Perceptron Learning for Reuse Prediction

Elvira Teran† Zhe Wang§ Daniel A. Jiménez†
†Texas A&M University §Intel Labs
{eteran,djimenez}@tamu.edu zhe2.wang@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 proposes perceptron learning for reuse prediction. The proposed predictor greatly improves accuracy over previous work. For multi- programmed workloads, the average false positive rate of the proposed predictor is 3.2%, while sampling dead block prediction (SDBP) and signature-based hit prediction (SHiP) yield false positive rates above 7%. The improvement in accuracy translates directly into performance. For single-thread workloads and a 4MB last-level cache, reuse prediction with perceptron learning enables a replacement and bypass optimization to achieve a geometric mean speedup of 6.1%, compared with 3.8% for SHiP and 3.5% for SDBP on the SPEC CPU 2006 benchmarks. On a memory-intensive subset of SPEC, perceptron learning yields 18.3% speedup, versus 10.5% for SHiP and 7.7% for SDBP. For multi- programmed workloads and a 16MB cache, the proposed technique doubles the efficiency of the cache over LRU and yields a geometric mean normalized weighted speedup of 7.4%, compared with 4.4% for SHiP and 4.2% for SDBP.**

I. Introduction

The last-level cache (LLC) mitigates the large disparity in latency and bandwidth between CPU and DRAM. An efficient cache keeps as many useful blocks as possible. Reuse predictors detect whether a given block is likely to be accessed again before it is evicted, allowing optimizations such as improved placement, replacement, and bypass. Reuse predictors use features such as data and instruction addresses to find correlations between past behavior and future accesses, but extracting accuracy from these features has been problematic in the LLC. We propose using perceptron learning to combine features in a way that overcomes this difficulty, resulting in better accuracy. The predictor yields significant performance improvements when used to drive a replacement and bypass optimization.

A. Better Accuracy with Perceptron Learning

Figure 1 shows violin plots for the accuracy of three reuse predictors: sampling dead block prediction (SDBP) [1], signature-based hit prediction (SHiP) [2], and our proposed perceptron-based reuse predictor over multi-programmed workloads. The coverage rate (shaded darker) is the percentage of all predictions for which a block is predicted not to be reused. The false positive rate (shaded lighter) is the percentage of all predictions for which a block was incorrectly predicted as not reused. The plots show the mean as a bar in the center of the plot along with the probability density of the data. The coverage for perceptron-based reuse prediction is higher than that for SDBP and SHiP (52.4% versus 47.2% and 43.2%, respectively) while the false positive rate is lower (3.2% versus 7.4% and 7.7%, respectively). Coverage represents the opportunity for optimization given by the predictor, while false positives represent the potential cache misses caused when an incorrect prediction leads to a live block being replaced. Thus, perceptron learning leads to increased opportunity for optimization while reducing false positives by a factor of 2. For more on the violin plots, see Section V-B1.

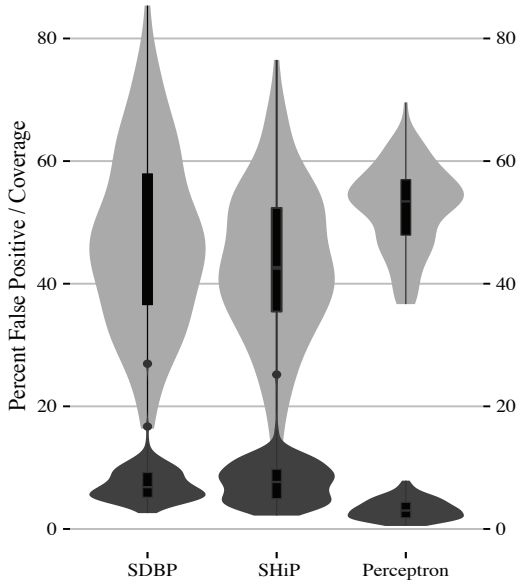

Fig. 1. Violin plots for false positive (darker) and coverage (lighter) rates. Perceptron learning is significantly more robust, with higher coverage, fewer false positives, and less variance in accuracy across workloads.

B. Perceptron Learning

This paper describes an algorithm that uses perceptron learning for reuse prediction. Perceptrons are a simple model of neurons in neural networks [3], [4] modeled by vectors of signed weights learned through online training. The output of a perceptron is the dot product of the weights and a vector of inputs. In this work, we do not actually use perceptrons, but we make use of the perceptron learning algorithm. There are two components to the abstract perceptron learning algorithm:

1) To predict true or false, a vector of signed weights is chosen according to some criteria. The dot product of

978-1-5090-3508-3/16/$31.00

the weights and input vectors, called y_{out}, is computed. If y_{out} exceeds some threshold, the prediction is true, otherwise it is false.

2) To update the weights after the true outcome of the predicted event is known, we first consider the value of y_{out}. If the prediction was correct and $|y_{\text{out}}|$ exceeds a threshold θ, then the weights remain unchanged. Otherwise, the inputs are used to update the corresponding weights. If there is positive correlation between the input and the outcome of the event, the corresponding weight is incremented; otherwise, it is decremented. Over time, the weights are proportional to the probability that the outcome of the event is true in the context of the input and the criterion for choosing that weight. The weights saturate at maximum and minimum values so they will fit in a fixed bit width.

In the original paper describing branch prediction with perceptrons, the input vector was the global history of branch outcomes [5]. In this work, as in some subsequent branch prediction work [6], [7], the weights in the vectors are chosen by indexing independent tables using indices computed as hashes of features such as branch pattern and path history. Such *hashed perceptron* predictors do not use a vector of binary weights. Rather, instead of a dot product, they simply compute a sum which can be thought of as a dot product of the weights vector with $\vec{1}$. In this work, we use features relevant to cache management such as the address of memory instructions and bits from the block address, tag, or page number. Thus, although the technique does not use classical perceptrons, it uses perceptron learning to adjust the weights.

C. Contributions

This paper makes the following contributions:

1) It shows that perceptron learning can be used to effectively combine multiple inputs features to greatly improve the accuracy of reuse prediction.
2) It applies perceptron-learning-based reuse prediction to a replacement and bypass optimization, outperforming the state-of-the-art policies for both single and multi-programmed workloads. For 1000 multi- programmed workloads, perceptron learning gives a geometric mean normalized weighted speedup of 8.3%. For single-threaded workloads, perceptron learning gives a geometric mean speedup of 6.2% over LRU, with a speedup of 18.3% on a memory-intensive subset of the benchmarks.
3) It shows that cache management based on perceptron learning more than doubles cache efficiency over the LRU policy.
4) It evaluates the proposed technique as well as previous work in the presence of a stream prefetcher. Previous work had been evaluated without considering prefetching.

II. Background and Related Work

This section reviews the most relevant recent work in reuse prediction and microarchitectural perceptron learning. Other work is described and cited later in the paper as appropriate; see in particular Section III-A. To our knowledge, the term "reuse predictor" as applied to predicting near-term reuse of cache blocks originates in recent work by Pekhimenko *et al.* [8]. We prefer this term to "dead block predictor" or "hit predictor" [2] because it unifies the two ideas and emphasizes the insight of the predictors.

a) Reuse Distance Prediction: Previous work predicts the distance to the next reuse, rather than simply whether the block will be reused or not. Re-reference Interval Prediction (RRIP) [9] is an efficient implementation of reuse-distance prediction [10]. RRIP categorizes blocks as near-immediate re-reference interval, intermediate re-reference intervals, and distant re-reference interval [9]. Static RRIP (SRRIP) always places into the same position while Dynamic RRIP (DRRIP) uses set-dueling [11] to adapt the placement position to the particular workload. RRIP is a simple policy, requiring little overhead and no complex prediction structures, while resulting in significant improvement in performance.

b) Dead Block Prediction: Dead block predictors predict whether a block will be used again before it is evicted. There are numerous dead block predictors applied to a variety of applications in the literature [12], [13], [14], [15], [16], [17], [18], [1]. We extend Sampling Dead Block Prediction (SDBP) of Khan *et al.* [1]. In this work, a *sampler* structure keeps partial tags of sampled sets separate from the cache. Three tables of two-bit saturating counters are accessed using a technique similar to a skewed branch predictor [19]. For each hit to a block in the sampled set, the program counter (PC) of the relevant memory instruction is hashed into the three 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. For an LLC access, the predictor is consulted by hashing the PC of the memory access instruction into the three tables and taking the sum of the indexed counters. When the sum exceeds some threshold, the accessed block is predicted to be dead. Tags in sampled sets are managed with true LRU and a reduced associativity, but the LLC may be managed by any policy. The paper applies its predictor to replacement and bypass.

c) Signature-Based Hit Prediction: Wu *et al.* [2] propose a signature-based hit predictor (SHiP) that predicts the likelihood that a block will be hit again. This technique uses a table of 3-bit saturating counters indexed by a signature of the block, *e.g.*, the memory instruction PC that caused the initial fill of the block. When a block with a given signature is hit, the counter associated with that signature is incremented. When a block with a given signature is evicted without having been reused, the counter is decremented. The hit predictor is used to determine insertion position for RRIP [9]. When the counter corresponding to the signature of an incoming block exceeds some threshold, the block is predicted to have an "intermediate re-reference interval," otherwise it is predicted to have a "distant re-reference interval." This work proposes a sampling-based predictor using a small subset of cache sets to

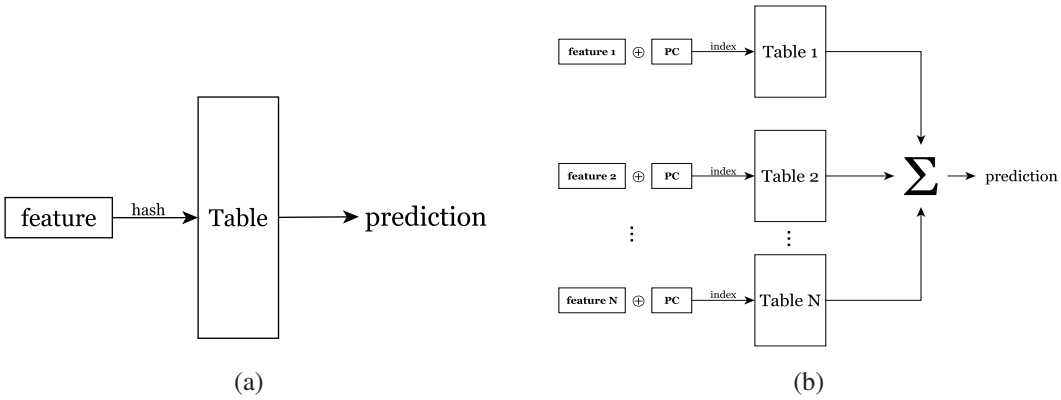

Fig. 2. (a) Previous PC-based Reuse Predictors and (b) Perceptron-based Reuse Predictor

keep metadata such as signatures, and the predictor generalizes to the whole cache.

d) Perceptron Learning in Microarchitecture: Perceptron learning was proposed for predicting conditional branches [5]. In this work, a perceptron weights vector is selected by a hash of the branch address and its dot product with an input vector of branch history outcomes. If the product is at least 0, the branch is predicted as taken, otherwise it is predicted not taken. The weights are updated with the perceptron update rule: if the prediction is incorrect, or the dot product does not exceed some threshold magnitude, then the weights are incremented or decremented based on their correlation to the corresponding history bits.

Subsequent work on predicting branches with perceptron-like algorithms improved the latency of the operation by using history bits to generate indices into weights tables rather than as input to the dot product computation [20], [7], [6]. Our work takes this approach as well, using input features to index weights tables and updating the weights with perceptron learning.

III. Perceptron Learning for Reuse Prediction

In this section we present the prediction algorithm. The structures are similar to those of sampling-based dead block prediction [1], but the algorithm and inputs are altered to provide superior accuracy.

A. Features Correlated with Last Access to a Block

Previous research has observed that multiple features are correlated with the last access to a block. Some of the features are:

1) **The trace of memory access instructions** (PCs) beginning when the block is placed and ending at the current access [13], [17]. This sequence of addresses can be concisely summarized as a truncated sum of those PCs, generating an integer signature to be used to index a prediction table. In an L1 cache, this signature is correlated with the last access to a block. However, in an LLC, particularly in the presence of a middle-level cache, many PCs are filtered, leaving a porous signature that is unsuitable for prediction [1]. In this work, we keep track of recent PCs accessing the LLC, but we treat each PC as a separate feature.
2) **The PC of the memory instruction** that caused the access. This feature is a succinct proxy for the program behavior that led to the ultimate eviction of the block. Unlike the instruction trace, it has reasonable accuracy in the last-level cache [1]. However, its accuracy is limited because a single PC can exhibit multiple behaviors depending on the control-flow context, *e.g.*, an access in a given subroutine may or may not be the last access to a block depending on the caller.
3) **Bits from the memory address**, *e.g.*, the page number or tag. Data in the same memory region are often treated similarly by programs, so the last access (or the only access) to one block may correlate with the last access to a neighboring block. There are many more memory addresses than memory access PCs, so using bits from the memory address as an input feature may lead to destructive interference in small prediction tables. Liu *et al.* combine the trace signature with 3 bits from the block address [17] to form a single index. Our work treats bits from the memory addresses separately, isolating the effects of interference.
4) **A compressed representation** of the data itself [8]. Data that are similar in blocks may lead to similar behavior.
5) **Time/reference count**. Some reuse predictors work on the idea that a block may be accessed a certain number of times before it is evicted, and that this count may be predicted. This technique requires keeping a count for each block in the cache, an idea we dismiss as too complex for implementation in a large last-level cache.

B. The Main Idea

Our predictor combines multiple features. To make a prediction, each feature is used to index a distinct table of saturating counters (hereafter *weights*) that are then summed. If the sum exceeds some threshold, then the accessed block is predicted not to be reused. A small fraction of the accesses are sampled to update the predictor using the perceptron update rule: if

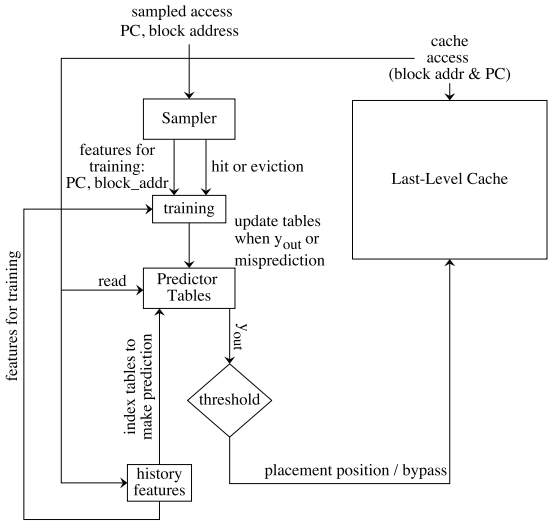

Fig. 3. Datapath from extracting features from an access, to making a prediction and acting on it

the prediction is incorrect, or the sum fails to exceed some magnitude, then the weights are decremented on an access or incremented on an eviction.

Combining features by summing weights avoids the destructive interference and exponential blowup that can be caused by combining them into a single index. That is, if we combine PC bits and memory address bits by, say, summing or XORing them directly into an index, the predictor must learn all the likely patterns that lead to different behavior before it can predict well. Using the perceptron update rule allows the predictor to adapt to changes in program behavior. In previous work, both summing and perceptron update have contributed to significant gains in branch prediction accuracy, motivating this work [5], [7].

Figure 2(a) shows the organization of recent reuse predictors SDBP and SHiP. A single feature is used to index a table of weights to make a prediction. SHiP and SDBP use as the feature a hash of the PC of the memory instruction that causes the reference. Other dead block predictors use a signature formed by truncated addition of all the PCs from the placement of the block to the current reference [13], [17], as well as some bits from the address of the references block [13], but these features are all combined into a single index into the table. It has been shown that the truncated addition idea works poorly for a LLC [1] due to the fact that first- and middle-level caches filter out many of the useful PCs, leaving a noisy signal.

Figure 2(b) shows the organization of the perceptron-based reuse predictor. Each feature is hashed, then XORed with the PC to index a separate table. The corresponding weights are summed and thresholded to get an aggregate prediction. This scheme works well because, at any given time, one or another feature may carry correlation with reuse behavior, while others do not. The perceptron algorithm discovers correlation and ignores the lack of correlation, to give a more accurate prediction.

C. Example

To illustrate why perceptron learning yields superior accuracy to previous reuse predictors, let us consider a simple example. Suppose we must design a reuse predictor indexed by either the PC of the memory instruction, the page number of the memory access, or a combination of both. Figure 4 illustrates two instructions: load A and load B. Load A randomly accesses cache blocks across thousands of pages, but these blocks experience little to no reuse after being accessed by load A. (Perhaps load A is in a destructor in a pointer-chasing object-oriented program.) Load B accesses blocks with good spatial locality over a small number of pages. Cache blocks accessed by load B on a certain set of pages experience reuse, but on another set of pages blocks experience little reuse. (Perhaps load B is a factory method handing out objects to various callers with different access patterns.) Load A would be predicted well by a PC-based predictor, but load B would be predicted poorly because of the variable nature of subsequent accesses. Load A would be predicted poorly by a page-based predictor because thousands of page references would cause aliasing (*i.e.,* collisions) in the prediction table, while load B might be better predicted by a page-based predictor since its behavior is page-based. Neither one would be predicted by indexing a table by a combination (*e.g.*, XOR) of PC and page number: load A continues to have the problem with aliasing, and the prediction table for load B now has more aliasing pressure because we are multiplying the number of indices into the table by the number of other PCs in the working set of load and store instructions.

By contrast, perceptron learning separates the two features into two different tables. Where there is correlation in one table, the magnitude of the weight selected for prediction is high. Where there is no correlation, or poor detection of correlation due to aliasing, the magnitude of the weight is low. The prediction is based on a sum of the weights. The signal of reuse or no reuse comes clearly through the noise of low correlation or aliasing because the noisy weights have low magnitude.

Note that SDBP uses three separate tables indexed by different hashes of the same PC [1]. This slightly reduces the impact of aliasing, but does not help in situations like load B where there is page-based correlation that is not discovered by a PC-based predictor.

D. Training with a Sampler

The sampler is a separate array for a small number of sampled sets. Some sets in the last-level cache are chosen to be represented by a set in the sampler. Each sampler set consists of entries with the following fields:

1) A partial tag used to identify the block. We need only store enough tag bits to guarantee a high probability that a tag match is correct, since the predictions are by their nature imprecise. We find that 15 bits is sufficient.

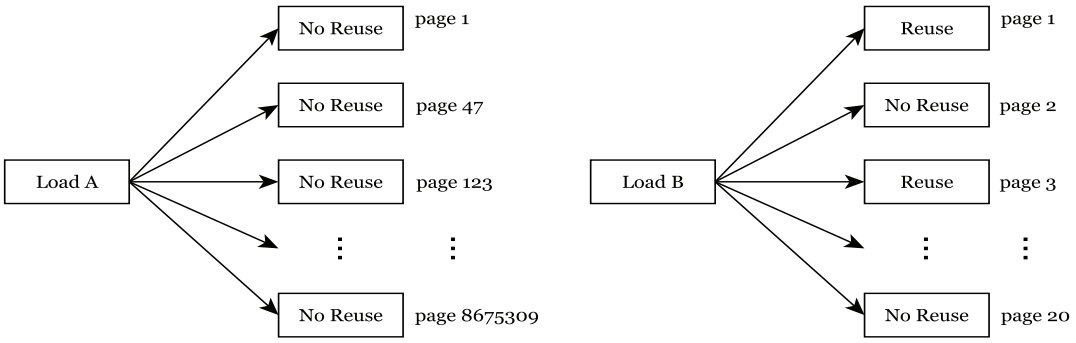

Fig. 4. Two different kinds of correlation. The PC of one load (left) always correlates with no reuse on thousands of pages, while another load (right) has no correlation through the PC but does have correlation through page number.

2) A small integer sum (y_{out}) representing the most recent prediction computation. y_{out} is used to drive the *threshold rule* of the perceptron learning algorithm: if the magnitude of y_{out} exceeds some threshold and the prediction is correct, the predictor does not need to be updated. As this strategy prevents the weights from all saturating at their extreme values, it improves prediction accuracy by ensuring that the predictor can respond quickly to changes in program behavior.
3) A sequence of hashes of the input features used to index the prediction tables. In this sequence, each feature is hashed, then XORed with the lower-order bits of the PC of the memory instruction that last accessed this entry. These hashes will be used to index the predictor's tables.
4) LRU bits for replacement within the sampler.

These sets are kept outside the main cache in a small and fast SRAM array. When a sampled set is accessed, the corresponding set and block in the sampler are accessed. A partial tag from the block is used to match a tag in the sampler set. If there is no match, the tag is placed in the sampler set, replacing the LRU entry in that sampler set. Otherwise, the access is treated as a hit in the cache and perceptron learning is used to train the predictor that this sequence of input features leads to a hit. Then, the current values of the input features (*e.g.*, a hash of the PC or the page number) are placed in the sampled entry. When an entry is evicted from a sampler set, the previous access to that entry is treated as the last access to the corresponding block. Before the entry's values are overwritten, they are used with perceptron learning to train the predictor that the previous access is the last access.

E. Predictor Organization

The predictor is organized as a set of tables, one per input feature. Each table has a small number of weights. In our experiments, we find that providing each table with 256 entries of 6-bit signed weights (ranging in value from -32 to +31) is sufficient. A sequence of hashed features are kept in per-core vectors that are updated on every memory access. We have listed several possible features in Section III-A. We find that using the last few PCs to access the cache as well as bits extracted from the address of the currently accessed block yield the best accuracy. Figure 3 shows the datapath from extracting features from memory accesses to making a prediction.

F. Making a Prediction

The predictor is consulted when a block is hit as well as when a block may be placed in the cache. When there is an access to the LLC, the predictor is consulted to determine whether there is likely reuse for that block. When an incoming block is considered for placement in the LLC, the predictor is used to decide whether to bypass the block if it is predicted to have no reuse. In both situations, the predictor is accessed by indexing the prediction table entries corresponding to the hash of each feature XORed with the PC of the instruction making the access. The signed weights read from each table are summed to find a value y_{out}. If y_{out} is less than some threshold τ, then the block is predicted to be reused; otherwise, it is predicted to have no reuse, *i.e.,* it is predicted dead. An incoming block predicted as dead bypasses the cache. Each cache block is associated with an extra bit of data that records its most recent reuse prediction that will be used to drive the replacement policy.

G. Training the Predictor

When the sampler is accessed, there are two possibilities resulting in two different training outcomes:

e) When there is a replacement in the sampler: A miss in the sampler requires eviction of the LRU entry. The fact that this block is evicted shows it is unlikely to be reused, so the predictor learns from this evidence. If the value of y_{out} for the victim entry is less than a threshold θ, or if the prediction was incorrect, then the predictor table entries indexed by the corresponding hashes in the victim entry are incremented with saturating arithmetic. Thus, future similar sequences of features for this instruction are more likely to be predicted as not reused. (Note that the threshold θ for invoking training may be different from the threshold τ for predicting reuse; see Section III-F).

f) When a sampled entry is accessed: A hit in the sampler is evidence that the current set of features is correlated with reuse. If the value of y_{out} for the entry exceeds a threshold $-\theta$, then the predictor table entries indexed by the hashes in the accessed sampler entry are decremented with saturating arithmetic. Subsequent accesses in the context of similar features are more likely to be predicted as reused (*i.e.,* live).

H. A Replacement and Bypass Optimization

The predictor is applied to the optimization of dead block replacement and bypass. As in previous work [1], the LLC is replaced with a default replacement policy such as LRU, but a block predicted with no reuse (*i.e.,* a dead block) will be replaced before the candidate given by the default policy. In this work, the LLC uses tree-based PseudoLRU as the default replacement policy [21]. Each cache block is associated with a prediction bit that holds the most recent reuse prediction for that block.

When a block is hit in the LLC, the predictor is consulted using the current vector of features to compute the prediction bit for that block. When there is a miss in the LLC, the predictor is consulted to predict whether the incoming block will have reuse. If not, the block is bypassed. Otherwise, a replacement candidate is chosen. The set is searched for a block predicted not to have reuse. If one is found, it is evicted. Otherwise, the block chosen by the PseudoLRU policy is evicted. We call this bypass and replacement optimization *Perceptron* with a capital P to distinguish it from the predictor itself.

Tree-based PseudoLRU requires $n - 1$ bits per set for an n-way set associative cache. Thus, including the prediction bit, each block requires about two bits of replacement state, which is equivalent to the per-block overheads for DRRIP [9] and SHiP [2].

I. Prefetches

When the hardware prefetcher attempts to bring a block from the memory into the LLC, the predictor makes a reuse prediction for that block. Since hardware prefetches are not associated with instruction addresses, a single fake address is used for all prefetches for the purpose of hashing into the predictor. Other features, such as bits from the block address, continue to be used in the predictor.

IV. Methodology

A. 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 4MB L3 cache while the multi-core simulations use a 16MB 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 dead block predictor accuracy.

B. Workloads

We use the 29 SPEC CPU 2006 benchmarks. Each benchmark is compiled using a configuration file for the X86 instruction set distributed with the SPEC CPU 2006 benchmarks. 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 each workload.

g) Single-Threaded Workloads: For single-threaded workloads, the results reported per benchmark are the weighted average of the results for the individual simpoints. The weights are generated by the SimPoint tool and represent the portion of all executed instructions for which a given simpoint is responsible. Each program is run with 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.

h) Multi-Core Workloads: For 8-core multi-programmed workloads, we generated 1000 workloads consisting of mixes from the SPEC CPU 2006 simpoints described above. We follow the sample-balanced methodology of FIESTA [23]. We begin by selecting 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 8 of these regions chosen uniformly randomly without replacement. For each workload the simulator warms microarchitectural structures until 500 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 8 cores are active during the entire measurement period.

i) Avoiding Overfitting of the Parameters: Our proposed technique has several parameters whose values affect accuracy and performance. We took steps to avoid overfitting the parameters to the workloads.

For single-threaded workloads, we use a leave-out-one cross-validation methodology. That is, for each of the 29 SPEC CPU 2006 benchmarks, we explore the space of parameters that give the best aggregate performance for the other 28 benchmarks, then use those parameters for the benchmark in question. The parameters varied were the values of θ, τ_{bypass}, τ_{replace}, the bit width of the weights, and parameters related to selecting the features listed in Section IV-E. Thus, for each of the benchmarks, we report results based on parameters that were not tuned for that benchmark. We find remarkable consistency in the parameters chosen for each benchmark. For instance, when $|y_{\text{out}}| < \theta$, training is invoked. We find that for every combination of 28 benchmarks but one, the best value

is $\theta = 74$. When the benchmark sphinx3 is held out the best value is $\theta = 71$.

For multi-programmed workloads, we use a smaller set of 100 mixes to explore the parameter space to find the set of parameters that give the best aggregate performance. These 100 workloads are separate from the 1000 workloads used for the performance results. Thus, for the 1000 workload mixes, we report results based on parameters that were not tuned for those mixes.

C. Replacement Policies

We compare our proposed technique against two closely related techniques: sampling based dead block prediction driving replacement and bypass (SDBP) [1] and signature-based hit prediction (SHiP) [2]. We use SDBP code provided by the original authors. The SHiP authors provided code for their hit predictor in a form that was readily adaptable to RRIP code they also provided. We modified this code to implement the sampling-based policy described in their paper and use the program counter (PC) as the signature for the hit predictor. Thus, what we hereafter refer to as SHiP in this paper is called SHiP-PC-S in that paper's nomenclature [2]. This modification allows us to control the overhead consumed by SHiP's structures. Thus, we may allocate the same amount of storage to SHiP, SDBP, and perceptron-based reuse prediction for comparison. To be clear, we implement sampling-based SHiP using the PC as the signature and deciding the insertion position for a baseline RRIP policy with a maximum re-reference prediction value of three. In the following text and figures, *Perceptron* with a capital P refers to the replacement and bypass optimization that uses perceptron learning for reuse prediction.

We evaluate SDBP because its structure is similar to that of our predictor and it also uses summation and thresholding to make a prediction. We evaluate SHiP because it provides the best speedup in the literature for cache management based on reuse prediction. There is significant other work in reuse prediction [13], [15], [16], [17], [18], [9], [24], [21] but because of SHiP's superior performance and for space reasons we do not report results for those techniques.

1) SHiP and Bypass: SHiP was described as a placement policy for RRIP without considering bypass. It uses a threshold on the 3-bit confidence counter read from the prediction table to decide whether to place a block in the distant re-reference interval or the immediate re-reference interval. Our proposed technique and SDBP both implement bypass, so we attempt to use bypass in SHiP also. We modify SHiP to use two thresholds: one to indicate that the probability of a hit is so low that the block should be bypassed rather than placed, and a higher one to decide whether to place in the distant or immediate re-reference interval. Sampled sets are not bypassed so that the hit predictor can continue to learn from them. We exhaustively search all pairs of feasible thresholds. The configuration with the best performance is a threshold of 0 for bypass and 1 for distant re-reference interval placement. We find that SHiP with bypass yields no better speedup than SHiP without bypass on average. Thus, we report results for SHiP without bypass.

2) SDBP and SHiP with Prefetching: In the original papers, SDBP and SHiP were evaluated without modeling prefetching. To provide a realistic evaluation, we model a stream prefetcher. We found that the unmodified SDBP and SHiP algorithms perform poorly in the presence of prefetching because the prefetcher sometimes issues prefetches that hit in the cache. These hits artificially boost the apparent locality of the blocks, causing SDBP and SHiP to promote blocks that should remain near LRU. We modify the code for both SDBP and SHiP to ignore hitting prefetches and observe that the change restores the good performance of these algorithms. In Section V-B we describe results without prefetching to evaluate our work in the same methodological context as the previous work.

D. Overhead for Predictors

SHiP, SDBP, and perceptron-based reuse prediction require extra state for their prediction structures. Table I summarizes the overheads for the various techniques. SHiP uses a 14-bit signature derived as a hash of the PC indexing a table of 16,384 three-bit saturating counters. It also keeps 14-bit signatures and one reuse bit for each block in the sampled sets. SDBP keeps three tables of 8,192 2-bit saturating counters each indexed by a different 13-bit hash of the PC. It also keeps a *sampler*, an array of cache metadata kept separately from the main cache. Each block in the sampler includes a 15-bit partial tag, a prediction bit, a valid bit, a 4-bit LRU stack position and a 15-bit signature. The associativity of the sampler is 12. To keep the same amount of state for each predictor, we use 192 sampled sets for SHiP and 96 sampled sets for SDBP. Both predictors consume approximately 11KB.

The state required for perceptron-based reuse prediction is proportional to the number of features used. We empirically determined that six features provided the best trade-off between storage and accuracy (see Section IV-F). We find that using an associativity of 16 for the sampler provides the best accuracy, as opposed to the associativity of 12 that was used in the SDBP work [1]. The prediction tables for the perceptron sampler have 6-bit signed weights, and there are six such tables. We find that using 256-entry tables provides a good trade-off between accuracy and storage budget. Thus, the prediction tables consume 1.125KB.

Each perceptron sampler entry contains six 8-bit hashes of the features, a valid bit, a 9-bit sum resulting from the most recent prediction computation, a 4-bit LRU stack position, and a 15-bit partial tag. Each sampler set contains 16 such entry. To stay within the 11KB hardware budget of the other two techniques, and accounting for the prediction tables, we choose 64 sampler sets, resulting in 10.75KB of storage for the perceptron reuse predictor. The sampler structure is larger than in SDBP, but the storage for the prediction tables is smaller. In addition to the tables, the predictor keeps per-core histories of the three most recent PCs accessing the LLC. The additional storage for all of these histories is less than 100 bytes.

Technique	Overhead
SDBP	3 tables of 8,192 2-bit counters + 96-set, 12-way sampler = 11.06KB
SHiP	1 table of 16,384 3-bit counters + 192-set, 16-way sampled PCs = 11.25KB
Perceptron	6 tables of 256 6-bit signed weights + 64-set, 16-way sampler = 10.75KB

TABLE I
OVERHEAD REQUIRED BY THE VARIOUS TECHNIQUES IS APPROXIMATELY THE SAME IN OUR EXPERIMENTS. SEE TEXT FOR DETAILS OF SAMPLER ENTRY OVERHEADS.

Each of the three policies also keeps storage associated with each cache block. SHiP is built on RRIP [9] requiring two bits per cache block. The perceptron-based reuse predictor uses tree-based PseudoLRU as its default replacement policy using one bit per cache block. It adds another bit for each block to store the most recent reuse prediction for that block. Thus, SHiP and perceptron-based reuse prediction keep the same number of bits associated with each block. SDBP uses true LRU instead of PseudoLRU, so it requires 4 LRU bits plus one prediction bit for each block in the cache. We use the unmodified SDBP code rather than hack PseudoLRU into it. We stipulate that PseudoLRU and true LRU ought to behave similarly so the 5-bit overhead of SDBP should not be counted against SDBP.

The implementation complexity of the perceptron-based reuse predictor is slightly higher than that of SHiP or SDBP, but it is still reasonable. Conditional branch predictors based on perceptron learning have been implemented in Oracle and AMD processors [25], [26]. Branch predictors must operate under very tight timing constraints. Predictors in the last-level cache have a higher tolerance for latency, so we are confident perceptron learning can be easily adapted to reuse predictors.

E. Measuring Performance

In Section V we report performance relative to LRU for the various techniques tested. For single-threaded 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-core workloads, we report the weighted speedup normalized to LRU. That is, for each thread i sharing the 16MB cache, we compute IPC_i. Then we find SingleIPC_i as the IPC of the same program running in isolation with a 16MB cache with LRU replacement. Then we compute the weighted IPC as $\sum \text{IPC}_i / \text{SingleIPC}_i$. We then normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

F. Features for the Predictor

We determined that the following six features provide a good trade-off between hardware overhead and prediction accuracy. Suppose PC_i is the address of the i^{th} most recent instruction accessing the LLC from the current thread, where PC_0 is the PC of the current memory access instruction. Then the features are:

1) PC_i shifted right by 2, for $i = 0$.
2) The three values of PC_i shifted right by i, for $1 \leq i \leq 3$.
3) The tag of the current block shifted right by 4,
4) The tag of the current block shifted right by 7.

We choose not to use features such as the number of accesses to a block [18] or other history of the block itself because of the overhead required in storing such information per block.

1) Discussion of Features: The first feature, the PC of the current memory access instruction, must be shifted because all of the features will be XORed with the PC before indexing the tables. Without shifting, the value would simply be 0. Shifting and XORing gives a hash of the PC, which is similar to the hashed PC used by SDBP.

The next three features are the most recent PCs of memory access instructions that accessed the LLC, each shifted right by an increasing number. These features give the predictor an idea of the "neighborhood" the instruction was in when it made the access. Shifting by increasing amounts allows a wider perspective for events further in the past, *i.e.,* rather than specifying the instruction at PC_3, the predictor learns from the 8-byte region around that instruction.

The last two features are two shifts of the tag of the currently accessed block. These features trade off resolution for predictor table capacity: shifting the tag right by 4 allows the predictor to distinguish between more memory regions at the cost of a higher chance of aliasing, while shifting right by 7 allows grouping many pages that could have the same behavior into the same predictor entry, at the cost of coarser granularity. If one of the features has higher correlation with reuse, its weight will have high magnitude and contribute more to the prediction.

It is important to note that **not all features are expected to correlate with reuse behavior for all blocks**. Some features will have no correlation. Only the features with high correlation will contribute significantly to the prediction.

G. Other Parameters

The parameter τ is the threshold below which a block is predicted to be reused. We chose two different values for τ based on the purpose of the prediction: τ_{bypass} for predicting whether a block should be bypassed, and τ_{replace} for predicting whether a block may be replaced after a hit. On 100 multi-programmed workloads (as described in Section IV-B), we empirically found the best values were $\tau_{\text{bypass}} = 3$ and $\tau_{\text{replace}} = 124$. The best value for θ, for which training is triggered when the magnitude of a correct prediction is below θ, was 68. Recall that, even though $\theta = 68$, training will still be invoked if the prediction is incorrect. These parameters were used to provide the results for the 1000 multi-programmed workloads (that exclude the 100 used for training). For single-thread workloads, the parameters were chosen according to

the cross-validation methodology described in Section IV-B to avoid overfitting.

V. Results

This section gives results for the various policies tested. It presents accuracy, cache misses, and speedup results. First, results are given for multi-core workloads. Then, results are given for single-core workloads.

A. Multi-Core Results

This section gives results for the 1000 8-core multi-programmed workloads.

1) Performance: Figure 5 shows weighted speedup normalized to LRU for SDBP, SHiP, and Perceptron with a 16MB last-level cache. See IV-E for the definition of weighted speedup. The figure shows the speedups for each workload in ascending sorted order to yield S-curves. SDBP yields a geometric mean 4.3% speedup and SHiP yields a 4.4% speedup. Perceptron gives a geometric mean speedup of 7.4%. The superior accuracy of Perceptron gives a significant boost in performance over the other two reuse predictors.

2) Misses: Figure 6 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. Perceptron, at an arithmetic mean 7.2 MPKI, delivers fewer misses than the other techniques. LRU, SDBP, and SHiP yield 9.4 MPKI, 7.8 MPKI, and 7.6 MPKI, respectively.

B. Without Prefetching

The original SDBP and SHiP studies were presented in simulation infrastructures that did not include prefetching. Prefetching has a significant effect on cache management

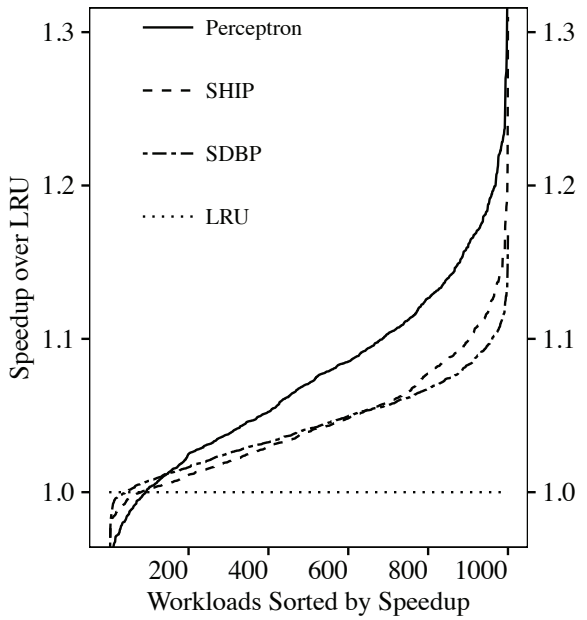

Fig. 5. Normalized Weighted Speedup over LRU for 8-Core Multi-Programmed Workloads

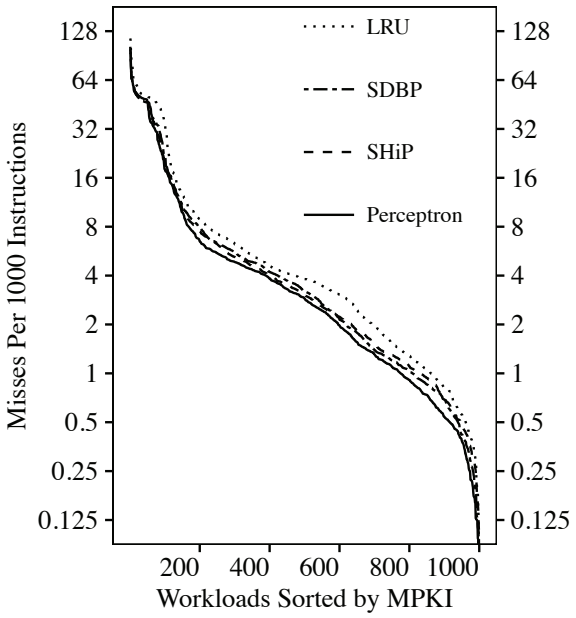

Fig. 6. Misses per 1000 Instructions for 8-Core Multi-Programmed Workloads

studies because many more references to memory are made. We believe cache management is best studied alongside prefetching since modern high performance processors include hardware prefetchers. As stated in Section IV-C2, we modify SDBP and SHiP to remove the confusing influence of prefetching, boosting their performance. Nevertheless, to put our work in the proper context of previous work, we run experiments with prefetching disabled for all techniques including the baseline LRU policy, SDBP, SHiP, and Perceptron. We do not illustrate the results with graphs for space reasons. The geometric mean normalized weighted speedups for SDBP, SHiP, and Perceptron are 7.5%, 8.3%, and 11.2%, respectively. The first two numbers are consistent with the conclusions of the previous work: SHiP has an advantage over SDBP. The third number shows that Perceptron continues to yield significant speedup over SHiP and SDBP. Note that all of the speedups are higher than the speedups obtained with prefetching. Prefetching eliminates some of the opportunity for optimization by replacement policy since it turns many potential misses into hits. Our LRU baseline with prefetching achieves a geometric mean 37% speedup over no prefetching, so obviously we are satisfied to accept a slight decrease in the advantage provided by an improved replacement policy.

1) Accuracy: Figure 1 (from Section I, the introduction) illustrates the accuracy of the three predictors, giving violin plots for the false positive and coverage rates of each predictor. Violin plots show the mean as a bar in the center of the plot along with the probability density of the data. The coverage rate is the percentage of all predictions for which a block is predicted not to be reused. The false positive rate is the percentage of all predictions for which reuse was incorrectly predicted.

To summarize the figure, the plots show that SDBP and

SHiP have higher mean false positive rates and lower coverage rates than Perceptron, but also higher variance among workloads for the false positive and coverage rates. The Perceptron probability density graphs are shorter, showing that, in addition to superior accuracy and coverage, Perceptron delivers a more dependable and consistent range of accuracy and coverage rates.

The coverage for perceptron-based prediction is higher than that for SDBP and SHiP (52.4% versus 47.2% and 43.2%, respectively) while the false positive rate is lower (3.2% versus 7.4% and 7.7%, respectively). Note that, while SHiP has a slightly higher false positive rate than SDBP, it delivers superior performance to SDBP. SHiP only predicts at placement, while SDBP predicts on every access, so the denominators in the false positive and coverage rates are different. Coverage represents the opportunity for optimization given by the predictor, while false positives represent the potential cache misses caused when an incorrect prediction leads to a live block being replaced. Thus, the perceptron scheme leads to increased opportunity for optimization while reducing false positives by a factor of 2. The plots also show that the coverage and false positive rates have far less variance for perceptron versus the other schemes. The standard deviation false positive rates for SDBP, SHiP, and perceptron learning are 3.3%, 3.6%, and 1.7%, respectively. Thus, we may more reliably depend on the perceptron scheme to deliver consistent accuracy.

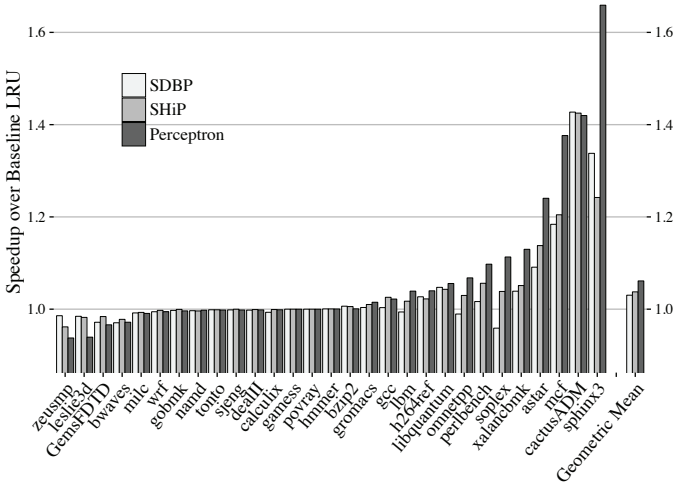

Fig. 7. Speedup over LRU for Single-Thread Workloads.

C. Single-Core Results

This section discusses the single-thread performance and misses for the 29 SPEC CPU benchmarks with a 4MB LLC. Prefetching is enabled.

1) Performance: Figure 7 shows single-thread speedup of the techniques over LRU. The benchmarks are sorted by speedup with Perceptron. SDBP and SHiP achieve a geometric mean 3.5% and 3.8% speedup over LRU, respectively. Perceptron yields a 6.1% geometric mean speedup. For 25 out of the 29 benchmarks, Perceptron matches or exceeds the performance of LRU, and for most workloads it exceeds the performance of the other two techniques. Many of the benchmarks experience few misses on a 4MB cache. We define a memory-intensive subset of SPEC as benchmarks with an MPKI of at least 1.0 under the LRU replacement policy. These benchmarks are GemsFDTD, sphinx3, xalancbmk, cactusADM, lbm, astar, soplex, omnetpp, milc and mcf. On this subset (not separately illustrated), perceptron learning yields a geometric mean 18.3% speedup, versus 10.5% for SHiP and 7.7% for SDBP.

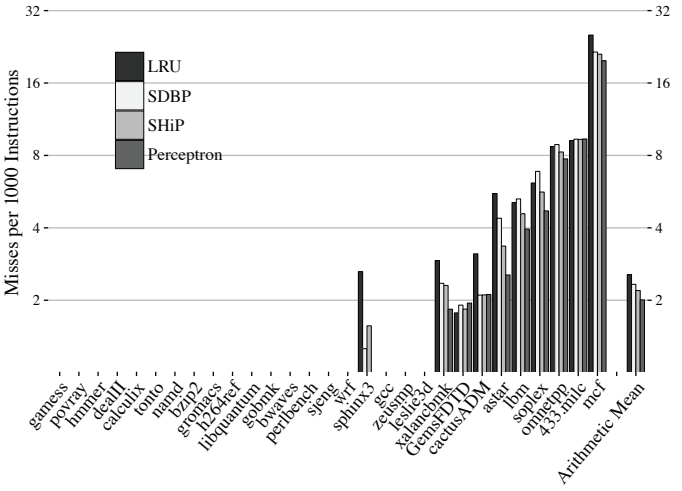

Fig. 8. Misses per 1000 Instructions for Single-Thread Workloads.

2) Misses: Figure 8 gives the MPKI for the 29 benchmarks Note the y-axis is a log scale. SDBP and SHiP have 2.3 and 2.2 MPKI, respectively. Perceptron gives 2.0 MPKI. The average MPKI figures are low because most of the benchmarks fit a large part of their working sets into a 4MB cache. For omnetpp, a benchmark with considerable misses, SDBP gives 8.3 MPKI, SHiP yields 9.2 MPKI, and Perceptron gives 7.7 MPKI.

D. Cache Efficiency

Cache management with reuse prediction improves cache efficiency. Cache efficiency is defined as the fraction of time that a given cache block is live, *i.e.,*, the number of cycles that a block contains data that will be referenced again divided by the total number of cycles the block contains valid data [27]. A more efficient cache makes better use of the available space and wastes less energy storing dead blocks. Figure 9 illustrates cache efficiency for the various techniques on a typical multi-programmed workload. In the four 128×128 heat maps, each pixel represents the average efficiency of one set in the 16MB cache, with higher efficiency represented by lighter values. The LRU cache is quite dark. SDBP and SHiP are clearly more efficient, but Perceptron is the lightest, thus it is the most efficient. On average over all blocks and 1000 multi-programmed workloads, LRU yields an efficiency of 21%, *i.e.,* at any given time 21% of blocks are live and 79% are

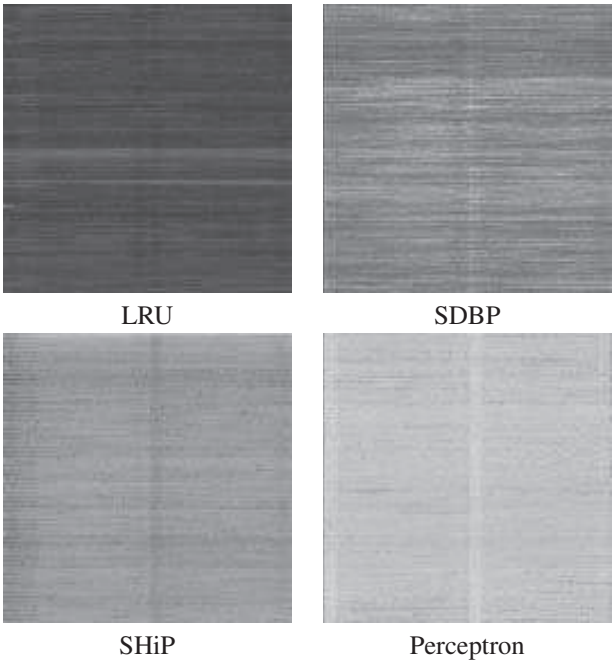

Fig. 9. Heat map of cache efficiency in cache sets for the various techniques on a typical multi-programmed workload. Darker signifies poor efficiency while lighter means better efficiency.

dead. SDBP gives an average efficiency of 47% and SHiP gives an efficiency of 43%. Although SHiP outperforms SDBP, its efficiency is somewhat lower because it does not bypass, so some dead blocks inevitably enter the cache and depress efficiency. Perceptron has 54% efficiency. Thus, Perceptron more than doubles the efficiency of the cache over LRU.

E. Analysis

Perceptron learning improves accuracy over previous techniques by being able to incorporate more input features such that highly correlated features make a large contribution to the prediction while uncorrelated features make little contribution.

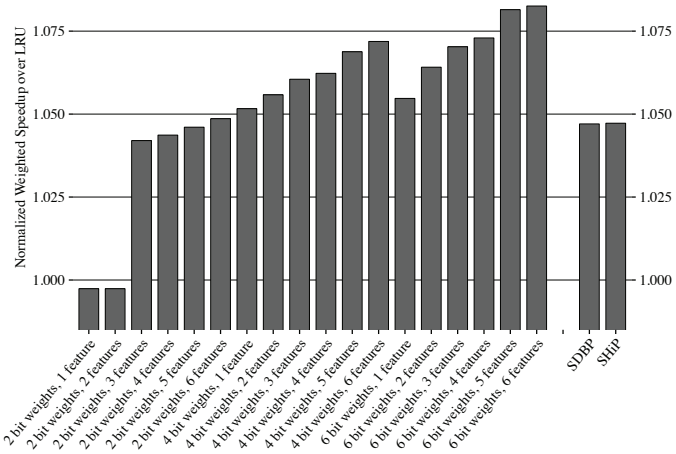

Fig. 10. Cumulative impact of increasing numbers of features and weight widths.

Wider weights allow more discrimination between levels of correlation. More features allow more correlations to be found. Figure 10 shows the results of experiments using different weight widths and numbers of features, as well as for SDBP and SHiP for reference. The graph shows the geometric mean normalized weighted speedup over the multi-programmed workloads with a 16MB LLC. Features are added in the order they appear in Section IV-F. Clearly, wider weights and more features contribute to improved performance. Considering the global history of PCs and memory addresses individually allows finding correlations not apparent from any one feature or from combining features into a single index as in previous work.

VI. Conclusion

The conclusion goes here. This paper describes reuse prediction based on perceptron learning. Rather than using a single feature, or a hashed combination of features indexing a single table, perceptron learning allows finding independent correlations between multiple features related to block reuse. The accuracy of perceptron-based reuse prediction is significantly better than previous work, giving rise to an optimization that outperforms state of the art cache replacement policies. The complexity of perceptron-based reuse prediction is no worse than that of branch predictors that have been implemented in real processors. In this work, the features we focus on are the addresses of previous memory instructions and various shifts of the currently accessed block. In future work, we intend to explore how other features may be incorporated without increasing the overhead of the technique. We also plan to explore how perceptron-based prediction might help with other cache management tasks such as reuse distance prediction and prefetching.

Acknowledgments

Elvira Teran and Daniel A. Jiménez are supported by NSF grant CCF-1332598 as well as a gift from Intel Labs. We thank the anonymous reviewers for their helpful feedback. We thank Paul Gratz, Akanksha Jain, Sangam Jindal, Jinchun Kim, Calvin Lin and Samira Mirbagher for their feedback on drafts of this paper. We thank Samira M. Khan who helped us with prefetching methodology.

References

[1] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *MICRO*, pp. 175–186, December 2010.

[2] 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*, MICRO-44, (New York, NY, USA), pp. 430–441, ACM, 2011.

[3] H. D. Block, "The perceptron: A model for brain functioning," *Reviews of Modern Physics*, vol. 34, pp. 123–135, 1962.

[4] M. L. Minsky and S. A. Papert, *Perceptrons, Expanded Edition*. MIT Press, 1988.

[5] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 197–206, January 2001.

[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] A. Seznec, "Genesis of the o-gehl branch predictor," *Journal of Instruction-Level Parallelism (JILP)*, vol. 7, April 2005.
[8] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting compressed block size as an indicator of future reuse.," in *HPCA*, pp. 51–63, IEEE, 2015.
[9] 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.
[10] 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)*, pp. 245–250, 2007.
[11] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, (Washington, DC, USA), pp. 167–178, IEEE Computer Society, 2006.
[12] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *International Symposium on Computer Architecture*, pp. 139 – 148, 2000.
[13] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," *SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 144–154, 2001.
[14] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Memory coherence activity prediction in commercial workloads," in *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, (New York, NY, USA), pp. 37–45, ACM, 2004.
[15] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: predicting and optimizing memory behavior," *SIGARCH Comput. Archit. News*, vol. 30, no. 2, pp. 209–220, 2002.
[16] J. Abella, A. González, X. Vera, and M. F. P. O'Boy le, "Iatac: a smart predictor to turn-off l2 cache lines," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 1, pp. 55–77, 2005.
[17] 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), pp. 222–233, IEEE Computer Society, 2008.
[18] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.
[19] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 292–303, June 1997.
[20] G. H. Loh and D. A. Jiménez, "Reducing the power and complexity of path-based neural branch prediction," in *Proceedings of the 2005 Workshop on Complexity-Effective Design (WCED'05)*, pp. 28–35, June 2005.
[21] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 284–296, 2013.
[22] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, 2003.
[23] A. Hilton, N. Eswaran, and A. Roth, "FIESTA: A sample-balanced multi-program workload methodology," in *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2009.
[24] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, (Washington, DC, USA), pp. 389–400, IEEE Computer Society, 2012.
[25] 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, "Sparc t4: A dynamically threaded server-on-a-chip," *IEEE Micro*, vol. 32, no. 2, pp. 8–19, 2012.
[26] A. Fog, "The Microarchitecture of Intel, AMD, and VIA CPUs." http://www.agner.org/optimize/microarchitecture.pdf, 2014.
[27] D. Burger, J. R. Goodman, and A. Kagi, "The declining effectiveness of dynamic caching for general-purpose microprocessors," *Technical Report 1261*, 1995.