

Dynamic Set Stealing to Improve Cache Performance

Brady Testa
brady.testa@gmail.com
Texas A&M University

Samira Mirbagher-Ajorpaz
smirbag@ncsu.edu
North Carolina State University

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

Abstract—In the last-level cache (LLC), replacement policy is dependent on workload characteristics. Adapting the policy to the current workload has been an active area of research. Previous works include set dueling exemplified by DIP [40] which uses static replacement policies and machine learning based models such as Glider [45] or Multiperspective Reuse Prediction [19]. Both provide improvement over the least-recently-used (LRU) policy, but additional improvement is possible. DIP suffers from wasted resources as leading sets of the competing policies are fixed in size. Machine learning approaches each use a fixed set of features that were selected offline that are not optimal for all workloads. We introduce Set Stealing with Perceptron Tables (SSPT), a novel resource management policy that allows combining the strengths of many replacement policies to maximize performance and eliminate wasted processor resources.

This policy achieves a 9.45% geometric mean speedup over a baseline LRU policy on a set of 81 SPEC benchmark workloads, compared to Glider's 9.28% and Multiperspective Reuse Prediction at 7.62% assuming a 2 MB LLC. We achieve a 9.70% geometric mean speedup over LRU on a set of 90 big data (GAPS and XS) workloads, compared to Glider's 8.79% and Multiperspective Reuse Prediction at 7.86% assuming a 2 MB LLC.

Index Terms—microarchitecture, prediction, cache replacement

I. INTRODUCTION

Recent decades have seen a large increase in the disparity between DRAM and processor speeds. It has become increasingly important to improve cache performance in order to improve overall system performance. The last-level cache (LLC) is the last line of defense against large DRAM latencies. Each LLC miss requires a costly access to DRAM. Thus, it is very important to minimize cache misses in order to avoid expensive DRAM accesses. A good cache replacement policy is essential to the effectiveness of the cache. The most famous of these policies is the Least Recently Used (LRU) policy. Often used as a baseline for other policies, LRU forms a recency stack of blocks in every cache set, ordered by the most recent access to the block. On every cache miss, the LRU block is evicted from the cache and the new block is placed at the Most Recently Used (MRU) position.

State of art cache replacement uses a simpler linear learning model *i.e.* support vector machine (SVM) [45] and perceptron learning [18] to predict the reuse behavior of the current memory access online, driving the replacement policy to retain blocks likely to be reused and evict blocks unlikely to be reused [20], [49]. These policies are based on the assumption that offline models should be able to reason about time ordered events online. Glider [45] trains a RNN model offline for individual programs and interpret this offline model to reveal an important insights. Multiperspective reuse prediction

(MPPPB) [20] uses a fixed set of features found by stochastic search offline on a set of training workloads.

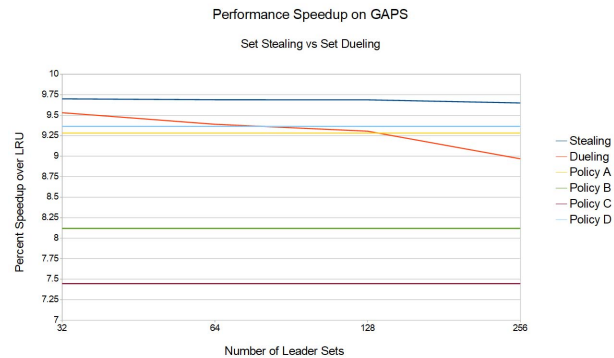


Fig. 1. Speedup obtained using different configurations of SSPT. Stealing depicts our novel stealing process, using different numbers of starting sets, with a minimum of 8 sets. Dueling shows our algorithm without allowing for sets to be reallocated. Policies A,B,C,D depict the individual policies' performance without using any set stealing or set dueling.

Unfortunately, the fixed set of features is a compromise that achieves good performance over general-purpose workloads but may perform suboptimally for any given workload. In a perfect world we would pick multiple high performing policies and select the best policy for each workload on a case by case basis, unfortunately, allocation of the optimal policy for each workload cannot be done in real time.

Set dueling [40], [41] uses *leader sets* that are distributed between two policies. These leader sets are dedicated to tracking a specific policy. Because of this, set dueling requires locking those leader sets to a sub-optimal policy, *i.e.* some leader sets will always be allocated to the policy that does not work as well. Two dueling policies on a given workload may not reach the same performance as one policy running on the workload because we may accidentally choose the wrong policy for part or all of the program, in some extreme cases, a single policy may even exceed the maximum performance of the two dueling policies. Set dueling has a fixed number of sets that are allocated to each policy regardless of which policy is being currently utilized, resulting in suboptimal utilization of valuable processor resources. This is the key opportunity of improvement for set dueling.

In this work we introduce *dynamic set stealing*, allowing to shift processor resources between the dueling policies as needed to maximize performance while minimizing wasted

resources resulting in improved overall performance. While set stealing initially may on the surface seem simple, in practice it is more nuanced. Simply shifting sets on a one to one basis between policies will bias the algorithm towards the policies with fewer sets because fewer sets results in fewer cache accesses and fewer associated cache misses. Thus we also introduce a balancing mechanism and new sets of thresholds and positions to counteract this effect.

Figure 1 shows the speedup derived from different initial choices of number of leader sets (N_{start}) when using set dueling vs set stealing. In set dueling, we have a fixed number of sets dedicated to each policy, which degrades efficiency at larger numbers of leader sets as a fixed number of sets are tracking suboptimal policies. Set stealing averts this issue by allowing more leader sets for the better policy.

A. Contributions

This paper makes the following contributions: 1) We propose the first replacement policy that dynamically allocates leader set configurations between arbitrarily many competing predictive replacement policies. The accuracy of our technique is superior to previous work. 2) We apply this strategy with perceptron tables, providing a vast design space while remaining within a reasonable hardware budget. 3) The new policy achieves a 9.45% geometric mean speedup over a baseline LRU policy on a set of 81 single-threaded SPEC traces, compared to Glider's 9.28% and Multiperspective Reuse Prediction at 7.62% assuming a 2 MB LLC. We show a 9.70% geometric mean speedup over a baseline LRU policy on a set of 90 big data benchmark single-thread traces, compared to Glider's 8.79% and Multiperspective Reuse Prediction at 7.86% assuming a 2 MB LLC.

II. RELATED WORK

A discussion of previous work regarding the topic of cache management policies begins with Bélády's [4] paper. He describes a theoretically optimal cache management policy in which the block that is chosen to be evicted is the one that will be referenced farthest into the future. This of course requires knowledge of future cache accesses, and thus cannot be implemented in practice. Bélády's work provides a strict lower bound on how few cache misses we can expect from a cache replacement policy on a given trace of memory accesses. Note that this is not always synonymous with performance, particularly in the case of out-of-order execution and memory-level parallelism [41].

Set-dueling was introduced by Qureshi *et al.* in a paper on using memory-level-parallelism to guide replacement policy [41] and further developed in work from the same group on dynamic insertion policy [40]. Khan *et al.* extend this work to consider dueling multiple policies in a space efficient way [26]. Our set-stealing idea is orthogonal to previous work on set-dueling; that work used a fixed number of leader sets per policy, while we explore varying the number of leader sets per policy.

A. Hawkeye and Glider

Another machine learning approach to cache management was introduced by Shi *et al.* with their algorithm Glider [45]. Glider is an adaption of an offline long short-term memory

(LSTM) model that builds on their previous Hawkeye [14] algorithm that uses an approximation of Bélády's algorithm on past cache accesses to provide predictions on future cache accesses. It currently represents the state of the art in cache replacement algorithms. The LSTM model in Glider is simplified into an Integer Support Vector Machine (ISVM) that uses a k -sparse binary feature. This feature uses a one-hot representation to denote whether a certain PC was accessed within a sequence of cache accesses. The most recent PCs seen by each core are stored in the PC History Register. 2048 ISVMs of 16 weights are used to form the predictor. Each group represents a different PC, and the 16 weights correspond to the k -sparse binary feature. Whenever a prediction is made, the ISVM corresponding to the current PC is queried, and five weights are added together to form a prediction.

Techniques from machine learning have already been found to be both effective and efficient to implement in hardware when applied to other problems in microarchitecture prediction. Jiménez and Lin [18] describe a branch predictor that uses perceptrons to predict the outcome of conditional branches.

B. Other Previous Work in Reuse Prediction

Reuse prediction is an extension of dead block prediction, the idea predicting whether a block will be referenced again before it is evicted. There are many dead block predictors described in the literature [1], [12], [28], [29], [31], [32], [35], [38], [39], [47], [54]. Dead block prediction has been evaluated in the context of making replacement decisions in the L1 data cache [7], [19], [28], [35], [50], [54], L1 instruction cache and branch target buffer [38], last-level cache [27], [28], [35], prefetching [7], [12], bypassing [6], [8], [9], [16], [22], [23], [34], [37], [42], [43], [48], [52], [55], translation look aside buffer [39], power reduction [2], [24], and cache coherence protocol optimization [31], [33], [47].

Our work uses of the idea of a *sampler* from Sampling Dead Block Prediction (SDBP) [28]. 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 [36]. 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 a 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. SHiP is a similar policy that uses prediction to drive placement in a cache replaced by re-reference interval prediction [15], [54]. SHiP improves over SDBP by placing predicted live blocks in a more favorable RRIP position than predicted dead blocks, and only acting

during placement rather than subsequent accesses to a block. The underlying RRIP policy is also superior to LRU.

C. Interaction with Prefetching

More recent previous work has focused on the importance of taking into account prefetching into cache replacement. Kill-the-PC [30] is a combined cache replacement and prefetching policy in which the cache replacement policy and prefetcher cooperate to make placement decisions. Harmony [13] is a cache replacement algorithm similar to Hawkeye that incorporates prefetching into Bélády's algorithm. Our work is focused on replacement, but our simulations model an L2 prefetcher so that the work is relevant to real systems, the vast majority of which use data prefetchers.

III. BACKGROUND AND MOTIVATION

A. Set Dueling

Qureshi *et al.* introduce the notion of set dueling [40], [41] that we extend in this work. Dynamic insertion policy (DIP) [40] uses LRU and MRU replacement policies that are tested on a small group of "leader sets." Counters are kept to track which policy is registering fewer misses, and the cache will use that policy on the remaining sets in the system. In this work, we extend the notion of leader sets and set dueling to allow the leader set partition to be reallocated to the winning policy. Later work discusses splitting each cache set into two partitions, one for dirty lines, and one for clean lines [25]. A sampler is used in this system to determine the number of lines that should be allocated to the dirty and clean partitions. The work also uses instruction PCs to better determine the behaviour of each instruction, and whether or not it is associated with block reuse.

B. Perceptron Learning for Cache Replacement

Teran and Jiménez [20], [49] use a form of perceptron learning well suited to microarchitecture. They use the notion of a sampler to train a set of prediction tables of weights on a set of features. When a prediction is made, the tables are referenced at indices corresponding to the features being used, and the weights in the table are summed and compared to different thresholds to decide whether or not a block should be bypassed, promoted, or left as is. The predictor predicts whether the block is dead, *i.e.* not reused. When the block is reused or evicted, the predictor is updated using the perceptron learning rule: if the prediction is incorrect, or if the magnitude of the sum fails to exceed a certain threshold, the weights that were summed are incremented if the block was dead, or decremented if the block was not dead. Saturating arithmetic is used to ensure weights stay within a certain range.

Recently, Jamet *et al.* propose a technique to choose between sets of perceptron features in an optimization aimed at cache replacement for big-data workloads [17]. That work uses four sets of features in two optimizations using multiple samplers, necessitating a very large hardware budget compared to other techniques. Our technique has a smaller hardware budget than previous work.

C. Motivation

We first examine the motivations behind extending the notion of set dueling by introducing the informal concept of policy orthogonality. Informally, we note that some policies perform well on some workloads, but worse on others. When we have another policy that works better on the workloads that its counterpart does not work well on, we call these policies *orthogonal*.

Observation 1: *It is important to note that multiple dueling policies on a given workload may not reach the same performance as one policy running on the workload because we may accidentally chose a poor policy for part or all of the program, or in some extreme cases, we may even exceed the maximum performance of the policies.*

Ideally, we would first like to pick policies with a high level of orthogonality that both perform well, and we would like to select the optimal policy for each workload on a case by case basis. The former can be done reasonably well, the latter cannot be done in real-time. What we can do instead is use set dueling to attempt to have our cache use the best of the multiple policies in real time. If dueling is occurring between multiple policies, then we want to select those policies in a manner such that if a policy performs poorly on a workload, another policy performs well.

Observation 2: *In the case of set dueling, we have a fixed number of sets being used on each policy, and in all cases, these sets will continue to be used to test the lower performance policies.*

With set stealing, if one policy is clearly showing better performance, then we want to be able to reallocate some of the leader sets to the better policy. This on average will increase performance and allow us to use more leader sets with less fear of having too many sets being stuck on a worse policy. Our algorithm facilitates this process by reassigning leader sets to the winning policies so that more sets are running on the better policy.

IV. SET STEALING

In this section we introduce the notion of set stealing and the intuition behind it, which allows dynamically reallocating the sets sampled for set dueling. A few parameters are introduced into this algorithm for two reasons: to give a high level of customizability, and to account for various problems. For example, the values by which the score changes differs depending on the number of leader sets in each policy. This is to account for the fact that set access is not uniform, as well as account for the imbalance of sets as more and more are stolen.

Observation 3: *When more sets become stolen, then a policy with fewer leader sets allocated to it will be less likely to register a miss. This would favor that policy, and would be more likely to force a steal in the opposite direction. The end result is a bias toward policies with fewer leader sets allocated.*

purposed by Teran and Jiménez [20], [49], but have several novel additions.

We use the predictor for three separate cases: (1) to calculate y_{out} for the sampler set on every access, (2) to determine whether an incoming block should be bypassed, and (3) to determine how to treat a block that has been reused in the LLC already. Each circumstance differs in the thresholds used, and the actions taken. Figure 2 describes the datapath at a high level.

Tables of counters We introduce a set of 10 tables of 256 6-bit counters. Whenever we make a prediction, we index the predictor with 10 indices from the table, and take the sum of all of the values read. This resulting value is then compared against a threshold that is dependent on the context of the prediction.

We also introduce 32 per-PC sets of tables with 7 features, 64 counters per feature, and 6-bit counters. These tables are updated according to the PC that accesses them, and the features used are taken from indices already stored by the sampler.

Sampler In order for us to gain useful information on the behaviour of the program, we need to train these perceptron tables to best predict future reuse behaviour. The first step in doing so is collecting the data to be used. To this end, we introduce m_{start} sampler sets¹. These are not to be confused with leader sets described previously. These sampler sets will simulate a smaller cache that we will use to train the predictor, mirroring $|m_{start}|$ cache sets in the LLC. Whenever one of those sets are accessed in the LLC, we will also update the sampler for policy P_i and the perceptron tables if it was a hit or if the incoming block is predicted to be above a threshold F_i . These sampler sets will be updated in LRU fashion. Each line in the sampler will contain a tag, LRU bits, 10 1-8 bit indices for the perceptron corresponding to the accessing instruction, and a 13-bit y_{out} field that stores the last prediction value.

Live or Dead Features Each feature is associated with a value $A \in [0, 1, \dots, assoc - 1, assoc]$ in which a feature is considered dead for training purposes. Specifically, the table is incremented on any access if it matches the recency parameter, and is subsequently ignored afterwards. This is to roughly simulate the feature being present in a sampler with smaller associativity.

Sampler Miss Whenever we miss in the sampler, we need to update the sampler according to LRU policy, and update the perceptron tables. If the incoming block is predicted to be above a threshold F_i , and if for the evicted block $|y_{out}| \leq \theta$ or $y_{out} < t_{revive}$, then we increment the counters referenced in the indices stored by evicted block, and insert the incoming block into the sampler. Otherwise, we do nothing. Per-PC tables will ignore the thresholds mentioned.

Sampler Hit Whenever we hit in the sampler, we likewise update the sampler sets by the LRU policy, and update the perceptron tables. If for the accessed block $|y_{out}| \leq \theta$ or $y_{out} > t_{revive}$, then we decrement the counters referenced

¹Note that both policies can sample the same set. In this case, we run this procedure for both samplers. This is in contrast to leader sets, which are part of the cache and are associated with one policy.

in the indices by one. Otherwise, we also do nothing. After any cache access, a block in the LLC has a single bit modified that represents whether it is predicted to be reused. We refer to these bits as the reuse bits. This reuse bit is set to one if our prediction is less than t_{revive} . Note that two such reuse bits exist for every line, one being used for each policy. Per-PC tables will ignore the thresholds mentioned.

Algorithm 2: Pseudocode for updating samplers

```

hit ← false
if cache access is a sampler2set S then
  for each tag  $t'$  in S do
    if  $t' = t$  then
      hit ← true
    end if
  end for
  pred ← predict(B)
  if hit == true then
    if  $B.Y_{out} \leq \theta$  or  $B.Y_{out} > T_{revive}$  then
      for x from 0 to the number of features do
        if current LRU value != feature associativity parameter then
           $P[x][get(B)[x]] - \{decrement the corresponding weight\}$ 
          Update PC-table
        end if
      end for
    end if
  else if hit == false then
    if  $B.Y_{out} \leq \theta$  or  $B.Y_{out} > T_{revive}$  then
      for x from 0 to the number of features do
        if current LRU value != feature associativity parameter then
           $P[x][get(B)[x]] + \{increment the corresponding weight\}$ 
          Update PC-table
        end if
      end for
    end if
  end if
  newInd ← get()
  update(newInd, predict(newInd))
end if

```

A. Using the Predictors

Make a Prediction To make a prediction for a block, we take the sum of the associated features in the per-PC table, and the general tables.

Bypass To determine whether or not to bypass the incoming block, we query the predictor, bypassing if our prediction is greater than some t_{bypass} . If we do not bypass, then we need to determine which block to evict. The algorithm will first try and find the closest block to LRU that is marked dead by a one-bit field. If no such block exists, then we simply evict the block in the LRU position.

Updating Metadata in the Cache When we access a block in the LLC, after updating the single bit reuse field, we also move it according to the prediction. The reason we shift the recency position when we access a block already in the cache is that we want to be able to consider how likely reuse for the block is, even after already being accessed.

We introduce four integers $\pi_0, \pi_1, \pi_2, \pi_3$, and τ_1, τ_2, τ_3 . The only restrictions are that $\pi_0, \pi_1, \pi_2, \pi_3 < t_{revive}$, and $\tau_1, \tau_2, \tau_3 \in [0, 1, \dots, assoc - 1, assoc]$. These parameters serve as thresholds for how confident the predictor is that the incoming block will be reused. When we receive higher values

from the predictor, we are not as confident that the block will be reused, so we place it in a position closer to MRU so that it will leave the cache sooner unless it is referenced.

Let our prediction be referred to as p . On cache hits, then if $\pi_0 \leq p \leq t_{revive}$, then do not move the recency position of the block. If $\pi_{i+1} \leq p \leq \pi_i$, then move the recency position back to the τ_{i+1} position. For example, if our prediction lies between π_1 and π_2 , and $\tau_2 = 7$, then a block in the LRU position should be placed in the 8th recency position.

When we insert a block into the LLC, we follow a similar pattern as above. We introduce a new set of thresholds and positions: $\pi_0^i, \pi_1^i, \pi_2^i$, and $\tau_1^i, \tau_2^i, \tau_3^i$. On cache misses, when $Y_{out} > \pi_0^i$, we insert at position τ_1^i . When $\pi_0^i > Y_{out} > \pi_1^i$, we insert at τ_2^i . When $\pi_1^i > Y_{out} > \pi_2^i$, we insert at τ_3^i , and when $\pi_2^i > Y_{out} > \pi_1^i$, we insert at LRU. Similarly, when we receive a higher prediction value, we are less certain about whether the block will be reused, so we don't want to promote it to LRU and risk having a dead block in the cache for longer than is needed.

VI. FEATURES

We now turn our attention to the features that will be used. Features are properties of the program in a given context, including PC, tag, block offset, and previous program behaviour. These features are limited to 1-8 bit values used as the indices for the tables. When we store these features into a sampler block, or reference them to make a prediction, we will store those 1-8 bits, depending on the feature. The set of possible features is derived from previous work by Teran and Jiménez [20], [49]. The list of possible features is as follows:

- $pc(A, S, W, X)$
- $tag(A, S, W, X)$
- $insert(A, X)$
- $burst(A, X)$
- $lastmiss(A, X)$
- $bias(A, X)$
- $offset(A, X)$

The A term refers to the recency parameter discussed above. The S term refers to the s -bit right shift that is applied to the feature. W refers to the w^{th} accessing instruction history (for example, $w = 1$ means the value from the previous cache access.)

The X Boolean determines if the whole feature should be XORed with the current PC. For example, if we see $pc(4, 2, 2, 1)$, we should interpret that to mean that the feature is the PC from two instructions ago, shifted right by 2, XOR'd with the current PC, and treated dead after reaching position four in the sampler set.

Note that the PC and tag features will require that a running history of the most recent values of the features will need to be stored. This paper assumes 16 positions will be needed, although the exact number will vary depending on the features selected.

Tag follows a similar logic, where the w^{th} tag is shifted right s bits, XORed with the current PC if X is true, and treated dead past the A^{th} position in the sampler sets.

Insert is 1 if the current access was a cache miss, and 0 if it was a cache hit. If X is true, then this feature is indexed by

a table with 256 indices, if it is not, then it is indexed by a table with 2 indices.

Burst is 1 if the current access is one to a block in the MRU position. If X is true, then this feature is indexed by a table with 256 indices, if it is not, then it is indexed by a table with 2 indices.

Lastmiss is a 1 if the current access to a block was to a block that had previously missed on the last access to it. If X is true, then this feature is indexed by a table with 256 indices, if it is not, then it is indexed by a table with 2 indices.

Bias is simply always 0. It will only need a single counter for its table, unless X is true, in which case we use 256 counters.

Offset takes the offset from the tag. Different offsets suggest different fields are being accessed, and this information may suggest a block is more likely to be reused. This will have up to 256 counters (dependent on the number of offset bits) in the case that X is false. If it is true, then it will have 256 counters guaranteed.

A. Feature Selection and Thresholds

In this section we explain the process of generating the features and thresholds necessary to use these techniques. Unfortunately, the search space for this problem is astronomically large; merely checking every possible recency position and every possible n -th entry in the PC/tag history chain while keeping every other parameter constant would require more than 2^{256} evaluations. This unfathomably large number renders a complete brute force approach computationally intractable.

To solve this problem, we used a combination of manual modifications to the configuration as well as hill climbing on a set of 35 traces that include SPEC, XS, and GAPS. None of the traces used for training were used in the performance reported below.

The policies we chose at the end of this process are superior to prior policies *i.e.* LRU and MRU which compete in the original set dueling. We show that set stealing increases the accuracy of the final design over the original multi perspective perceptron policy and the state of the art policy.

B. Handling Writebacks

In the case of writebacks, we ignore updating the sampler and recency positions, but we do execute all procedures related to set stealing (steal sets if scheduled, reset scores, find dominating policy, etc.), except we do not log writeback misses. On bypasses, we ignore updating the sampler and cache, but we execute all procedures related to set stealing.

C. Pseudocode Description

We present pseudocode in Algorithm 2 for updating the samplers upon cache access. Let t be the partial tag of the current access, $get()$, $get(B)$ to refer to generating the current PCs features or getting a block B set of indices, $predict(x)$ refer to making a prediction on an array of indices x (or stored array indices of a block in the sampler), S, B to refer to the sampler set corresponding to the cache access and block that the partial tag hits respectively, $P[x][y]$ to refer to the perceptron table counters' x^{th} feature at index y . $update(x, y)$ updates the block in the sampler set with an index array x and

prediction y , and Y_{out} , A refer to the predicted y value and the associativity parameter.

VII. IMPLEMENTATION OVERHEAD

In this section we discuss the overhead of our proposed technique in terms of complexity and storage requirements.

A. Complexity

The complexity of our technique is very similar to the original multiperspective proposal [20]. On the critical path to making a prediction, a set of 10 tables are indexed with hashed feature values to read 10 counters that are summed and thresholded. This operation is essentially the same that is required for hashed perceptron-based branch predictors that have been implemented in recent processors [11]. Those branch predictors operate under very tight timing constraints and with more tables (16), yielding a prediction in at most 3 cycles. Tricks from computer arithmetic such as using an adder tree can bring the time complexity for computing the sum of n items down to $O(\log n)$. For our purposes, the predictions can be made simultaneously with last-level cache hits or misses which take at least 10 cycles. Thus, the timing of the critical computation can easily be achieved. Other updates such as updating the prediction tables, sampler metadata, and counters for set dueling are not on the critical path and can be accomplished quickly, simultaneously with normal maintenance of LLC replacement state. The Glider predictor operates similarly with a perceptron-like predictor [45]. For training, Glider relies on a sampler as in our work, as well as on the *OPTgen* algorithm [14]. *OPTgen* approximates Bélády's algorithm by using an *occupancy vector* to keep track of the replacement decisions Bélády's algorithm would have made. A longer vector results in a more accurate predictor. *OPTgen* takes time linear in the size of that window. Authors report that a quantized window size of 32 achieves good performance. This additional *OPTgen* component represents significant extra complexity which our simpler algorithm avoids, while resulting in superior performance.

B. Storage

We have made certain choices to minimize the storage overhead needed, and we report the resulting requirements. This section will assume a choice of 32 sample sets, three 20 and a 32 way sampler with 32 sets, and perceptron set policies with 10 tables of 256 6-bit counters, all in the context of a 2 MB LLC cache with 2048 sets and 16-way set associativity. For the set stealing metaheuristic, the only contributor to memory usage is the score counters, which will consume six 64-bit counters.

The total memory taken by the perceptron tables is dependent on the features selected. If only Boolean features are selected, our total cost will be significantly lower than if we include none. P_1 contains three such features, lowering the number of counters it needs to use. The PC-tables use 32 sets of 448 6-bit counters, using 10.5 KB. They are indexed by P_2 , so extra memory in the sampler is not used. Global history of PC and Tags will take 256 bits, to store 16 16-bit entries.

The samplers contain entries for a partial tag of 18 bits, 13-bit previous prediction, LRU bits, and feature indices. Depending on the specific configuration, the policies used

consume 92 to 114 bits per block, with 640 or 1024 blocks per sampler, depending on the associativity of the sampler. We use one alive bit, to be used by only P_4 . LRU will cost us 16 KB. Altogether, the given configuration in our results for the single core takes around 76.89 KB. Table VII summarizes these calculations.

C. Power and Latency

The original multiperspective reuse predictor has a certain overhead in terms of power and latency. Our proposal adds a small amount of random logic to implement set-stealing; the additional overhead is negligible. It is not obvious that a neural reuse predictor should not have a significant overhead in terms of power and latency. However, we note that neural branch predictors with much the same structure have been incorporated into processors for quite some time, and have been disclosed in detail for a recent processor [11]. Branch prediction requires much tighter timing constraints than cache management, and the neural branch predictor must be exercised on virtually every cycle, unlike a reuse predictor in the last-level cache which is used much less frequently, only when there are last-level cache misses that require using the replacement policy. Thus, our proposal incurs an overhead comparable in area and much more favorable in power to branch predictors already implemented in modern processors.

VIII. METHODOLOGY

We evaluate SSPT by using ChampSim [10], a microprocessor simulator that models out-of-order execution on a set of program traces. We model a configuration approximating Intel's Ice Lake microarchitecture. We simulate a 64 set, 8 way L1 instruction cache, a 64 set, 12 way L1 data cache, a 1024 set, 8 way L2 unified cache, and a 2,048 set L3 cache with 16 ways. We simulate a hashed perceptron branch predictor, and next line prefetchers on the L1 data cache and the L2 unified cache. Table II gives the microarchitectural parameters used for our simulations. Figure 5 depicts the performance gain over LRU of SSPT to Glider and MPRU over different cache sizes. While 2 MB cache sizes are modelled in Multiperspective Reuse Prediction [21] and Glider [46], we depict 1, 4 and 8 MB cache sizes as well.

We use a set of 81 benchmarks from SPEC CPU, and 90 benchmarks from the GAP benchmark suite [3] and XSBench benchmarks [51] for the single core benchmarks. SPEC CPU is a widely used benchmark suite for measuring processor performance. The GAP benchmarks focus on graph processing workloads. The XSBench benchmarks contain a computational kernel calculating macroscopic neutron cross sections accounting for 85% of the runtime from the OpenMC application from Lawrence Livermore National Labs. The traces from SPEC and GAP are chosen as the single highest-weighted SimPoint for each benchmark [44]. We warm the cache for 500 million instructions, and simulate execution for another 500 million instructions. This methodology of using about one billion instructions for warming and measuring is consistent with recently published work on cache management using ChampSim and these same traces [5], [17], [50], [53]. In our experience, earlier cache management work using only 100 million instructions for warming and measurement resulted in unclear conclusions, but using 500 million instructions for

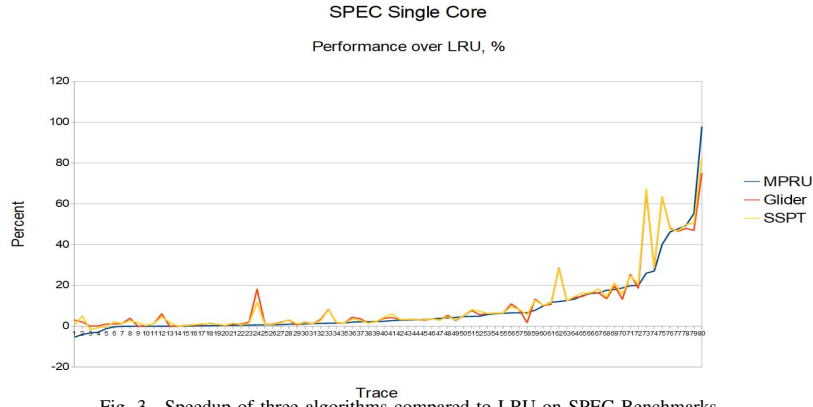


Fig. 3. Speedup of three algorithms compared to LRU on SPEC Benchmarks.

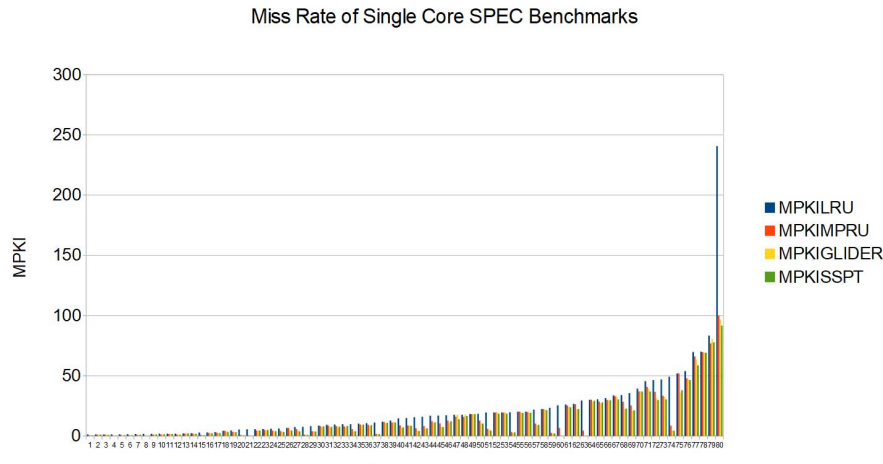


Fig. 4. Misses per 1000 instructions on SPEC benchmarks

Description	Overhead	Total
Set Stealing Score	64 bits * 6 counters	384 bits
Four Perceptron Tables	3 policies * 10 features * 6 bits * 256 counters + 1 policy * 3 features * 6 bits * 2 counters + 1 policies * 7 features * 6 bits * 256 counters	10788 bits
Sampler A	32 sets * 20 ways * (92 bits/block)	58,880 bits
Sampler B	32 sets * 20 ways * (113 bits/block)	72,320 bits
Sampler C	32 sets * 32 ways * (114 bits/block)	116,736 bits
Sampler D	32 sets * 20 ways * (113 bits/block)	72,320 bits
Per block Alive and LRU	5 bits * 16 ways * 2048 sets	163,840 bits
Global History of PC and Tag	16 bits * 16 depth	256 bits
Per-PC tables	32 tables * 6 bits * 7*64 counters	86016 bits
Total		76.89 KB

TABLE I
SUMMARY OF HARDWARE OVERHEAD

these same benchmarks provided the same consistent results as using several billion instructions.

For all experiments, we use four different sets of perceptron tables, or policies P_1, P_2, P_3, P_4 with different parameters. P_3 uses a 32-way sampler, P_3 ignores the per-PC tables, and P_4 uses an *isAlive* bit. For the single core benchmarks, we report a geometric mean speedup over LRU. This is depicted as the

ratio behind the policy being tested versus a baseline LRU policy. We also report MPKI for each benchmark.

We do not model multi-threaded or multi-core workloads due to time constraints and the large computational burden. For future work we plan to extend the ideas presented here to the multi-core context. We expect that a straightforward application of the ideas to the multi-core concept would show

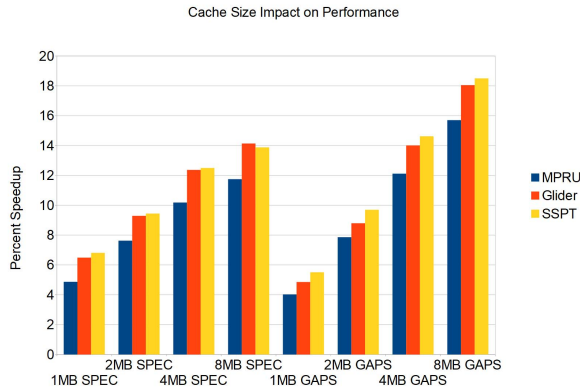


Fig. 5. Speedup of SSPT vs Glider and MPRU on different cache sizes.

similar benefits, as earlier work in similar cache management techniques has done [21].

L1 ICache	32KB, 8-way, 4-cycle, 8-entry MSHR
L1 DCache	32KB, 8-way, 4-cycle, 8-entry MSHR, next line prefetcher
L2 Cache	512KB, 8-way, 10-cycle, 16-entry MSHR, next line prefetcher
LLC	2MB per core, 16-way, 20-cycle, 64-entry MSHR
DRAM	4GB, DDR4, 4GHz, 1600 MT/s

TABLE II
CHAMPSIM SIMULATION PARAMETERS

IX. RESULTS

A. SPEC CPU

Figure 3 shows the performance on 81 SPEC CPU traces of Multiperspective Reuse Prediction, Glider, and SSPT as a ratio against the baseline LRU policy in the 2MB setting. We report a geometric mean speedup of 9.45% for SSPT, 9.28% with Glider, and 7.62% with Multiperspective Reuse Prediction. We note that SSPT performs better or equal to Glider on 46 out of the 81 benchmarks. Similarly, SSPT performs better than or equal to Multiperspective Reuse Prediction for 66 out of the 81 benchmarks.

Figure 4 shows the misses per 1000 instructions (MPKI) of Multiperspective Reuse Prediction, Glider, SSPT, and LRU, sorted by LRU. The arithmetic mean of MPKI values for LRU is 20.94 MPKI, Multiperspective is 15.13 MPKI, Glider at 13.92 MPKI, and SSPT at 13.78 MPKI.

We note that on 470.lbm, performance on SSPT trails Glider. 470.lbm contains many cache re-references far into the future, the most common being a peak around 80-90 accesses in the future before re-reference. Excluding writebacks, of all blocks re-referenced within 275 accesses, about .2% of them occur within 32 accesses. The samplers, relying on LRU, can struggle to adequately capture this behaviour since most sampled blocks will be evicted from the sampler before being re-referenced.

B. Big Data Workloads

Figure 6 depicts the performance of SSPT against Glider and Multiperspective Reuse Prediction over 90 GAPS and XS

benchmarks. We report a speedup of 9.70% over a baseline LRU policy, in comparison to Glider at 8.79%, and Multiperspective Reuse Prediction at 7.86%.

Figure 7 shows the misses per kilo instruction (MPKI) of Multiperspective Reuse Prediction, Glider, SSPT, and LRU, sorted by LRU. The arithmetic mean of MPKI values for LRU is 60.54 MPKI, Multiperspective is 48.99 MPKI, Glider at 46.48 MPKI, and SSPT at 44.77 MPKI. SSPT provides a speedup compared to Glider on 69 out of the 90 benchmarks tested. Similarly, SSPT performs better than or equal to Multiperspective Reuse Prediction for 67 out of the 90 benchmarks.

We note that on bfs.twitter, performance on SSPT leads Glider. Unlike 470.lbm, bfs.twitter contains a lot of short term cache re-references, peaking at around a distance of 8, tapering off on the right tail. Excluding writebacks, of all blocks re-referenced within 275 accesses, this time about 61.15% of them occur within 32 accesses.

X. CONCLUSION

We have introduced the idea of set stealing, modifying the perceptron tables found in [49] and [20]. SSPT allows for customizable cache management that can be modified to fit the designer's needs. When one policy is used that is not optimal, SSPT allows us to dynamically run on the better of the two, to ensure the best performance.

These results were observed in a highly configurable feature space that takes up similar hardware as previous work. Future research directions around SSPT could be focused on further analysis of the design space, or more analysis into further lowering the hardware budget.

XI. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants CNS-1938064 and CCF-1912617, generous gifts from Intel, and the Semiconductor Research Corporation project GRC 2936.001. Portions of this research were conducted with the advanced computing resources provided by Texas A&M High Performance Research Computing.

REFERENCES

- [1] J. Abella, A. González, X. Vera, and M. F. P. O'Boyle, "IATAC: a smart predictor to turn-off l2 cache lines," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 1, pp. 55–77, 2005.
- [2] J. Abella, A. González, X. Vera, and M. F. O'Boyle, "Iatac: a smart predictor to turn-off l2 cache lines," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 1, pp. 55–77, 2005.
- [3] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.
- [4] L. A. Bélády, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, June 1966.
- [5] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 1–13.
- [6] C.-H. Chi and H. Dietz, "Improving cache performance by selective cache bypass," in *System Sciences, 1989. Vol. 1: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, vol. 1. IEEE, 1989, pp. 277–285.
- [7] A. chow Lai, C. Fide, and B. Falsafi, "Dead-block prediction and dead-block correlating prefetchers," in *In Proceedings of the 28th International Symposium on Computer Architecture*, 2001, pp. 144–154.
- [8] H. Dybdahl and P. Stenström, "Enhancing last-level cache performance by block bypassing and early miss determination," in *Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer, 2006, pp. 52–66.

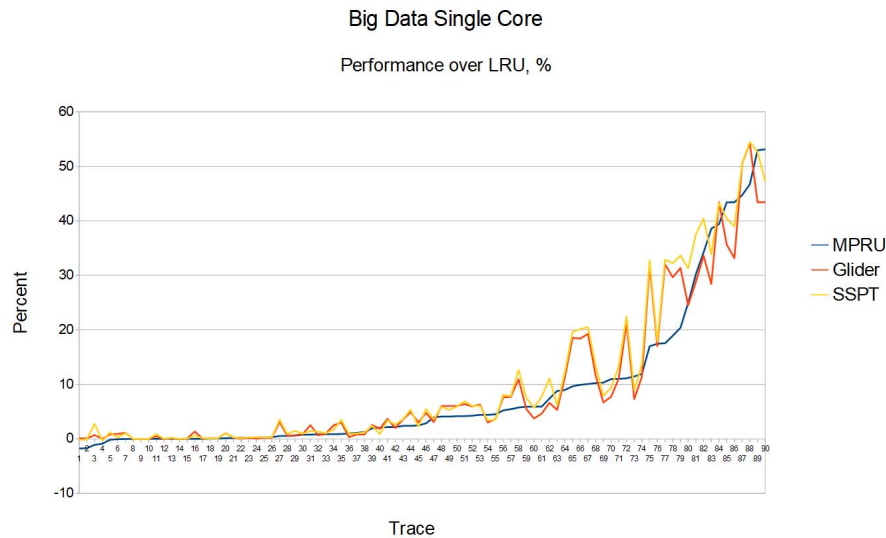


Fig. 6. Speedup of three algorithms compared to LRU on GAPS + XS Benchmarks.

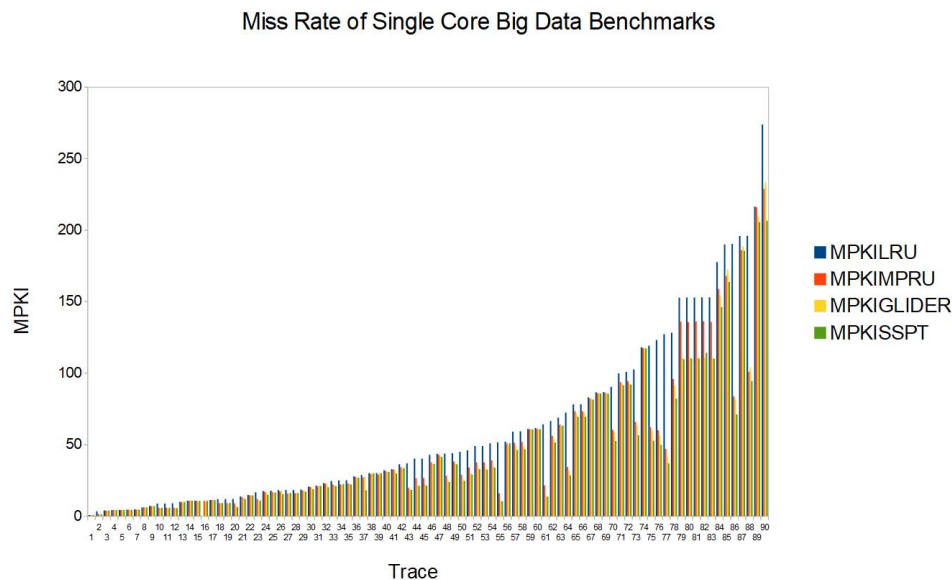


Fig. 7. Misses per 1000 instructions on GAPS + XS benchmarks

- [9] A. González, C. Aliagas, and M. Valero, "A data cache with multiple caching strategies tuned to different types of locality," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 2014, pp. 217–226.
- [10] P. V. Gratz, J. Kim, and G. Chacon. THE 2nd CACHE REPLACEMENT CHAMPIONSHIP – co-located with ISCA june 2017. [Online]. Available: <https://crc2.ece.tamu.edu/>
- [11] B. Grayson, J. Rupley, G. Z. Jr., E. Quinell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, , and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May/June 2020*, 2020.
- [12] 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.
- [13] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 110–123.
- [14] A. Jain and C. Lin, "Back to the future: Leveraging belady's algorithm for improved cache replacement," in *Proceedings of the 43rd ACM/IEEE International Symposium on Computer Architecture*, ser. ISCA '16, 2016, pp. 78–89.
- [15] A. Jaleel, K. Theobald, S. S. Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA-37)*, June 2010.
- [16] J. Jalminger and P. Stenstrom, "A novel approach to cache block reuse predictions," in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*. IEEE, 2003, pp. 294–302.
- [17] A. V. Jamet, L. Alvarez, M. Casas, and D. A. Jiménez, "Characterizing the impact of big data workloads on cache replacement policies," in *Proceedings of the 2020 IEEE International Symposium on Workload Characterization*.
- [18] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptors," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, January 2001, pp. 197–

- 206.
- [19] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 436–448. [Online]. Available: <http://doi.acm.org/10.1145/3123939.3123942>
- [20] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 436–448. [Online]. Available: <http://doi.acm.org/10.1145/3123939.3123942>
- [21] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 436–448, 2017.
- [22] T. L. Johnson, D. A. Connors, M. C. Merten, and W.-M. Hwu, "Run-time cache bypassing," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1338–1354, 1999.
- [23] T. L. Johnson and W.-M. W. Hwu, "Run-time adaptive cache hierarchy management via reference analysis," in *ACM SIGARCH Computer Architecture News*, vol. 25, no. 2. ACM, 1997, pp. 315–326.
- [24] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proceedings of the International Symposium on Computer Architecture*. Los Alamitos, CA, USA: IEEE Computer Society, 2001, p. 240.
- [25] S. Khan, A. R. Alameldien, C. Wilkerson, O. Mutlu, and D. A. Jiménez, "Improving cache performance using read-write partitioning," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 452–463.
- [26] S. Khan and D. A. Jiménez, "Insertion policy selection using decision tree analysis," in *2010 IEEE International Conference on Computer Design*, 2010, pp. 106–111.
- [27] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi, "Using dead blocks as a virtual victim cache," in *Proceedings of the 4th Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, January 2010.
- [28] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *MICRO*, December 2010, pp. 175–186.
- [29] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.
- [30] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the program counter: Reconstructing program behavior in the processor cache hierarchy," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 737–749. [Online]. Available: <http://doi.acm.org/10.1145/3037697.3037701>
- [31] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *International Symposium on Computer Architecture*, 2000, pp. 139 – 148.
- [32] 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.
- [33] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 48–59.
- [34] L. Li, I. Kadayif, Y.-F. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and A. Sivasubramanian, "Leakage energy management in cache hierarchies," in *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*. IEEE, 2002, pp. 131–140.
- [35] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 222–233.
- [36] 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*, June 1997, pp. 292–303.
- [37] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay, "The split temporal/spatial cache: initial performance analysis," in *Proc. of the SC1zzL-5, Santa Clara, CA, USA*, 1996, pp. 72–78.
- [38] S. Mirbagher-Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, 2018, pp. 519–532. [Online]. Available: <https://doi.org/10.1109/ISCA.2018.00050>
- [39] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jiménez, "Chirp: Control-flow history reuse prediction," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 131–145.
- [40] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 381–391. [Online]. Available: <https://doi.org/10.1145/1250662.1250709>
- [41] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ser. ISCA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 167–178. [Online]. Available: <http://dx.doi.org/10.1109/ISCA.2006.5>
- [42] J. A. Rivers and E. S. Davidson, "Reducing conflicts in direct-mapped caches with a temporality-based design," in *Parallel Processing, 1996. Vol. 3. Software., Proceedings of the 1996 International Conference on*, vol. 1. IEEE, 1996, pp. 154–163.
- [43] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens, "Utilizing reuse information in data cache management," in *Proceedings of the 12th international conference on Supercomputing*. ACM, 1998, pp. 449–456.
- [44] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
- [45] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 413–425. [Online]. Available: <https://doi.org/10.1145/3352460.3358319>
- [46] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 413–425. [Online]. Available: <http://doi.acm.org/10.1145/3352460.3358319>
- [47] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, "Memory coherence activity prediction in commercial workloads," in *WMP1 '04: Proceedings of the 3rd workshop on Memory performance issues*. New York, NY, USA: ACM, 2004, pp. 37–45.
- [48] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. S. Davidson, "Active management of data caches by exploiting reuse information," *IEEE Transactions on Computers*, vol. 48, no. 11, pp. 1244–1259, 1999.
- [49] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 2:1–2:12. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3195638.3195641>
- [50] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 2:1–2:12. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3195638.3195641>
- [51] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, 2014. [Online]. Available: <https://www.mcs.anl.gov/papers/P5064-0114.pdf>
- [52] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A modified approach to data cache management," in *Proceedings of the 28th annual international symposium on Microarchitecture*. IEEE Computer Society Press, 1995, pp. 93–103.
- [53] G. Vavoulitis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jiménez, and M. Casas, "Exploiting page table locality for agile tlb prefetching," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 85–98, 2021.
- [54] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, J. Simon C. Steely, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 430–441.
- [55] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang, "Compiler managed micro-cache bypassing for high performance epic processors," in *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*. IEEE, 2002, pp. 134–145.