2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer

Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A. Jiménez
Department of Computer Science & Engineering
Texas A&M University

***Abstract*—Modern processors support instruction fetch with the instruction cache (I-cache) and branch target buffer (BTB). Due to timing and area constraints, the I-cache and BTB must efficiently make use of their limited capacities. Blocks in the I-cache or entries in the BTB that have low potential for reuse should be replaced by more useful blocks/entries. This work explores predictive replacement policies based on reuse prediction that can be applied to both the I-cache and BTB.**

Using a large suite of recently released industrial traces, we show that predictive replacement policies can reduce misses in the I-cache and BTB. We introduce *Global History Reuse Prediction* (GHRP), a replacement technique that uses the history of past instruction addresses and their reuse behaviors to predict dead blocks in the I-cache and dead entries in the BTB.

This paper describes the effectiveness of GHRP as a dead block replacement and bypass optimization for both the I-cache and BTB. For a 64KB set-associative I-cache with a 64B block size, GHRP lowers the I-cache misses per 1000 instructions (MPKI) by an average of 18% over the least-recently-used (LRU) policy on a set of 662 industrial workloads, performing significantly better than Static Re-reference Interval Prediction (SRRIP) [1] and Sampling Dead Block Prediction (SDBP)[2]. For a 4K-entry BTB, GHRP lowers MPKI by an average of 30% over LRU, 23% over SRRIP, and 29% over SDBP.

I. Introduction

Modern processors rely on efficient instruction fetch to keep the pipeline fed with right-path instructions [3], [4]. To maintain that stream of instructions, the front-end relies on structures such as the instruction cache (I-cache) and branch target buffer (BTB). The I-cache stores blocks of recently used instructions, improving instruction throughput and latency. The BTB caches targets of previously-taken branches to minimize target re-computation latency [5]. Because of timing and area constraints, these structures require efficient organization and management to achieve high caching accuracies and speed.

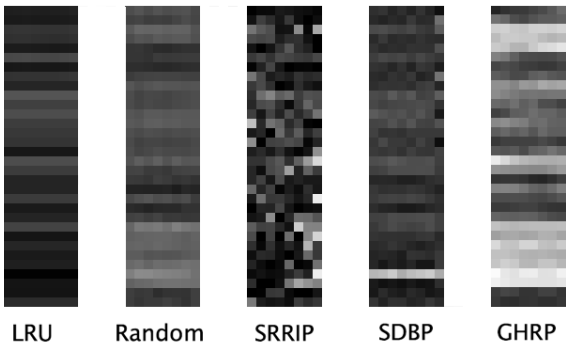

Fig. 1. Heat map illustrating cache efficiency [6] of a 16KB 8-way I-cache using five replacement policies for a given trace. Lighter pixels represent longer live times.

A block is said to be *live* in a cache if it will be used again before it is evicted. A block is *dead* if it will not be used before evicted. Much recent work in data caches relies on predicting and replacing dead blocks. This paper explores such predictive policies for the I-cache and BTB.

Figure 1 illustrates the *cache efficiency* [6] of a 16KB instruction cache using several replacement policies. Cache efficiency is the fraction of time a block is live in the cache. Each pixel represents a cache block in an 8-way set-associative, 16KB I-cache with each row corresponding to one set. Clearly, the replacement policy has a large impact on cache efficiency. Our proposed replacement policy, Global History Reuse Prediction, results in significant improvements in cache efficiency that translate into improved performance.

A. Instruction Cache and BTB Management

Most literature on I-cache management has focused on either prefetching techniques [7], [8], [9],

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00050

[10], [11], [12], or software-based techniques [13], [14], [15], [16], [17], [18], [19] to increase I-cache performance. However, very little work has been done on developing and evaluating replacement policies expressly for the I-cache. Smith and Goodman [20] evaluate replacement policies for the I-cache but, due to the earliness of the work, analyze nascent policies like first-in, first-out (FIFO), least-recently-used (LRU), and random replacement. We find very little work on high-performance BTB replacement policies, although we know through private communications that industry is very interested in good BTB replacement.

To improve replacement policies in the I-cache and BTB, we look to recent work in data cache replacement [2], [21], [1], [22], [23], [24]. A key idea in recent work is that sequences of recently accessed instructions correlate strongly with the likelihoods of block reuse. Training a predictor on control-flow traces based on sampled sets yields a high reuse prediction accuracy. We find this correlation holds within both I-cache and BTB, but due to the nature of instruction streams, the current literature's sampling approaches are unsuitable as we will see in Section II-A. In response to this analysis, we develop Global History Reuse Prediction (GHRP), a block replacement technique for the I-cache and BTB that can predict the reuse behaviors of the entries in these components.

B. Contributions

The contributions of this paper are:

1) An exploration of techniques for I-cache and BTB management using a set of over 600 industry-sourced workloads recently released through the Fifth Championship Branch Prediction Competition [25].
2) Evaluation of replacement policies adapted from recent work on data cache management to demonstrate their potential for improving I-cache and BTB hit rates. In particular, we explore whether using sampling-based policies benefit I-cache replacement. We compare these results to LRU as well as Static Re-reference Interval Prediction (SRRIP) [1].
3) Based on the analysis of the reference patterns of I-caches and BTBs, we describe why sampling-based policies like SDBP fail to predict dead blocks/entries. We thus propose *Global History Reuse Prediction*, a replacement policy for both the I-cache and BTB that exploits these reference pattern behaviors.

II. Background and Motivation

We have not found much recent work on the impact of replacement policy on BTB performance beyond the work of Perleberg *et al.* [26]. New mobile and server workloads motivate us to take a fresh look at this area. As we explore I-cache design, exploring the BTB alongside it is a natural extension of our work.

A. Sampling-Based Dead Block Prediction is Unsuitable

Our original intent was to apply PC-based dead block predictors such as SDBP and SHiP to instruction caches and BTBs. These predictors use set-sampling [22], [2], [21] to generalize the behavior of accesses to a small number of cache sets over the entire cache. For example, SDBP can yield a good speedup from sampling only 32 of the 2,048 sets of a 2MB last-level cache. We expected to see good performance improvements for the I-cache and BTB as well, but were surprised that the results were disappointing. In retrospect the reason seems obvious but we will explain it here. PC-based predictors exploit the observation that, if a block in the data cache is accessed by a given PC and becomes dead, other blocks accessed by the same PC in other sets are likely to become dead as well. Set-sampling allows the predictor to learn from only a small number of sets, allowing the metadata store to be very small. Unfortunately, instruction streams do not allow for set-sampling in this way since the PC itself forms the index into the I-cache or BTB. Thus, set-sampling cannot generalize behavior on a small number of sets over the entire structure, as a given PC only accesses one set. Figure 2 illustrates this difference. Hereafter, we evaluate SDBP for I-cache and BTB with a sampler the same size as the cache, to avoid this inability to generalize.

B. Dead Block Prediction

It has been observed that caches often retain *dead blocks*, *i.e.* blocks in the cache that will not be used again until they are evicted [27]. Dead blocks waste space and energy in the cache. Dead block prediction has been evaluated in the context of making replacement decisions in L1 Data cache [27], [28], Last level cache [2], [28], [29] and prefetching [27],

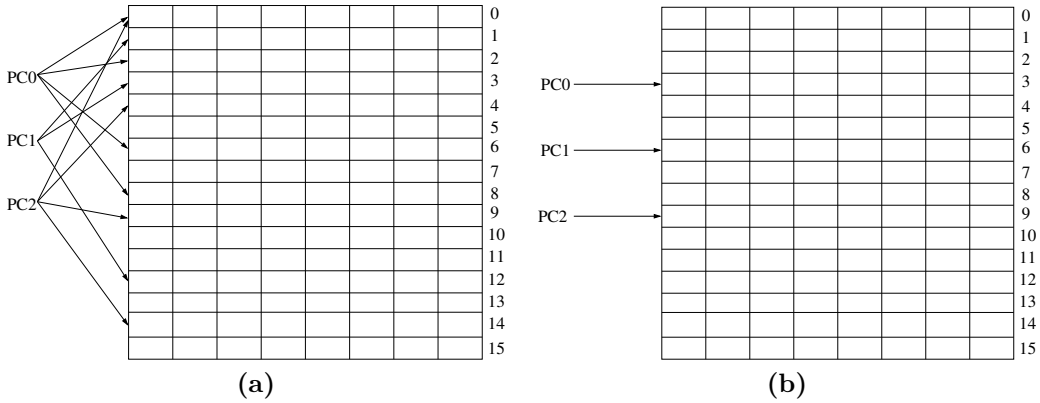

Fig. 2. Set-sampling for PC-based dead block predictors does not work for I-cache or BTB. For an 8-way, 16-set data cache (a), it is sufficient to sample the first two sets to generalize behavior over the entire cache. For an 8-way, 16-set I-cache or BTB (b), one PC accesses only one set, so the behavior of all sets must be measured. Hereafter SDBP is modeled sampling all cache sets.

[30], bypassing [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], power reduction [44], [45], and cache coherence protocol optimization [46], [47], [48]. Note that no cache replacement policy has been proposed for I-cache or BTB based on dead block prediction.

Lai *et al.* propose a trace-based dead block predictor [27] used to prefetch data into predicted dead blocks and thus improve prefetching and replacement decisions for the L1 data cache. In this technique, a trace of instruction addresses that make reference to a block is summarized in a block signature associated with that block. The signature is used to index a table of saturating counters. The corresponding counter is incremented when a block is evicted and decremented when a block is reused, thus keeping track of the tendency of an instruction trace to lead to a dead block.

Kharbutli *et al.* propose a counter-based dead block prediction approach [23] to drive cache replacement and bypassing. Each cache block is associated with a counter keeping track of the number of accesses to a block before it is evicted. The live time and access time of a block are tracked using Live time (LvP) and access time predictor (AIP). When the counter reaches a threshold, the block is predicted as dead. Blocks predicted dead on their first access are bypassed to the L1 cache.

Hu *et al.* propose timekeeping techniques [30] that can be used to predict dead blocks by learning number of cycles a block is accessed. A block is considered dead if it is not accessed twice the number of cycles. Virtual Victim Cache [29] uses a dead block predictor to reuse dead regions of the cache as a victim cache, effectively reducing conflict and capacity misses.

Liu *et al.* [28] propose dead block prediction based on cache bursts, *i.e.*, repetitive accesses to the most-recently-used (MRU) position. In this scheme, the prediction is made when a block becomes non-MRU.

Sampling-based Dead Block Prediction (SDBP) [2] uses only the address (PC) of the most recent instruction, allowing it to be useful in the last-level cache and eliminating the need to store signatures with blocks. In this method, a predictor learns the pattern of accesses and evictions from a small number of sets. There is also some work [49], [50] on software-based learning, in which hints from the compiler are learned to predict dead block information.

Our policy is the first to use dead block prediction in the I-cache and BTB. We compare it with the state of the art replacement policies LRU, SRRIP [1], SBDP [2].

C. Static Re-reference Prediction(SRRIP)

SRRIP [1] keeps track of the recency of blocks by predicting blocks that will be re-referenced again in the cache. Each block is associated with a two-bit re-reference prediction value. An initial prediction is made on block placement and revised when a block is reused or replaced.

D. Hashed Perceptron Branch Predictor

Alongside the I-cache and BTB, we need a highly accurate branch predictor. We have used a hashed perceptron branch predictor [51] that merges the concepts behind the gshare [52], path-based [53] and perceptron branch predictors [54]. The main idea is that the one-to-one correlation of weights to the number of history bits in a perceptron is not necessary. The hashed value of a history of instruction

addresses is used in combined with global branch outcome to index the weight tables. Perceptron-based branch predictors are used in high-performance ARM processors from Samsung [55] as well as processors from AMD and Oracle [56], [57].

E. I-Cache Optimizations

Previous work on I-cache management has focused on instruction prefetching [7], [8], [9], [10], [11], [12], code placement optimization [13], [14], [15], [16], [17], [18], [19], and basic block reordering [58]. Prefetching schemes [7], [8], [9], [10], [11], [12] focus on keeping a record of committed instruction streams, which better predict future control transfers and accesses to instruction blocks. Work on cache block alignment [58] and other software solutions [13], [14], [15], [16], [17], [18], [19] help minimize conflict misses and improve locality within the I-cache.

Several proposed solutions incur a large hardware overhead. For example, Shift [10] has 240KB storage overhead (750% of I-cache capacity) just for the index table, compared to 5KB of GHRP overhead for a larger I-cache. Similarly, some previous work explores unrealistically low associativities resulting in higher MPKI, while we model a realistic I-cache from the Samsung Mongoose processor. Little work has focused on evaluating and creating replacement policies for the I-cache [20].

F. BTB Optimizations

Previous literature [59], [26], [60], [61], [62], [63] on the BTB has focused on its design structure, including the branch target selection algorithm, the utilization of multi-level BTB organizations, and using different replacement policies. To reduce the bit storage overhead in the BTB, Fagin *et al.* propose an encoding scheme requiring the storing of partial tags [64]. Kobayashi *et al.* [65] further the work by also encoding target addresses' higher-order and lower-order bits separate tables. Other schemes for the BTB [66], [67] attempt to better capture large workloads by augmenting the effective BTB size via secondary structures.

G. Joint I-Cache/BTB Management

Some previous work [11], [68] addresses the BTB and I-cache design problem together. Kaynak *et al.* [11] take advantage of in-common metadata to simultaneously prefetch blocks and branch instructions into the I-cache and BTB, respectively. Similarly, Rakesh *et al.* [68] propose a branch predictor-directed prefetching strategy that also populates the I-cache and BTB concurrently, however, without the need for metadata. In this paper, we have focused on maintaining efficient utilization of both the I-cache and BTB by targeting the removal of dead blocks/BTB entries.

Algorithm 1 GHRP

```
1:  int cntrsNew[numPredTables]
2:  int predTables[numCounts][numPredTables]
3:  int indices[numPredTables]
4:  procedure Access(int PC)
5:      sign ← signature(PC, history)
6:      indices ← ComputeIndices(sign)
7:      cntrsNew ← GetCounters(predTables, indices)
8:      set ← calcSet(PC, cache)
9:      tag ← calcTag(PC, cache)
10:     isMissed ← isTagMatch(PC, cache)
11:     if isMissed = true then                          ▷ miss
12:         bypass ← majorityVote(cntrsNew, bypassThresh)
13:         if bypass = false then
14:             block ← victimBlock(set)
15:             indices ← ComputeIndices(block.signature)
16:             isDead ← true
17:             updatePredTables(indices, isDead)
18:             pred ← MajorityVote(cntrsNew, deadThresh)
19:             block.dead ← pred
20:             block.tag ← tag
21:     else                                             ▷ hit
22:         block ← matchedBlock(set, tag)
23:         indices ← ComputeIndices(block.signature)
24:         isDead ← false
25:         updatePredTables(indices, isDead)
26:         pred ← MajorityVote(cntrsNew, deadThresh)
27:         block.dead ← pred
28:     block.signature ← sign
29:     history ← UpdatePathHist(PC)
30:     updateLRUstackPosition()
```

III. Global History Reuse Predictor

It is often the case that a majority of the blocks in the last-level cache (LLC) are dead [2]. LLC dead block predictors are adept at determining whether a block is dead after its first access. However, their accuracy declines for subsequent accesses. Because common access patterns to the BTB and I-cache tend to consist of multiple reuses before eviction, current PC-based dead block prediction schemes such as SDBP and SHiP are ill-fit for these structures as explained in section II-A. An algorithm that takes into account control-flow history should have

Component	Number of Bits
Prediction tables	$3 \times 4,096$ entries $\times$ 2 bit counters = 3KB
Prediction bits	1 bit $\times$ 1,024 blocks = 128B
Signature	16 bits $\times$ 1,024 blocks = 2KB
History register	16 bit $\times$ 1 = 2B
Total	**5.13KB**

TABLE I
STORAGE OVERHEAD FOR GHRP FOR A 64KB, 8-WAY I-CACHE WITH 64B BLOCKS

higher accuracy [27], [69]. While PC-based policies can only efficiently predict dead blocks at insertion time, Global History Reuse Predictor (GHRP) is able to predict dead blocks more generally, either at insertion or at the last reuse. The main steps taken by GHRP are shown in Algorithm 1. We first propose GHRP algorithm for I-cache replacement and present results. Then we describe an enhancement for BTB replacement that uses minimal extra overhead.

A. Signature formula in GHRP

Like previous dead block predictors, GHRP indexes a table of counters with a signature generated from features correlated with reuse behavior. The GHRP signature uses the global path history of instruction addresses. Algorithm 2 line 1 shows the steps of updating the global path history. To update the global history, on every access we shift the three lowest-order bits of the PC into the history followed by one zero bit. The history register is 16 bits, allowing four previous accesses to be recorded. The signature is constructed by exclusive-ORing the history with the PC of the access to be predicted (Algorithm 2 line 4) . The zero bits in the history allow some of the PC bits to pass into the signature unmodified, yielding a useful hash of the history and PC. To compute indices into the prediction tables, we compute three different 12-bit hashes of the 16-bit signature with the function COMPUTEINDICES in Algorithm 2 line 7.

B. GHRP Prediction State

GHRP stores metadata for each I-cache block, consisting of 3 LRU stack position bits, a valid bit, a 16-bit signature and a prediction bit. Let us consider a real-world example I-cache: the Samsung Exynos M1 processor has a 64KB I-cache with 128B blocks [55]. The extra metadata and prediction tables of GHRP would impose an additional 5.13 KB of metadata, or 8% of the capacity of the I-cache. All of GHRP's operations are off the critical path to hitting in the I-cache. The small additional area of GHRP pays for itself in terms of reducing misses without affecting hit latency.

Algorithm 2 Updating path history and computing signatures

```
1:  procedure UpdatePathHist(int PC, int history)
2:      history ← history << 4
3:      history ← (history | PC_{0..2})  mod 2^16

4:  function Signature(int PC, int history)
5:      int signature ← history ⊕ PC
6:      return signature  mod 2^16

7:  function ComputeIndices(int signature)
8:      for i = 1 to numPredTables do
9:          indices[i] ← Hash(signature, i)
10:     return indices
```

Algorithm 3 Returns true if corresponding counters are more than specified threshold in majority of tables.

```
procedure MajorityVote(int[] counters, int threshold)
    int vote ← 0
    for  i = 1 to numPredTables do
        if counters[i] > threshhold then
            vote ← vote + 1
    if vote >= (numPredTables/2) then return true
    else return false
```

C. GHRP Predictor Table Indexing

GHRP uses three tables to provide a prediction via majority vote. This technique is shown in Algorithm 3. Each table is indexed by a distinct hash function similarly to the three tables in SDBP [2]. The indexing step is shown in Algorithm 2 line 7

and Algorithm 4. Each of the three corresponding counters is thresholded. If two or more counters exceed the threshold, the aggregate prediction is that the entry is dead. SDBP also utilizes three tables but aggregates the results via summation rather than majority vote. We find majority vote [70] superior to summation due to the nature of instruction cache accesses versus LLC accesses: with a high threshold and summation, SDBP is more conservative when it decides to predict a block as dead. Aliasing in the prediction tables leads to a lack of confidence and less coverage, rather than costly LLC misses. Instruction accesses are less likely to be dead, requiring lower thresholds for reasonable coverage. Majority vote avoids the effects of aliasing without needing a high threshold, so coverage and accuracy can both improve.

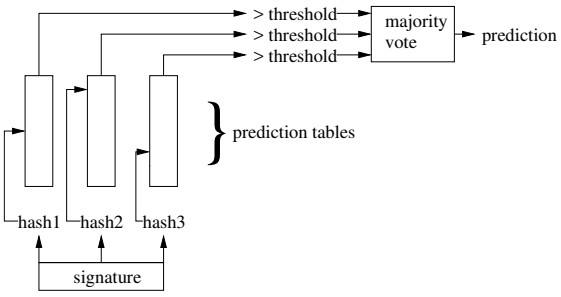

Fig. 4. Prediction datapath. Three hash functions of the signature index 3 tables to read 2-bit counters that are thresholded to make predictions. Aggregate prediction is by majority vote.

Algorithm 4 Index prediction tables and get counters

```
procedure GetCounters(int[][] predTables, int[] indices)
    for t = 1 to numPredTables do
        counters[t] ← predTables[indices[t], [t]]
    return counters[]
```

D. Accessing Predictor Table and Metadata in GHRP

On each access to the I-cache, the global path history is used to make the signature (Algorithm 1 lines 5)(Algorithm 2 line 4). Next, the prediction tables are indexed by hashing the signature. The corresponding counters are read out from the arrays (Algorithm 1 lines 6, 7, Algorithm 4 and Algorithm 2 line 7). Figure 4 illustrates the datapath for making a prediction.

The prediction table needs to be accessed on each access to I-cache because for each reuse the future prediction for one block may vary as the global history changes. Even if there were a hit during one access the next accurate prediction for that block may be *dead*. Thus, this prediction will be obtained from the prediction table indexed by the signature made with the current history and will be used to update the prediction bit in I-cache. The fact that the block received a hit during an access reveals an information about its future reuses and so the prediction table entry indexed with the old signature will be updated. On a miss, the prediction bit of the evicted block is updated with a prediction obtained by indexing the prediction table with the current history. Reading from prediction table is also necessary on each access to the I-cache to allow bypass.

The counters are used to decide if the incoming block should be bypassed or placed. In case of a miss, the tables vote and decide to bypass the block if the majority of the corresponding counters are above the bypass threshold. If the vote is to bypass, there are no more accesses to the prediction tables, and no metadata is updated. On the other hand, if it was a miss that is not bypassed (Algorithm 1 line 13) then a victim block is chosen to be replaced with the new one (Algorithm 1 line 14). GHRP first tries to find a predicted dead block by reading the prediction bit of each block: *block.dead*. If no block was predicted as dead then GHRP evicts the LRU block. The details of victim block selection is shown in Algorithm 5.

Algorithm 5 Returns a predicted dead block if there is one, otherwise returns the LRU block

```
function victimBlock(Set set)
    for int i = 1 to associativity do
        block ← set.blocks[i]
        if block.isDead = true then return block
    return LRUblock()
```

As soon as one block is chosen to be evicted, the prediction tables need to be updated by the new information about this eviction. First, the signature bits of the victim block are used to index the prediction tables and each corresponding counter is

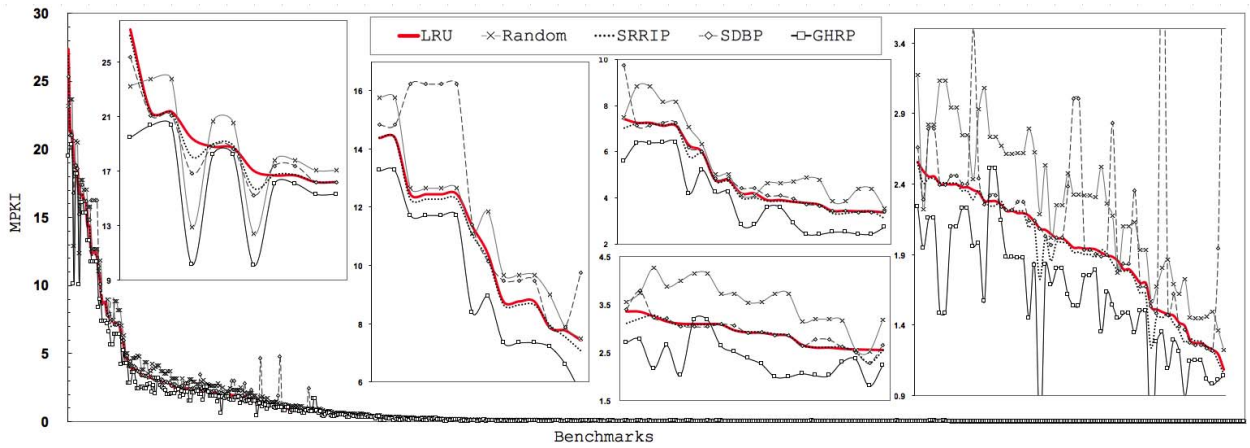

Fig. 3. MPKI comparison of various policies for a 8-way 64KB I-cache with 64B cache lines. The horizontal axis shows the benchmarks in the order of sorted MPKI for LRU. Multiple areas of the graph are shown in zoomed windows for different ranges of MPKI.

increased by one since the block was just shown to be dead (Algorithm 6).

Algorithm 6 Index prediction tables and update counters

```
procedure UPDATEPREDTABLES(int[] indices, bool isDead, )
    for t = 1 to numPredTables do
        if isDead = true then
            predTables[ indices[t], [t]]++
        else
            predTables[ indices[t], [t]]−−
```

After replacing the new block, the corresponding metadata for that block is updated (Algorithm 1 lines 20, 28 and 19). The dead block prediction for the signature made based on the current PC is obtained (Algorithm 1 line 18) and the prediction bit is updated (Algorithm 1 line 19). Algorithm 1 lines 15 to 18 summarizes the steps taken for eviction. Note that GHRP does not update the predictor table for the new signature related to current access. Rather, it updates the signature bits of the corresponding block which will be used in the future accesses.

In the case of a hit access (Algorithm 1 line 21) first, the prediction tables will be indexed based on the old signature in the block (Algorithm 1 line 23) and the corresponding counters will be decreased by one to make sure this block will be predicted as live under the same conditions in the future (Algorithm 1 lines 25 and Algorithm 6). Then the old signature is replaced with the new one and the prediction bit is replaced with the prediction obtained from the prediction table.

E. Adapting GHRP for BTB Replacement

The BTB is another cache-like structure that relies on a replacement policy to do a good job of providing branch targets. We use the GHRP algorithm to enable improved replacement of BTB entries as well as I-cache block replacement. Every BTB access comes from a branch that occupies some I-cache block. When a BTB access is made, the metadata corresponding to that branch's block in the I-cache is used to make a GHRP prediction. That is, the signature recorded for that I-cache block is used to index the I-cache GHRP prediction tables to generate three predictions that are thresholded and majority-voted to yield a dead-entry prediction for that BTB entry. Each BTB entry maintains one additional bit of metadata: a prediction bit that records whether that BTB entry is predicted as dead. The history register is also shared and is only updated with branch PCs using the same formula. All of the other structures for the GHRP algorithm are already present for use by the I-cache dead block prediction, so BTB replacement comes with almost no additional overhead. As in the I-cache, BTB replacement and bypass are guided by GHRP. A predicted dead block will be evicted, or if there is no such block, the LRU block will be evicted.

We first modeled GHRP as a stand-alone replacement policy with its own metadata, but realized that the size of the predictor would be so large that it would make more sense to simply increase the BTB size. However, we noticed that the prediction tables and metadata from the I-cache did just as well for the BTB. This result is somewhat counterintuitive; as cache blocks may have multiple branches, we would

make the same prediction for each BTB entry in a cache block. However, the algorithm still provides good BTB replacement for the following reasons:

1) Most branches are highly biased to be taken or not taken. A branch that is never taken will not get a BTB entry, so it will never need to replace another entry or be replaced. A branch that is seldom taken will have its BTB entry quickly reach the LRU position, and when it is occasionally taken the BTB miss will not have a large impact on MPKI.
2) On the other hand, a branch that is always or often taken will cut off fetching to other branches in the same cache block, so those branches will seldom or never need BTB entries.
3) Due to modulo indexing of the BTB, branches in the same cache block will map to distinct BTB sets. Figure 5 illustrates a BTB heat map for the various policies similar to the one shown in Figure 1. One noticeable feature of this image is that the different sets experience different levels of access, *i.e.* there are hot and cold sets. If a cache block is mostly live, the corresponding BTB entries will be predicted as live. This will protect vulnerable BTB entries in hot sets and not matter for BTB entries in cold sets.
4) It is possible that a dead BTB entry will be falsely predicted as live, but this sort of misprediction only reduces the opportunity to evict that block, rather than the other kind of misprediction that falsely predicts a live block as dead and can lead to a miss. By tuning the threshold for BTB predictions separately from I-cache predictions, we find a balance that minimizes false dead predictions while allowing the best coverage for the replacement optimization.

F. Impact of Misspeculation on Training

To prevent wrong-path information from polluting the prediction tables, GHRP only updates the tables of counters at commit time with right-path branches, a practice consistent with branch predictor implementation [71], [72]. GHRP uses path history to make predictions. The history used to predict the current I-cache or BTB access must be current, so GHRP uses the stream of fetch addresses from the branch predictor to update its history speculatively. For recovery from mispredictions, GHRP borrows a technique from branch prediction speculative history management [72]. GHRP maintains two path histories: the speculative history updated using the outcome of the branch predictor, and a non-speculative history updated when a branch is retired from the reorder buffer (ROB). On a misprediction, the speculatively history is restored from the non-speculative one. To support speculation on a modern processor, the I-cache and BTB may be updated according to wrong-path cache accesses and targets. Thus, GHRP updates its prediction tables based on the speculative history so that its predictions reflect the liveness of actual accesses to the BTB and I-cache.

G. Integrating GHRP into a Modern Decoupled Front-End

Modern processors use decoupled front-ends to fill an instruction queue. To ensure efficient instruction delivery, a new component must not introduce extra latency into the critical path for dequeuing instructions. Fortunately, none of GHRP's operations are on the critical path. Predictions are used to drive BTB and I-cache replacement, and can occur simultaneously with target computation and reading the L2, respectively. The speculative path history is updated simultaneously with branch predictor histories which, in a modern processor, are far deeper than the simple history used by GHRP. Using 8T SRAM cells for the prediction tables would allow updates to be made simultaneously with prediction. More area-friendly 6T cells would require some queuing of updates, but since the counters tend to saturate quickly most accesses would not require training.

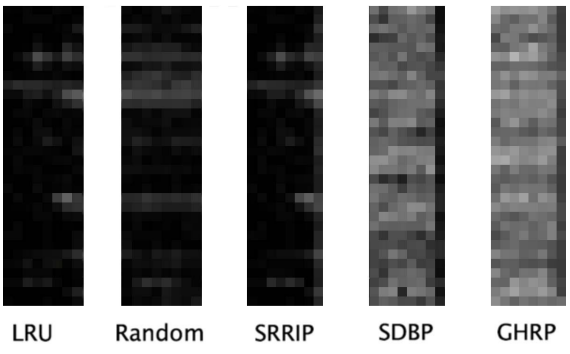

Fig. 5. Heat map showing the efficiency of a 256-entry, 8-way BTB using five replacement policies for a given industrial trace. The darker the pixel, the block remains unevicted while dead. GHRP improves live time over the other policies.

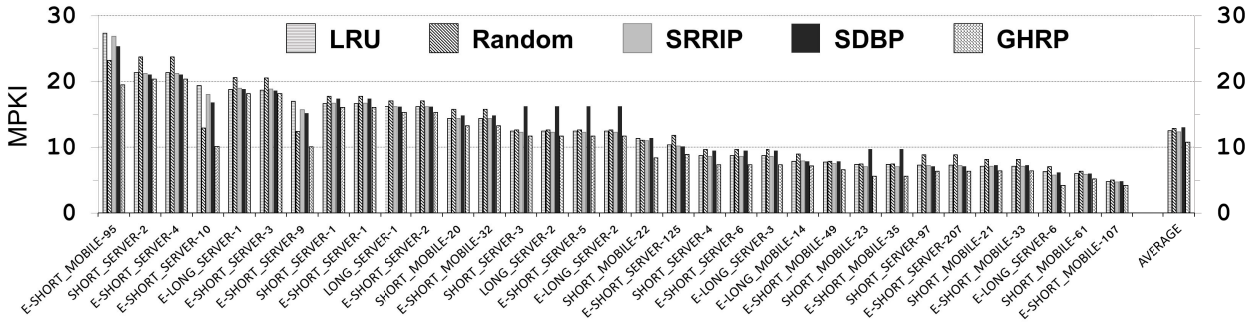

Fig. 6. MPKI for a 64KB 8-way I-cache with 64B cache lines for various policies over benchmarks. The last group of bars shows the average among the benchmarks illustrated.

IV. Methodology

A. Simulation Infrastructure

We use the simulator and traces released for the recent Championship Branch Prediction competition(CBP5) [25]. As it is intended for branch prediction studies, we augment it with additional code to study the I-cache and BTB. We use a hashed perceptron predictor as the branch direction predictor [51]. Perceptron-based branch predictors are used in high-performance ARM processors from Samsung [55] as well as processors from AMD and Oracle [56], [57].

The original traces contain one record for every branch, including conditional, unconditional, indirect, and returns. From these traces we reconstruct the block address of every instruction fetch group by inferring the missing instructions between branch targets. The simulator is not cycle accurate, so we use misses per 1000 instructions (MPKI) as our figure of merit. For a given benchmark, MPKI is roughly proportional to cycles per instruction (CPI). Arithmetic mean MPKI gives a good overall indication of the relative benefits of I-cache and BTB replacement policies.

We measure performance of the I-cache and BTB using LRU replacement as the baseline. We study static re-reference interval prediction (SRRIP), sampling-based dead block prediction (SDBP) and global history reuse prediction (GHRP) for different configuration parameters such as associativity, block size and cache size. We characterize the workloads as described in the following section.

For this study, we use a modified version of SDBP. The following modifications are applied to achieve the best possible results for I-cache and BTB based on SDBP design: 1) The sampler is as large as the cache, *i.e.* it has the same number of sets and same associativity. 2) Tuned dead block threshold to decrease number of false positives. 3) Tuned bypass threshold to avoid costly incorrect dead predictions. The original SDBP work used 2-bit counters, but in the context of I-cache and BTB we find the best performance comes with 8-bit counters. This is mainly to cover high bypass value which was required to get better accuracy. Our SDBP uses three skewed prediction table to reduce the possibility of miss prediction due to conflicts. Each entry in the sampler consist of 1 valid bit, 1 prediction bit, 3 bits to maintain LRU positions, 12 bits as partial PC (signature), and 16 bits of tag. Taking these steps adapts SDBP to work well with instruction streams as explained in Section II-A.

GHRP also keeps metadata for each I-cache block and BTB entry. GHRP also uses three skewed prediction tables. Each of the 4,096 entries in the tables contains a two-bit counter. The additional metadata for each block consists of 1 prediction bit, 3 bits to maintain LRU positions, and 16 bits of signature. Table I summarizes the storage requirements for GHRP for a 64KB I-cache with 8-way associativity. The modified SDBP requires considerably more storage due to the wider prediction tables.

B. Workloads

We use the set of 662 traces provided as a part of CBP5. The distribution of benchmarks is a mixed set of SHORT-MOBILE, LONG-MOBILE, SHORT-SERVER, and LONG-SERVER workloads. Short traces are simulated completely, while long traces are allowed to run for the first one billion instructions.

C. Warm-up

In each simulation, we warm the cache using the first half of the instructions in the trace, or up to two hundred million instructions, whichever comes first. We stop the simulation after one billion instructions.

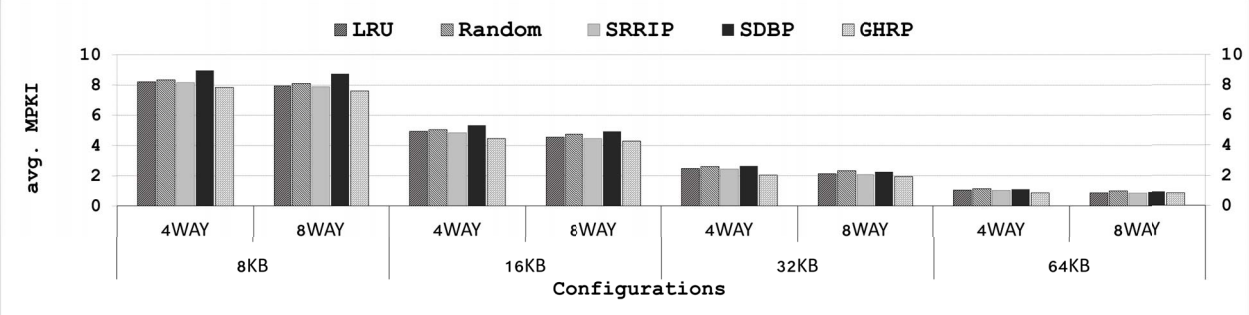

Fig. 7. Average MPKI for multiple I-cache configuration with various policies.

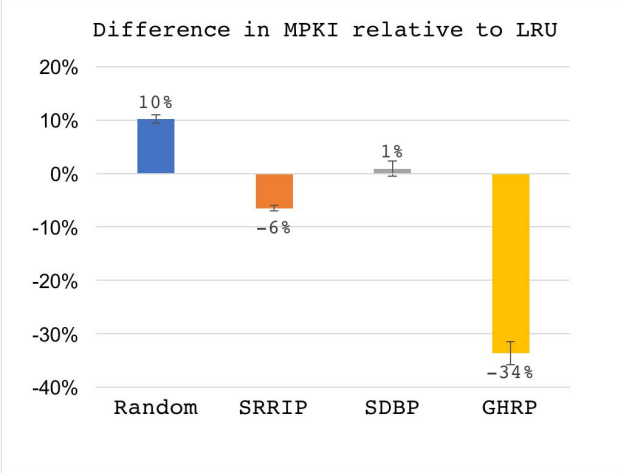

Fig. 8. Average *difference-relative-to-LRU* with error bars (95% confidence interval).

V. Results

In this section we describe the results of experiments simulating the GHRP policy as well as policies adapted from previous work. We first present results on I-cache replacement, then give results for BTB replacement.

A. I-Cache Results

Figure 6 depicts per-benchmark results for the various replacement policies for a 8 way 64KB I-cache with 64B blocks. The x-axis shows the benchmarks in order of sorted MPKI for LRU and is compared with other policies. Since it is not possible to depict the bar charts for all 662 benchmarks, an S-curve is shown in Figure 3 for the 64KB I-cache on all 662 benchmarks. On average for the 662 workloads, GHRP achieves 0.86 average MPKI, compared with 1.05 for LRU, 1.14 for Random, 1.02 for SRRIP, and 1.10 for SDBP. GHRP improves average MPKI by 18% over LRU, 24% over Random, 16% over SRRIP and 22% over SDBP. For a subset of 123 benchmarks experiencing at least 1 MPKI under the LRU policy on average, GHRP achieves 4.32 MPKI, compared with 5.11 for LRU, 5.53 for Random, 4.50 for SRRIP, and 5.38 for SDBP. GHRP improves average MPKI by 26% over LRU, 32% over Random, 15% over SRRIP and 20% over SDBP. GHRP is the only policy that significantly improves MPKI over LRU.For the vast majority of cases, GHRP provides lower MPKI than the other policies. In only 14 out of the 662 traces, GHRP fails to improve over LRU. This number is 106 for SDBP, 110 for SRRIP and 541 for Random.

We focus on the 64KB I-cache as it represents a common configuration in existing processors. However, we have also explored other configurations to verify our approach. I-cache MPKI values averaged over all 662 benchmarks mentioned above for multiple configurations are shown in Figure 7. The figure shows various combinations of 8KB, 16KB, 32KB and 64KB caches with 64B blocks and 4-way or 8-way associativity. For each configuration, the trend is the same: Random performs poorly, SRRIP performs better than LRU and SDBP perform better in some benchmarks and worse than LRU in some others, and GHRP provides significantly lower MPKI than the other policies.

1) Statistical Analysis: Figure 8 is the average of difference relative to LRU with error bars (95% confidence interval). The relative difference for GHRP is negative more than 95% of the times meaning that the MPKI was lower compared to LRU, and the average of this relative difference is -33% meaning that on average there is a 33% reduction in MPKI using GHRP compared to LRU. In more than 95% of tests the relative difference is at least 31% reduction (-31%).

Figure 9 shows that the number of traces in which Random performed worse than LRU (541) is 5 times more than this measurement in SDBP (106). This

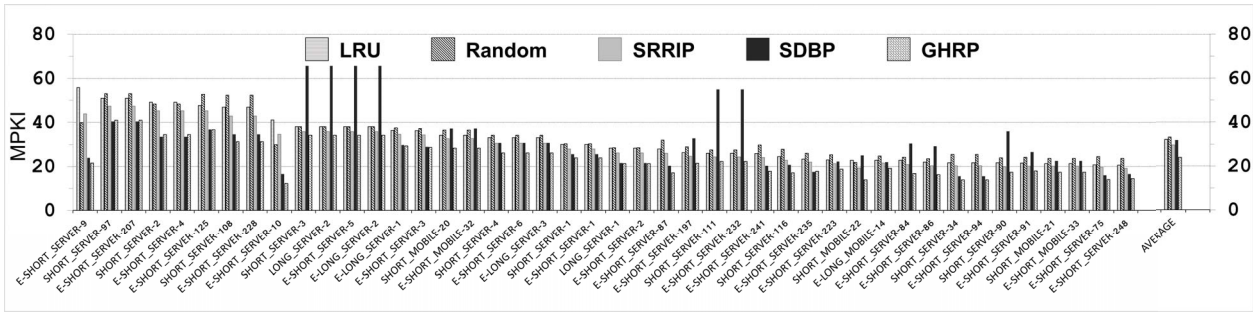

Fig. 10. BTB MPKI for a 4-way 4K-entry BTB with various policies.

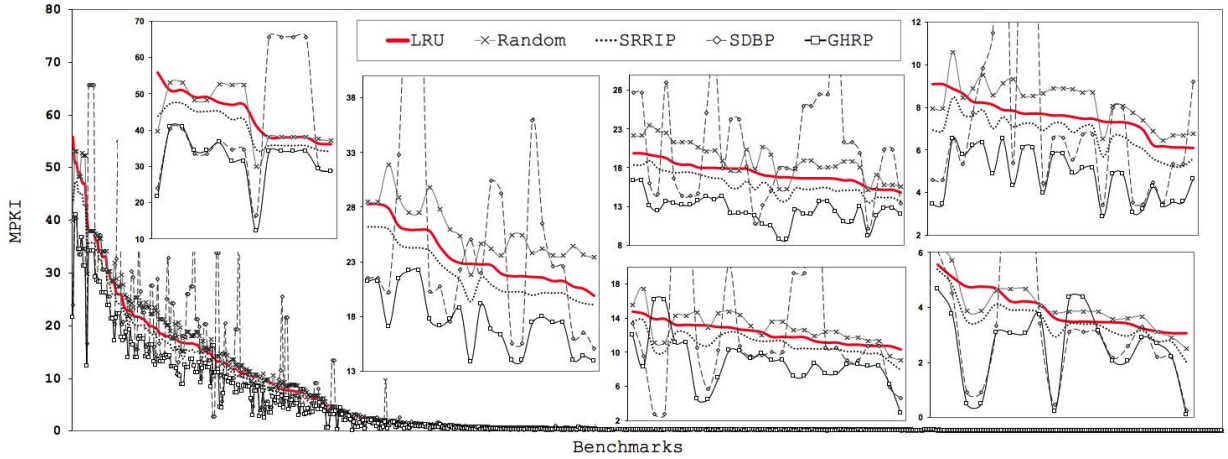

Fig. 11. MPKI comparison of various policies for a 8-way 4K BTB. The horizontal axis is showing the benchmarks in the order of sorted MPKI for LRU. Multiple areas of the graph are shown in zoomed windows for different ranges of MPKI.

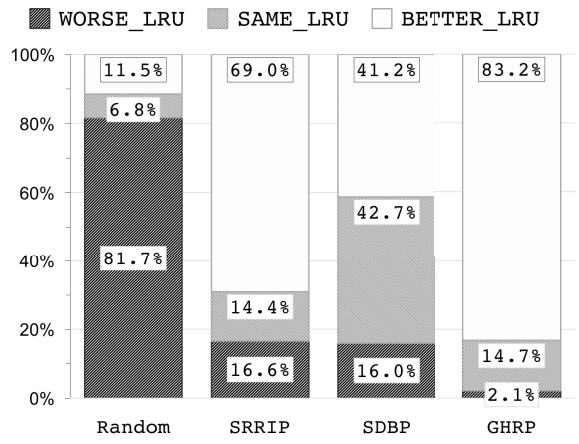

Fig. 9. For a 64K 8-way 64B-block size I-cache, this graph shows the percentage of traces for which the given policy performs worse than (black) or similar to (gray) or better than (white) LRU.

difference shows that SDBP can still a benefit large number of workloads (41% of traces) while harming 16% and performing similarly to LRU for the others (42% of traces). On the other hand, Figure 9 shows that the number of traces in which SRRIP performs worse than LRU (110) is close to this measurement for SDBP (106).

The figure also shows that GHRP is sufficient for 98% of traces (benefits 83% of traces and being similar to LRU for 14% of traces) while only harming 2% of traces. This observation did not indicate any dependency on trace category (SHORT-MOBILE, LONG-MOBILE, SHORT-SERVER and LONG-SERVER).

B. Branch Target Buffer

BTB misses for the various policies are shown in Figure 10. Since it was not possible to depict the bar charts for all 662 benchmarks so Figure 11 shows an S-curve for the 64KB I-cache on all 662 benchmarks. This BTB has 4,096 entries and is modeled after the Mongoose [55] BTB.

We find that more traces experience high MPKIs with smaller BTBs, but to keep our research rele-

vant to future processors we choose a modern BTB capacity. Over these traces, the LRU policy yields an average 4.58 MPKI. Random is worse at 4.81 MPKI, SRRIP and SDBP are slightly better at 4.17 and 4.57 MPKI, respectively. GHRP has the lowest average MPKI at 3.21, a 30.0% improvement over LRU, 33.3% over Random, 23.1% over SRRIP and 29.1% over SDBP.

VI. Conclusions and Future Work

This paper has demonstrated that predictive replacement policies can significantly reduce I-cache and BTB misses over a large suite of industrial traces. Previous replacement policies adapted from data cache replacement policies have potential, but a policy specifically designed for instruction streams has greater benefits. We introduce Global History Reuse Prediction, a dead block/entry predictor designed for instruction streams. In future work we will explore how our techniques interact with high-performance indirect branch prediction.

Acknowledgments

This research was funded in part by NSF grants CCF-1649242 and CCF-1216604/1332598. Thanks to the anonymous reviewers for their many helpful suggestions, and heartfelt thanks in particular to Mike Ferdman for his help preparing the final version of this paper.

References

[1] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 60–71, ACM, 2010.

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

[3] A. Ramirez, O. J. Santana, J. L. Larriba-Pey, and M. Valero, "Fetching instruction streams," in *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pp. 371–382, IEEE Computer Society Press, 2002.

[4] T.-Y. Yeh and Y. N. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," in *ACM SIGMICRO Newsletter*, vol. 23, pp. 129–139, IEEE Computer Society Press, 1992.

[5] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, pp. 6–22, Jan 1984.

[6] D. Burger, J. R. Goodman, and A. Kagi, "The declining effectiveness of dynamic caching for general-purpose microprocessors," *Technical Report 1261*, 1995.

[7] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pp. 1–10, IEEE, 2008.

[8] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 152–162, ACM, 2011.

[9] G. Chadha, S. Mahlke, and S. Narayanasamy, "Efetch: optimizing instruction fetch for event-driven webapplications," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 75–86, ACM, 2014.

[10] C. Kaynak, B. Grot, and B. Falsafi, "Shift: Shared history instruction fetch for lean-core server processors," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 272–283, ACM, 2013.

[11] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: unified instruction supply for scale-out servers," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 166–177, ACM, 2015.

[12] A. Kolli, A. Saidi, and T. F. Wenisch, "Rdip: return-address-stack directed instruction prefetching," in *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*, pp. 260–271, IEEE, 2013.

[13] S. McFarling, "Program optimization for instruction caches," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183–191, ACM, 1989.

[14] N. Gloy and M. D. Smith, "Procedure placement using temporal-ordering information," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 977–1027, 1999.

[15] S. Carr, K. S. McKinley, and C.-W. Tseng, *Compiler optimizations for improving data locality*, vol. 29. ACM, 1994.

[16] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *ACM SIGPLAN Notices*, vol. 33, pp. 139–149, ACM, 1998.

[17] A. H. Hashemi, D. R. Kaeli, and B. Calder, "Efficient procedure mapping using cache line coloring," in *ACM SIGPLAN Notices*, vol. 32, pp. 171–182, ACM, 1997.

[18] W.-m. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *ACM SIGARCH Computer Architecture News*, vol. 17, pp. 242–251, ACM, 1989.

[19] J. Torrellas, C. Xia, and R. Daigle, "Optimizing instruction cache performance for operating system intensive workloads," in *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pp. 360–369, IEEE, 1995.

[20] J. E. Smith and J. R. Goodman, "Instruction cache replacement policies and organizations," *IEEE Transactions on Computers*, vol. 34, no. 3, pp. 234–241, 1985.

[21] 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.

[22] 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*, ISCA '06, (Washington, DC, USA), pp. 167–178, IEEE Computer Society, 2006.

[23] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008.

[24] 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 *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pp. 51–63, 2015.

[25] The Journal of Instruction-Level Parallelism, *The 5th JILP Championship Branch Prediction Competition (CBP-5), https://www.jilp.org/cbp2016*, June 2016.

[26] C. H. Perleberg and A. J. Smith, "Branch target buffer design and optimization," *IEEE transactions on computers*, vol. 42, no. 4, pp. 396–412, 1993.

[27] 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*, pp. 144–154, 2001.

[28] 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.

[29] 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.

[30] 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.

[31] C.-H. Chi and H. Dietz, "Improving cache performance by selective cache bypass," in *System Sciences, 1989. Vol. I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, vol. 1, pp. 277–285, IEEE, 1989.

[32] 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*, pp. 134–145, IEEE, 2002.

[33] 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.

[34] 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.

[35] 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, pp. 315–326, ACM, 1997.

[36] 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*, pp. 449–456, ACM, 1998.

[37] 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, pp. 154–163, IEEE, 1996.

[38] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay, "The split temporal/spatial cache: initial performance analysis," in *Proc. of the SCIzzL-5, Santa Clara, CA, USA*, pp. 72–78, 1996.

[39] 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*, pp. 217–226, ACM, 2014.

[40] 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*, pp. 93–103, IEEE Computer Society Press, 1995.

[41] L. Li, I. Kadayif, Y.-F. Tsai, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, and A. Sivasubramaniam, "Leakage energy management in cache hierarchies," in *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pp. 131–140, IEEE, 2002.

[42] 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*, pp. 52–66, Springer, 2006.

[43] J. Jalminger and P. Stenstrom, "A novel approach to cache block reuse predictions," in *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pp. 294–302, IEEE, 2003.

[44] 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.

[45] 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), p. 240, IEEE Computer Society, 2001.

[46] 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.

[47] 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, pp. 48–59, ACM, 1995.

[48] 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.

[49] J. B. Sartor, S. Venkiteswaran, K. S. McKinley, and Z. Wang, "Cooperative caching with keep-me and evict-me," in *Interaction between Compilers and Computer Architectures, 2005. INTERACT-9. 9th Annual Workshop on*, pp. 46–57, IEEE, 2005.

[50] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems, "Using the compiler to improve cache replacement decisions," in *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pp. 199–208, IEEE, 2002.

[51] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, pp. 280–300, September 2005.

[52] S. McFarling, "Combining branch predictors," tech. rep., Technical Report TN-36, Digital Western Research Laboratory, 1993.

[53] D. A. Jiménez, "Fast path-based neural branch prediction," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, pp. 243–252, IEEE Computer Society, December 2003.

[54] 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.

[55] B. Burgess, "Samsung's exynos-m1 cpu," in *Hot Chips: A Symposium on High Performance Chips*, August 2016.

[56] 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.

[57] C. Williams, "'neural network' spotted deep inside samsung's galaxy s7 silicon brain," August 2016.

[58] B. Calder and D. Grunwald, "Reducing branch costs via branch alignment," in *ACM SIGPLAN Notices*, vol. 29, pp. 242–251, ACM, 1994.

[59] J. K. Lee, "Branch prediction strategies and branch target buffer design," *IEEE computer*, vol. 17, no. 1, pp. 6–22, 1984.

[60] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*, pp. 135–148, IEEE Computer Society Press, 1981.

[61] R. Holgate and R. N. Ibbett, "An analysis of instruction-fetching strategies in pipelined computers," *IEEE Transactions on Computers*, vol. 4, no. C-29, pp. 325–329, 1980.

[62] B. D. Hoyt, G. J. Hinton, D. B. Papworth, A. K. Gupta, M. A. Fetterman, S. Natarajan, S. Shenoy, and R. V. D'sa, "Method and apparatus for implementing a set-associative branch target buffer," November 12 1996. US Patent 5,574,871.

[63] B. D. Hoyt, G. J. Hinton, A. F. Glew, and S. Natarajan, "Branch target buffer for dynamically predicting branch instruction outcomes using a predicted branch history," December 10 1996. US Patent 5,584,001.

[64] B. Fagin and K. Russell, "Partial resolution in branch target buffers," in *Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on*, pp. 193–198, IEEE, 1995.

[65] R. Kobayashi, Y. Yamada, H. Ando, and T. Shimada, "A cost-effective branch target buffer with a two-level table organization," in *Proceedings of the 2nd International Symposium of Low-Power and High-Speed Chips (COOL Chips II)*, 1999.

[66] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. Prasky, and A. Saporito, "Two level bulk preload branch prediction," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 71–82, IEEE, 2013.

[67] I. Burcea and A. Moshovos, "Phantom-btb: a virtualized branch target buffer design," in *Acm Sigplan Notices*, vol. 44, pp. 313–324, ACM, 2009.

[68] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 493–504, IEEE, 2017.

[69] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *In Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 222–232, 2008.

[70] A. Seznec, "An optimized 2bcgskew branch predictor," *Technical Report*, 2003.

[71] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO-33)*, pp. 67–76, December 2000.

[72] K. Skadron, M. Martonosi, and D. Clark, "Speculative updates of local and global branch history: A quantitative analysis," *Journal of Instruction-Level Parallelism*, vol. 2, January 2000.