Dynamically Sizing the TAGE Branch Predictor

ABSTRACT

We observe that the statically assigned size of each table in TAGE limits the branch predictor accuracy for many benchmarks. This paper describes a mechanism for allocating table sizes of TAGE at run time, based on the needs of the individual benchmark, and incorporating the resulting dynamically reconfigurable TAGE into our branch predictor. This results in a MPKI of 5.37 on an 8KB budget, and 4.265 on a 64KB budget.

1. INTRODUCTION

TAGE [1] has remained 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 perhaps 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. The rationale is that most benchmarks are dominated by short history branches; therefore, these branches should be allocated more storage. We show, however, that this is not true for all benchmarks; in particular, some benchmarks require only the use of 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, where we allocate the the bulk of the storage to the history lengths that need it most.

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 discusses our algorithms for reallocating storage. Section 4 analyzes our results.

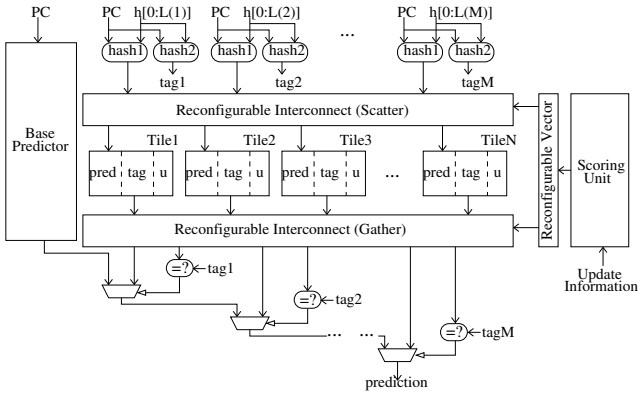

Figure 1: Block diagram of the Reconfigurable TAGE architecture

2. THE ARCHITECTURE

2.1 Overview

Our design consists of several parts: (1) a set of reconfigurable tables, (2) algorithms for determining the reallocation of storage to tables, (3) a scoring unit that is need by the reallocation algorithm, and (4) a sophisticated interconnect for connecting the tables to the rest of the branch predictor. Figure 1 shows the block diagram.

2.2 The Tables

TAGE historically has required n tables, each having storage assigned prior to run time. We have chosen n=16, and allocate storage to each table from a set of 32 tiles, each consisting of 64 entries. Each table consists of 0 tiles (nothing allocated), or 2^k tiles, where the value of k at each reconfiguration point is determined by our reconfiguration algorithm. An example configuration might be 2-0-0-0-0-0-0-0-2-0-2-2-4-4-8-8, where each of the 16 digits correspond to the number of tiles in the corresponding table. In this example 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. But we also have the luxury of allocating no entries corresponding to that history register if we determine that those entries will be useless. This means that we can assign up-front many history registers and only allocate tables for those histories which we determine at runtime will be useful.

Like TAGE, each entry in each table consists of 3 fields: tag, prediction counter, and useful bits. The prediction comes from the matching entry that has the longest history (or the second longest, in the case of alternate prediction). If there is no matching entry, the prediction from the base table is used. The predictor is updated exactly as in TAGE [5].

2.3 Phases of Execution

We break execution into two phases: the reconfiguration phase and the adaptive phase. In the reconfiguration phase we generate and evaluate different tile configurations. In the adaptive phase we try and lock onto the best performing tile configuration.

The reconfiguration phase is divided into seven quanta. In each quanta we run the current configuration (starting from a default configuration in the first quantum) and collect statistics. At the end of the quantum, the scoring unit uses the collected statistics to generate a new configuration for the next quantum. This iterative process continues for all seven quanta. At the end of the reconfiguration phase we will have generated and evaluated seven different tile configurations

Additionally, we track the number of mispredictions that occur during each quantum. We do this by adding a structure called the misprediction vector. The misprediction vector is a 7 entry table that contains a slot for each configuration that was run during the configuration phase. At the end of each quantum, we log the number of mispredictions in the corresponding entry in the misprediction vector. This way we have a record of which configuration ran with the lowest number of mispredicts. This algorithm is very similar to that in [10]. Once we are finished with the reconfiguration phase, we move into the adaptive phase. Just like in [10], the adaptive phase selects the configuration that ran with the lowest number of mispredictions and locks onto it. At the beginning of the first quantum in the adaptive phase, we scan the misprediction vector for the lowest number of mispredicts. The index of that entry tells us which configuration to lock on to. At the end of each quantum, we update the misprediction vector with the number of mispredictions, scan for the smallest entry, then lock on to that configuration. We repeat this process until the end of the program.

Switching configurations can be costly, so it is necessary to avoid as many unnecessary switches as possible. To this end, we introduce a 2-bit counter that we use to 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 remain in 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. The only time we actually switch configurations is when the counter is equal to zero. This simple mechanisms prevents unnecessary switches due to an outlier quantum.

2.4 The Scoring Unit

The scoring unit works in two phases. The first phase collects as many tiles as possible from tables that are not fully utilizing them. This is done by examining the number of useful entries in a table (i.e., entries that have non-zero useful bits) and rounding that number up to the nearest power of 2. This value represents the smallest table size that is capable of holding all of the useful entries. If this size is smaller than the current size of the table, we reduce the size of the table by reclaiming all of the surplus tiles, thus shrinking the table to its minimum size. 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, we experimented with less aggressive policies for reclaiming tiles and found that being more aggressive was generally better.

The second phase reassigns the reclaimed tiles to tables that are most congested. We have two metrics for measuring congestion: the number of attempted allocations and the number of conflicts that each table experiences. Each attempted allocation can either result in a conflict or a success. Either way, it is included in our statistic. A conflict occurs when an attempted allocation fails (i.e., the useful bits of the entry to be allocated are non-zero). We use these two metrics to form two different policies for reconfiguration: the attempt policy and the conflict policy. The policies and the reasoning behind them will be discussed in detail in section 3. The policies are responsible for producing a score that quantifies the level of congestion at each table. The second phase of the scoring unit uses these scores as priorities for which tables to increase first. Because the tables are direct mapped, any increase in size must be to double the size of the table. If two tables have the same score, the storage is distributed equally among them with a slight priority given to the smaller table. This works to prevent starving (doubling a large table could prevent the smaller tables to get any storage).

Once the scoring unit has decided how much storage to take away and give to each table, it produces the reconfiguration vector (RV). The RV is a vector that contains all the information needed by the reconfigurable interconnects (RI) to group the appropriate tiles into tables. At the end of each quantum, the scoring unit runs, producing these vectors. Once the vector is latched into the RV register, the reconfigurable interconnect will connect the appropriate tiles, and the next quantum will begin.

2.5 The Reconfigurable Interconnect

There are two RIs in our design. The first interconnect runs between the history registers and the tiles and is responsible for mapping each history to its appropriate tile. The second interconnect runs between the tiles and the output muxes and is responsible for mapping each tile back to its appropriate history. Essentially, the first interconnect works as a scatter, mapping some smaller set of histories to a larger set of tiles, while the second interconnect works as a gather, mapping the larger set of tiles down to the smaller set of histories. Figure 2 shows an example of our RI. For brevity, the figure uses 2 histories and 4 tiles, but the full network works similarly. Note that each of the x's in the figure represent a switch, which is either enabled or disabled by the reconfiguration vector (RV). The RV drives each switch in our design thus connecting each of the tiles to their appropriate history register and output mux.

We organize our tiles into direct mapped tables, which

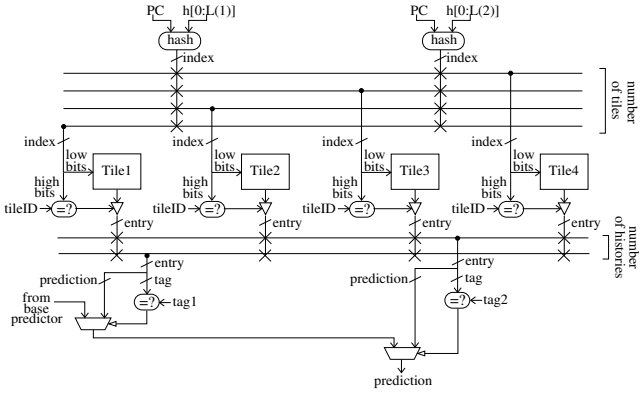

Figure 2: The reconfigurable interconnect

means that 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. In our design, 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). Alternatively, we could have used multiple hashing functions, each with a different output width, and selected the appropriate hashing function based on our configuration. Our method, however, has the advantage that the tile index never changes. This means that a tile that remains associated with the same table can continue to use its already warmed-up entries, which provides a small reduction in the warm-up time after we reconfigure.

The last piece of the RI are the comparators, which check that the high bits of the hashing function match the tile ID of the appropriate tile. This tile ID is the index of the tile within its table. For example, in Figure 2, the 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 what the configuration is. The tile ID is supplied by the reconfiguration vector and compared to the high bits of the hashing function. If the bits are equal, the entry from that table is allowed to pass through the buffer and is placed on the appropriate output line, based on which of the switches is set. From there, the entry is split into the tag and prediction fields. The tag field is compared with the computed tag, and the prediction is supplied to the mux as in TAGE.

2.6 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 1 shows the tag sizes that we used for each of our submissions. This significantly impacts the overall storage available to our predictor, which goes against our overall goal of storage efficiency. To combat this, we could have had an asymmetric design where each entry in the lower tables actually contained two sets of tag, usefulness, and prediction. This would effectively double the capacity of tiles mapped to lower tables. However, in our experiments we saw that doing so would reduce the size of each tag to a sufficiently low number of bits that hurt performance. Alternatively, we could have made each entry contain two tags and one set of useful bits and prediction bits. In this method we would allow two branches to share the same entry if their corresponding predictions were in the same direction. We were not able to evaluate this method, however, due to time constraints.

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, then the scoring unit would to more carefully decide where to map tiles, as some decisions may cause there to be a remainder of unmapped tiles.

3. THE POLICY

We have developed three policies for identifying the most highly congested tables. Each policy producing a score, which the scoring unit uses to produce the reconfiguration vector as discussed in section 2.3. The policies recognize two key observations:

1. In general, a highly congested table will have a lot of conflicts
2. However, some tables are so highly congested that entries get overwritten by new allocations before they can even be used. These tables will paradoxically have high allocation counts, few useful entries, and few conflicts

Increasing the size of a table that has a large number of conflicts will typically provide some reduction in MPKI, even if the storage does not increase by a very large factor. Tables that suffer from the second observation, however, typically require a lot more storage before gains can be seen. Moreover, it is difficult to distinguish between a table that is behaving this way due to high congestion (and therefore low reuse) or a table that is behaving this way simply because it is not very useful.

3.1 Conflict Policy

The conflict policy is targeted at observation 1. During each quantum, the number of conflicts a table experiences is collected. At the end of the quantum, we compute the average of these values. Any table that experienced more conflicts that the average has its score incremented. The initial score for each table is 1, so if a table has an above average number of conflicts, its score will be 2. At the end of the quantum, the scoring unit will then collect all unused storage and distribute it fairly among the tables with a score of 2.

3.2 Attempt Policy

The attempt policy is targeted at observation 2 and is similar to the conflict policy. Just as in the conflict policy, the

tables that received an above average number of attempted allocations will all have their scores incremented. The attempt policy, however, differs in that the table with the maximum number of attempted allocations and each table after it (i.e. tables with longer histories) will have their scores incremented yet another time. This recognizes that the tables most likely to suffer from this problem are the tables after the highest attempted allocation table. This is because 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.

3.3 Hybrid Policy

The attempt policy can provide significant gains when there is enough storage to allocate to the effected tables. However, many benchmarks suffer from a large number of mispredictions. These mispredictions cause a high number of allocations, which limit the overall storage available. Therefore, we have created a hybrid policy that, after the first quantum, selects between the conflict and attempt policy based on the number of mispredictions. If the number of mispredictions is high, the conflict policy is used. If it is low, the attempt policy is used. If the number of mispredicts is very low, then we do not reconfigure at all as the cost of doing so is very significant.

4. 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 off correctness of the cache (i.e. false positive rate) for capacity and thus store more entries for fewer number of 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 pushed into the victim cache. Because the victim cache is so small, we only allow one of the tables to use it. We select that table by choosing the table with the lowest number of conflicts, while still having an above average number of allocations. Choosing a table with a lot of allocations will help us improve the reuse for entries that get overwritten before they are used. Choosing a table with few conflicts is desirable because it increases the likelihood that the victim cache will be able to restore the entry without getting a conflict itself.

Table 1: 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

5. ANALYSIS OF RESULTS

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 2 and 3 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 as a 16 numbers separated by dashes. Each number represents the number of tiles used for each history. For example, the first number for SS53 is 1. This means that for SS53, the first history has 1 tile allocated to it. 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 the geometrically increasing the history lengths allocates most of the storage to the tables with smaller history lengths. Therefore it is no surprise that the benchmarks we see the most improvement on are the ones that require more storage in the higher tables. Moreover, most traces in the CBP4 suite prefer higher amounts of storage in the low tables. This makes it very difficult to justify statically allocating storage in the higher tables. Because we are able to dynamically reallocate storage to the higher tables, we are able to see a big win on some traces. Note that we are also able to get modest improvements on traces that utilize the low tables (SS53, SS56, SS57). This is because we are able to completely abandon the high histories and allocate all the storage in the predictor to the lower histories.Our gains in the 64KB category were not as large as the 10KB category. This is because of the problems discussed in section 2.5 become more serious as the total size of the predictor increases.

Table 2: Improvement in MPKI for our 5 best traces on the 8KB storage budget

Trace	Diff	Improv	Configuration
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 3: Improvement in MPKI for our 5 best traces on the 64KB storage budget

Trace	Diff	Improv	Configuration
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

6. CONCLUSION

Our key observation is that some benchmarks suffer from the static allocation of storage in TAGE. In this paper, 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 three best benchmarks. Ultimately, we conclude that reconfiguration is a necessary feature to ensure that all benchmarks are achieving their minimum MPKI.

7. 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.