Multiperspective Reuse Prediction

Abstract

Multiperspective reuse prediction is a technique that predicts the future reuse of cache blocks using several different types of features. We demonstrate the technique using a placement, promotion, and bypass optimization.

1 Introduction

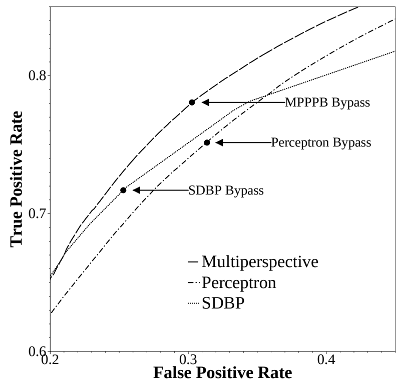

Figure 1: Receiver operating characteristic (ROC) curves for three reuse predictors.

This paper introduces *multiperspective reuse prediction*. We propose using many features that examine various properties of program and memory behavior to give a prediction informed from multiple perspectives. The features are drawn from an abstract set of seven parameterized features, each tracking a distinct property related to block reuse. The result is a highly accurate reuse predictor capable of driving a cache management optimization, Multiperspective Placement, Promotion, and Bypass (MPPPB).

Figure 1(a) shows a portion of the receiver operating characteristic (ROC) curves for three reuse predictors: sampling-based dead block prediction (SDBP) [4], perceptron-learning-based reuse prediction (hereafter, Perceptron) [8], and our multiperspective technique. On an access to a block, each predictor gives a confidence value. If this value exceeds some threshold, the block is classified as "dead," *i.e.*, it is predicted not to be reused before it is evicted. ROC curves plot the false positive rate against the true positive rate for all the feasible threshold values. The false positive rate is the fraction of live blocks that are mispredicted as dead, while the true positive rate is the fraction of dead blocks that are correctly predicted. A higher false positive rate is a liability as it increases the chances of a cache miss, while a higer true positive rate increases the opportunity for applying the optimization.

We consider an aggressive bypass optimization. The reuse predictor decides whether a block brought to the LLC from DRAM is "dead-on-arrival," *i.e.* it will not be reused in the cache. In this case, the block is not placed in the LLC, but bypassed to the core cache. For each predictor, the false positive rate giving the best performance is between 25% and 31%, with the exact value decided by the granularity of the possible threshold values. At every point in this region, the multiperspective technique provides a lower false positive rate and higher true positive rate, resulting in higher performance for the bypass optimization.

2 Multiperspective Reuse Prediction

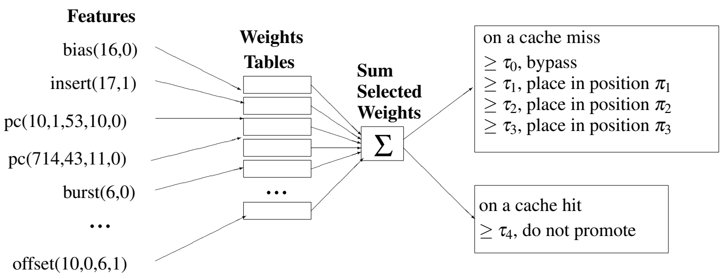

Figure 2: Prediction drives bypass, placement, and promotion.

In this section we present the multiperspective reuse prediction technique. The predictor is organized as a hashed perceptron predictor [6] indexed by a diverse set of features and trained with a modified version of perceptron learning that allows features to be parameterized by variable associativities.

2.1 Combining Multiple Features

As with perceptron-based reuse prediction, our predictor combines multiple features. To make a prediction, each feature is used to index a distinct table of 6-bit integer weights that are then summed. The sum is used as a confidence estimate to drive three cache management decisions: bypass, placement, and promotion. A *sampler* is used to train the predictor using accesses to a small number of cache sets in a manner similar to SDBP [4]. The sampler can be thought of as a simulator for a subset of the cache. It is managed with LRU replacement. Each access to the sampler presents an opportunity to train the predictor, as the sampler keeps track of the vector of input features used to estimate the confidence on the last access to that block. If the sum is below 0 for dead (evicted) blocks, or above 0 for live (reused) blocks, the corresponding counters are incremented or decremented, respectively. Unlike previous work, the training of each table is selective based on an associativity parameter for the feature used for that table: a block might be considered dead for one table but live for another. Thus, for a given training opportunity, some counters may be incremented, some left alone, and some decremented. The magnitudes of the weights is proportional to the correlation of the feature to block reuse for that access. Weights with values near 0 contribute very little to the sum, while weights with large values can affect the sum much more. Thus, the prediction can be thought of as an aggregate of many predictions taking into account each prediction's confidence.

2.2 The Features

We introduce seven parameterized features used to form indices into the prediction tables. The range of the features is from 1 to 8 bits. We have found that each feature has some correlation with whether a block will be reused. The features are very general, with thousands of possible parameterizations. In Section 3 we describe our heuristics for finding a good set of features and parameters.

Each feature has as its first parameter the LRU stack position (A) beyond which a block is considered dead for the purpose of training the corresponding table. The original SDBP work found that a sampler with a different associativity from the main cache improved predictor accuracy [4]; in this work, we extend this observation to each feature. Intuitively, a block passing a certain stack position may signal a high probability that it is on its way to being evicted, so considering a variety of such positions enriches the feature space for the predictor.

Each feature also has as its last parameter a mask (X) that. If bit 0 of X is true, the PC of the current memory instruction is exclusive-ORed with the feature bits, allowing features to be distributed across the weights and exploit correlations between features and PCs. If bit 1 of X is true and the memory access is a prefetch, then bit 1 of the index is toggled, giving the predictor the information that this feature is being used to predict a prefetch.

Following is a list of the features with their parameters:

pc(A, B, E, W, X): Bits B to E of the PC of the W^{th} current memory access instruction. A is the recency stack position beyond which a block is considered dead.

address(A, B, E, X). Bits B to E of the physical address for the memory access.

bias(A, X). The value 0. If X is false, this feature yields a simple global counter that is incremented when any block goes beyond position A and decremented with it is accessed below position A, tracking the general short-term bias of blocks to be dead or live. If X is true, this feature yields a traditional PC-based predictor like SDBP or SHiP [9], tracking the tendency of a given PC to lead to dead or live blocks.

burst(A, X). Single bit that is 1 if and only if this access is to a most-recently-used (MRU) block, indicating a *cache burst* [5].

insert(A, X). Single bit that is 1 if and only if this access is an insertion, *i.e.* the block is being placed in the cache as a result of a cache miss.

lastmiss(A, X). Single bit that is 1 if and only if the last access to the cache set in question was a miss.

offset(A, B, E, X). Bits B to E of the block offset of this memory access.

2.3 The Sampler

Our predictor uses a sampler [3,4,8]. A small number of sets in the main cache are *sampled sets* for which a set of partial tags and other metadata are kept and managed with LRU replacement. When a sampled set is accessed, the corresponding set in the sampler is also accessed and used to train the predictor.

Each feature has an associativity (A) parameter. When a sampled set is accessed, if the access causes a block to be demoted beyond the associativity specified for a given feature, the table for that feature is trained that the block is dead. If a sampled block is accessed beyond the associativity for a given feature, the table for that feature is *not* trained that the access is a reuse, since it would have been a miss if the cache had associativity A.

We empirically determine that the best trade-off between number of sets and associativity yields a sampler with 18 ways. Each entry in a sampled set consists of the following fields:

1. A partial tag to identify the block. We find that using 16 bits for each tag gives a good trade-off between accuracy and making best use of hardware resources.

2. A 9-bit signed integer giving the most recent confidence value for that block.

3. The vector of indices into the prediction tables that were used to compute the current confidence value for that block. These indices are used to index the prediction tables for training. No table is larger than 256 entries requiring an 8-bit index, and some are very small, e.g. 1 entry for a *bias*(A,0) feature requiring no index bits.

4. Four bits storing the LRU recency stack position for that block.

2.4 Predictor Organization

The predictor is organized as a set of independently indexed tables, one per feature. Each table has a small number of weights. Features that use the PC, physical address, or exclusive-OR with the PC generate 8-bit indices requiring 256 weights per table. The *offset* feature requires up to 64 weights. Single-bit features such as *insert*, *burst*, *lastmiss* require two weights per table. The *bias* feature requires one weight. We find that 6 bit weights ranging from -32 to +31 provide a good trade-off between accuracy and area. A vector of feature values is kept per core and updated on every memory access. The *lastmiss* feature requires keeping a single extra bit for every set.

2.5 Driving Bypass, Placement, and Promotion

On a last-level cache access, the predictor is consulted. Each feature is used to produce an index into a distinct prediction table to select a weight. The weights are summed to produce a confidence value.

On a cache miss, the confidence value is used to decide whether the bypass the block or place it in one of three recency positions π_1, π_2, or π_3 in the set. Four thresholds are used to guide this decision: τ_0, τ_1, τ_2, and τ_3. If the confidence value exceeds τ_0, the block is bypassed; otherwise, the block is placed in position π_i such that the confidence value exceeds τ_i but not τ_{i-1}. If the confidence value is below τ_4 it is placed in the position corresponding to most-recently-used.

On a cache hit, if the value exceeds a threshold τ_4, then the block is not promoted but rather remains in the same recency position it previously occupied.

2.6 Default Replacement Policies

In this work, we explore two default replacement policies: static minimal disturbance placement and promotion (static MDPP) [7], and static re-reference interval prediction (SRRIP) [2]. Static MDPP uses tree-based pseudoLRU with an enhanced promotion policy. In a 16-way cache, MDPP allows placement or promotion into one of 16 distinct recency positions. SRRIP classifies blocks into one of four recency positions, initially placing blocks into a less favorable position and then promoting them as they are accessed. We tune our thresholds and positions (π_i) to minimize misses for each default replacement policy.

2.7 Training the Predictor

When a sampled set is accessed, the predictor has an opportunity to be trained. Evictions from the sampler have no special significance because each feature has its own maximum recency position (the *A* parameter). Thus, the only event triggering training is an access that places or hits in the sampler. Such an access may cause several instances of training: one for the block that is placed or reused, and more for any block demoted beyond a particular feature's *A* parameter.

Training on Block Placement or Reuse When a block is placed or reused in the sampler, the vector of feature indices for that block is used to index each table. For each feature F_i, the corresponding prediction table T_i may be trained. Suppose a reused block occupies recency position p. If p is less than the *A* parameter for F_i, then the selected weight in T_i is incremented with saturating arithmetic.

Training on Block Demotion After a reused block is used to train the predictor, it is promoted to the MRU position according to the LRU replacement policy. This promotion may result in the demotion of other blocks. For each feature F_i, if a block is demoted to that feature's *A* parameter, it is treated as an eviction for the purposes of that feature. The weight in T_i indexed in the vector of feature indices for that demoted block is decremented with saturating arithmetic.

(a)	(b)
offset(15,1,6,1)	*insert*(15,0)
pc(7,14,43,11,0)	*insert*(16,1)
pc(16,3,11,16,1)	*insert*(6,1)
insert(16,1)	*offset*(14,0,7,2)
offset(10,0,6,1)	*bias*(17,3)
pc(10,1,53,10,0)	*burst*(8,2)
bias(16,0)	*pc*(6,5,48,0,3)
insert(8,1)	*lastmiss*(15,2)
pc(17,6,20,0,1)	*address*(17,1,32,2)
burst(6,0)	*pc*(17,6,20,0,1)
lastmiss(9,0)	*pc*(6,4,11,2,2)
pc(17,6,20,0,1)	*bias*(13,2)
insert(16,0)	*offset*(8,1,6,2)
insert(17,1)	*pc*(6,5,77,4,1)
pc(16,8,16,5,0)	*address*(11,8,19,0)
pc(17,6,20,14,1)	*address*(16,8,16,0)

Table 2: Single-thread features for configurations 1 (a) and 2 (b)

2.8 Overhead for Predictors

Table 1 shows the number of bytes used for each of the configurations. It gives the number of bits for each major structure, then sums the number of bits into a total number of bytes. For each configuration, the hardware budget does not exceed 32KB per core.

3 Developing Features and Parameters

We empirically determined that a set of 16 features provided enough diversity of features to yield good accuracy while not requiring too much hardware. For each configuration (single vs. multi, prefetching vs. non-prefetching) we generate 4,000 sets of 16 features randomly, then refine the set of features that minimizes MPKI over a set of benchmarks with hill-climbing. Tables 2 and 3 show the final set of features used for the competition. The four thresholds τ_0, τ_1, τ_2, τ_3, and τ_4 as well as the three placement positions π_1, π_2, and π_3 are tuned empirically. As an optimization, we tune two values of τ_0 and set-duel between them.

4 Results

We compare our proposed technique against two recent proposals: Hawkeye [1] and perceptron-learning-based reuse prediction [8]. For both techniques, we used code generously provided by the original authors. Figure 3 gives misses per 1,000 instructions (MPKI) for 900 4-core workloads that were not used in developing the features. Prefetching is enabled. MPPPB yields an average 10.8 MPKI. Hawkeye gives 11.7 MPKI, Perceptron gives 12.5 MPKI, and LRU gives 14.1 MPKI.

Configuration 1:
2,048 × 15 = 30,720 plru bits
2,048 lastmiss bits
table 0: 6 × 256 = 1,536 bits
table 1: 6 × 256 = 1,536 bits
table 2: 6 × 256 = 1,536 bits
table 3: 6 × 256 = 1,536 bits
table 4: 6 × 256 = 1,536 bits
table 5: 6 × 256 = 1,536 bits
table 6: 6 × 2 = 12 bits
table 7: 6 × 256 = 1,536 bits
table 8: 6 × 256 = 1,536 bits
table 9: 6 × 2 = 12 bits
table 10: 6 × 2 = 12 bits
table 11: 6 × 256 = 1,536 bits
table 12: 6 × 2 = 12 bits
table 13: 6 × 256 = 1,536 bits
table 14: 6 × 256 = 1,536 bits
table 15: 6 × 256 = 1,536 bits
100 trace bits for global trace buffer
129 × 18 ways × 80 sampled sets = 185,760 sampler bits
53 × 16 = 848 bits for global address array
10 bits for set dueling counter
total bytes: 29,751

Configuration 2:
2,048 × 15 = 30,720 plru bits
2,048 lastmiss bits
table 0: 6 × 2 = 12 bits
table 1: 6 × 256 = 1,536 bits
table 2: 6 × 256 = 1,536 bits
table 3: 6 × 256 = 1,536 bits
table 4: 6 × 256 = 1,536 bits
table 5: 6 × 4 = 24 bits
table 6: 6 × 256 = 1,536 bits
table 7: 6 × 4 = 24 bits
table 8: 6 × 256 = 1,536 bits
table 9: 6 × 256 = 1,536 bits
table 10: 6 × 256 = 1,536 bits
table 11: 6 × 4 = 24 bits
table 12: 6 × 256 = 1,536 bits
table 13: 6 × 256 = 1,536 bits
table 14: 6 × 256 = 1,536 bits
table 15: 6 × 256 = 1,536 bits
96 trace bits for global trace buffer
125 × 18 × 80 sets = 180,000 sampler bits
77 × 16 = 1,232 bits for global address array
10 bits for set dueling counter
total bytes: 29,078

Configuration 3:
262,144 RRPV bits
8,192 lastmiss bits
table 0: 6 × 2 = 12 bits
table 1: 6 × 256 = 1,536 bits
table 2: 6 × 4 = 24 bits
table 3: 6 × 256 = 1,536 bits
table 4: 6 × 256 = 1,536 bits
table 5: 6 × 256 = 1,536 bits
table 6: 6 × 256 = 1,536 bits
table 7: 6 × 256 = 1,536 bits
table 8: 6 × 256 = 1,536 bits
table 9: 6 × 256 = 1,536 bits
table 10: 6 × 256 = 1,536 bits
table 11: 6 × 256 = 1,536 bits
table 12: 6 × 256 = 1,536 bits
table 13: 6 × 256 = 1,536 bits
table 14: 6 × 256 = 1,536 bits
table 15: 6 × 256 = 1,536 bits
106 trace bits for global trace buffer
135 × 18 ways × 308 sets = 748,440 sampler bits
54 × 16 × 4 = 3,456 bits for per-core address arrays
10 bits for set dueling counter
total bytes: 130,486

Configuration 4:
262,144 RRPV bits
8,192 lastmiss bits
table 0: 6 × 2 = 12 bits
table 1: 6 × 256 = 1,536 bits
table 2: 6 × 256 = 1,536 bits
table 3: 6 × 256 = 1,536 bits
table 4: 6 × 256 = 1,536 bits
table 5: 6 × 4 = 24 bits
table 6: 6 × 4 = 24 bits
table 7: 6 × 256 = 1,536 bits
table 8: 6 × 2 = 12 bits
table 9: 6 × 256 = 1,536 bits
table 10: 6 × 8 = 48 bits
table 11: 6 × 256 = 1,536 bits
table 12: 6 × 256 = 1,536 bits
table 13: 6 × 256 = 1,536 bits
table 14: 6 × 256 = 1,536 bits
table 15: 6 × 256 = 1,536 bits
87 trace bits for global trace buffer
116 × 18 × 337 sets = 703,656 sampler bits
23 × 16 × 4 = 1,472 bits for per-core address arrays
10 bits for set dueling counter
total bytes: 124,072

Table 1: Space overhead for the various configurations

bias(1,0)	*lastmiss*(9,0)
pc(16,9,25,9,1)	*pc*(8,6,8,14,3)
insert(8,2)	*bias*(13,3)
pc(6,9,28,12,1)	*offset*(8,2,2,2)
offset(14,1,4,3)	*address*(16,3,14,2)
lastmiss(7,1)	*burst*(16,2)
pc(10,1,54,13,3)	*insert*(10,2)
pc(10,3,32,5,1)	*pc*(14,3,18,10,2)
pc(14,5,24,0,1)	*insert*(6,0)
offset(13,4,4,2)	*pc*(17,1,14,5,0)
address(8,4,47,2)	*offset*(11,2,5,0)
address(2,24,32,1)	*offset*(15,0,7,3)
pc(12,10,30,0,1)	*address*(9,2,13,2)
pc(12,9,28,0,2)	*address*(15,4,34,2)
pc(12,5,31,2,2)	*offset*(10,0,6,1)
address(8,10,8,1)	*pc*(11,7,23,0,2)
(a)	**(b)**

Table 3: Multi-core features for configurations 3 (a) and 4 (b)

References

[1] A. Jain and C. Lin, "Back to the future: Leveraging Bélády's algorithm for improved cache replacement," in *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture*, ser. ISCA '16, 2016, pp. 78–89.

[2] A. Jaleel, K. Theobald, S. S. Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA-37)*, June 2010.

[3] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *In Proceedings of the 25th International Conference on Computer Design (ICCD-2007)*, 2007, pp. 245–250.

[4] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2010, pp. 175–186.

[5] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 222–233.

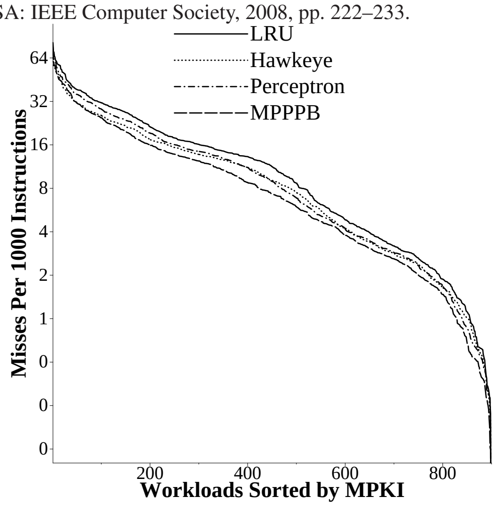

Figure 3: Misses per 1,000 Instructions for 900 4-Core Multi-Programmed Workloads

[6] D. Tarjan, K. Skadron, and M. Stan, "An ahead pipelined alloyed perceptron with single cycle access time," in *Proceedings of the Workshop on Complexity Effective Design (WCED)*, June 2004.

[7] E. Teran, Y. Tian, Z. Wang, and D. A. Jiménez, "Minimal disturbance placement and promotion," in *2016 IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA)*, March 2016.

[8] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *Proceedings of the 49th ACM/IEEE International Symposium on Microarchitecture (MICRO-49)*, October 2016.

[9] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, J. Simon C. Steely, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 430–441.