shorter history than that dictated by the size of the branch history tables. For these small predictors, the best tuned predictor using our optimizations consistently used more history length than a tuned predictor without our optimizations. If we had not followed this methodology, and instead assumed all predictors use the expected amount of history, our reported results would be higher. For example, again for the smallest gshare, our techniques provide a 21% reduction in mispredicts for SPEC-select and 10% reduction for CBP (as opposed to the 6.5% and -0.5%, respectively, reported in Table 2).

6.5 Summary

These results demonstrate that simple heuristics can be used for improved branch prediction accuracy at little cost by reducing the amount of noise in the global path history. For most predictors, modifying the GHR when a backward branch is not taken or when encountering a function call provides a reasonable reduction in MISP/KI.

In general, our results show that the technique of modifying the GHR is most useful for smaller predictors. However, the techniques are not limited to simple predictors and are shown to eliminate a significant percentage of MISP/KI in more complicated predictors.

7. CONCLUSION

This paper demonstrates that artificially modifying the GHR before regions of limited branch correlation (RLBC) can improve branch predictor performance during single threaded execution.

By performing GHR modifications based on program heuristics - when backward branches are not taken and when encountering function call and return instructions - improved branch predictor accuracy can be achieved. For 32Kb predictors, our techniques offer up to a 12% overall decrease in MISP/KI. For small gshare predictors, these techniques can provide as much as a 14% reduction in MISP/KI. All predictors examined benefit from these techniques and only a minor hardware modification is required for implementation.

These techniques enable processors to achieve higher branch prediction accuracy, increased performance, and reduced power consumption with simple branch predictors. Simple branch predictors have the advantages of easier design and verification as well as lower hardware cost and lower power consumption.

8. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their helpful insights. We would also like to thank Gabriel Loh for his help in improving the quality of this paper. This research was supported in part by the Reserve Officers Association Henry J. Reilly Memorial Scholarship, NSF Grant CCF-0702349, and Semiconductor Research Corporation Grant 2005-HJ-1313.

9. REFERENCES

[1] P.-Y. Chang, M. Evers, and Y. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *International Conference on Parallel Architectures and Compilation Techniques*, Oct 1996.

[2] I.-C. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[3] B. Choi, L. Porter, and D. M. Tullsen. Accurate branch prediction for short threads. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[4] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *31st Annual ACM/IEEE International Symposium on Microarchitecture*, Nov. 1998.

[5] M. Evers, S. J. Patel, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *In Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[6] F. Gao and S. Sair. Exploiting intra-function correlation with the global history stack. In *5th International Conference on Systems, Architectures, Modeling, and Simulation*, 2005.

[7] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. In *30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[8] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. In *ACM Transactions on Computer Systems*, Feb. 2002.

[9] T. Juan, S. Sanjeevan, and J. J. Navarro. Dynamic history-length fitting: a third level of adaptivity for branch prediction. In *25th Annual International Symposium on Computer Architecture*, 1998.

[10] C.-C. Lee, I.-C. Chen, and T. Mudge. The bi-mode branch predictora. In *30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[11] G. Loh. The frankenpredictor. In *The 1st JILP Championship Branch Competition (CBP-1)*, 2004.

[12] G. H. Loh. A simple divide-and-conquer approach for neural-class branch prediction. In *14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[13] S. McFarling. Combining branch predictors. In *DEC WRL Technical Note TN-36*. DEC Western Research Laboratory, 1993.

[14] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *SIGPLAN Not. Vol. 27 No. 9*, 1992.

[15] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Sixth International Symposium on High-Performance Computer Architecture*, 2000.

[16] Y. Sazeides, A. Moustakas, K. Constantinides, and M. Kleanthous. The significance of affectors and affectees correlations for branch prediction. In *3rd*

International Conference on High Performance and Embedded Architectures and Compilers, 2008.

[17] A. Seznec. Analysis of the o-geometric history length branch predictor. In *32nd Annual International Symposium on Computer Architecture*, 2005.

[18] A. Seznec. The l-tage branch predictor. In *Journal of Instruction-Level Parallelism, vol. 9*, May 2007.

[19] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, June 2002.

[20] A. Seznec and P. Michaud. De-aliashed hybrid branch predictors. In *Technical Report RR-3618, Inria*, Feb. 1999.

[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[22] J. E. Smith. A study of branch prediction strategies. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[23] J. Stark, M. Evers, and Y. N. Patt. Variable length path branch prediction. *SIGPLAN Not.*, 33(11), 1998.

[24] M.-D. Tarlescu, K. B. Theobald, and G. R. Gao. Elastic history buffer: A low-cost method to improve branch prediction accuracy. In *International Conference on Computer Design*, 1997.

[25] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark. Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In *30th Annual International Symposium on Computer Architecture*, 2003.

[26] D. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *In 22nd Annual Computer Measurement Group Conference*, December 1996.

[27] 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*, May 1993.