Re-Defining the Tournament Predictor for Embedded Systems.

Yuval Peress peress@cs.fsu.edu
Dr. Gary Tyson tyson@cs.fsu.edu
Dr. David Whalley whalley@cs.fsu.edu

Abstract

Looking at trends in recent publications on computer architecture we find that there has been a steep decline in the number of published works relating to branch prediction. Further inspection reveals that while older trends include development of new predictor designs [12][15][8][10], more modern trends have been favoring the consumption of larger history and storage to attain better hit rates [4][3][14][11]. The few new developments in new designs of branch predictors, excluding neural predictors[5][6], generally involve slight modifications to the baseline predictors (bimodal, gshare, and tournament predictors) and can be seen in works such as [9][1][2][7][13]. In this work we present the need for application specific branch prediction in power constrained embedded systems, along with an example demonstrating the performance gains and power reduction for a fraction of the cost paid in advanced predictors mentioned above.

1 Evaluation of Branch Behavior

Branch prediction is an important component of the fetch stage in any processor. It provides the ability to continue instruction execution in a speculative mode as opposed to stalling the processor for each branch execution (roughly 25%-30% of dynamic instructions). In some processors, such as the Alpha EV8[14], the miss rate of the branch predictor could indeed be the limiting factor of the processor's performance. The EV8 was designed to allow up to 16 instructions to be fetched per cycle, with the capability of making 16 branch predictions per cycle. With increasing penalties for misprediction (minimum of 14 cycles in the EV8) and high clock frequency limiting the number of the table accesses of the predictor, the EV8 processors are designed with 352K bits allocated for its branch predictor.

The initial advancements in branch predictor performance can be summorized by the development of the *Bimodal*, *GShare*, *Local*, and *Tournament* predictors from which most designs are derived. These predictors can be classified into PC based (*Bimodal*), history based (*GShare* and *Local*), and combination (*Tournament*). The trend in improvement for such predictors follows from the additional information used to index the *Pattern History Table* (PHT); in most predictors the PHT is a table of 2 bit saturating counters with '00' and '01' translating to Not Taken, and '10' and '11' translating to a Taken prediction. The *Bimodal* predictor simply uses some of the PC bits to index into the PHT. Next, a single global history register is maintained and XOr'ed with the PC to produce the *GShare* predictor. The *Local* predictor is similar, but maintains local history by first indexing into a table of history registers using the PC, then using the local history as an index into the PHT. Finally, the *Tournament* predictor will access a *Bimodal* or *Local* predictor at the same time as *GShare* and a meta predictor in the form of a *Bimodal* predictor. The meta predictor is simply used to decide which prediction is best for the given PC and

is updated in the direction of the correct predictor.

Similar to caches, as transistor size became smaller, larger PHTs were built and were able to be accessed in a single clock cycle. A simple trend was then followed where a larger table allowed for less conflicts and thus more accurate predictions. Many designs preferred to leverage off this property to the point where a single branch prediction could no longer be made in a single clock cycle. To combat these latencies several attempts have been made to quicken the access to larger predictive structures. (1) Using a large GShare, a single cycle access is made possible by caching the more recent N PHT accesses[4]. (2) Pipelined designs have been migrated to branch predictors to allow multiple successive accesses[3]. (3) An overriding design was made such that a quick, 1 cycle, prediction can be made and later overridden by a longer (3 cycle) prediction if it differed[14].

Finally, new predictors have been made but can be easily traced to one of the base predictors mentioned above. These predictors include the *Bi-Mode* predictor[7] and the *(M,N) Correlating* predictor[9]. Other predictors have been known to detect correlations on data[2] or memory addresses[1] but are not currently implemented in processors due to design complexity or power overhead. The most relevant to embedded systems is the *Agree* predictor[13]. The *Agree* predictor appends a single bit to the *Branch Target Buffer* (BTB) to decide if the processor should go with the predictor's result or invert it for the given branch address. This became a powerful tool for benchmarks that include a few branches with $> 50\%$ miss rate, thus hurting performance but not quite justifying adding another predictor and meta predictor. This approach works very well if a branch has a very high miss rate, but works poorly for branches with roughly 50% miss rate. The initial runs of the *Agree* predictor set the bit to '0' if the first prediction was wrong and '1' if it was accurate creating a large dependency on the first execution of the branch. Later, better results were attained by setting the bit according to later executions or (for the best results) according to profiling data and an additional bit encoded in the branch instruction.

2 Limitations of Embedded Systems on Branch Prediction

Since current trends in improving branch prediction require larger tables and longer access times, they do not quite fit the model set forth by embedded systems. Such systems often have between 2-5 stage pipelines, and very tight space and power constraints. Conversely, the traditional predictors are a great fit for such constraints and using a simple model run of the benchmarks the best performing one can be selected for the type of applications executed. While the *Agree* predictor provides a good solution for embedded systems in dealing with branches having a very high miss rate with the selected simple branch predictor, there is no equally elegant approach for frequent branches with near 50%.

Execution of benchmarks from the MediaBench benchmark set was used to collect per branch data providing the per-branch miss rate, actual Taken/Not-Taken pattern, and the predicted Taken/Not-Taken pattern. Of the 20 benchmarks we started with, 8 were selected for their branch behavior as examples of application specific predictor design. The model used a 1k-entry *Bimodal*, *GShare*, and *Tournament* predictors with average miss rates of 6.96%, 6.30%, 4.85% respectively (Figure 4). Branches executing

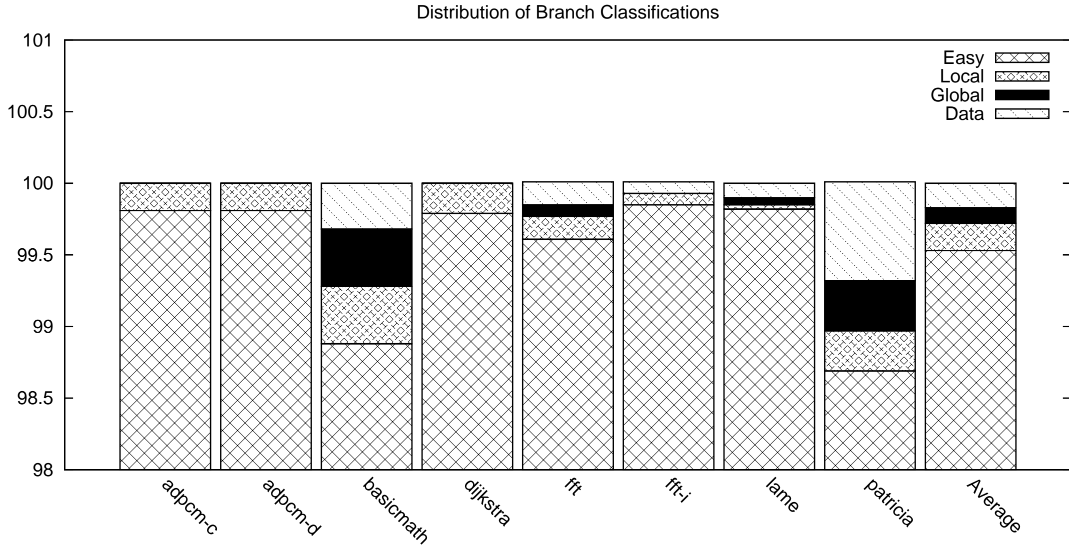

Figure 1: Static Branch Distribution in Easy/Local/Global/Data Classification.

more than 10,000 times and having a miss rate between 30% and 70% were examined. From these dumps, we were able to classify branches, those that were able to be predicted well by all the baseline predictors were classified as *Easy* branches. Those that were captured by the *Bimodal* and *Tournament* predictors but not the *GShare* predictor were classified as *Local*. Similarly, branches that did well with the *GShare* and *Tournament* predictors exclusively were deemed *Global*. Lastly, those branches that performed equally poorly on all baseline predictors were considered *Data Dependent*. Over-all we found that 99.53% of all static branches were *Easy*, 0.19% were *Local*, 0.11% were *Global*, and 0.17% were *Data Dependent* (Figure 1). Such figures are not surprising considering we already get 93% of our success rate from the simple *Bimodal* predictor and only an additional 2% from the *Tournament* predictor which encompasses local and global patterns greater than one in length.

Due to the relatively higher percentage of *Local* static branches in the selected applications, highly missed branches having such patterns as 'TNTN', 'TTNTTN', or 'NTTNTT' were revealed in the execution dumps. Such patterns would normally be easy to capture using a *GShare* or *Local* predictor, but the tight constraints of an embedded device do not allow for the implementation of the *Tournament* predictor. Alternatively, the use of *GShare* instead of the *Bimodal* predictor would eliminate the opportunities to capture such easy patterns in a Finite Sate Machine (FSM) and introduces misses in branches that were easy to predict using the *Bimodal* predictor. Such misses are accounted for by additional conflicts to simple branches and is the driving force that pushed computer design to use the *Tournament* predictor capturing the best of both predictor styles.

Our first intent was to append the *GShare* predictor with a FSM, but found that the additional misses are so sporadic and spread out among many branches that it would be easier to append a FSM to the *Bimodal* predictor. The *Bimodal* predictor is capable of captur-

ing all the branches categorized as *Easy* while missing the simpler of the *Local* branches captured by a *Local* predictor and *GShare* predictor of long enough history. Such simple patterns are short, 2-3 branch, sequences which can be captured as local patterns or global patterns (See Figure 2 for code example).

```
for (int i = 0; i < size; ++i)
  if (i%3 == 0 || i%3 == 1)
    ...
```

Figure 2: The above if statement has 'NNT' local pattern and a 'NTNTTT' global pattern requiring only 6 bits of global history to be captured.

True global patterns do exist and can be found on branches with high correlation such as in Figure 2. In that figure, data dependency will increase the miss rate of the loop, but some global history can provide a guaranteed 50% prediction rate on the second branch. In this case, the first if statement will be dependent on data being fetched from an array; if that data is '0' we set it to '1' so as to not divide by '0'. We then compare $\frac{num}{dnum}$ and $\frac{num}{2}$, it is easy to see in this situation that in every instance where the first branch executed "dnum[i] = 1;" the second if statement will pass and execute its code, thus demonstrating global dependency and data dependency for the initial if statement.

```
for (int i = 0; i < size; ++i) {
  if (dnum[i] == 0) {
    dnum[i] = 1;
  }
  if (num / dnum[i] >= num / 2) {
    ...
  }
  ...
}
```

Figure 3: The above compound if...else statement is an example of global patterns due to the dependency between the conditional statement.

The initial design of a FSM would naturally approach the most common branches, in this case the *Local* branches which encompass 0.19% of static branches as oppose to the *Global* branches encompassing 0.11%. It is important to note that FSMs can be designed for either *Local* or *Global* branches using similar signals to update: correct/incorrect prediction, taken/not-taken prediction, and the PC. While we currently do not address *Data Dependent* branches, one could design a FSM that probes the contents of registers for information regarding the future of a branch in a similar, but in a statically determined, fashion to the Data Correlating Predictor[2] and the Address Correlating Predictor[1].

3 Designing an Application Specific Branch Predictor

Our goal is to provide the best prediction rate similar to that exhibited by the *Tournament* predictor without the additional cost of 2 PHT accesses on every cycle. We already know that the *Bimodal* predictor can provide a great baseline miss rate of 6.96% and cover all of the branches classified as *Easy*. More importantly, a significant number of the branches that are still missed by the *Bimodal* predictor include ones that can be easily captured by both a *GShare*/*Local* predictor or a FSM. Using profiling data we can identify such branches having the simple alternating pattern 'TNTNTN' and route them to the FSM and away from the *Bimodal* predictor.

With the above in mind we can see how it can be possible to mimic the *Tournament* predictor by simply capturing those patterns

that *GShare* easily captures while ignoring the complex patterns that require the additional PHT. Using M5, we profiled each branch finding its taken/not-taken pattern. From these patterns we were able to decide if the branch would benefit from being mapped to a simple FSM or not. Branches that fit such criteria were flagged in the binary so that the M5 simulator can route the prediction of those branches to the FSM instead of the PHT. It is important to note that not every benchmark contained branches that needed improving. In some cases, such as *gsm untoast*, we find an average miss rate of $< 1\%$ and no branches with local patterns that were not already captured by the simple *Bimodal* predictor, thus having little to no room for improvement. Other benchmarks, while having a higher miss rate, also contain data dependent branches as well as complex global patterns making a simple FSM impossible to design. Alternatively, the selected 8 benchmarks made for good examples where simple patterns can be captured and used as examples for our ability to improve embedded application execution at little to no cost.

A FSM was designed to capture the simple case of 'TNTNTN' branches and the addresses of said branches were flagged for future executions. The following run of M5 (Denoted as Bimodal+FSMs for *Bimodal* predictor with static FSM) used a single bit in the instruction, labeling branches which should use the FSM to route the prediction away from the PHT. On average, the additional FSM provided a miss rate of 4.91% translating to a miss rate reduction of 29.36% and 22.04% from the *Bimodal* and *GShare* predictors, respectively (specific results can be seen in Figure 4). As expected, the FSM enhanced *Bimodal* predictor did not out-perform the *Tournament* predictor (average miss rate 4.85%). While a large portion of that reduction comes from the adpcm benchmarks, the average miss rate excluding those benchmarks still shows a miss rate reduction when compared to the *Bimodal* predictor and equivalent miss rate when compared with the *GShare* predictor. We expect that with an additional FSM (one that captures the 'TNNTNN' and 'NTTNTT' patterns), the FSM enhanced *Bimodal* predictor will be able to beat the *GShare* predictor for every benchmark.

The FSM itself is very simple. A single bit is initially set to '0' and is used to make the prediction. When a branch instruction is fetched, having the FSM bit set, the FSM prediction is used instead of the PHT access. Updates to the FSM bit occur only on a branch hit for a branch that used the FSM. When an update signal reaches the branch predictor carrying a correct prediction signal, the FSM bit is simply inverted. More complex updates, including speculative updates, were attempted but found to be either unnecessary or poorly performing. This elegant solution fits well in embedded systems since an update signal will reach the FSM before the next execution of the same branch in all tested cases and likely in all cases implemented on short pipelines. Due to the repeat length of the pattern (2 branch executions), an update on a miss is unnecessary since the next prediction being predicted the same as the missed prediction is likely to be a success.

Some benchmarks did not have branches that fit our previous criteria and thus exhibit no performance improvement from the additional FSM. Curiosity led us to make an attempt at capturing branch execution phases which may exhibit patterns similar to that which is captured by the FSM. The final design step allows for dynamic mapping to the FSM of any branch with no compiler support or profiling. To make the decision, the use of the PHT saturating counters was altered. Traditionally, if the counter value is $< \frac{1}{2}$ of the maximum value, then the branch is simply predicted as not taken. This trans-

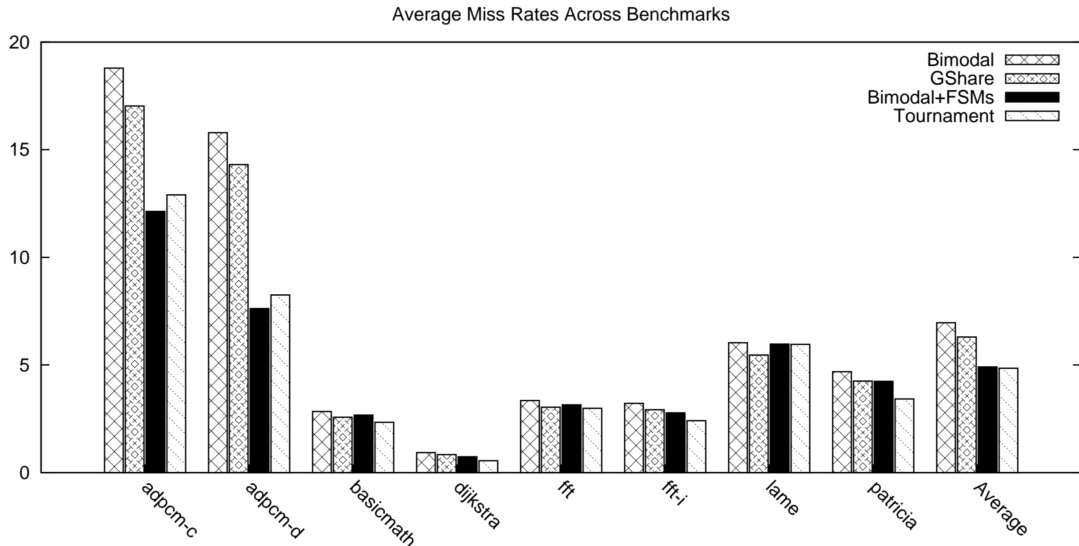

Figure 4: Miss rates for varying FSM approaches in Media-Bench bench suite.

lates to '0' and '1' being not taken, while '2' and '3' being taken values. Most of the time, a saturating counter exists in its extreme values '0' and '3' (in a 2-bit counter), while extended durations in the middle values will hint toward a 50% miss rate phase cause by a 'TNTNTN' pattern. Using a 2, 3, and 4 bit saturating counter model for the PHT, benchmarks were run having the additional condition that if the value fetched from the PHT is $\frac{1}{2}$ of the maximum value, then the branch prediction will be routed to the FSM. This model turned out to fail due to phase transitions leading to the use of the FSM and throwing other branch predictions off. To make the model successful, a more clever method of keeping track of individual miss rates or global/local patterns needs to be developed and it would likely rival the *Tournament* predictor in power/size.

Alternatively, an attempt was made to add a FSM which captures patterns of length 4 or less. Such patterns were hand picked from the profiling data described earlier, and include 'NTT', 'TNN', 'NTTT', 'TNNN', and 'NNTT'. Each pattern was hard coded as a looping register, meaning that on a bit shift, the discarded bit is appended to the other end so that prediction can be made from the first bit followed by a bit shift moving that bit to the end of the pattern. Branch instructions were then mapped to these pattern registers. While most benchmarks have already gained all they could from the simple FSM described earlier, 6 of the 20 did find benefits. Of those 6, 4 belong to the original 8 that were selected for the study. Some benchmarks (dijkstra and jpeg compress) were able to beat the *Tournament* predictor. Overall, the miss rate reduction brought the average down to 4.81% from the original 4.91%, now beating the *Tournament* predictor by 0.06%. Such reduction, in our opinion, is not worth the additional hardware space or the additional flagging bits per branch instruction. Further, these reductions only took place in a limited number of benchmarks, far less than the original FSM.

The current design requires the processor to perform a partial decode of the instruc-

tion at the fetch cycle. While most embedded systems will have no problem doing so, an alternative method exists. Since a branch can never be predicted as taken the first time, due to the target not existing in the BTB, we can send a signal along with the update to flag the BTB entry for a FSM use decoded after fetch from the instruction. This system would remove the pressure from the fetch cycle at the cost of an additional bit to each entry in the BTB. This modification would especially be beneficial when multiple FSMs are present and the partial decode becomes more complex.

4 Conclusion

The development of branch predictors has followed trends similar to those of the processor leaving it with larger structures, pipelined accesses, and parallel redundancy for the sake of slight performance gains. Very few developments have been made toward a branch predictor for smaller embedded systems, with the exception of the *Agree* predictor. Our work has shown that the *Tournament* predictor's ability to utilize the simple *Bimodal* predictor along with the *GShare* ability to capture simple patterns can be emulated utilizing a FSM. Combined, the *Bimodal* predictor and FSM are able to out perform both the *Bimodal* and *GShare* predictors, just like the *Tournament* predictor.

The statically assigned FSM was able to attain great performance with the use of only a single bit added to the branch predictor, reducing the average miss rate by 29.36% and 22.04% when compared to the *Bimodal* and *GShare* predictors and nearing that of the *Tournament* predictor. A second trial using captured patterns of 3 or 4 proved the additional simple patterns were able to close the gap between the modified *Bimodal* predictor and the traditional *Tournament* predictor. This second trial demonstrated that additional, more specific FSMs were able to be designed and implemented at low cost.

It is important to keep in mind that the goal of the work was to approach the miss rate of the *Tournament* predictor via simple and low power devices. We believe that for most applications, complex FSMs are unnecessary. As for the cost of the *Bimodal* + FSM, the ability to perform near the *Tournament* predictor miss rates is achieved by the addition of a single bit to the processor and a single flag bit in the branch instructions added by identifying a branch fitting into a category pattern. The ability to perform beyond that of the *Tournament* predictor came at the cost of a few 3 and 4 bit registers. A second design alternative was proposed to remove pressure from the fetch stage by adding a bit to the BTB to be marked by the update signal sent back after decoding and calculating the branch target.

The above work is an example of a single FSM capturing some behavior in 8 of 20 benchmarks. Each benchmark was profiled and its static branches flagged as to be mapped to a standard *Bimodal* branch predictor or a FSM designed to capture simple patterns. Improvements in branch prediction are used to demonstrate the gains made from the addition of such FSMs and the potential of creating more application specific predictors to rival the performance of complex and power hungry predictors such as the *Tournament* design. It would not be far fetched to assume that application and branch specific designs for *Global* and *Data Dependent* branches could provide further benefits in the applications studied and other applications as well.

References

[1] Hongliang Gao, Yi Ma, M. Dimitrov, and Huiyang Zhou. Address-branch correlation: A novel locality for long-latency hard-to-predict branches. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 74 –85, Feb. 2008.

[2] Timothy H. Heil, Zak Smith, and J. E. Smith. Improving branch predictors by correlating on data values. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 28–37, Washington, DC, USA, 1999. IEEE Computer Society.

[3] D.A. Jimenez. Reconsidering complex branch predictors. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 43 – 52, Feb. 2003.

[4] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 67–76, New York, NY, USA, 2000. ACM.

[5] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 197, Washington, DC, USA, 2001. IEEE Computer Society.

[6] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.*, 20(4):369–397, 2002.

[7] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. The bi-mode branch predictor. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 4–13, Washington, DC, USA, 1997. IEEE Computer Society.

[8] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, 1984.

[9] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 76–84, New York, NY, USA, 1992. ACM.

[10] Branch Predictors and Scott McFarling. Combining branch predictors, 1993.

[11] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. *SIGOPS Oper. Syst. Rev.*, 30(5):116–127, 1996.

[12] James E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[13] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. The agree predictor: a mechanism for reducing negative branch history interference. *SIGARCH Comput. Archit. News*, 25(2):284–291, 1997.

[14] Andre Seznec Stephen, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha

ev8 conditional branch predictor. In *in 29th Annual International Symposium on Computer Architecture*, pages 295–306, 2002.

[15] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 257–266, New York, NY, USA, 1993. ACM.