Multiperspective Reuse Prediction

Daniel A. Jiménez
Texas A&M University
djimenez@acm.org

Elvira Teran
Intel Labs
elvira.teran@intel.com

ABSTRACT

The disparity between last-level cache and memory latencies motivates the search for efficient cache management policies. Recent work in predicting reuse of cache blocks enables optimizations that significantly improve cache performance and efficiency. However, the accuracy of the prediction mechanisms limits the scope of optimization.

This paper introduces *multiperspective reuse prediction*, a technique that predicts the future reuse of cache blocks using several different types of features. The accuracy of the multiperspective technique is superior to previous work. We demonstrate the technique using a placement, promotion, and bypass optimization that outperforms state-of-the-art policies using a low overhead. On a set of single-thread benchmarks, the technique yields a geometric mean 9.0% speedup over LRU, compared with 5.1% for Hawkeye and 6.3% for Perceptron. On multi-programmed workloads, the technique gives a geometric mean weighted speedup of 8.3% over LRU, compared with 5.2% for Hawkeye and 5.8% for Perceptron.

CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; • **Hardware** → *Static memory*;

KEYWORDS

cache management, locality, prediction, microarchitecture

ACM Reference format:
Daniel A. Jiménez and Elvira Teran. 2017. Multiperspective Reuse Prediction. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 13 pages.
https://doi.org/10.1145/3123939.3123942

1 INTRODUCTION

LLC misses are a significant source of performance loss because of the large number of CPU cycles required to access DRAM. Recent work has focused on predicting whether a block will be reused in the last-level cache (LLC). If blocks with a low probability of reuse can be accurately identified, they can be quickly evicted or even bypassed altogether, avoiding costly capacity misses by allowing blocks with more locality to remain in the cache. Previous work has identified features correlated with block reuse that may be used in the design of reuse predictors. Each proposed technique uses one or two features resulting in increasingly accurate reuse predictors.

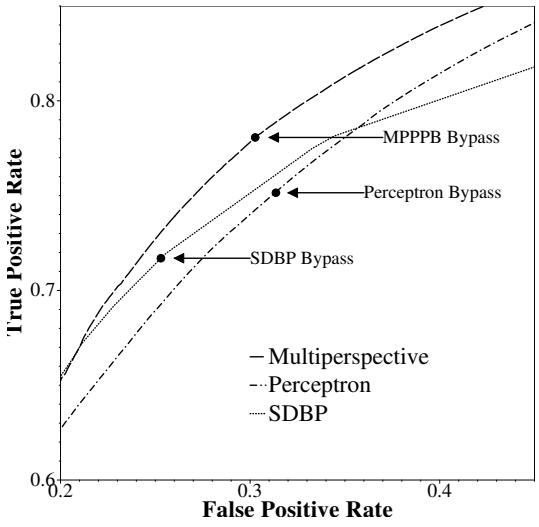

Figure 1: Receiver operating characteristic (ROC) curves for three reuse predictors. In a range of false positive rates allowing for a bypass optimization, multiperspective prediction identifies more true positives (dead blocks).

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 8(a) shows a portion of the receiver operating characteristic (ROC) curves for three reuse predictors: sampling-based dead block prediction (SDBP) [15], perceptron-learning-based reuse prediction (hereafter, Perceptron) [28], 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

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123942

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 higher 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.

1.1 Contributions

This paper makes the following contributions:

(1) It introduces a last-level cache block reuse predictor based on combining seven parameterized features correlated with block reuse. The features and selection of their parameters for accuracy are described.
(2) Each feature may use a different recency position for deciding the last access to a given block. Thus, each feature may simulate a different associativity, allowing the set of parameterized features to be greatly expanded.
(3) The predictor drives an optimization for three aspects of cache management: block placement, replacement, and bypass. On a set of 33 single-thread benchmarks, the technique yields a geometric mean 9.0% speedup over LRU, compared with 5.1% for Hawkeye [11] and 6.3% for Perceptron [28], two state-of-the-art cache management policies. On a set of 900 4-core multiprogrammed workloads, the technique yields a geometric mean weighted speedup of 8.3% over LRU, compared with 5.2% for Hawkeye and 5.8% for Perceptron.

2 BACKGROUND AND RELATED WORK

This section reviews recent work in reuse prediction.

Reuse Distance Prediction. Recent work proposes techniques to determine the distance to the next reuse of a block. Re-reference Interval Prediction (RRIP) [12] is an efficient implementation of reuse-distance prediction [14]. RRIP groups blocks into recency categories based on the predicted interval to the next reference [12]. Static RRIP (SRRIP) places all incoming blocks into the same category, while Dynamic RRIP (DRRIP) uses set-dueling [23] to adapt the placement position to the particular workload. RRIP is simple, requiring little overhead and no complex prediction structures, while resulting in significant performance improvement. The multi-programmed version of our technique uses SRRIP with two-bit re-reference interval values as the default replacement policy.

Dead Block Prediction. Dead block predictors attempt to detect whether a block will be referenced again before it is evicted. There are numerous dead block predictors applied to a variety of applications in the literature [1, 10, 15–19, 25, 29]. Our work uses of the notion of a *sampler* from Sampling Dead Block Prediction (SDBP) [15]. The sampler structure keeps partial tags of sampled sets corresponding to a set in the cache. When a sampled cache set is accessed, the matching set in the sampler will be accessed as well. Three tables of two-bit saturating counters are accessed using a technique similar to a skewed branch predictor [20]. When a block is hit in a sampled set, the program counter (PC) of the relevant memory instruction is hashed into the tables and the corresponding counters are decremented. For each eviction from a sampled set, the counters corresponding to the PC of the last instruction to access the victim block are incremented. To make the predictions for the rest of the blocks in the LLC, when there is an access, the predictor is consulted by hashing the PC of the memory access instruction into the tables and taking the sum of the indexed counters. If the sum exceeds some threshold, the accessed block is predicted dead. The sampler is managed by the LRU policy with a reduced associativity from that of the LLC. The SDBP work applies the predictions to a replacement and bypass optimization.

Perceptron Learning for Reuse Prediction. Teran *et al.* propose using perceptron learning for reuse prediction (hereafter Perceptron) driving a bypass and replacement optimization [28]. In that work, perceptron learning is used to set weights selected by hashes of multiple features consisting of the PC, several recent PCs, and two different shifts of the tag of the referenced block. Each feature is hashed to index its own table of weights. The selected weights are summed and thresholded to make a prediction. When a sampled block is reused or evicted, the corresponding weights are decremented or incremented, respectively, according to the perceptron learning rule. That work is inspired by the organization of the hashed perceptron branch predictor [26]. The structure of our predictor is similar to Perceptron, but the set of features is greatly expanded, and the confidence value produced by the predictor is used for block placement as well as bypass and replacement. Crucially, Perceptron requires keeping an extra bit per block to keep track of whether a block is dead, where our technique uses block placement position to implicitly record whether a block has been predicted dead.

Features Correlating with Reuse. Several features correlate with block reuse in caches. The sequence, or *trace*, of memory access instruction addresses (PCs) leading to a block's use is highly correlated with whether that block will be used again [18, 19]. Accesses to LLCs are filtered by first- and second-level caches, so this trace does not provide good accuracy for the LLC [15]. The PC of the memory instruction that last touched the block in question is a simpler feature. The PC gives good accuracy in the LLC, but since there

are multiple paths to the same PC, some information about program behavior is lost, limiting the accuracy that can be achieved. Bits from the memory address can also be used [19]. Typical programs reference many more memory addresses than PCs, so there is a trade-off between memory address bits used as input features and destructive interference and training times in prediction tables. Other features such as the compressibility of a block [21] or reference counts [16] have been proposed; however, these ideas have a significant overhead in terms of complexity or space overhead.

Combining Features. Bits from the memory address and PC traces can be combined by appending or hashing to generate an index into the predictions table [19]. However, this combination risks destructive interference and longer training times in the predictor as each possible pattern may index a different counter. Perceptron learning for reuse prediction [28] uses recent PCs and bits extracted from the memory address to index distinct tables, mitigating the effects of interference in a given table in a way similar to hashed perceptron branch predictors [26]. Our work uses the same idea of multiple distinct tables, but greatly augments the set of available features.

3 MULTIPERSPECTIVE REUSE PREDICTION

In this section we present the multiperspective reuse prediction technique. The predictor is organized as a hashed perceptron predictor [26] 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.

3.1 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 [15]. The sampler is like a simulator for a subset of the cache. It is managed with LRU replacement. Each access to the sampler is 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.

3.2 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 5 we describe our heuristics for finding a good set of features and parameters.

The first feature for each parameter is the LRU stack position (A) beyond which a block is considered dead for the purpose of training the corresponding table. Our original SDBP work found that a sampler with a different associativity from the main cache improved predictor accuracy [15]. 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 Boolean (X) that, if true, causes the PC of the current memory instruction to be exclusive-ORed with the feature bits, allowing features to be distributed across the weights and exploit correlations between features and PCs.

Following is a list of the features with their parameters:

(1) *pc*(A, B, E, W, X). This feature is bits B to E of the PC of the W^{th} most recent memory access instruction. As with every parameter, A is the recency stack position beyond which a block is considered dead for this feature, and if X is true then the feature is XORed with the PC of the current memory access instruction. As described above, the PC and history of PCs is known to correlate with block reuse. A "fake" PC address is used for all hardware prefetches.

(2) *address*(A, B, E, X). This feature is bits B to E of the physical address for the memory access. Bits from the address are known to correlate with block reuse, as certain regions of memory are reused more than others.

(3) *bias*(A, X). This feature is simply 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 [29], tracking the tendency of a given PC to lead to dead or live blocks.

(4) *burst*(A, X). This single-bit feature is 1 if and only if this access is to a most-recently-used (MRU) block. Successive accesses to an MRU block, or a *cache burst* [19], may signal high locality for given access and thus affect the probability of reuse.

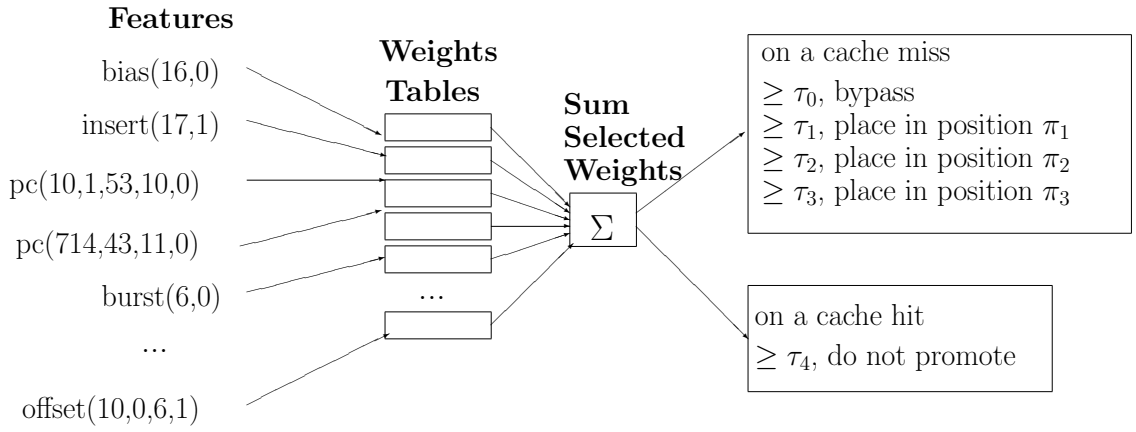

Figure 2: Prediction drives bypass, placement, and promotion. When a block is placed or reused, the predictor is consulted. Features index tables of weights that are summed to a confidence value. On a miss, the sum is thresholded to decide bypass or placement. On a hit, the block is not promoted if the sum exceeds a threshold.

(5) *insert*(A, X). This single-bit feature 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. The behavior of newly inserted blocks may differ from that of re-referenced blocks, so this feature may provide useful information about block reuse.

(6) *lastmiss*(A, X). This single-bit feature is true if and only if the last access to the cache set in question was a miss. This feature gives some indication of the pressure on this set, allowing the predictor to distinguish between hot and cold sets when estimating the probability of reuse.

(7) *offset*(A, B, E, X). This feature, from 1 to 6 bits in a system with 64B blocks, gives bits B to E of the block offset of this memory access. The offset may have some correlation with reuse because it can given an indication of which fields of an object are being access, and thus provide some insight into program behavior.

3.3 The Sampler

Our predictor uses a sampler inspired by similar structures in previous work [14, 15, 28]. A small number of sets in the main cache are designated as *sampled sets*. For each sampled set, a corresponding set of partial tags and other metadata are kept in the sampler and managed with LRU replacement. Each time a sampled set is accessed, the corresponding set in the sampler is also access and used to train the predictor.

In previous work, hits and evictions in the sampler determine how the predictor is trained. If a sampled block is evicted, the predictor is trained positively, *i.e.* the predictor learns that the block is dead and the counters corresponding to the input features are incremented. If a sampled block is accessed, the predictor is trained negatively with the corresponding counters being decremented.

In our sampler, we extend the notion of associativity to be a parameter to each feature. When a sampled set is accessed, if the access causes a block to be demoted beyond the recency position 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 recency position (A) 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.

Each set in the sampler has 18 ways; we find empirically that representing more than 18 possible recency positions uses hardware resources than the benefit it provides. Each entry in a sampled set consists of the following fields:

(1) A partial tag used to identify the block. Since access to the sampler do not have to be correct, it is permissible to allow a small number of distinct tags to map to the same block. We find that using 16 bits for each tag gives a good trade-off between low a aliasing rate and making best use of hardware resources.

(2) A 9-bit signed integer giving the most recently computed 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, so the number of bits needed for this vector is small.

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

3.4 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.

3.5 Making a Prediction

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.

3.6 Driving Bypass, Placement, and Promotion

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.

3.7 Default Replacement Policies

In this work, we explore two default replacement policies: static minimal disturbance placement and promotion (static MDPP) [27], and static re-reference interval prediction (SRRIP) [12]. 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.

3.8 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.

Complexity of Training. Although different A parameters may trigger training for several blocks on a given access to the sampler, the training can be completed in at most two rounds of table accesses: one for the reused block, and one for any demoted blocks. Since LRU recency positions are distinct, only one block at a time might be demoted to a given feature's A parameter. Thus, the second round of training requires no more than one access per table. **Note:** although MDPP or SRRIP are used in the main cache, only true LRU is used in the sampler.

4 METHODOLOGY

4.1 Performance Models

We model performance with an in-house simulator using the following memory hierarchy parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way, DRAM latency: 200 cycles. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. The single-thread simulations use a 2MB L3 cache while the multi-programmed simulations use an 8MB L3 cache. The simulator models a stream prefetcher. It starts a stream on a L1 cache miss and waits for at most two misses to decide on the direction of the stream. After that it starts to generate and send prefetch requests. It can track 16 separate streams. The replacement policy for the streams is LRU. This infrastructure enables collecting instructions-per-cycle figures as well as misses per kilo-instruction and reuse predictor accuracy information.

4.2 Workloads

We use the 29 SPEC CPU 2006 benchmarks as well as three server workloads from CloudSuite [7]: *data_caching*, *graph_analytics*, *sat_solver* as well as a machine learning workload *mlpack_cf* from mlpack [5]. We use SimPoint [22] to identify up to 6 segments (*i.e. simpoints*) of one billion instructions each characteristic of the different program phases for the SPEC and mlpack workloads. For CloudSuite, we fast-forward at least 30 billion instructions to get past the initialization phases. In total, we use 99 segments representing 33 benchmarks.

Single-Thread Workloads. For single-thread workloads, the results reported per benchmark are the weighted average of the results for the individual simpoints. The weights are generated by SimPoint and represent the portion of all executed

instructions for which a given simpoint is responsible. Each program runs the first `ref` input provided by the `runspec` command. For each run, the 500 million instructions previous to the simpoint are used to warm microarchitectural structures, then the subsequent one billion instructions are used to measure and report results.

Multi-Programmed Workloads. For 4-core multi-programmed workloads, we generated 1000 distinct workloads consisting of mixes from the 99 segments described above. We follow the sample-balanced methodology of FIESTA [9]. We select regions of equal standalone running time for each simpoint. Each region begins at the start of the simpoint and ends when the number of cycles in a standalone simulation reaches one billion cycles. Each workload is a mix of 4 of these regions chosen uniformly randomly without replacement. For each workload the simulator warms microarchitectural structures until 100 million total instructions have been executed, then measures results until each benchmark has executed for at least one billion additional cycles. When a thread reaches the end of its one billion cycle region, it starts over at the beginning. Thus, all 4 cores are active during the entire measurement period.

4.3 Cache Management Policies Simulated

We compare our proposed technique against two recent proposals: Hawkeye [11] and perceptron-learning-based reuse prediction [28]. For both techniques, we used code generously provided by the original authors. For single-thread benchmarks, we also simulate Bélády's optimal replacement policy (MIN) [3] adapted to also provide optimal bypass. (For multi-programmed workloads, the simulation of MIN is problematic due to the replacement policy's ability to affect the inter-thread ordering of memory operations.) There is voluminous other work in reuse prediction [1, 2, 6, 10, 12, 13, 15, 16, 18, 19, 27, 29]. We observe that Hawkeye and Perceptron provide the best performance out of previous work.

We explore two versions of multiperspective placement, promotion, and bypass (MPPPB). For single-thread workloads, we use a default static MDPP replacement policy. This policy requires only 15 bits per set, allowing for an inexpensive implementation. For multi-programmed workloads, we use a default SRRIP replacement policy. SRRIP provides performance comparable to MDPP, but because it has only four recency levels instead of 16, it allows for simpler design-space exploration of the space of threshold parameters. This simplicity is important because the design space exploration of multi-programmed workloads is far more compute-intensive than it is for single-thread benchmarks.

4.4 Overhead for Predictors

The area overhead for the multiperspective predictor is dominated by various tables. We design the structures to consume a very small fraction of the capacity of the cache, and ensure that Hawkeye and Perceptron use an equivalent amount of storage. In choosing the parameters affecting hardware budget, we target the single-core budget used by Hawkeye of 28KB. We choose 64 sampled sets per core. Each block in a sampler entry consists of the vector of feature indices, 9 bits of confidence value, 16 bits of partial tag, and 4 bits of LRU state. The sampler is 18-way set-associative.

Single-Thread Overhead. The set of single-thread features given in Table 1(b) uses 118 vector total index when taking into account the sizes of the various tables used for those features. The other single-core feature require fewer bits as they have more single-bit features; thus, we use (b) for the area estimate. Thus, the sampler consumes $(9 + 16 + 4 + 118) \times 80 \times 16 = 20.67$KB. The 16 variable-sized prediction tables of 6 bit weights consume 2.64KB total. The vector of feature values kept for the current memory access consumes 0.44KB. The default MDPP replacement policy uses 15 bits for each of the 2,048 sets or 3.75KB. Thus, the total size required to support single-core MPPPB is 27.5KB, or 1.3% of the capacity of the cache. This is comparable to Hawkeye, which uses 28KB total for its structures. Perceptron uses only about 10KB. We compensate by allowing it to have more sampler sets so that it uses an equivalent hardware budget.

Multi-Core Overhead. For the 4-core set of features, there are more single-bit and *offset* features, so only 93 bits are required to store the vector index bits per sampler entry, and the predictor is slightly smaller. We scale up the sampler by a factor of four to contain 256 sampled sets, requiring 68.63KB for the sampler. The 16 variable-sized prediction tables consume 1.94KB. The four per-core vector of feature values consume 1.29KB total. We use SRRIP for multi-core MPPPB, using two bits per cache block in an 8MB cache, requiring 32KB. Thus, the total overhead for multi-core MPPPB is 104KB, or 1.3% of the capacity of the cache. The code provided by the Hawkeye authors scales its resource usage based on the number of sets in the cache resulting in a roughly equivalent sized structure. Again, we augment Perceptron with a larger number of sampler sets so that it uses an equivalent hardware budget.

The implementation complexity of the perceptron-based reuse predictor is approximately the same as Perceptron. The small tables, sampler, and summing logic are conceptually similar to the Perceptron algorithm. Conditional branch predictors based on perceptron learning have been implemented in Oracle, AMD, and Samsung processors [4, 8, 24]. Branch predictors operate under very tight timing constraints. Predictors in the last-level cache have a much higher tolerance for latency. We are confident that the multiperspective predictor can be implemented given the timing constraints of a last-level cache reuse predictor.

4.5 Reporting Performance

In Section 6 we report performance relative to LRU for the various techniques tested. For single-thread workloads, we report the speedup over LRU, *i.e.* the instructions-per-cycle (IPC) of a technique divided by the IPC given by LRU. For the multi-programmed workloads, we report the weighted speedup normalized to LRU. That is, for each thread i sharing

the 8MB cache, we compute IPC_i. Then we find $SingleIPC_i$ as the IPC of the same program running in isolation with a 8MB cache with LRU replacement. Then we compute the weighted IPC as $\sum IPC_i/SingleIPC_i$. We then normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

5 DEVELOPING SETS OF FEATURES

In this section, we describe our search for a good set of features and parameters for MPPPB. For both single-thread and multi-programmed workloads, we use use a methodology of using different training and testing sets. We empirically determined that a set of 16 features provided enough diversity of features to yield good accuracy while not requiring too much hardware.

5.1 Finding Features

Our methodology for finding good features and parameters is to start with a large set of randomly chosen features, evaluate them with a fast simulator that only measures average MPKI for a set of workloads, and then choose the best set of features for further refinement with a hill-climbing algorithm. The hill-climbing algorithm randomly chooses a feature from the current set of features and changes it randomly by either replacing it with a randomly generated feature, replacing it with a copy of another feature, or slightly perturbing one of its parameters. If the change lowers average MPKI, it is kept, otherwise it is discarded. The hill-climbing procedure is allowed to continue until it appears to have reached a state of convergence, *i.e.* no random changes benefit the set of features within a certain time limit. Previous work has used genetic algorithms for a more robust search [13]. We tried using a genetic algorithm for this work, but found that the computationally intensive nature of evaluating average MPKI led to unsatisfactory results. The design space exploration for this project consumed approximately 10 CPU years spread across our local cluster and a supercomputer to which we had access.

5.2 Single-Thread Features

For single-thread workloads, we used a cross-validation methodology to develop two sets of features. We first randomly generated 4,000 sets of 16 parameterized features and evaluated them on all 99 workloads. We divided the 99 program segments described in Section 4.2 randomly into two subsets of 50 and 49. Somewhat surprisingly, we found that the same set of parameterized features provided the lowest average MPKI for both subsets. We then proceeded with separate hill-climbing for each subset at which point the set of features diverged. For reporting performance and miss rate results for each segment, we used the set of features developed for the other segment. Thus, there is no overfitting of features to benchmarks. Table 1 shows the two sets of features developed for the single-thread workloads.

Figure 3 illustrates the search methodology. For the 99 single-thread program segments, we evaluate 4000 randomly chosen sets of 16 features. The figure shows the simulated MPKI of these feature sets sorted in descending order of MPKI. It also shows the MPKI for LRU, MIN, and the result of the cross-validated hill-climbed feature sets used to for our final single-thread numbers. Random feature selection yields MPKIs ranging from worse than LRU to almost halfway between LRU and MIN. Hill-climbing provides an additional boost in performance, but most of the benefit comes from the initial random search. As hill-climbing is prone to getting caught in local minima, we believe a better search technique will improve performance.

(a)
bias(16,0)
burst(6,0)
insert(16,0)
insert(16,1)
insert(17,1)
insert(8,1)
lastmiss(9,0)
offset(10,0,6,1)
offset(15,1,6,1)
pc(10,1,53,10,0)
pc(16,3,11,16,1)
pc(16,8,16,5,0)
pc(17,6,20,0,1)
pc(17,6,20,0,1)
pc(17,6,20,14,1)
pc(7,14,43,11,0)

(b)
address(11,8,19,0)
bias(6,1)
insert(15,0)
insert(16,1)
insert(6,1)
offset(15,1,6,1)
offset(15,3,7,0)
pc(11,2,24,4,1)
pc(15,14,32,6,0)
pc(15,5,28,0,1)
pc(16,0,16,8,1)
pc(17,6,20,0,1)
pc(6,12,14,10,1)
pc(7,1,24,11,0)
pc(7,14,43,11,0)
pc(8,1,61,11,0)

Table 1: Two sets of features developed for single-thread benchmarks using cross-validation.

bias(6,0)
address(9,9,14,5,1)
address(9,12,29,0)
address(13,21,29,0)
address(14,17,25,0)
lastmiss(6,0)
lastmiss(18,0)
offset(13,0,4,0)
offset(14,0,6,0)
offset(16,0,1,0)
pc(6,13,31,4,0)
pc(9,11,7,16,0)
pc(13,16,24,17,0)
pc(16,2,10,2,0)
pc(16,4,46,9,0)
pc(17,0,13,5,0)

Table 2: Set of features developed for multi-programmed workloads using 100 training mixes.

5.3 Multi-Programmed Workloads

For the multi-programmed workloads, we randomly generated 1000 mixes of 4 program segments without replacement. We used the first 100 of these mixes as a training set to develop features. For reporting performance and miss rate results, we use the remaining 900 mixes. Thus, the performance and miss rate numbers reported in this paper are for multi-programmed workloads that were not used for finding features. Table 2 shows the features found for multi-programmed workloads.

5.4 Discussion of Features

We make no claim that the features found by the hill-climbing methodology are optimal. However, we can make interesting observations about the features found:

(1) The single-thread features appear to have little use for the *address* feature. It appears only once in one set of features. On the other hand, the multi-programmed features use four instances of *address*. We hypothesize that different benchmarks within a multi-programmed workload have different working sets of physical address ranges, allowing the predictor to distinguish among behavior from the four different programs. Single-thread behavior is more uniform so the address may provide less information than the other features.

(2) Both single-thread and multi-programmed features make heavy use of the *pc* feature. This confirms previous research that finds high correlation between previous PC values and program behavior [15, 18, 19, 28, 29].

(3) The *burst* feature is only used in one of the single-thread feature sets, and is not used for the multi-programmed features. Recall that the Boolean *burst* feature is true when an access is to a most-recently-used block. Since the sampler and the main cache use different replacement policies, the signal given by the *burst* feature might be inconsistent between training and prediction. For the multi-programmed cache that uses SRRIP as the default replacement policy, this feature is particularly problematic. In SRRIP, recency positions are not exclusive among cache blocks. Theoretically, all of the blocks in an SRRIP-replaced set could occupy the most-recently-used position simultaneously.

(4) The *lastmiss* feature is used only once in the two sets of single-thread features. It may be of limited use in that context. Eliminating it from consideration would cut down the area expense of storing an additional bit per set tracking misses.

(5) One of the single-thread feature sets, as well as the multi-programmed feature set, include the *bias* feature without exclusive-ORing the PC. This gives a simple global up/down counter that tracks the tendency of recent blocks to be dead.

(6) The *insert* feature figures prominently in both sets of single-thread features, but does not appear at all in the multi-programmed features. In Section 6.4 we will see that the value of the *insert* feature is unclear.

Note that the two sets of single-thread features share some elements, for instance, *pc*(17,6,20,0,1) appears in both sets of features, and even twice in one of them. This is due to the fact that the same initial random set of features minimized MPKI for both randomly chosen subsets of program segments, and because the hill-climbing algorithm may choose to duplicate a feature.

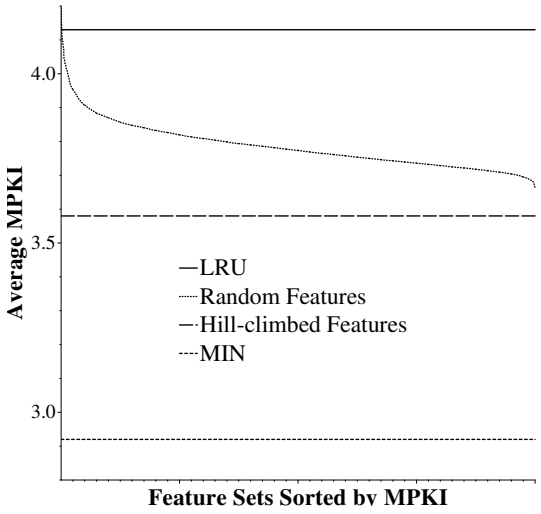

Figure 3: Results for developing features for single-thread benchmarks. Four thousand randomly chosen sets of 16 features yield a range of MPKI values. Hill-climbing improves MPKI but most of the benefit comes from initial random search.

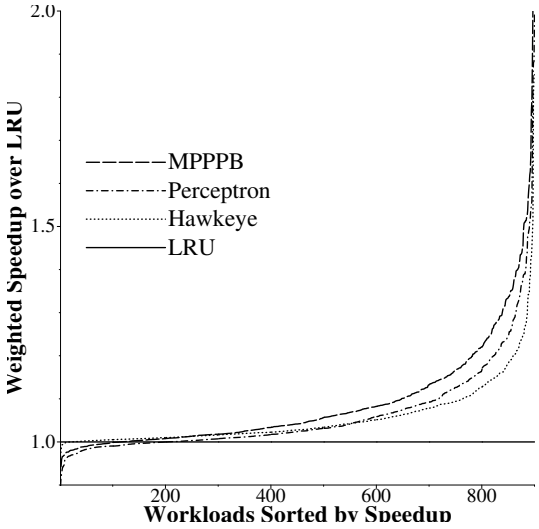

Figure 4: Normalized Weighted Speedup over LRU for 4-Core Multi-Programmed Workloads

5.5 Other Parameters

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 chosen using the same cross-validation methodology as the feature sets. For each subset of workloads uses to train the parameters, the bypass threshold τ_0 is set first by an exhaustive search of all possible values. Then the values of τ_1, τ_2, τ_3, π_1, π_2, and π_3 are searched by generating thousands of random feasible combinations of these values and selecting the combination yielding the minimum average MPKI. For the multi-core version of MPPPB, we use SRRIP as the default replacement policy to limit the number of combinations of π_1, π_2, and π_3, allowing us to exhaustively enumerate the combination of placement positions with randomly generated thresholds.

6 RESULTS

This section gives results for the various policies tested. It presents misses per kilo-instruction (MPKI) and speedup results. First, results are given for multi-programmed workloads. Then, results are given for single-thread workloads.

6.1 Multi-Programmed Results

This section gives results for the 900 4-core multi-programmed workloads.

6.1.1 Performance. Figure 4 shows weighted speedup normalized to LRU for Perceptron, Hawkeye, and MPPPB with an 8MB last-level cache. See 4.5 for the definition of weighted speedup. The figure shows the speedups for each of the 900 testing workloads in ascending sorted order to yield S-curves. Perceptron yields a geometric mean 5.8% speedup and Hawkeye yields a 5.2% speedup. MPPPB gives a geometric mean speedup of 8.3%. The superior accuracy of multiperspective prediction gives a significant boost in performance over the other two reuse predictors. Interestingly, Hawkeye yield performance slightly below LRU for only 18 workloads. Perceptron gives performance inferior to LRU for 201 workloads, while MPPPB giver lower performance than LRU for 115 workloads. Both Perceptron and MPPPB more than make up for that poor showing in terms of average and best-case speedup, but it is clear that Hawkeye has some advantage over the other techniques in delivering stable performance. We intend to study this advantage in future research.

6.1.2 Misses. Figure 5 shows misses per 1000 instructions (MPKI) various techniques sorted in descending order, *i.e.* worst-to-best from left-to-right, to yield S-curves with a log scale y-axis. MPPPB, at an arithmetic mean 10.97 MPKI, delivers fewer misses than the other techniques. LRU, Perceptron, and Hawkeye yield 14.1 MPKI, 12.49 MPKI, and 11.72 MPKI, respectively.

6.2 Single-Thread Results

This section discusses the single-thread performance and misses for the 29 SPEC CPU benchmarks with a 2MB LLC. Prefetching is enabled.

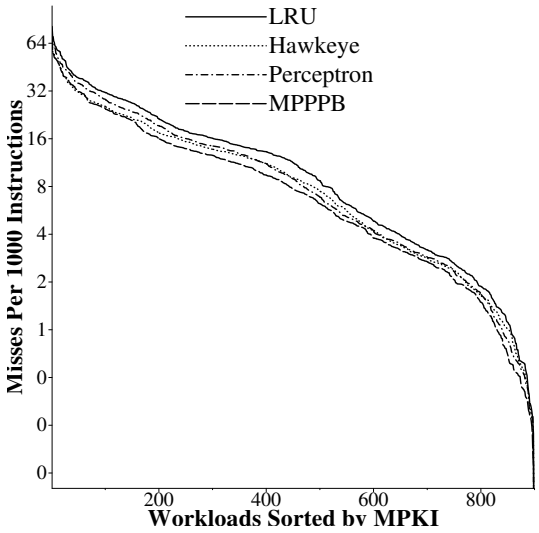

Figure 5: Misses per 1000 Instructions for 4-Core Multi-Programmed Workloads

6.2.1 Performance. Figure 6 shows single-thread speedup of the techniques over LRU. The benchmarks are sorted by speedup with MPPPB. Perceptron and Hawkeye achieve a geometric mean 6.3% and 5.1% speedup over LRU, respectively. MPPPB yields a 9.0% geometric mean speedup. The optimal MIN policy achieves a 13.6% speedup. Thus, MPPPB yields two thirds of the maximum speedup achievable. For 22 out of the 33 benchmarks, MPPPB yields the best speedup over Hawkeye and Perceptron. For 28 out of the 33 benchmarks, MPPPB exceeds the performance of LRU, and never performs below 95% of the performance of LRU. For 8 benchmarks for which MPPPB does not provide the best speedup of the realistic techniques, Hawkeye gives the best speedup. This result suggests that MPPPB might be combined with Hawkeye to provide superior performance.

6.2.2 Misses. Figure 7 gives the MPKI for the 33 benchmarks. Note the y-axis is a log scale. Perceptron and Hawkeye have an average 3.7 and 3.8 MPKI, respectively. MPPPB gives 3.5 MPKI. The average MPKI figures are somewhat low because most of the benchmarks fit a large part of their working sets into a 2MB cache. For *mcf*, a benchmark with a high number of misses, Perceptron gives 25.2 MPKI, Hawkeye yields 26.9 MPKI, and SPPPD gives 23.5 MPKI.

6.3 Accuracy

Figure 8(a) shows receiver operating characteristic (ROC) curves for three reuse predictors with wide-ranging confidence values: sampling-based dead block prediction (SDBP) [15], perceptron-learning-based reuse prediction (hereafter, Perceptron) [28], and our multiperspective technique. For each technique, we modify the simulator to make the prediction but not apply the optimization so that we can measure the

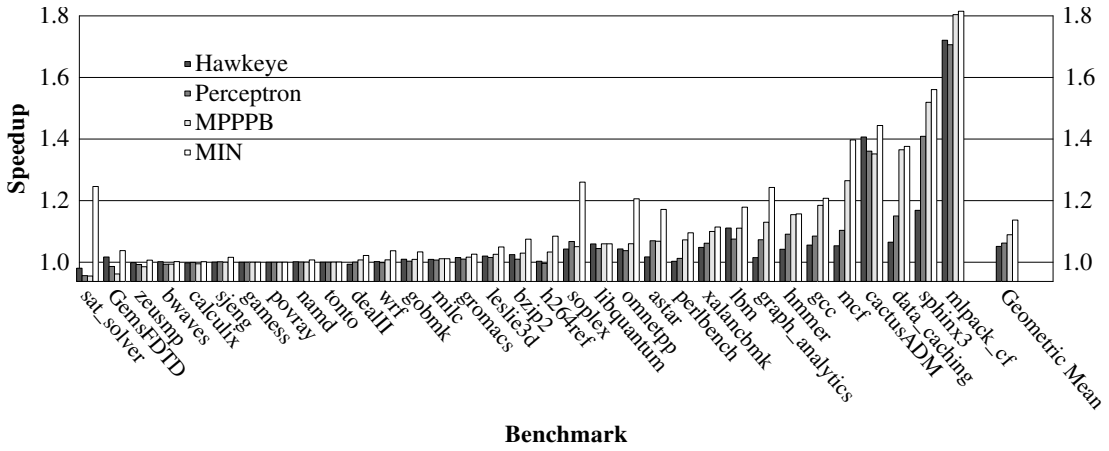

Figure 6: Speedup over LRU for Single-Thread Workloads.

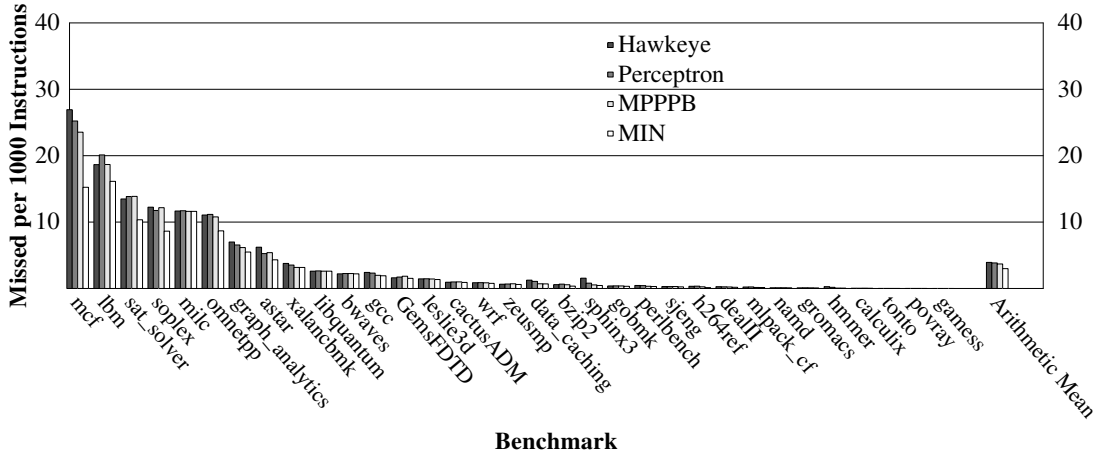

Figure 7: Misses per 1000 Instructions for Single-Thread Workloads.

accuracy of the predictors without feedback from their decisions affecting the measurement. The curves are averaged over the 33 single-thread benchmarks.

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 higher true positive rate increases the opportunity for applying the optimization.

We do not report ROC for Hawkeye. SDBP, Perceptron, and multiperspective each classify blocks as "dead" or "live" learning from an LRU-replaced sampler. Hawkeye classifies blocks as "cache friendly" or "cache averse" learning from an approximation of MIN. The false and true positive rates given by Hawkeye are not directly comparable to the other predictors. Naïvely mapping "cache averse" to "dead" and "cache friendly" to "live," results in false and true positive rates worse than the other predictors.

Consider the bypass optimization. For each predictor, the maximum tolerable false positive rate for best performance is between 25% and 31%, with the exact value decided by the granularity of the possible threshold values (*i.e.*, possible values for π_0 for MPPPB). See Figure 8(b). 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 bypass.

6.4 Analysis

We perform three experiments to examine the impact of two aspects of the features on performance.

Figure 9 illustrates the effect of allowing each feature to have its own associativity parameter. For the 900 multi- programmed workloads, we fix the A parameter for each feature from 1 through 18 and observe the resulting performance.

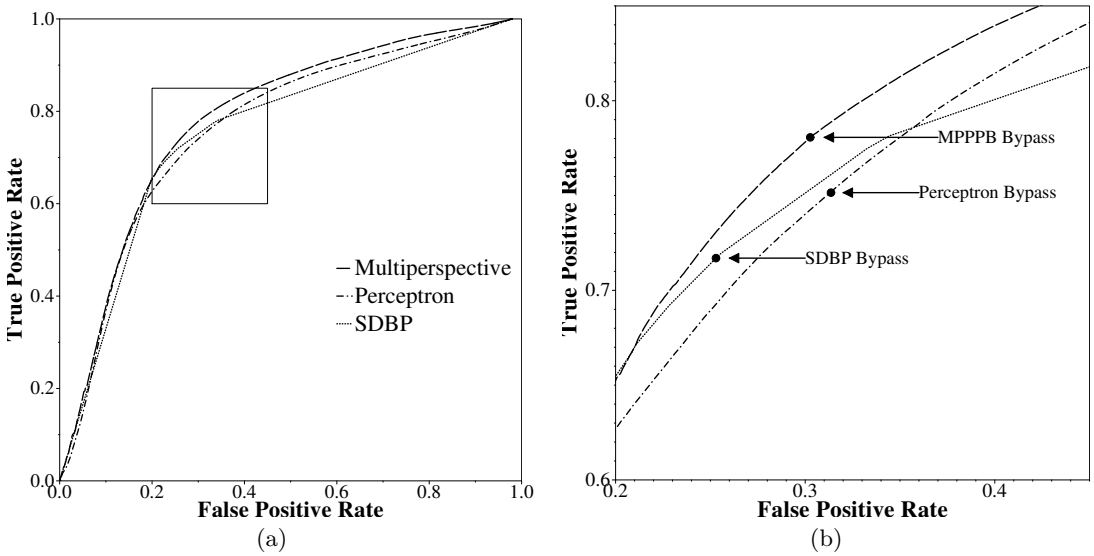

Figure 8: Receiver operating characteristic (ROC) curves for three reuse predictors. Each predictor has a wide range of false and true positive rates (a). In a range of false positive rates (inset box) allowing for the bypass optimization, multiperspective prediction identifies more true positives (dead blocks).

With $A = 1$ for all features, performance reaches a 6.4% geometric mean weighted speedup over LRU. With $A = 18$, the maximum associativity allowed by the sampler, MPPPB reaches a speedup of 7.8% over LRU. Allowing the original set of features with its variable associativities per feature results in an 8.0% speedup. Thus, variable associativities help performance, but not by as large a margin as we had expected.

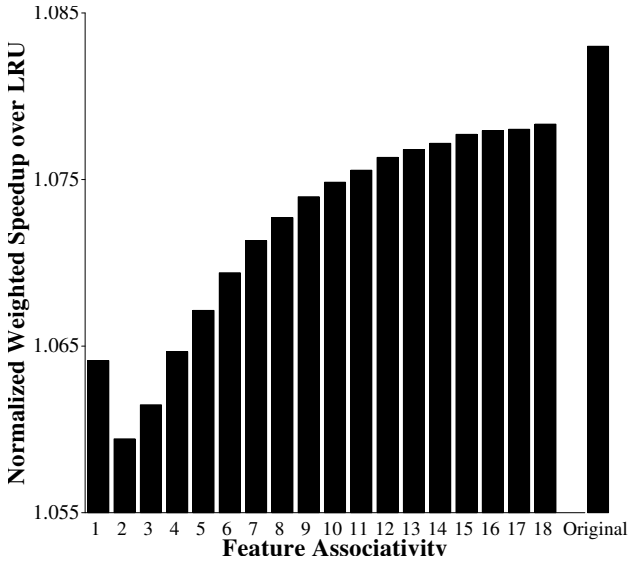

Figure 9: Performance impact of uniform associativity for each feature

Figure 10 analyzes the feature set given in Table 1(a). Each bar shows the speedup obtained over the 900 multi-programmed workloads when a given feature is removed from

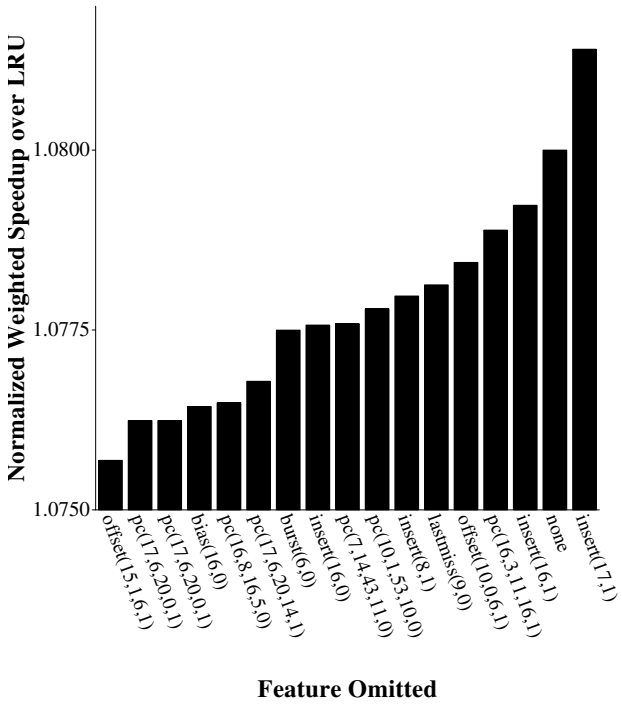

Figure 10: Performance impact of removing each feature from the predictor

Simpoint	Instruction Footprint	Memory Footprint	Feature	MPKI without	MPKI with	Percent Increase
603.bwaves_s-891B	9KB	1.6GB	*bias*(6,1)	4.2	4.2	0.02%
623.xalancbmk_s-10B	233KB	30MB	*insert*(15,0)	8.2	7.6	7.82%
621.wrf_s-575B	1.1MB	75MB	*insert*(16,1)	3.2	3.2	1.11%
657.xz_s-2302B	43KB	43MB	*insert*(6,1)	1.3	1.2	1.88%
654.roms_s-1390B	83KB	387MB	*offset*(15,1,6,1)	6.8	6.6	2.24%
602.gcc_s-2226B	23KB	196MB	*offset*(15,3,7,0)	1.5	1.4	8.82%
657.xz_s-3167B	43KB	48MB	*pc*(11,2,24,4,1)	1.0	1.0	1.99%
605.mcf_s-472B	11KB	845MB	*pc*(15,14,32,6,0)	15.7	12.7	18.88%
619.lbm_s-2677B	4KB	2.23GB	*pc*(15,5,28,0,1)	44.3	44.3	0.06%
605.mcf_s-1554B	13KB	184MB	*pc*(16,0,16,8,1)	28.1	25.8	8.38%
654.roms_s-293B	48KB	1.1GB	*pc*(17,6,20,0,1)	3.7	3.6	2.67%
641.leela_s-334B	148KB	5.8MB	*pc*(6,12,14,10,1)	0.1	0.0	14.06%
621.wrf_s-8065B	1.2MB	68MB	*pc*(7,1,24,11,0)	1.8	1.8	0.99%
625.x264_s-12B	299KB	39MB	*pc*(7,14,43,11,0)	0.5	0.5	1.06%
620.omnetpp_s-141B	158KB	165MB	*address*(11,8,19,0)	7.0	6.9	1.04%

Table 3: Features contributing the most benefit for some SPEC CPU 2017 simpoints

the set of features, showing the impact of that feature. One surprising result is that removing the *insert*(17,1) feature actually improves performance; thus, that *insert* feature seems to be useless. Other *insert* features provide some value, but it is possible *insert* altogether in favor of other features could help performance.

The most valuable feature is *offset* with associativity 15 considering bits 1 to 6 of the block offset exclusive-ORed with the PC. When it is removed, speedup drops from 8.0% to 7.6%. Two *pc* features as well as the global *bias* counter are similarly valuable. Note that, although this set of features was developed for single-thread benchmarks, it provides reasonable performance for the multi-programmed workloads: 8.0% speedup versus 8.3% for the features specifically developed for multi-programmed workloads.

Table 3 shows how different features dominate predictor performance for different programs and phases. We find 95 simpoints from the SPEC CPU 2017 benchmarks. We choose the SPEC CPU 2017 benchmarks as they became available between the acceptance and camera ready versions of this paper, providing a good testing set as they had not been used at all in the design of the features. For each simpoint, we run the same experiment as in Figure 10 using the features from Table 1(b), *i.e.* we run the simulator 16 times, omitting a different feature each time. The table shows, for 15 of the 16 features, a simpoint where that feature contributes the most to MPKI reduction (one feature was never the best for any simpoint). The table shows the name of the simpoint as the name of the benchmark followed by the instruction count where the interval starts. It shows the name and parameters of the feature, the MPKI both with and without the given feature as well as the percentage increase in MPKI when the feature is removed. It also shows the size of the instruction and memory footprints for the simpoint.

We can see that a particular *pc* feature, *pc*(15,14,32,6,0), improves performance for a `605.mcf` simpoint by over 18%, showing strong PC correlation for that benchmark. An offset feature improves a `602.gcc` feature by 8.8%. `602.gcc` is a compiler that makes heavy use of field dereferencing in objects, an activity which is likely to trigger the offset feature. An *insert* feature improves `621.wrf` by 1.1%. The instruction footprint for this simpoint is somewhat large, so the *pc* features might experience significant destructive aliasing in the predictor tables. Thus, the availability of the *insert* feature, which is less affected by aliasing, gives an advantage. `623.xalancbmk` is also improved by an *insert* feature, this time with the XOR flag set to false, *i.e.*, the insert feature is a single bit indexing one of two counters. This simple feature improves performance by 7.8% for this simpoint.

7 CONCLUSION AND FUTURE WORK

Many features correlate with reuse behavior. Previous work focuses on one or two features at a time. Multiperspective prediction uses many features, each contributing to the overall prediction. It enables a placement, promotion, and bypass optimization that improves performance over prior work. In future work, we will improve the search strategy for finding good sets of features and explore other optimizations to which multiperspective prediction can be applied.

ACKNOWLEDGMENTS

This research was supported by NSF grants CCF-1332598 and CCF-1649242 as well as a gift from Intel Labs. Some of this research was conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

REFERENCES

[1] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. 2005. IATAC: a smart predictor to turn-off L2 cache lines. *ACM Trans. Archit. Code Optim.* 2, 1 (2005), 55–77. https://doi.org/10.1145/1061267.1061271

[2] Nathan Beckmann and Daniel Sanchez. 2017. Maximizing Cache Performance Under Uncertainty. In *Proceedings of the 23rd international symposium on High Performance Computer Architecture (HPCA-23)*.

[3] László A. Bélády. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. *IBM Systems Journal* 5, 2 (June 1966), 78–101.

[4] Brad Burgess. 2016. Samsung's Exynos-M1 CPU. In *Hot Chips: A Symposium on High Performance Chips* (Cupertino, California).

[5] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. 2013. MLPACK: A Scalable C++ Machine Learning Library. *Journal of Machine Learning Research* 14 (2013), 801–805.

[6] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V. Veidenbaum. 2012. Improving Cache Management Policies Using Dynamic Reuse Distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '12)*. IEEE Computer Society, Washington, DC, USA, 389–400. https://doi.org/10.1109/MICRO.2012.43

[7] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 37–48.

[8] Agner Fog. 2014. The Microarchitecture of Intel, AMD, and VIA CPUs. http://www.agner.org/optimize/microarchitecture.pdf. (2014).

[9] Andrew Hilton, Neeraj Eswaran, and Amir Roth. 2009. FIESTA: A Sample-Balanced Multi-Program Workload Methodology. In *Workshop on Modeling, Benchmarking and Simulation (MoBS)* (Austin, Texas).

[10] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. 2002. Timekeeping in the memory system: predicting and optimizing memory behavior. *SIGARCH Comput. Archit. News* 30, 2 (2002), 209–220. https://doi.org/10.1145/545214.545239

[11] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture (ISCA '16)*. 78–89.

[12] Aamer Jaleel, Kevin Theobald, Simon Steely Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA-37)*.

[13] Daniel A. Jiménez. 2013. Insertion and Promotion for Tree-based PseudoLRU Last-level Caches. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. 284–296.

[14] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache Replacement Based on Reuse-Distance Prediction. In *In Proceedings of the 25th International Conference on Computer Design (ICCD-2007)*. 245–250.

[15] Samira M. Khan, Yingying Tian, and Daniel A. Jiménez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In *MICRO*. 175–186.

[16] Mazen Kharbutli and Yan Solihin. 2008. Counter-Based Cache Replacement and Bypassing Algorithms. *IEEE Trans. Comput.* 57, 4 (2008), 433–447. https://doi.org/10.1109/TC.2007.70816

[17] An-Chow Lai and Babak Falsafi. 2000. Selective, accurate, and timely self-invalidation using last-touch prediction. In *International Symposium on Computer Architecture*. 139 – 148. https://doi.org/10.1109/ISCA.2000.854385

[18] An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block prediction & dead-block correlating prefetchers. *SIGARCH Comput. Archit. News* 29, 2 (2001), 144–154. https://doi.org/10.1145/384285.379259

[19] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Los Alamitos, CA, USA, 222–233. https://doi.org/10.1109/MICRO.2008.4771793

[20] Pierre Michaud, André Seznec, and Richard Uhlig. 1997. Trading Conflict and Capacity Aliasing in Conditional Branch Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*. 292–303.

[21] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2015. Exploiting compressed block size as an indicator of future reuse.. In *HPCA*. IEEE, 51–63.

[22] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.* 31, 1 (2003), 318–319. https://doi.org/10.1145/885651.781076

[23] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A Case for MLP-Aware Cache Replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 167–178. https://doi.org/10.1109/ISCA.2006.5

[24] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja. 2012. Sparc T4: A Dynamically Threaded Server-on-a-Chip. *IEEE Micro* 32, 2 (2012), 8–19.

[25] Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. 2004. Memory coherence activity prediction in commercial workloads. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*. ACM, New York, NY, USA, 37–45. https://doi.org/10.1145/1054943.1054949

[26] David Tarjan, Kevin Skadron, and M. Stan. 2004. An Ahead Pipelined Alloyed Perceptron with Single Cycle Access Time. In *Proceedings of the Workshop on Complexity Effective Design (WCED)*.

[27] Elvira Teran, Yingying Tian, Zhe Wang, and Daniel A. Jiménez. 2016. Minimal Disturbance Placement and Promotion. In *2016 IEEE 22nd International Symposium on High Performance Computer Architecture (HPCA)*.

[28] Elvira Teran, Zhe Wang, and Daniel A. Jiménez. 2016. Perceptron Learning for Reuse Prediction. In *Proceedings of the 49th ACM/IEEE International Symposium on Microarchitecture (MICRO-49)*.

[29] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Jr. Simon C. Steely, and Joel Emer. 2011. SHiP: Signature-based Hit Predictor for High Performance Caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 430–441.