Dynamically Sizing the TAGE Branch Predictor

Stephen Pruett, Siavash Zangeneh, Ali Fakhrzadehgan, Ben Lin, and Yale N. Patt
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX USA
stephen.pruett@utexas.edu, siavash.zangeneh@utexas.edu, alifakhrzadehgan@utexas.edu, bencplin@utexas.edu, patt@ece.utexas.edu

***Abstract*—The statically assigned size of each table in TAGE limits the branch predictor accuracy for many benchmarks. We provide a mechanism for allocating TAGE table sizes at run time, based on the needs of the individual benchmark. The resulting dynamically reconfigurable TAGE branch predictor results in an MPKI of 5.370 (8KB budget) and 4.265 (64KB budget).**

I. Introduction

TAGE [1] has been the leader among state of the art branch predictors since its introduction in 2006. The TAGE algorithm is an approximation of the prediction by partial matching (PPM) data compression algorithm, first introduced in [3], which enables more efficient storage of detected patterns in the branch result stream than any prior work. This compression, coupled with its moderate use of long, expensive global histories, has made TAGE arguably the most storage efficient branch predictor to date.

The key idea behind the TAGE predictor is that most of its storage is allocated to tables with relatively small histories. TAGE reasons that since most benchmarks are dominated by short history branches, these branches should be allocated more storage. We show, however, that this is not true for all benchmarks; in particular, some benchmarks require mostly long histories and rarely use shorter histories. Allocating *any* storage to shorter histories results in an increase in MPKI. To combat this, we propose a reconfigurable architecture for the TAGE algorithm, allocating most of the storage to the history lengths that need it most. We partition the running time of an application into quanta, and reallocate the storage at the start of each quantum based on information collected during the previous quanta.

Our contributions are:

- A reconfigurable architecture that can easily reallocate storage at run time among the various histories.
- Algorithms for determining at run time the storage needs of the running application.
- The addition of a victim cache to assist the most loaded table.

The remainder of the paper is organized into 3 sections. Section 2 discusses the architecture and its limitations, Section 3 analyzes our results, and Section 4 provides some concluding remarks.

II. The Architecture

Our design consists of several parts (see Figure 1): (1) the set of geometrically increasing histories and cascading

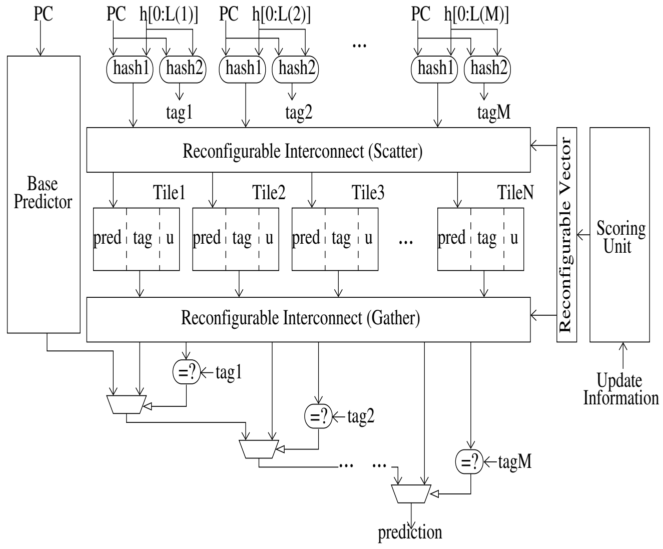

Fig. 1. Block diagram of the Reconfigurable TAGE architecture

MUXes, as originally specified in TAGE, (2) an array of tiles and two interconnects which are responsible for grouping tiles into tables, (3) a scoring unit that collects run time information during each run-time quantum to determine the size of all the tables for the next quantum, and (4) a Configuration Vector that is specified by the scoring unit at the end of each quantum and is used by the reconfigurable interconnect to resize the tables at the start of the next quantum.

A. Tables, Tiles, and Configuration Vectors

TAGE specifies M tables, each having storage assigned prior to run time. We have chosen M=16, and allocate storage to each table from a set of 32 tiles, each tile consisting of 64 entries. We represent the assignment of tiles to tables by a 16 digit Configuration Vector, where each digit d_i corresponds to the number of tiles in table i. In each case, we restrict d to be 0 or a power of 2. For example, the Configuration Vector 2,0,0,0,0,0,0,0,2,0,2,2,4,4,8,8 indicates that the second table consists of no entries and the 16th table consists of 512 entries (8 tiles of 64 entries each). Note that although some tables may be empty, we still have 16 history registers. We allocate no entries corresponding to a history register if we determine that any entries will be useless. That is, although we assign up-front many history registers we only allocate tables for those histories which we determine at the start of each run-time quantum will be useful.

Entries in our tables are used exactly as in TAGE-SC-L[5]. Each consists of 3 fields: tag, prediction counter, and useful bits; prediction and update are both done as in TAGE.

B. The Reconfigurable Interconnect

Our design contains two reconfigurable Interconnects (RIs). The first connects the history registers to the tiles and is responsible for mapping each history to its appropriate tiles. The second connects the tiles to the output muxes and is responsible for mapping each tile back to its appropriate history. Figure 2 illustrates the mechanism with two histories and four tiles. Each of the x's in the figure represents a switch which is either enabled or disabled by the Configuration Vector (CV). The CV drives each switch in our design, connecting each of the tiles to its appropriate history register and output mux.

Tiles are organized as direct mapped tables, i.e., the hash function that produces the index into a table must produce the index into a tile as well as the select signals between a variable number of tiles. The hash functions produce an index that is wide enough to index into each tile (low bits of index) and to select between the maximum number of tiles that can be grouped together (high bits of index).

Note that we could have used multiple hashing functions, each with a different output width, and selected the appropriate hashing function based on our configuration. Not doing this, however, has the advantage that the tile index never changes so a tile that remains associated with the same table can continue to use its already warmed-up entries, providing a small reduction in warm-up time after reconfiguration.

The last piece of the RI are the comparators, which compare the high bits of the hashing function to the tile ID of the appropriate tile. This tile ID is the index of the tile within its table. For example, in Figure 2, Tile3 can either be the 3rd tile in table 1, the first tile in table 2, or the 3rd tile in table 2 depending on the configuration. The tile ID is supplied by the Configuration Vector and compared to the high bits of the hashing function. If the bits are equal, the entry from that table passes through the buffer and is placed on the appropriate output line, based on which switch is set. From there, the entry is split into its tag and prediction fields. The tag field is compared with the computed tag, and the prediction is supplied to the mux as in TAGE.

C. The Scoring Unit

The scoring unit determines the allocation of tiles to tables for the next quantum. It separates the run-time quanta into two phases: a learning phase (the first seven quanta of a running program) and an adaptive phase (the rest of the quanta of the running program). We have fixed each quantum as the time it takes to accumulate 100,000 predictions.

1) The Learning Phase: The purpose of the learning phase is to produce the seven configurations that will be used during the rest of the program's execution. Seven quanta are

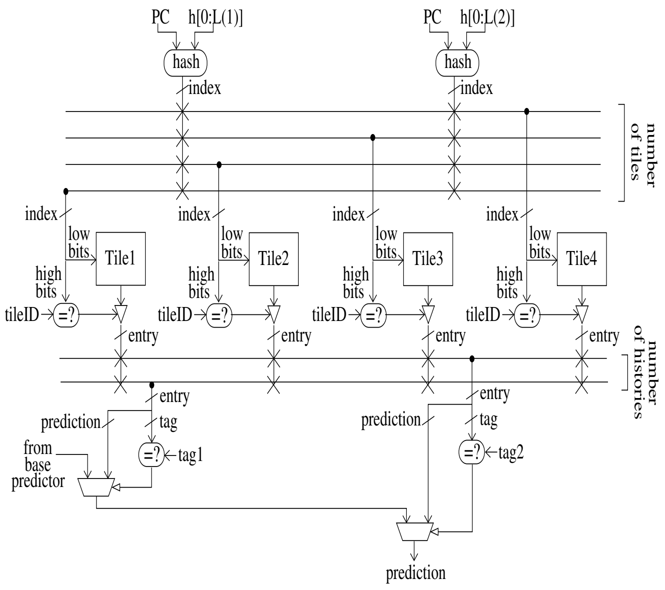

Fig. 2. The reconfigurable interconnect

needed to produce these configurations. Each quantum starts with tiles allocated to tables according to a Configuration Vector. The Configuration Vector for the first quantum is 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2. During the quantum, statistics such as the number of attempted allocations, the number of conflicts, and the number of mispredictions are collected. At the end of the quantum, the scoring unit uses this information to produce a new Configuration Vector for the next quantum, and saves the Configuration Vector and misprediction count of the current quantum for use during the adaptive phase.

2) The Reallocation Algorithm: The reallocation algorithm consists of two parts, (1) collecting tiles from tables that are not fully utilizing them, and (2) reassigning them where needed. Tiles are collected from tables as follows. For each table, the number of entries that have non-zero useful bits is rounded up to the nearest power of 2. This value (call it k) represents the smallest table size that can hold all the table's useful entries. If k is smaller than the current size of the table, we reduce the table size to k, and reclaim the remaining tiles.

This policy for collecting unused tiles is very aggressive. If the number of useful entries is very close to the new table size, the number of collisions to that table will increase, and many useful entries may end up being evicted. However, our experiments with less aggressive policies for reclaiming tiles found that being more aggressive was generally better.

The reclaimed tiles are reassigned to tables that are most congested using two metrics: the number of attempted allocations to each table (whether resulting in a conflict or a success) and the number of conflicts each table experiences. A conflict occurs when an attempted allocation fails (i.e., the useful bits of the entry to be allocated are non-zero). Both counts were collected during the quantum just completed. We reassign tiles to tables in one of three ways. The attempt algorithm uses the number of attempted allocations, the conflict algorithm uses the number of conflicts, and the hybrid algorithm is a combination of both.

The algorithms recognize two key observations:

1) Highly congested tables usually have many conflicts.
2) However, some tables are so highly congested that entries get overwritten by new allocations before they can ever be used. These tables will paradoxically have high allocation counts, few useful entries, and few conflicts.

Increasing the size of a table having a large number of conflicts typically provides some reduction in MPKI, even if the storage does not increase much. Tables that suffer from the second observation, however, typically require a lot more storage before we see any gains. Moreover, it is difficult to distinguish a table's behavior due to high congestion (and therefore low reuse) from a table's behavior due simply to it not being very useful.

a) Conflict Algorithm: The conflict algorithm targets observation 1. At the end of a quantum, we compute the average of the number of conflicts for each table. The scoring unit distributes fairly the reclaimed tiles to all the tables that had more conflicts than the average.

b) Attempt Algorithm: The attempt algorithm targets observation 2. The average number of attempted allocations is computed, and those tables that had more than the average are assigned reclaimed tiles. In addition the table with the largest number of attempted allocations and all tables associated with longer histories are designated for assignment of additional reclaimed tiles.

The Attempt Algorithm recognizes that the tables most likely to suffer from observation 2 are the tables with histories larger than the highest attempted allocation table. That is, if entries are not being reused, then we do not know whether the entry would have predicted correctly in the current table, or would have mispredicted again, thereby causing an allocation in a higher table. Therefore we must increase the storage in not only the highest attempted allocation table, but each table with a longer history as well.

c) Hybrid Algorithm: The attempt algorithm can provide significant gains when there is enough storage to allocate to the affected tables. However, many benchmarks suffer from a large number of mispredictions, which cause a large number of allocations, which limit the overall storage available. To combat this, we have created a hybrid algorithm that, after the first quantum, selects between the conflict and attempt algorithms based on the number of mispredictions. If the number of mispredictions is high, the conflict algorithm is used. If it is low, the attempt algorithm is used. If the number of mispredicts is very low, then we do not reconfigure at all since the cost of doing so is too large.

d) Fairness: These three algorithms are responsible for producing a priority for which tables to increase first. Because the tables are direct mapped, every increase doubles the size of that table. If two tables are equally chosen, the storage is distributed equally among them with a slight priority given to the smaller table. This tends to prevent starving (i.e., doubling a large table could prevent smaller tables from getting any storage).

3) The Adaptive Phase: In the adaptive phase, we select the Configuration Vector for the next quantum based on the number of mispredictions associated with each. We established seven configurations during the learning phase. We keep track of the number of mispredictions for each with a seven entry misprediction vector. At the start of the adaptive phase, the misprediction vector contains the number of mispredictions that occurred for each configuration during its quantum in the learning phase.

At the end of each quantum during the adaptive phase, we update the misprediction vector entry of the configuration that was active during that quantum with the number of mispredictions that occurred during the quantum. Unless there is reason to do otherwise, we select the Configuration Vector for the next quantum corresponding to the one with the smallest number of mispredictions in the misprediction vector. This algorithm is very similar to that in [10]. We repeat this process until the end of the program.

Switching configurations can be costly, so we try to avoid unnecessary swithes when possible. We add a 2-bit counter to help us make sure the current configuration does not change too often. When we switch to a new configuration, the counter is initialized to 0. Each time we continue with the same configuration, the counter is incremented. When we decide to switch configurations (because the misprediction vector has a new lowest entry) we decrement the counter. We do not switch configurations unless the counter is zero. This simple mechanism prevents unnecessary switches due to an outlier quantum.

D. The Victim Cache

The victim cache is a small cache that utilizes the remainder of our storage budget. It is organized as a bloom filter [11] to maximize its capacity. The bloom filter allows us to trade correctness of the cache (i.e. false positive rate) for capacity and thus store more entries with fewer bits. The victim cache stores evicted entries that were never used. We track unused entries in TAGE by taking advantage of some of the unused bit combinations at each entry. Doing this allows us to track unused entries at no additional storage overhead. Any entry that is overwritten and never used is moved to the victim cache. Because the victim cache is so small, we only allow one of the tables to use it – the table with the fewest conflicts, while still having an above average number of allocations. Choosing a table with many allocations improves the reuse for entries that get overwritten before they are used. Choosing a table with few conflicts increases the likelihood that the victim cache will be able to restore the entry without getting a conflict itself.

E. Limitations of our Design

Our design has two serious limitations on performance. First, we must use the same tag size in all the tiles. Since any tile can be associated with any history length, we must choose a tag size that is appropriate for dealing with aliasing in the longest history table, where the aliasing problem is at its worst. In contrast, TAGE is able to choose different tag sizes for each table, which allows it to save storage in short history tables while reducing aliasing in long history tables. Table I shows the tag sizes that we used for each of our submissions. This significantly impacts the overall storage available to our predictor, contrary to our overall goal of storage efficiency. We could have had an asymmetric design where each entry in the lower tables actually contained two sets of tag, usefulness, and prediction, which would have effectively doubled the capacity of tiles mapped to lower tables. However, our experiments showed that doing so would hurt performance due to the reduced size of each tag. Alternatively, each entry could contain two tags and one set of useful bits and prediction bits. This would allow two branches to share the same entry if their corresponding predictions were in the same direction. We did not have time to evaluate this method,

The second limitation is that the total number of tiles in our design must be a power of 2. This is to reduce the complexity of the configuration logic in the scoring unit. If the total number of tiles is a power of 2, then it is impossible to create a configuration where not every tile is mapped. If we did not have this constraint, the scoring unit would need to more carefully decide where to map tiles, as some decisions may result in a remainder of unmapped tiles.

TABLE I
CHOICE OF PARAMETERS FOR EACH CATEGORY

Parameter	8KB	64KB
# of Tiles	32	64
Size of Tile	64	512
Tag Size	15	10
Quantum	100k	50k
# of Quantums in Reconfigure Phase	7	7

III. Analysis of Results

Our predictor, which dynamically configures itself to each benchmark in the 2016 CBP list, results in an average MPKI of 5.370 (8KB budget) and 4.265 (64KB budget). We compare this to the running of the same predictor with configuration disabled (i.e., a TAGE predictor with a static configuration of 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2), which achieves an average MPKI of 5.520 (8KB budget) and 4.288 (64KB budget). In our experiments, we use a TAGE-SC-L [5] as our baseline. The SC and L components were added to our reconfigurable TAGE so that we can fairly evaluate the usefulness of reconfiguration against the most competitive version of TAGE.

Tables II and III show the difference in MPKI, the improvement in MPKI as well as the most frequently used configuration for our 5 most improved traces for the 8KB and 64KB track respectively. The difference column shows the raw difference in MPKI with and without configuration (old MPKI - new MPKI). The most improved traces were selected based on which traces had the highest differences in MPKI. The configuration is represented by its 16 digit Configuration Vector. Each digit represents the number of tiles used for each history. For example, the first digit for SS53 is 1, which means that for SS53, the first history has 1 tile allocated. SHORT_MOBILE-2 (SM2) and SHORT_MOBILE-43 (SM43) give us the most improvement. Note that those traces are also the ones with the most abnormal configurations. Recall that geometrically increasing the history lengths allocates most of the storage to the tables with smaller history lengths. It is no surprise that the benchmarks showing the most improvement are the ones that require more storage in the longer history tables. Moreover, most traces in the CBP4 suite prefer more storage in the smaller history tables. This makes it very difficult to justify statically allocating storage in the longer history tables. Because we can dynamically reallocate storage to the longer history tables, we can see a big win on some traces. Note that we can also get modest improvements on traces that utilize the smaller history tables (SS53, SS56, SS57) because we can completely abandon the long histories and allocate all the storage in the predictor to the smaller histories. Our gains in the 64KB category were not as large as the 8KB category since the problems discussed in section 2.6 become more serious as the total size of the predictor increases.

TABLE II
IMPROVEMENT IN MPKI FOR OUR 5 BEST TRACES ON THE 8KB STORAGE BUDGET

Trace	Diff	Improv	Configuration Vector
SS53	2.32	6.59%	1,8,8,4,4,4,1,1,1,0,0,0,0,0,0,0
SS56	2.49	6.17%	1,8,8,4,4,4,1,1,1,0,0,0,0,0,0,0
SS57	2.61	7.57%	2,8,8,4,4,4,1,1,0,0,0,0,0,0,0,0
SM2	2.88	65.58%	1,1,1,1,1,1,4,8,4,4,4,1,1,0,0,0
SM43	5.07	435.97%	0,0,0,0,1,1,2,1,1,8,4,4,4,4,1,1

TABLE III
IMPROVEMENT IN MPKI FOR OUR 5 BEST TRACES ON THE 64KB STORAGE BUDGET

Trace	Diff	Improv	Configuration Vector
SS57	1.1	3.85%	4,4,4,8,8,8,4,4,4,4,2,2,2,2,2,2
SS53	1.22	4.26%	4,4,8,8,8,8,4,4,2,2,2,2,2,2,2,2
SM42	1.71	151.15%	2,1,1,4,4,4,8,8,8,8,4,4,2,2,2,2
SS41	1.99	15.35%	1,1,1,1,4,4,4,4,8,8,8,4,4,4,4,4
SS58	2.45	25.52%	2,2,1,1,1,1,4,8,8,8,8,4,4,4,4,4

IV. Conclusion

Because some benchmarks suffer from the static allocation of storage in TAGE, we introduced a new architecture for the TAGE branch predictor, which uses run time information to dynamically allocate its storage to the most highly congested tables. We discussed three policies to measure congestion and showed the improvement in our five best benchmarks. We submit that reconfiguration is necessary to ensure that all benchmarks are achieving their minimum MPKI.

Acknowledgment

We thank the anonymous reviewers and Santhosh Srinath for their valuable suggestions and feedback. We thank the rest of the HPS Group for contributing to our daily working environment. Finally, we thank Intel, Oracle, and Microsoft for their continued financial support or our research.

References

[1] A. Seznec and P. Michaud, “A case for partially TAgged GEometric history length branch prediction.” in *Journal of Instruction Level Parallelism*, Vol. 8, Feb. 2006.

[2] A. Seznec, “Analysis of the O-GEHL branch predictor” in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[3] P. Michaud, “A ppm-like, tag-based predictor.” in *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[4] A. Seznec, J. San Miguel, and J. Albericio, “The inner most loop iteration counter: a new dimension in branch history.” In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. 2015.

[5] A. Seznec, “Tage-sc-l branch predictors” In *Proceedings of the 4th Championship on Branch Prediction, http://www.jilp.org/cbp2014/*. 2014.

[6] T.-Y. Yeh and Y. Patt, “Two-level adaptive branch prediction” In *Proceedings of 24th Intl. Symp. on Microarchitecture*. Nov. 1991.

[7] M. Evers, S. Patel, R. Chappell, and Y. Patt, “An analysis of correlation and predictability: What makes two-level branch predictors work” In *Proceedings of 24th Annual Intl. Symp. on Computer Architecture*. June 1998.

[8] J. Smith, “A study of branch prediction strategies” In *Proceedings of 8th Annual Intl. Symp. on Computer Architecture*. 1981.

[9] R. Nair, “Dynamic path-based branch correlation” In *Proceedings of 28th Annual Intl. Symp. on Microarchitecture*. 1995.

[10] T. Juan, S. Sanjeevan, and J. Navarro, “DynamicHistory-Length Fitting: A Third Level of Adaptivity for Branch Prediction” in *Proc. of the 25th Intl. Symp. on Computer Architecture (ISCA)*, July 1998.

[11] M. Yoon, “Aging bloom filter with two active buffers for dynamic sets” in *Knowledge and Data Engineering, IEEE transactions on,* vol. 22, no. 1, pp. 134-138, 2010.