The Triton Branch Predictor

Josh Karlin Darko Stefanovic Stephanie Forrest

Department of Computer Science
University of New Mexico
{karlinjf,darko,forrest}@cs.unm.edu

October 22, 2004

Abstract. *We describe a new branch predictor that is designed to balance multiple constraints—predicting branch biases versus predicting specific branch instance behavior. Most branch instances only require branch bias information for accurate predictions while a select few require more sophisticated prediction structures. Our predictor uses a cache mechanism to classify branches and dynamically adjust the balance of the predictor. On average, our predictor mispredicts 24% less often than YAGS and 19% less often than a global perceptron predictor with the same bit budget.*

1 Introduction

Effective and efficient branch prediction remains an important component in the design of modern microprocesors, essential to the goal of exploiting instruction-level parallelism. The hardware budget for branch prediction, however, is relatively limited. Many recently proposed predictors [4–7] focus on reducing destructive aliasing in the predictor's data structures (which arise in large applications with many branch instructions in the working set), while others [1,2] attempt to create entirely new prediction mechanisms (in order to capture the behavior of individual branches that are difficult to predict with simple mechanisms). In general-purpose microprocessors, a balance must be maintained between the two strategies, as applications favorable to each will arise.

We propose a new hybrid predictor, named *Triton*, in which we have carefully balanced the allocation of the bit budget between predicting branch biases

and predicting specific branch instance behavior. Like most modern predictors, the Triton predictor is built upon the results of previous work. Our design was inspired by the Yet Another Global Scheme (YAGS) cache structure. One of our components is a "hard"-branch predictor. Any good predictor can play this role, but we have found that a global perceptron works the best.

2 Related Work

The YAGS predictor is intended to reduce destructive aliasing [4]. Its structure is very similar to the popular bi-mode predictor [5], but the direction pattern history tables (PHTs) are replaced with direction caches. Instead of always using the direction PHT for the prediction, as in the bi-mode predictor, the YAGS predictor first looks for a branch bias in the choice PHT and then checks the appropriate direction cache for an entry. If an entry exists in the cache, then the cache's prediction will override the choice PHT's. An entry is inserted into the cache if the choice PHT's bias was wrong. The cache is indexed by the *xor* of the program counter and the global history register. Therefore, each entry in the cache corresponds to a branch instance (PC *xor* history) that disagreed with the branch bias at the time.

We have found this cache structure to be advantageous for resource-constrained predictors because, unlike bi-mode predictors, only a handful of important branch instances are kept in the cache. By not allowing biased branch instances to enter the cache, the cache is therefore less polluted than the bi-mode PHT.

The Triton predictor uses the cache mechanism found in the YAGS predictor, but makes two extensions. The bias field of the cache is used as a test of branch stability, and if it fails the branch instance is known to be difficult and passed off to a better predictor. We have also merged the taken and not taken caches found in the YAGS predictor into one; this improves performance because it is often the case that the majority of a program's branches are biased in the same direction.

The Perceptron branch predictor is based on neural networks. Its advantage over other prediction mechanisms such as *gshare* [3] is that it requires significantly less space to predict over long histories. An individual perceptron assigns a weight (an 8-bit integer suffices) to each position in the history to indicate correlation. If there is a strong positive correlation between the history position and the outcome of the branch's prediction, then the weight will be a large positive integer. Likewise, if there is a negative correlation the weight will be negative.

Since each perceptron is large (8 $\times$ history length), there can be only a few hundred perceptrons in a straight perceptron predictor with a 64Kb resource budget and history length at least 16; thus, many branches are mapped to each perceptron and the perceptrons become polluted. Our predictor alleviates this aliasing problem by applying its perceptron component only to the subset of branch instances which are not heavily biased.

3 Triton

Our predictor comprises three major components. The first is the bias table (1/4 of budget). It keeps track of each branch address' bias. The second component is the cache (1/2 of budget). Each branch instance that disagrees with its prior bias is inserted into the cache. The third component is the perceptron predictor (1/4 of budget).

The Bias Table. Our predictor uses a table of 8192 bi-modal counters to maintain each branch address' bias. It is indexed by branch PC, incremented if the instance is taken, and decremented otherwise. This takes up 16Kb of state.

The Cache. Each cache entry holds a bi-modal counter and a 6-bit address field. The cache is indexed exactly as a *gshare* PHT: The branch PC *xor*ed with the global history. An instance is inserted into the cache by writing the low 6 bits of the branch PC into the address field and initializing the bi-modal counter. The counter is initialized to 0 if the branch was not taken, otherwise it is initialized to 3. To check if a branch instance exists in the cache (a cache hit), the index is computed and then the low order bits of the branch PC and the cache entry are compared. The cache holds 4096 entries and uses 32Kb of state.

The Perceptron. We use a 16Kb perceptron predictor with a global history size of 64. This allows for 32 entries. The predictor is exactly as described in the literature [1].[1]

[1] Any other predictor can trivially be swapped in. The fast path-based neural predictor [2] may be a candidate, but could not be evaluated in time for this submission.

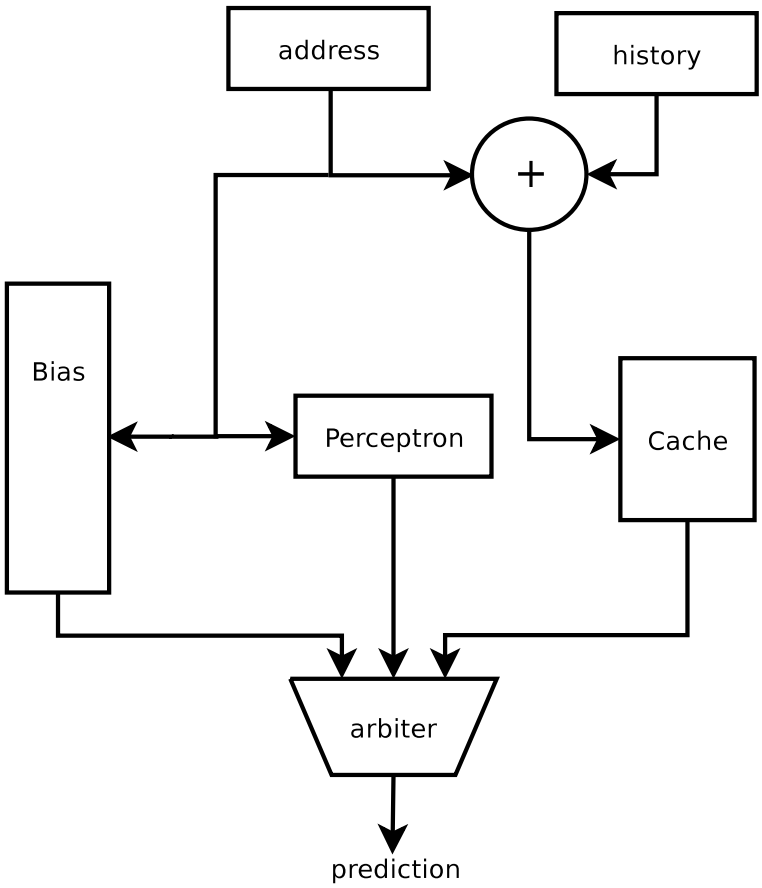

Figure 1: Triton Prediction

3.1 Prediction

The prediction stage is a simple parallel operation with priorities assigned to each component. Each component is first queried for its prediction. The bias table responds with its bi-modal prediction. The cache has a 3-bit response: one bit signifies a cache hit or miss, and the other two hold the index's bi-modal counter value. The perceptron responds with a 1-bit prediction.[2]

Once the components have been queried, arbitrator logic chooses the appropriate result. The bias table has least priority, and thus if the branch instance exists in the cache the cache bi-modal counter takes precedence. If the cache's counter value is 0 or 3, it is strongly biased and that will be used for the final prediction. However, if the cache's counter value is 1 or 2, the instance's bias is weak and the instance is therefore considered very hard to predict. Only in this case is the perceptron's prediction used. Note that once a cache entry's counter reaches 1 or 2 it will not change until the entry is evicted.

3.2 Update

The Triton update step is simple. Whichever component of the predictor was used in the prediction is updated. The bias table and cache simply update their bi-modal predictor with the result of the outcome, just as *gshare* does. The perceptron update is also no different from its original description.

Once the tables have been updated the cache may need to be updated. Since the cache is used to signal a branch instance that does not follow its bias, an instance is inserted into the cache if the bias table was used for the last prediction and it predicted incorrectly. The insertion changes the address field at the cache index to the bottom 6 bits of the branch's PC and sets its counter to 3 if the branch was taken and 0 otherwise.

The final step is to update the histories. The global *gshare*-like history is updated just as in *gshare*. The perceptron history is also shifted at each update, as described in [1].[3]

[2] For simulation efficiency, the provided code calls the perceptron predictor only when its result is actually needed.

[3] The longer perceptron history obviously subsumes the *gshare*-like history. However, for the sake of keeping the "hard" branch predictor pluggable, we maintain both.

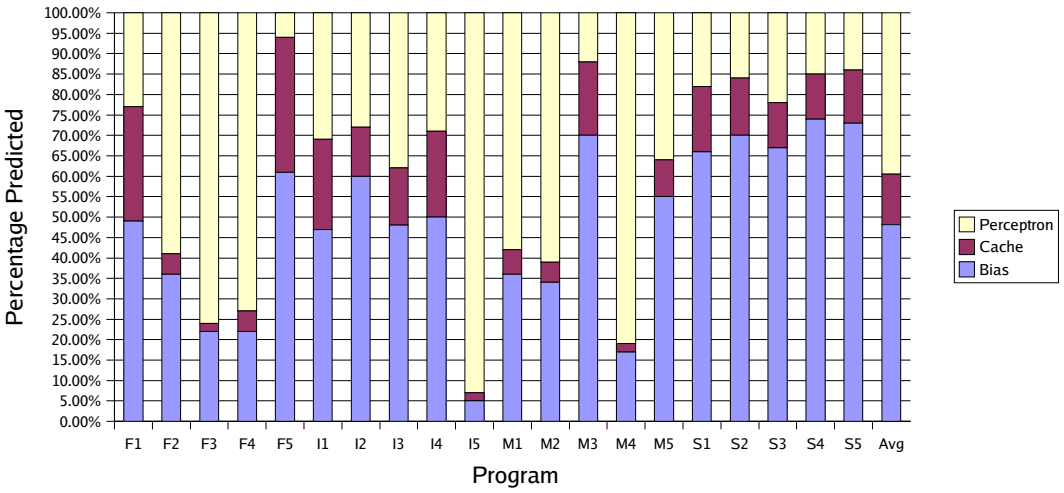

Figure 2: Component Usage

4 Results

We ran our predictor against 20 sample program execution traces containing only branch instruction addresses and their outcomes. The programs are split into categories (five each) of Floating Point, Integer, Multimedia, and Server, but are otherwise unknown to us. The language, compiler, and even architecture are also unknown to us. We show the overall performance (using the misprediction rate metric) of our predictor and its predecessors, each tuned optimally for a 64Kb budget. and we analyze the usage of the Triton predictor's components.

It can be seen in Figure 3 that the Triton predictor outperforms the other predictors by a wide margin. In fact, the Triton predictor has on average 31% fewer mispredictions than *gshare*, 24% fewer than YAGS, and 19% fewer than the global perceptron predictor, all of which are well tuned to a 64Kb constraint. Using the contests metric of average number of mispredicted branches per 1000 instructions, our predictor averages 3.592, which is 32% better than *gshare*, 23% better than YAGS, and 20% better than the global perceptron.

Of particular interest are three observations. First, our predictor outperforms the YAGS predictor in the server benchmarks by 14%. Second, Triton outperforms the global perceptron in all cases but two, and when the server applications are not considered, our predictor still bests it by 13%. Finally, owing to the perceptron's long (64 branches) history in our predictor, we are able to predict the M3 program

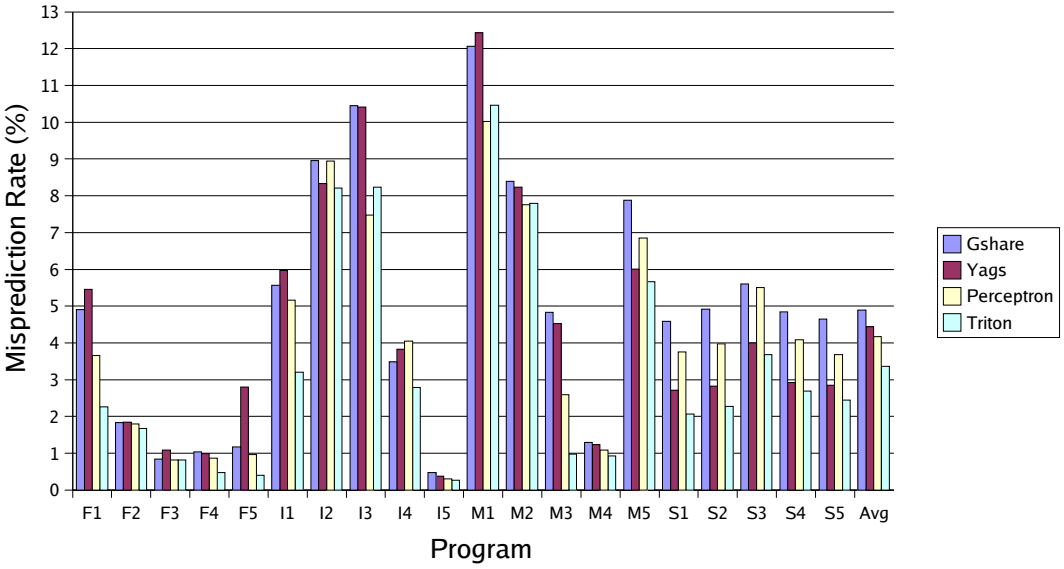

Figure 3: Predictor Comparison

nearly as well as a small local branch history predictor can.

Figure 2 breaks down the predictions by component that produced them. Here we can see that the server applications consistently use the bias table the majority of the time. This makes sense when Figure 4 is also considered. Figure 4 shows the number of branch addresses per program that are frequently accessed ($>$ 200 times) and that also disobey their bias at least 20% of the time. Since the server programs have so many frequent and difficult branches to deal with, they quickly get evicted from the cache. Therefore only those branch instances that are really frequent and are consistently resident in the cache will use the perceptron for their predictions. This shows that under high aliasing pressure, our predictor backs off from the highly trafficked cache and uses the bias table—which is better for high degrees of aliasing. Yet many of the most frequent and difficult branches will remain in the perceptron predictor.

We offer the following observations about our predictor's performance improvement over the global perceptron. In all cases save F4 where the Triton predictor beats the perceptron by at least 30%, the Triton predictor uses the bi-modal predictors at least 65% of the time. Likewise, in all cases save I2 where the Triton preditor uses the bi-modal predictors at least 65% of the time, the Triton predictor bested the perceptron by at least 30%. This is evidence of our predictor's ability to

distinguish easily predicted from difficult branches and to delegate them correctly.

Our last observation regards the tradeoff between keeping track of local and global history. In a previous predictor of ours, we were able to predict the M3 at a 0.64% mispredict rate using a 12-bit local history scheme. Notice that other predictors such as the 24-bit history global perceptron can at best achieve 2.59%. It is therefore conceivable that having a 64-bit global history is nearly as useful as keeping a 12-bit local history in some, if not many cases. This hypothesis requires further study.

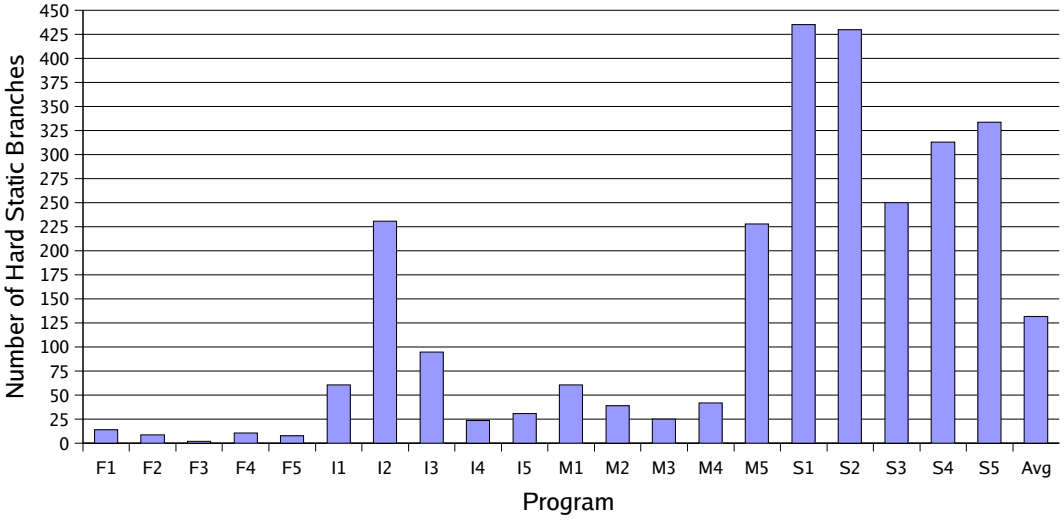

Figure 4: Hard Branches

5 Conclusions

The Triton predictor succesfully allocates its resources where they are needed. The bias table holds each branch's bias and is used to predict highly biased branches and branches that are seldom frequented and are hence evicted from the cache. The cache/perceptron combination handles the difficult (and generally frequent) branch instances. This balance is so effective that the Triton predictor mispredicts 19% less often than the global perceptron and 14% less than YAGS on benchmarks with large static branch signatures.

Acknowledgments. The authors gratefully acknowledge the partial support of the National Science Foundation (grants ANIR-9986555i, CCR-0219587, CCR-0085792, EIA-0218262, EIA-0238027, and EIA-0324845), Hewlett-Packard gift 88425.1, and Microsoft Research. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

References

[1] D.A. Jimenez and C. Lin, Neural Methods for Dynamic Branch Prediction, *TOCS* **20** (4), Nov. 2002.

[2] D.A. Jimenez, Fast Path-Based Neural Branch Prediction, *Proc. MICRO* 36, Dec. 2003.

[3] S. McFarling. Combining Branch Predictors. *Technical Report* 36, Digital Western Research Laboratory, June 1993.

[4] A. Eden and T. Mudge. The YAGS Branch Prediction Scheme. *Proc. MICRO* 31. Nov. 1998.

[5] C.-C. Lee, I.-C. Chen, and T. Mudge. The Bi-Mode Branch Predictor. *Proc. MICRO* 30, Dec. 1997.

[6] P. Michaud, A. Seznec, and R. Uhlig. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. *Proc ISCA* 24, May. 1997.

[7] E. Sprangle, R. Chappell, M. Alsup, and Y. Patt, The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference. *Proc. ISCA* 24, May. 1997.