Offline Program Re-mapping to Improve Branch Prediction Efficiency in Embedded Systems

Stephen S. Brown, Jeet Asher, and William H. Mangione-Smith

The Department of Electrical Engineering - The University of California at Los Angeles

Abstract – This work presents a technique for improving the efficiency of hardware branch predictors. The key approach is to apply techniques of off-line re-mapping of the program-space in order to reduce the incidence of conflict misses in the branch hardware. This work also presents a new model for organizing temporal information between blocks in the address space, which can be applied effectively to previous re-mapping systems as well. The increased efficiency can be translated to improved performance for fixed hardware specifications, or used to reduce the hardware cost for achieving targeted performance during the design cycle.

1 Introduction

The work presented in this paper can be beneficially applied to both embedded and desktop/server computer systems. We consider it to be embedded-systems research for two reasons. First, that is the context out of which the work was developed. Second, and more importantly, as the code base and working characteristics of a particular system become more stable over time, the profile-based optimizations developed here become more effective. Clearly, embedded systems have more stable applications and use patterns than general-purpose systems, even when they allow the introduction of third-party software in an after-market fashion (e.g. PDAs).

This work focuses on methods for improving the efficiency of branch prediction hardware. While dynamic branch prediction hardware has been far more common in desktop and server systems, the features are becoming more common in embedded microcontrollers, particularly as relatively high-performance 32-bit RISC processing cores are moved down into embedded applications.

One important observation with respect to branch prediction hardware is that it is a form of caching. The hardware attempts to synthesize dynamic state information that can be used to predict branch outcomes. For the most part, this state information consists of simple saturating up-down counters that track recent branching decisions, though more sophisticated techniques have been developed. Much of the innovation in branch prediction hardware, and the consequent increase in prediction accuracy, come as a result of the methods used for accessing this cached state information. While early techniques were generally based on direct indexing based on bits from the branch address [1, 2], in a method that is analogous to a direct mapped cache, more recent efforts have incorporated global recent branch results to achieve higher performance [3].

Nonetheless, the branch prediction hardware is fundamentally a cache, and it can fail to produce a correct prediction for four reasons: capacity misses, compulsory misses, conflict misses, and incorrect state information. The first three conditions correspond to failures of the caching mechanisms, while only the last is a unique component of branch prediction.

There are no known techniques for reducing the incidence of compulsory misses that have proven to be effective. The solution to capacity misses is well known: increase the size of the cache. This solution is particularly unappealing to the designers of embedded systems, where cost pressure is significant and thrift is at a premium. Consequently, most research in this area has focused on reducing conflict misses or improving the quality of the state information [4-7].

The approach that is used here to reduce conflict misses involves making sure that branches that are active close in time do not map to the same entries in the branch predictor cache. As almost all forms of branch predictions use some bits from the address of the branch to index into the branch prediction cache, we can control the mapping by controlling these bits. While the actual bits used are implementation dependent, embedded systems designers frequently turn to platform-specific optimizations to improve performance.

2 Previous Work

Pettis and Hansen [8] proposed one of the earliest pieces of research to increase performance by remapping the address-space. The authors constructed a graph, where vertices corresponded to basic blocks. An edge was placed between two vertices whenever the corresponding code blocks followed directly in sequential execution. The edge weight captured the number of such events. Thus, making sure that vertices with high edge weight would be unlikely to conflict in the instruction cache produced an optimized program. Pettis and Hansen developed a series of heuristics that attempted to either move high edge weight nodes to the same cache line, or guarantee that they had different set numbers. Hashemi et al. [9] applied graph-coloring techniques to improve the utilization of scarce cache set numbers, though without changing the method used for constructing the graph.

Concurrent research by Gloy et al. [10] and Kirovski et al. [11] advanced the approach by bringing increased information into the graph. The original work by Pettis and Hansen only

introduced an edge in the graph between basic blocks that followed directly in sequence. This approach works well when branch points are strongly biased. Consider a case where a branch occurs with equal probability down both paths. If all of the activations on one path are taken before any of the activations on the other, then the two targets can share locations in the cache without causing conflict misses. However, if their activations are interleaved in time then they should be mapped to separate sets in the cache. Both Gloy et al. and Kirovski et al. address this problem by introducing temporal information regarding activations between a broader set of vertices. Each approach has a goal of producing a heavy weight edge between the two target vertices above for the case when their executions are interleaved in time and a low weight edge between the nodes when they are not interleaved in time.

The work presented here contributes to the field in two distinct ways. First, the application of address space remapping to branch prediction is unique. Second, a new approach to incorporating temporal information in the graph is introduced that achieves significantly improved results.

3 Approach

A four-stage approach is used for optimizing branch predictor performance. First, the application programs must be compiled into their final basic blocks. Second, sets of training data sets are used to collect profile information. Third, the temporal correlation graph is constructed based on the profile information. Fourth, the graph is colored to reduce conflicts in the hardware. Finally, the address space is remapped to produce an optimized binary file.

3.1 Compilation and Profiling

Compilation is accomplished using the IMPACT compiler suite from Professor Hwu's group at the University of Illinois [12]. IMPACT is used to produce a load-map that ties basic blocks to addresses, as well as to produce an instrumented binary that can be used for acquiring runtime data. Running the instrumented binary, which produces a series of events directed to an architectural simulator, collects profile information. This simulator, LSIM, has been modified to produce a sequence of basic block activations.

3.2 Construct Temporal Correlation Graph

The system used here is able to construct three types of graphs, corresponding to the original approach developed by Pettis and Hansen, the temporal approach developed by Gloy et al., and the new approach being proposed here.

Pettis and Hansen graphs are constructed by processing the trace of basic block activations and keeping track of two distinct vertices, i.e. the current block and the last block. If the current vertex does not exist in the graph it is added. A clean up phase will account for all vertices that do not occur in the trace, by leaving their locations unmodified. If there is no edge between the current vertex and the last vertex then the edge is created. Finally, the edge weight between the two vertices is incremented. This process repeats until all blocks are processed.

Gloy et al. originally proposed to construct their graphs specifically for the target instruction cache configuration. They keep track of block (or procedure) sizes, and remember a set of the most recently seen blocks. The number of entries in this set varies over time, and is kept just small enough that the sum of the size of all of the entries will fit in the target cache. Thus, the size of the set of remembered blocks varies over time, as the dynamic average block size over the recent time neighborhood varies. This approach attempts to remember the state of any seen blocks that can usefully guide cache management, while reducing the overall amount of state saved.

We have made a number of modifications to the original Gloy et al. approach in order to apply it to branch prediction. First, notice that the size of the interfering object varies in the cases of instruction caches, according to the size of the basic block that is being cached. On the other hand, the size of the interfering object is always the same for branch prediction, as it is simply the state associated with branch outcomes for a particular branch instruction. Thus, it is natural when constructing Gloy et al. graphs for branch prediction to have a fixed size remembered set. Second, we have de-coupled the size of the remembered set from the size of the branch prediction cache (i.e. the number of available colors). This approach allows us to vary the amount of program state represented in the graphs and evaluate the marginal benefit of increased history.

The Gloy et al. graphs are constructed using the following pseudo-code:

```
1. while (n = next trace node) != EOF
2.   delete n from N if present
3.   for first < i < last
4.     N[i+1] = N[i]
5.   add n to graph if not previously seen
6.   N[first] = n
7.   for first < i < last
8.     E[N[first]:N[i]]++;
```

The process begins at line 1 by iterating over all of the vertices (branch instructions) that occur in the dynamic trace. The array N holds the set of most recently seen vertices in the trace, and the set size is determined by the value of ***first-last***. Lines 3 through 6 reorder the set N so that LRU order is maintained. Finally, lines 7 and 8 add one to the weight of each edge between the first vertex in the set and all other vertices.

This approach captures the temporal correlation effectively, by keeping a longer memory than Pettis and Hansen's approach and thus managing to distinguish between subtle variations in execution conditions. In fact, Pettis and Hansen's approach can be considered a degenerate form of the Gloy et al. algorithm with the remembered set limited to a single vertex.

One shortcoming of this approach is that the graphs generated do not distinguish between remembered vertices that have been seen recently and those that are close to being removed from the remembered set. Our proposed modification involves changing the function in line 8 for increasing the weight of a specific edge. Instead of simply adding a constant term (i.e. one), we believe it is more advantageous to add a term that decreases with the value of ***first-i***, thus giving more weight to edges where vertices occur more closely in time. We have experimented with two functions, a simple linear form that adds the value ***last-i***, and a logarithmic form $\boldsymbol{2^{last-i}}$. Results for the linear approach will be presented in Section 4, the logarithmic approach achieved essentially the same performance.

3.3 Graph Coloring

Once the interference graph is constructed from the dynamic trace, it is colored using techniques developed in [16]. The algorithm is a hybrid of several known techniques, and in the correct combination it has outperformed all other approaches reported in the literature by an order of magnitude. Heuristics and meta-algorithmic approaches are necessary, as graph coloring is known to be NP-complete.

Three key techniques are essential to improving the results and reducing the runtime of the coloring algorithm:

1. Estimated values that are semi-stable over time are calculated and stored on individual vertices or edges of the graph. This allows local decision making to occur quickly, without sacrificing much useful information.
2. A global approach of assigning the most-constrained vertices to the least-constrained colors provides the essential driving force.
3. A weighted lottery scheme (again using pre-computed values), is employed to introduce statistical variance and robustness in the face of the driving function.

The graphs are colored by assigning one color to each entry in the target branch prediction hardware. Thus, a four-entry branch predictor would have four colors with which to color the graph.

3.4 Remapping

Remapping consists of assigning binary values to the colors used in graph coloring, and producing a remapping table. This table is used to translate the original branch address into a new branch address. A binary rewriting tool is used to produce the optimized output.

4 Results

As mentioned above, the compiler and profiling tool are provided by the IMPACT compiler infrastructure. A custom implementation of the graph-coloring algorithm was developed in order to facilitate a broad range of experiments. A highly parameterized branch predictor was constructed that can be used to implement the gshare model [7].

For the purposes of this paper we have adopted the MediaBench benchmark suite [17]. These benchmarks encompass most of the media applications in use today. The original MediaBench application suite has been extended to provide two data sets for each program. This enhancement allows realistic profile-based optimization, which is the core of the research presented here.

After modifying the approach developed by Gloy et al. to use a fixed size history set, the graph topology is identical to the new method. However, the specific edge weights differ significantly, and this is where we hope to achieve improved performance.

The most time consuming component of the tool set involves processing the trace to produce the final graphs. While the time involved is linear in the size of the history set, the sizes of the data sets often cause memory thrashing (even on 128MB PCs) which results in sub-linear performance. The time required to construct a typical graph was 4 hours, while the time required to collect the trace, color the graph and execute one pass through the branch simulator was less than 20 minutes. We briefly considered doing periodic pruning of graph edges that fall below some threshold. This approach is particularly appealing as a post-processing step to clean up the graphs prior to coloring. Pruning was ultimately rejected as inappropriate, however, because it could cause relatively isolated regions of the graph with relatively low edge weights to be completely pruned. As these regions are isolated they can be optimized without significant impact on other regions. Clearly, pruning them would result in a performance loss when operating in those regions of code, with relatively little commensurate return on software development time.

All of the branch predictors considered here use the gshare design with four bits of global history. Address bits are colored if they fall into the fields used for indexing the branch predictor but are not covered by the XORed global history bits. While an increased coloring range could be used that covered these additional bits, it seems counter productive to reduce the control over layout by applying a runtime XOR operation.

Each entry in the branch predictor uses two bits to store branch state information. Results in the following figures are arranged to focus on the number of colors available for optimization. Thus, while the range is from 2 to 4096 colors, the number of branch predictor entries ranges from 32 to 64k (due to the 4 bits of global history). Clearly, the lower range values are of most interest.

Figure 1 illustrates the resulting raw performance on the cjpeg application. Four categories of results are shown

1. The original performance without modification
2. The approach developed by Gloy et. al, which is indicated by the ***sc*** prefix. The number after sc indicates the size of the history set. An x after the history set size indicates that the system trained on one data set and was evaluated on the other, while a missing x indicates both sets were the same.
3. The new graph construction approach, which is indicated as ***new*** and follows the modifiers in 2 above.
4. The original Pettis and Hansen approach, which uses the x modifier to indicate cross training using separate data sets and is identified as ***ph***.

The cjpeg data begins to bring out several important trends. First, as the degree of sophistication used to construct the graphs increases, the knee of the performance curve shifts to the left. Second, there is a significant amount of difference between training and testing on the same data set or different data sets. For this reason, we will provide no further discussion of results based on self-training. It is interesting to notice than even on self-trained data a number of cases fall below the original performance once the size of the branch predictor exceeds a particular size. For example, consider the case of unmodified Pettis and Hansen compared to the original performance. Pettis and Hansen performs significantly better for the extremely small 32-entry branch predictor (55% vs. 80%), is essentially equal for the 64-entry branch predictor, and trails in performance for all other cases.

The problem with considering data for individual applications on a small number of data sets is that idiosyncratic conditions can obscure important general behavior. Thus, we were driven to consider methods for aggregating results across the set of benchmarks. The problem with direct aggregation is that some benchmarks are much more predictable than others, thus skewing results in any simple mean of the prediction accuracy. Our solution to this problem has been to first normalize the results for each application against the best branch prediction seen under any circumstances for that application, considering only cross-trained conditions. For example, consider the raw data for cjpeg, which is show in Figure 1. All of individual data points will be normalized against the value of ***new 64x*** at the 4096-color mark. We then average these normalized values.

Figure 2 shows the aggregate results for the cross-trained data. Observe that the knee of the performance curve is shifted even more dramatically to the left, i.e. performance benefits of increased hardware are greatly reduced. While there was some advantage to considering branch predictors with 256 colors when reviewing the cjpeg performance, apparently there is no benefit to considering greater than 128 colors when considering the overall performance. In fact, very little benefit can be found for more than 64 colors.

The remapping methods almost always outperform the original (for the aggregates), suggesting that the cases for cjpeg where performance dropped are likely due to idiosyncratic behavior. The one exception to this trend is ***new 16***, which generally achieves approximately 1% lower performance than the original code.

sc 32 and ***sc 64*** achieve essentially the same performance, while ***sc 16*** is close with approximately 1% higher miss prediction rate. On the other hand, ***new 32*** and ***new 16*** are close in performance and approximately 2% higher than the Gloy et al. method and 3% higher than the original gshare hardware. One conclusion that can be drawn from this data is that no more than 32 values are likely to be needed to achieve all available optimization. This observation is important for reducing the time spent optimizing the overall program.

One interesting perspective from which to consider this data involves the relative differences in hardware resources needed to reach some specific performance goal. This approach to design is becoming more common in embedded systems that adopt techniques of high-level synthesis for SOC design. If the designers judge that meeting 96% of the available performance is sufficient, then the new graph algorithms can be used to justify a 256 entry branch predictor (16 colors), while the methods of Gloy et al. would requires 512 entries. Pettis and Hansen would not be able to achieve this performance target.

5 Conclusion

This paper has made two specific contributions to the research community.

First, we have shown that the techniques of profile-driven address space remapping can be extremely effective at improving the performance of stock branch prediction hardware. This capability can be used to either improve the performance of an existing system, or reduce the cost of a system that is being designed.

Second, we have developed a new method for constructing profile-based temporal correlation graphs that captures a richer view of branch interaction than previous techniques. This approach trivially breaks the equal branching conundrum that

confounds the original Pettis and Hansen approach along with Trace Selection techniques. The resulting graphs have been shown to produce significantly better results than previous results published in the literature.

Bibliography

[1] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, 1984.

[2] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, 1981, pp. 135--148.

[3] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," Proceedings of the 24th International Symposium on Microarchitecture, 1991.

[4] E. Hao, C. Po-Yung, and Y. N. Patt, "The effect of speculative updating branch history on branch prediction accuracy, revisited," Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994.

[5] S. T. Pan, K. So, and J. T. Rahmeh, "Correlation-based branch prediction," The Twenty-Sixth Asilomar Conference on Signals, Systems and Computers, 1992.

[6] G. S. Tyson, "The effects of predicated execution on branch prediction," Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994.

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

[8] K. Pettis and R. C. Hansen, "Profile guided code positioning," ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, 1990.

[9] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, 1997.

[10] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder, "Procedure placement using temporal ordering information," Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997.

[11] D. Kirovski, C. Lee, M. Potkonjak, and W. Mangione-Smith, "Synthesis of power efficient systems-on-silicon," Proceedings of 1998 Asia and South Pacific Design Automation Conference, 1998.

[12] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W.-m. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," Proc. of International Symposium on Computer Architecture, 1991.

[13] W.-m. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, 1993.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock," Proc. of Micro 25, 1992.

[15] B. Rau, "Iterative Modulo Scheduling," *International Journal of Parallel Programming*, 1996.

[16] D. Kirovski and M. Potkonjak, "Efficient coloring of a large spectrum of graphs," Design and Automation Conference, 1998.

[17] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," Proceedings of 30th Annual International Symposium on Microarchitecture, 1997.

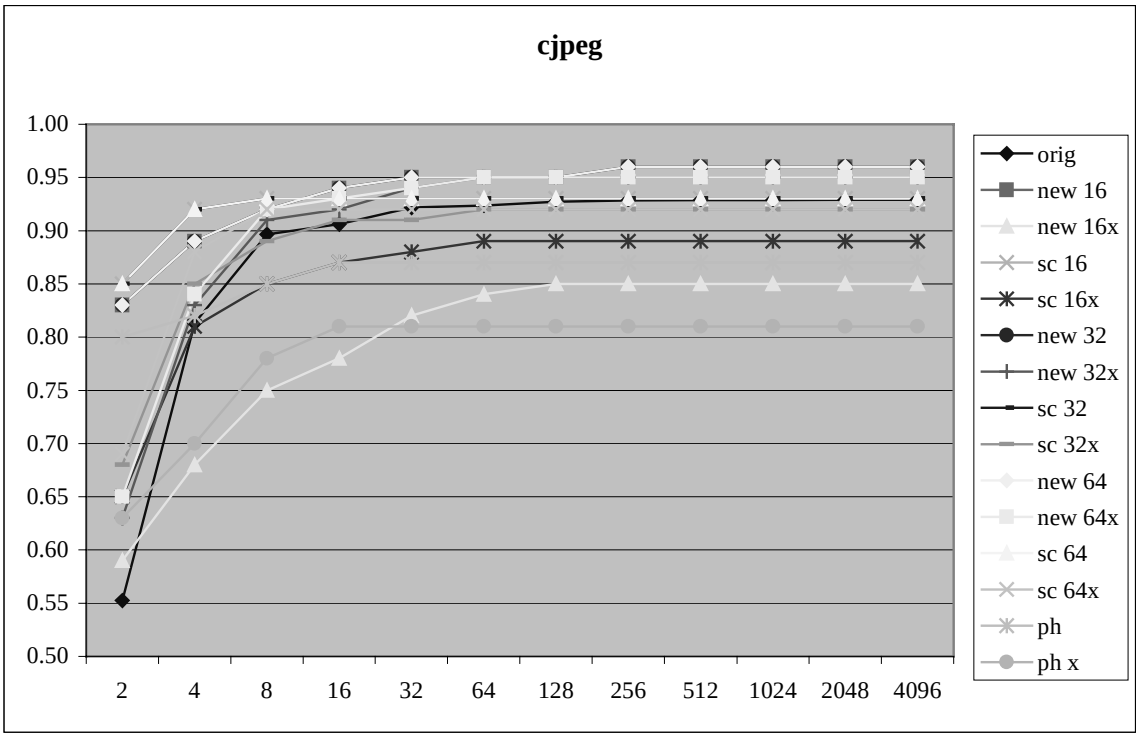

Figure 1: Branch prediction accuracy of various remapping schemes for cjpeg.

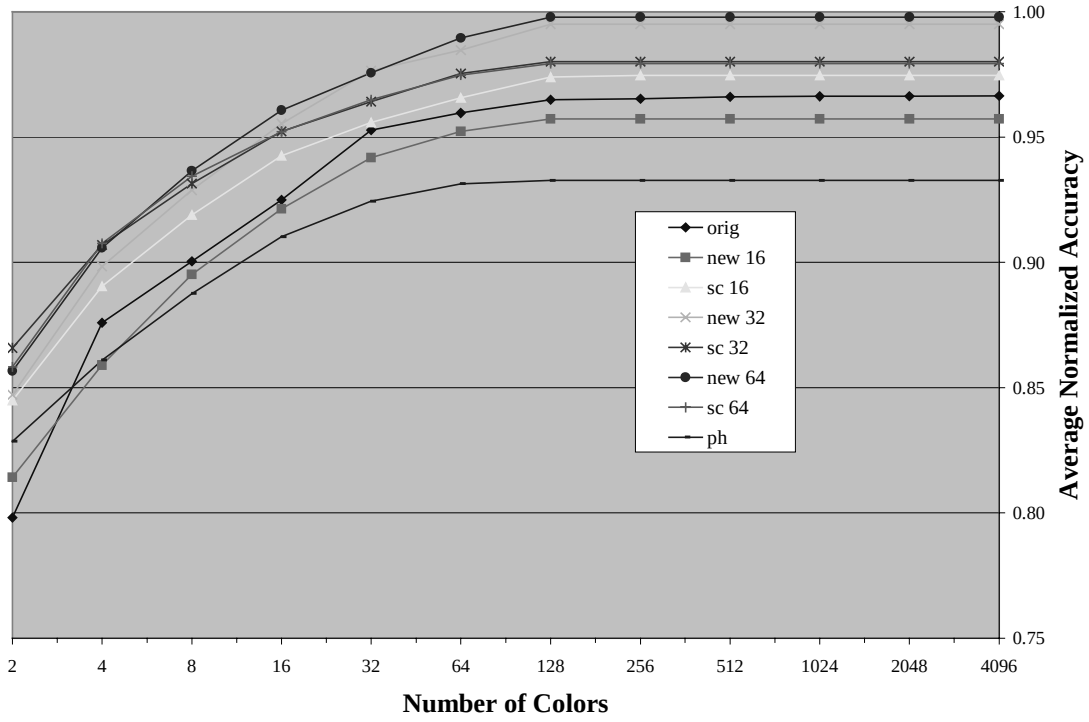

Figure 2: Average of all results normalized to individual best performance within each application