Dead Block Replacement and Bypass with a Sampling Predictor

Daniel A. Jimenez

Dead Block Replacement and Bypass with a Sampling Predictor

Daniel A. Jiménez
Department of Computer Science
The University of Texas at San Antonio

Abstract

We present a cache replacement and bypass policy driven by dead block prediction. A block is considered dead is it will be replaced before it will be used again. If dead blocks can be identified, then they can be replaced early. If a block is predicted to be "dead on arrival," i.e., it will not be accessed again after it is placed in the cache, then it can bypass the cache. The predictor is based on one simple observation: if a block becomes dead after being touched by a particular instruction, then other blocks touched by that instruction are also likely to become dead.

Ideally, we would track the tendency of blocks to become dead for every instruction accessing the cache. However, to fit within a realistic hardware budget, we sample only a few sets from the cache. This paper describes our sampling dead block predictor and the techniques used to make it fit within the allow hardware budget for the cache replacement contest.

1 Introduction

Our into the 1st JILP Cache Replacement Championship Competition is based on *sampling dead block prediction*. A block is considered *dead* between its last access and its eviction. A *dead block predictor* attempts to determine whether a block is dead.

2 The Basic Idea

The basic idea of our cache replacement and bypass technique can be summarized as:

- Track the tendency of blocks to become dead after being accessed by particular instructions.
- Predict a block dead if the instruction accessing it often leads to blocks becoming dead. The prediction is stored as one bit of meta-data per cache block.
- When a victim block is needed, choose a predicted dead block before falling back to the default least-recently-used (LRU) policy.
- Do not place a predicted dead block into the cache, i.e., predicted dead blocks bypass the cache.

3 Related Work

Lai *et al*. [4] observed that, if a given sequence of accesses to a given cache block leads to the death (i.e. last access) of the block, then that same sequence of accesses to a different block is likely to lead to the death of that block. Dead block predictors that exploit this idea are called *trace based predictors* [5]. Dead blocks can also be predicted depending on how many times a block has been accessed. Kharbutli and Solihin propose counting based predictors for the L2 cache [3].

3.1 Differences from Previous Work

Trace based predictors rely on consistent sequences of addresses leading to block death, somewhat the same way that correlating branch predictors rely on sequences of previous branch outcomes leading to a particular current branch outcome [9]. The intuition is that natural program behavior leads to regular, predictable sequences of accesses to blocks, ending in the last use of a block.

However, for the CRC contest, regular sequences of block accesses are not seen by the last-level cache (LLC) as they are filtered by a relatively large L2 cache. Thus, it makes more sense to use only one instruction address to predict whether a block is dead. This is analogous to bimodal or "Smith" branch predictors tracking the bias of a branch without regard to other branches [8].

The contest only allows eight bits of extra state per cache block, which is not enough to keep trace or counting information for each block. Thus, we use a small number of *sampler sets* that abstract program behavior over the entire cache. This precludes the use of counting-based predictors that need to keep timing information with cache blocks, but allows for a sampling trace-based predictor.

4 Structures and Algorithm

4.1 Sampler Sets

We choose certain sets as sampler sets. For these sets, we store additional meta-data in a separate structure we call the *sampler*. For each block in each sampler set of the sampler, we keep the following information: 1) A partial tag, consisting of the lower order bits of the real tag for this block, 2) A *trace*, giving lower order bits of the address of the last instruction to access this block, 3) A one-bit prediction, where "true" means the block is predicted dead, 4) An LRU stack position for this block, and 5) A valid bit.

Each time a block from a sampler set is accessed, the corresponding sampler set is updated. Sampler set entries are replaced with a dead block replacement policy backed

with a default LRU replacement policy. The LRU policy is implemented the same was as the default LRU policy provided by the organizers: each sampler set entry keeps track of its position in the LRU stack.

Sampler sets are distributed uniformly across the cache. The sampler has lower associativity than the LLC to allow for early discovery of dead blocks. Note that the blocks tracked by the sampler may or may not be the same blocks in the corresponding cache sets.

4.2 The Prediction Table(s)

4.2.1 One Table

Prediction is done using a table of saturating counters. The trace of a block is used to index the table. When a block in the sampler is evicted, the corresponding table entry is incremented. When a block in the sampler is used, the corresponding entry is decremented. When any block in the cache is accessed, the trace is computed (but not stored) and used to index the table. If the corresponding counter meets a certain threshold, then the block is predicted dead. When a block might be placed in the cache, the trace is computed and used to access the predictor table. If the block is predicted dead, then it is bypasses the cache.

4.2.2 Multiple Tables

To increase predictor accuracy, we use more than one table. A different hash function of the trace is used to form an index into each table. This idea is inspired by skewed caches and skewed branch predictors [7, 6]. Now, we predict a block dead only if the sum of the corresponding counters from each table exceeds some threshold. Summing different confidence counters for improved confidence estimation was suggested by Jiménez [2].

4.2.3 Different Update Methods

False positive predictions (i.e., predicting a block dead when it is not dead) are more harmful than false negatives, but using up/down counters treats these mispredictions equally. Thus, we enhance the predictor to decrease counters exponentially (i.e. halve them) and increase them linearly (i.e. count up by one) for odd-numbered tables, and we retain the up/down counters for even-numbered tables. This idea was inspired by the miss-distance counters used for branch prediction confidence from Jacobsen *et al.* [1].

4.3 The Algorithm

4.3.1 Accessing a Cache Block

The following events take place when an LLC block B is accessed.

- If B's set corresponds to a sampler set, the sampler is accessed.
- The default LRU policy is updated, moving B to the most-recently-used (MRU) position.
- The predictor is consulted with a trace constructed with the PC used to access B and a prediction is stored for B.

This process adds no latency to the normal access time of the LLC. In the common case, the predictor can be accessed in parallel with the cache tag array. In the 5% of cases that B's set is a sampler set, the access to the predictor must be serialized with the sampler's access the the predictor (assuming only one read/write port for each prediction table). However, this can be done in parallel with whatever magical logic is computing the LRU stack position for each cache block in the LLC.

4.3.2 Replacing a Cache Block

The following events take place when finding a victim in an LLC set S to replace with a block B:

- The set is searched for the LRU block.
- The set is searched for a predicted dead block.
- The predictor is consulted with a trace constructed from the PC used to access B. If B is predicted dead, then it bypasses the cache, i.e., no victim is selected.
- If B is not predicted dead, then if there is a predicted dead block in S, then that block is the victim. Otherwise, the LRU block is the victim.

This process adds virtually no latency to LRU replacement. The search for a predicted dead block and the LRU block can be carried out in parallel if the prediction bits are kept in a separate structure. The prediction and decision to bypass can also be carried out in parallel with the search.

4.3.3 Accessing the Predictor

The following events take place when the predictor is accessed:

- Each of three prediction tables is accessed with a different hash of a trace.
- The sum of the confidence counters from each table is computed.
- If this sum is at least some threshold, then the block is predicted dead.

This process is similar to the access of a *gskew*-style branch predictor [6]. The three hashes can be computed in parallel. The particular hash used in our entry mixes some constants with bits from the PC using addition and subtraction. It is likely that computing the hash, accessing the tables, and computing the final sum would consume

Parameter Name	Parameter Value (single-core)	Parameter Value (four-core)
Partial Tag Bits per Sampler Block	16	16
Trace Bits per Sampler Block	16	16
Bits per Saturating Counter	2	2
Counters per Table	4,096	16,384
Number of Prediction Tables	3	3
Sampler Associativity	12	13
Threshold	8	8
Number of Sampler Sets	55	200

Table 1: Best Parameter Values for single and four core configurations.

Item	Computation		Number of Bits	
	Single Core	**Four Core**	**Single Core**	**Four Core**
LRU stack position bits for LLC	$4 \times 1,024 \times 16$	$4 \times 4,096 \times 16$	$65,536$	$262,144$
Prediction bits for cache blocks	$1 \times 1,024 \times 16$	$1 \times 4,096 \times 16$	$16,384$	$65,536$
LRU stack position bits for sampler	$4 \times 55 \times 12$	$4 \times 200 \times 13$	$2,640$	$10,400$
Partial tag bits for sampler	$16 \times 55 \times 12$	$16 \times 200 \times 13$	$10,560$	$41,600$
Trace bits for sampler	$16 \times 55 \times 12$	$16 \times 200 \times 13$	$10,560$	$41,600$
Prediction bits for sampler	$1 \times 55 \times 12$	$1 \times 200 \times 13$	660	$2,600$
Valid bits for sampler	$1 \times 55 \times 12$	$1 \times 200 \times 13$	660	$2,600$
Prediction tables bits	$3 \times 2 \times 4,096$	$3 \times 2 \times 16,384$	$24,576$	$98,304$
Total bits for our entry			**131,576**	**524,784**
Total bits allowed by contest	$1,024 \times 16 \times 8 + 1024$	$4,096 \times 16 \times 8 + 1024$	$132,096$	$525,312$
Surplus bits			520	528

Table 2: Computation of storage for single and four core configurations.

about two or three clock cycles, which should quite comfortably meet the constraints of a last-level cache. Predictor access could be pipelined to provide one prediction per cycle, providing the tables can each be accessed in one cycle.

4.3.4 Accessing the Sampler

About 5% of accesses to the LLC cause the sampler to be accessed as well. The sampler is like a little cache with only meta-data. When the sampler is accessed for a set S and tag t, the following events take place:

- The sampler's tag array is searched for t in the sampler set corresponding to S.
- If there is a hit in the sampler, then the predictor is accessed using the last trace to access t and the saturating counters are decreased according to the method in Section 4.2.3.
- If there is a miss in the sampler, then a victim is chosen in the sampler for the placement of t. Victims are chosen in this order: invalid blocks, predicted dead blocks, LRU blocks. The predictor is accessed using the last trace to access the victim tag and the saturating counters are incremented. Tags never bypass the sampler.
- The sampler entry is filled with: 1) the tag t, 2) a trace for t using the current PC, 3) a prediction using that trace, and 4) a true valid bit.

This process involves the most sequential accesses. The sampler tag array is accessed to determine whether there is a hit, then the predictor is written, then the predictor is read, then the sampler entry is filled. This process could take a few cycles, but since the sampler is far smaller than the LLC's tag array, with smaller tags, less associativity, and far fewer sets, and the sampler is accessed on only 5% of LLC accesses, we believe this latency will be quite tolerable.

5 Choosing Parameters for the Contest

We used the 29 SPEC CPU 2006 benchmarks and the simulation infrastructure provided by the conference organizers to choose the best parameters for our algorithm. Table 1 shows the values of these parameters.

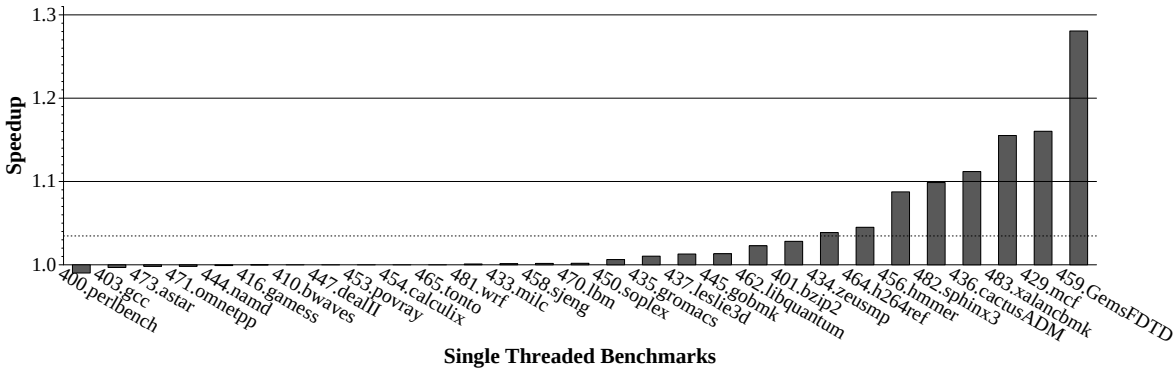

Figure 1: Speedup for single-core workloads. The dotted line is the geometric mean.

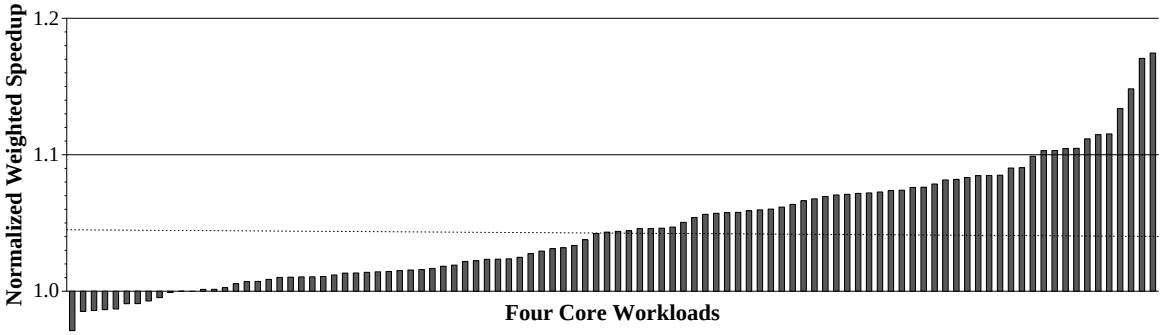

Figure 2: Speedup for multi-core workloads. The dotted line is the geometric mean.

6 The Size of the Cache Replacement State

Using the parameters from Table 1 and other information, we can compute the total number of bits consumed by the 1MB and 4MB caches for the contest. Table 2 shows this computation for both configurations. Each configuration leaves over 500 bits surplus, which should leave ample room for any local variables or run-time constants the organizers would like to count against our budget.

7 Results

Table 1 shows the speedup of our technique over a baseline LRU policy for a 1MB cache. The geometric mean speedup is 3.5%.

Table 2 shows the weighted speedup of our technique normalized to the LRU policy for a 4MB cache for 100 randomly chosen combinations (without duplication) of four chosen from the 29 SPEC CPU 2006 benchmarks. The weighted speedup is computed with respect to a baseline 4MB LRU cache with only one benchmark running. The geometric mean normalized speedup is 4.5%.

References

[1] Erik Jacobsen, Eric Rotenberg, and James E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 142–152, December 1996.

[2] Daniel A. Jiménez. Composite confidence estimators for enhanced speculation control. In *Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2009)*, pages 161–168, October 2009.

[3] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[4] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. *SIGARCH Comput. Archit. News*, 29(2):144–154, 2001.

[5] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[6] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, June 1997.

[7] André Seznec. A case for two-way skewed-associative caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178, New York, NY, USA, 1993.

[8] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, May 1981.

[9] T.-Y. Yeh and Yale N. Patt. Two-level adaptive branch prediction. In *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, November 1991.