# CHiRP: Control-Flow History Reuse Prediction

Samira Mirbagher-Ajorpaz
*Computer Science and Engineering*
*Texas A&M University*
College Station, USA
samiramir@tamu.edu

Elba Garza
*Computer Science and Engineering*
*Texas A&M University*
College Station, USA
elba@tamu.edu

Gilles Pokam
*Intel Labs*
Santa Clara, USA
gilles.a.pokam@intel.com

Daniel A. Jiménez
*Computer Science and Engineering*
*Texas A&M University*
College Station, USA
djimenez@acm.org

*Abstract*—Translation Lookaside Buffers (TLBs) play a critical role in hardware-supported memory virtualization. To speed up address translation and reduce costly page table walks, TLBs cache a small number of recently-used virtual-to-physical address translations. TLBs must make the best use of their limited capacities. Thus, TLB entries with low potential for reuse should be replaced by more useful entries. This paper contributes to an aspect of TLB management that has received little attention in the literature: replacement policy. We show how predictive replacement policies can be tailored toward TLBs to reduce miss rates and improve overall performance.

We begin by applying recently proposed predictive cache replacement policies to the TLB. We show these policies do not work well without considering specific TLB behavior. Next, we introduce a novel TLB-focused predictive policy, Control-flow History Reuse Prediction (CHiRP). This policy uses a history signature and replacement algorithm that correlates to known TLB behavior, outperforming other policies.

For a 1024-entry 8-way set-associative L2 TLB with a 4KB page size, we show that CHiRP reduces misses per 1000 instructions (MPKI) by an average 28.21% over the least-recently-used (LRU) policy, outperforming Static Re-reference Interval Prediction (SRRIP) [1], Global History Reuse Policy (GHRP) [2] and SHiP [3], which reduce MPKI by an average of 10.36%, 9.03% and 0.88%, respectively.

*Index Terms*—Translation Lookaside Buffers, Replacement Policies, Paging, Microarchitectures
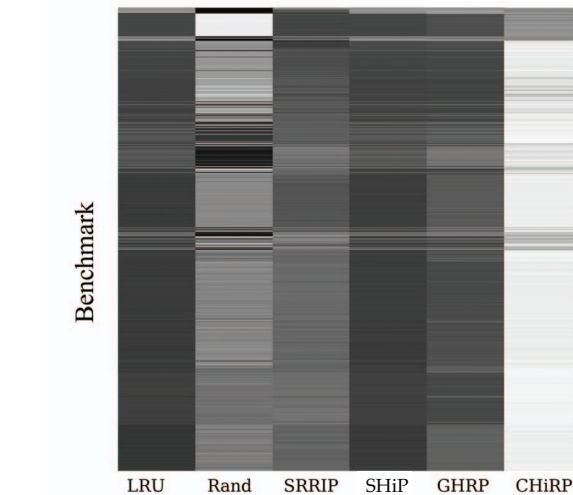
Fig. 1. Comparing predictive policy efficiency with a heat map shows CHiRP maintains more live TLB entries compared to other policies when analyzed on 870 different benchmarks. A lighter color block indicates higher TLB efficiency, while darker denotes lower efficiency.

## I. INTRODUCTION

Virtual-to-physical address translation is expensive [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. Translation lookaside buffers (TLBs) help minimize the need for costly page table walks by caching recently retrieved virtual-to-physical address mappings [16], [17].

Recent studies by Google [18], asmDB [19], and Facebook [20] confirm that modern deeply pipelined speculative OoO CPUs face increasing challenges associated with TLB performance. For example, server workloads show growing code footprints and working set sizes [18], [21], [22], [23], placing tremendous pressure on caches and TLBs [24]. The caches and TLBs of future systems will need to improve at a similar rate to maintain performance.

Unfortunately, TLBs are limited in size, and thus reach, due to power, timing, and area constraints [25]. The TLB lies on the critical path to accessing memory. Thus, *increasing L2 TLB sizes to reduce TLB misses is difficult* because larger TLBs incur higher access latencies [26].

Meanwhile, TLB misses are a first-order concern in terms of their negative impact on performance. Recently studies [27], [28], [29] indicate that many programs can spend hundreds of extra cycles conducting address translations that did not hit in the TLBs. This is despite the fact that the Skylake architecture includes special MMU/paging structure caches (or PSCs) to lessen the page walk penalty [30]. The study [27] finds L2 TLB miss costs range from 16.3 cycles for Sandy Bridge in 2011, increasing up to 212 for Skylake in 2015, 272 cycles for Broadwell Xeon in 2016, and 230 cycles for Coffee Lake in 2017. Such overhead is likely to be exacerbated in the

future[1] given that modern computing platforms can now be supplied with terabytes, and even petabytes, of main memory [32], [33] , all on various memory-intensive workloads that are rapidly emerging [18], [19], [20], [28].

Translation overheads running into 100+ cycles have also been reported in prior work [13], [14]. Address translation latencies due to TLB misses represent between 20% and 50% of system run-times today [9], [10], [13], [14], [34], [35], [36], [37], [38], [39], [40], [41] and consume a substantial share of processor energy [4], [5], [6], [7], [11], [15], [42].

Peng *et al.* conduct a thorough study of TLB behavior of Java applications [43], reporting 230+ cycle TLB miss latencies and indicate TLB miss overhead accounts for 5.5% to 19% of the total execution time. Their study finds that five out of seven benchmarks exhibit similar TLB overhead.

These concerns motivate us to investigate mechanisms to improve TLB performance that do not require increasing TLB sizes. Similar efforts to improve TLB performance have included using varied page sizes and superpages [24], [44], [45], [46], [47], [48] as well as prefetching [36], [49], [50], [51].

Fortunately, TLBs' organization makes them amenable to predictive replacement policies. TLBs are organized with tagged set-associative SRAM arrays much like cache memories. Predictive replacement policies have been well-explored and have been shown to perform well in data caches [3], [52], [53], [54] that depend on spatial and temporal locality of data accesses to maintain useful entries. Access patterns to TLBs are similar to cache accesses at a larger granularity. Thus, it is reasonable to apply previous work on cache replacement/management to TLBs.

TLB replacement policy has received little attention in the literature. Recent work [14], [34], [36], [37], [38], [55], [56], [57], [58] advocates using an LRU replacement policy for all levels of TLBs. Other prior work focuses either on reducing the cost of a page table walk upon a TLB miss [10], [34], [49], [50], [51] or reducing the TLB miss rate by extending the size of the TLB [26]. In this paper, we suggest tackling the fundamental problem of the TLB's insufficient capacity by improving its replacement policy.

Our work builds on prior predictive replacement policies geared toward the last-level cache (LLC), such as static re-reference interval prediction (SRRIP) [1], signature-based hit prediction (SHiP) [53], and Global History Reuse Prediction (GHRP) [2], to extract key insights for the TLB. We propose a novel mechanism, Control-flow History Reuse Prediction (CHɪRP), that provides superior prediction accuracy and performance by better correlating to TLB reuse behavior.

We begin with predictive policies adapted from the cache replacement literature, in particular the last-level cache (LLC), and show that they are not a good fit for TLBs. We show that features used by these schemes do not correlate well to TLB reuse, resulting in negligible performance gains. Moreover, LLC-focused prediction policies are designed with less stringent

---

[1] The new generation of Intel processors, Sunny Cove [31], introduced 5-levels radix page tabling.

cycle time requirements and can tolerate several accesses to their prediction tables. TLBs, on the other hand, have tighter timing requirements for TLB access. Based on this and other insights, we introduce a policy that efficiently indexes prediction tables using a novel signature specifically designed to correlate to TLB behavior. We focus on the L2 TLB as L2 TLB misses account for most of the cycles spent in the TLB miss handler [41].

This paper makes the following contributions:

1) A first study and exploration of TLB replacement policies by implementing and adapting policies from previous work on data caches and branch target buffers to the TLB.
2) An intuition on why previous predictive replacement policies may or may not be as effective on TLBs. We evaluate the impact of various optimizations on adapted predictive replacement policies over a large suite of industry-sourced traces.
3) A new predictive replacement policy, Control-flow History Reuse Prediction (CHɪRP). This policy indexes prediction tables using a signature specially designed to correlate with TLB behavior. It reduces L2 TLB misses by 28.21% on average over LRU, resulting in significant speedup. For example, for a page walk latency of 150 cycles, CHɪRP yields a geometric mean speedup of 4.8%.

## II. BACKGROUND

Processor performance is affected by the TLB in two ways: the *number of TLB misses* and the *TLB miss penalty* in cycles. While other solutions have mainly focused on reducing the TLB miss penalty, very little work has focused on reducing the number of misses in the TLB directly. There have been a handful of papers on prefetching into the TLB [36], [49], [50]. However, to the best of our knowledge, no previous work has proposed a predictive replacement policy specifically for TLB. Rather, recent work employs LRU or Random replacement policies [14], [34], [36], [37], [38], [55], [56], [57], [58], [59]. We advocate using a predictive replacement policy that relies on a variety of program features to guide TLB entry replacement to improve performance without needing to increase the TLB's size.

Recent work in cache and BTB replacement shows that reuse prediction can significantly reduce misses and improve performance [2], [3], [53], [54], [60], [61], [62], [63]. Predictive replacement policies attempt to predict whether a cached item will be used again before it is evicted. If not, then it is a prime candidate for eviction. This idea is superior to LRU replacement, in which a block with no near-term reuse must migrate all the way down the recency stack before being replaced. However, a highly accurate predictive replacement policy for one cache-like structure may not work for another cache-like structure. For example Mirbagher *et al.* [2] show that while PC-based policies such as SDBP [3] and SHiP [53] reduce the number of dead blocks in the LLC, it is detrimental to instruction caches and BTBs. We find the same applies to TLBs.

There are three main challenges in designing a predictive replacement policy. The first is finding the microarchitectural

features that correlate with reuse for a particular cache-like structure. These features vary for different structures such as the TLB and data caches, and even different applications [2], [3], [53], [54], [60], [61], [63]. The second is building an efficient *signature* by combining the identified correlating features. The features are combined to reduce their hardware storage budget and prediction time. The third is designing a fast/low-cost prediction algorithm to use this signature. The latter is particularly important for the TLB as it lies on the critical path to a memory access.

Once we identified highly correlating features of TLB entry reuse, we adapted previous algorithms to propose a novel, low-cost algorithm specifically tailored for reuse prediction in L2 TLBs. Previous work on LLC reuse prediction that uses prediction tables has used multiple features hashed to multiple indices [3] or signature [54], [63] to combine several predictions into one. Because the TLB is on the critical path to accessing memory, we reduce accesses to a *single table* with a signature combining several features as the most latency-sensitive approach.

We explore using predictive cache replacement policies such as static re-reference interval prediction (SRRIP) [1], signature-based hit prediction (SHiP) [53], and Global History Reuse Prediction (GHRP) [2] for the TLB, and propose a new mechanism, Control-flow History Reuse Prediction (CHiRP), to better guide TLB entry replacement.

### A. Static Re-Reference Interval Prediction

SRRIP [1] predicts which blocks will be referenced again (*i.e.* re-referenced) in the cache. Each block has a 2-bit re-reference prediction value (RRPV) placing the block into one of four categories ranging from near-immediate re-reference to distant re-reference. A first prediction is made on block placement and revised when a block is reused or replaced. Blocks with distant re-reference prediction are evicted. If there are none, the RRPV for each block in the set is incremented until there is at least one eviction candidate. We adapt SRRIP to work with TLB entries instead of cache blocks.

### B. PC-Based Dead Block Predictors

In sampling-based Dead Block Prediction (SDBP) [3], a predictor learns the pattern of accesses and evictions from a small number of sets kept in a structure called the sampler. When a load or store accesses the LLC, the address (PC) of that instruction is hashed to index prediction tables. Counters read from the tables are summed and thresholded to predict whether the block is dead. In the original SDBP paper, blocks are predicted on each access [3]. Signature-based Hit Prediction (SHiP) improves on this idea by using the prediction only for placement in a RRPV-replaced cache, reducing the number of predictions and significantly improving performance.

However, sampling is not suitable for structures indexed by instruction addresses such as the BTB and instruction cache [2]. Sampling works for data caches because the behavior of a memory access instruction, represented by its PC, generalizes over the entire cache. Instruction streams do not allow set sampling to generalize the behavior of accesses to such structures since the PC itself forms the index into the structure.

We find that sampling also does not work well for second-level TLBs. The reason is the coarser granularity of TLB entries versus cache blocks. A PC accesses different data addresses that are in the L2 TLB, which might lead one to believe sampling should generalize across the TLB. However, in the LLC, one sampled set may map to many cache sets all accessed by the same PC, which allows behavior to be generalized across sets. On the other hand, in the L2 TLB, one PC accesses data that are mapped to much fewer TLB entries than cache blocks. Spatial locality for data accessed by a single PC does not expand beyond a few TLB entries, so generalization fails.

Because of this failure, in this work we evaluate SHiP with the same general algorithm, but with bits of PC kept as metadata in each TLB entry, which is equivalent to keeping a sampler the same size as the structure. We consider SHiP to be the best cache replacement policy from previous work that would be implementable under the tight timing requirements of the TLB access critical path.

### C. Global History Reuse Prediction

Global History Reuse Prediction (GHRP) [2] is the state-of-the-art predictive replacement policy for BTB and i-cache replacement. We adapt GHRP for TLB replacement. GHRP has a structure similar to SHiP, but the signature used to index the prediction tables is specifically designed for instruction streams. Like a branch predictor, it uses the global history of conditional branch outcomes [64] as well as lower-order bits from branch addresses to form an index into a table of counters that keep track of reuse behavior.

### D. Offline Learning

We use insights from neural networks to design a new hand-crafted feature that represents a program's control-flow history compactly and that can be used with a much simpler linear learning model. Offline training has been used for designing replacement policies in the past through using genetic algorithm by Jiménez *et al.* [65] and LSTM by Shi *et al.* [66]. Their work shows how insights from offline training can improve learning model for online prediction in the LLC. We use ADALINE (ADAptive LINear Element) [67], [68] to find insights for TLB replacement policy.

ADALINE uses a vector of weights that records correlations between an input vector and a target value. It can be used to classify inputs into one of two classes.

ADALINE computes the weighted sum of the input patterns $x(n)$.

$$y(n) = w^T(n)x(n) + \theta$$

ADALINE weights are updated after the desired outcome $d(n)$ of the predicted event is known. If the prediction was correct then the weights remain unchanged. Otherwise, the inputs are used to update the corresponding weights.

$$w(n+1) = w(n) + \mu[d(n) - y(n)]x(n)$$

where $\mu$ is the learning-rate parameter and the difference $d(n) - y(n)$ is the error signal.

### E. CHiRP

We explored adapting predictive cache replacement policies to the TLB and observed that features that correlate well to cache reuse behavior may not necessarily correlate well to TLB reuse behavior. In contrast to a cache access, a TLB access is of coarser granularity with many PCs that map to the same TLB entry. Furthermore, depending on the context, each such PC may result in an eviction or a reuse of the same TLB entry. We find that predicting a TLB entry's reuse requires multiple features that we compose into a single signature for better prediction accuracy and overhead reduction.

## III. THE REUSE PREDICTION PROBLEM IN TLB & OUR SOLUTION

We find that predictive policies for the LLC, instruction cache, and BTB do not apply well to L2 TLBs, and describe the main reasons why in this section.

We simulated 870 workloads from a variety of categories provided publicly by Qualcomm [69] to prevent overfitting to one type of workload. More information about the full details of our simulation methodology can be found in Section V.

We first applied signature-based hit prediction (SHiP) [53], which was shown to be useful in the LLC. SHiP uses only the address (PC) of the most recent instruction. However, our results show that a solely PC-based reuse entry prediction does not perform much better than LRU, giving a reduction in MPKI of only 0.88%.

We investigated whether aliasing was the cause of the observed mispredictions, but found that even with an unlimited prediction table size (i.e. no aliasing), SHiP is not able to detect dead entries in the TLB, giving a reduction in MPKI of only 0.63%. Since prediction table size was not the source of the mispredictions, we further investigated by limiting the prediction to only a subset of the TLB sets and used LRU for the rest. This technique also just slightly improves accuracy, reducing MPKI by 1.28%, leading to the following observation:

***Observation 1**: The inaccuracy in previous predictive policies for the TLB is not due to conflicts among multiple sets but rather within the sets themselves.*

We find that a TLB entry may experience many hits from one or more PCs that map to the same entry before it is eventually evicted. This is because a larger range of unique addresses map to the same entry in the TLB compared to accesses to a block in a cache. Indeed, there is a nearly two order-of-magnitude difference between a 4KB page and a 64B block.

Therefore, we obtain our second observation:

***Observation 2**: The coarse-grained nature of TLB accesses results in increased aliasing in previous predictive policies, which cause the prediction counters to saturate too quickly, rendering the predictor ineffective.*

From Observation 2 we posit that in order to dissipate this noise, we need to slow down the rate at which the prediction counters are updated. We do this by limiting updates only to hits of a TLB set different from the one last accessed. We

call this method *Selective Hit Update*. Selective Hit Update improves accuracy by reducing average MPKI by 5.85%.

Previous work [2], [66] has shown that a longer history of past PCs would benefit predictive replacement policies in the LLC and i-cache. Figure 2 shows our results conducting a similar study for the TLB. Here, we analyze varying PC history lengths from 4 to 40 and their resulting speedups. We find that the benefits of using longer global PC history for TLB reuse prediction diminishes beyond a length of 15. This contrasts with prior work on predictive policies for the LLC, which show benefits of using global PC history length of 60 or more. This is likely due to the coarse-grain nature of TLB accesses that may limit the global history window from capturing enough information pertaining to TLB reuse. To improve on this, we augment the global PC history with branch path history information, resulting in a history length greater than 30 (Figure 2). Hence, our third observation is as follows:

***Observation 3**: TLB reuse prediction does not benefit from a global PC history of length 15 or more. However, by combining branch path history into a prediction signature, CHiRP can take advantage of a PC history length of 30 or more.*
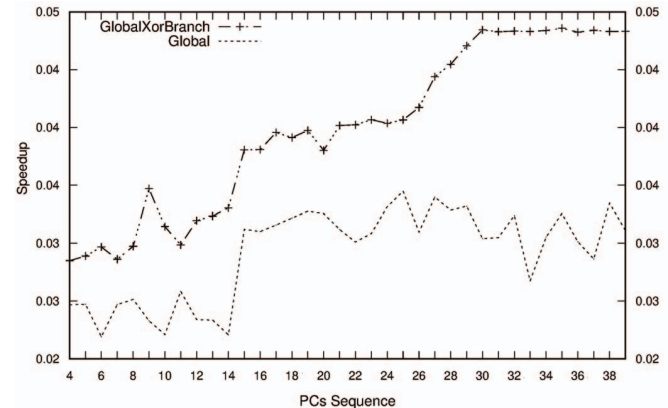


Fig. 2. Speedup does not increase for global PC history length more than 15. However, combining branch history into signature, CHiRP can benefit from history lengths longer than 30.

Branch history is effective because L2 TLB accesses come from both data and instructions in the first-level TLBs. Conditional branch histories can reflect the data accesses when global path history does not. Branch path history can also reveal high-level program semantics that also contribute to TLB misses.

### A. PC Bits Carry Uneven Weights

Previous work [65], [66] shows that certain features from program behavior are important to predicting reuse of a block in the LLC. We come to the same conclusion with regard to TLBs, recognizing that some bits of the PC carry more weight than others in reuse prediction. To show this for the case of TLBs, we use the weights of a trained ADALINE neural network to score the bits of PCs that we incorporate into the global history. The idea is based on the principle that the weights of the input
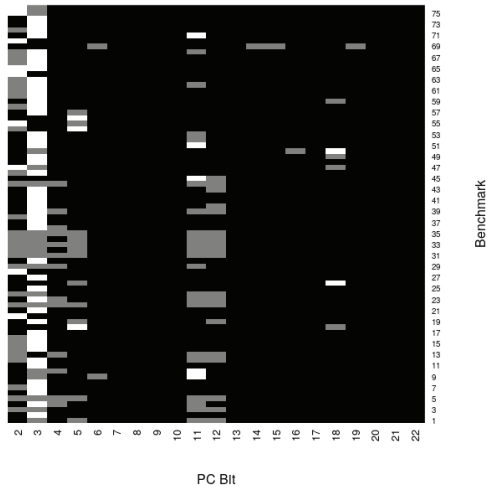
Fig. 3. Each row represents an offline-trained ADALINE weight vector for one benchmark. The $x$-axis shows the PC bit used as input. The white boxes show reuse prediction in TLB entries is strongly correlated with bits 2 and bit 3 of the PC.

nodes corresponding to less important features are expected to be smaller in trained ADALINE networks. The incorporation of appropriate regularization terms in the ADALINE update function encourages such weights to converge to zero and ultimately be eliminated.

Figure 3 shows that the two lower-order bits of a PC address (bits 2 and 3) contain important information, as indicated by their higher-weight color values. Thus, passing on these bits to the signature function yields a high chance of preserving information to reduce aliasing. In our proposed CHIRP policy described in Section IV, we keep these two correlated bits in the global path history.

### B. Modeling Efficient Signatures

Aliasing in the prediction table is harder to solve in the TLB than caches. With TLB reuse prediction, far too many PCs map to the same TLB entry, *i.e.*, 64 times more than to a cache block. The problem of aliasing is exacerbated further with large footprint applications.

If a counter in the prediction table changes direction frequently due to aliasing, the same problem will only be exacerbated with a smaller table size. To achieve high reuse prediction accuracy with a smaller table size, we have to solve aliasing first.

This problem can be addressed by coordinating how the input bits are transformed by designing a succinct signature. We found that employing shifting and scaling techniques as described by Lecun and Hinton [70], [71] improves prediction accuracy.

We accomplished this by injecting and shifting leading zeros into specific bit positions of different components of the signature including the global path history, conditional branch history, indirect branch history, and the shifted PC of the access (section IV).

Doing this both shifts the individual PCs and scales the

less salient history bits down to make them *less visible* to the learning process, allowing the prediction table to converge to an accurate counter value with 3 times fewer entries than GHRP.

The above techniques of shifting and scaling the signature bits are simple to implement in hardware and provide significant reduction in TLB MPKI. Figure 6 shows that while adding conditional branch path history to the signature would reduce MPKI by 23.88%, adding two leading zeros in the path history would allow the effect of conditional branch history to reduce MPKI by 26.98%.

In the next section we discuss our signature function and the individual effect of above optimizations.

## IV. CONTROL-FLOW HISTORY REUSE PREDICTION ALGORITHM

### A. Overview

CHIRP correlates TLB replacement with reuse history. CHIRP uses features that best correlate to reuse behavior and combines them into a signature that is used to uniquely tag each TLB entry (IV-B).

This signature is subsequently used to track the reuse behavior of the associated TLB entry by means of a prediction table indexed by the signature (IV-C). The prediction table is updated on an eviction or a reuse, and the resulting prediction status is written back into the corresponding TLB entry to inform the next TLB replacement operation (IV-D). Figure 4 describes the main components of CHIRP and Algorithm 5 provides the CHIRP algorithm.

### B. CHIRP *Signature*

CHIRP contributes four features that correlate with reuse behavior. The first is the global path history of PCs. The global path history in CHIRP is 64 bits wide and is updated on each access by shifting the two lower-order bits of the PC into the path history, followed by two zero bits (Figure 5, line 28), as previously discussed in subsections III-A and III-B, respectively. The global path history in CHIRP allows recording the last 16 accesses.

The second and third features are the conditional and unconditional indirect branch address history, respectively. Each of these histories is 64 bits and is updated by shifting the eight bits of the PC [11:4] into the branch history on every conditional (resp. unconditional indirect) branch instruction (Algorithm 5, line 31.), recording the last 8 branch accesses for each type.

The fourth feature is the current PC, shifted right by two bits. The signature is constructed by XOR-ing the global path history with the conditional branch history, the unconditional indirect branch history, and the shifted PC of the access (Figure 5, line 5).

To compute indices into the prediction table, CHIRP computes a 16-bit hash of the constructed signature. For hashing, we first use Robert Jenkins' 64-bit mix function [72]. The mix function enables a single-bit change in the key to influence widely disparate bits in the hash result. We then
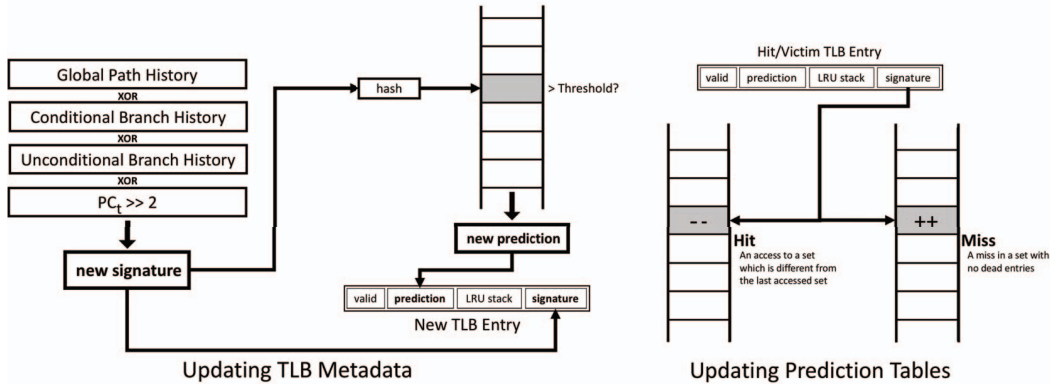
Fig. 4. CHIRP TLB metadata and prediction table update flow using a signature.

```
 1: int predTable[numCounters]
 2: procedure ACCESSTLB(int VA)
 3:     set ← calcSet(VA)
 4:     isMissed ← isTagMatch(VA)
 5:     sign ← VA»2 ⊕ pathHist ⊕ condBrHist ⊕ unCondBrHist
 6:     index ← Hash(sign)  mod 2^16
 7:     cntrNew ← predTable[index]
 8:     if isMissed = true then                    ▷ miss
 9:         entry ← victimEntry(set)
10:         if entry.isDead = false then           ▷ lru
11:             index ← Hash(entry.signature)
12:             updatePredTables(index, true)
13:         entry.firstHit = true                  ▷ insertion
14:     else                                       ▷ hit
15:         entry ← matchedEntry(set, tag)
16:         if entry.firstHit = true then          ▷ access table
17:             indices ← Hash(entry.signature)
18:             updatePredTables(index, false)
19:             entry.dead ← predict(cntrNew, deadThresh)
20:             entry.firstHit ← false
21:     entry.signature ← sign
22:     update LRU stack Position()
23:     UpdatePathHist(VA, pathHist)
24:     if instType = conditionalBranch then
25:         UpdateBrHist(VA, condBrHist)
26:     if instType = unConditionalBranch then
27:         UpdateBrHist(VA, uncondBrHist)

28: procedure UPDATEPATHHIST(int VA, int history)
29:     history ← history << 4
30:     history ← (history | VA_{2..3})

31: procedure UPDATEBRHIST(int VA, int history)
32:     history ← history << 8
33:     history ← (history | VA_{4..11})

34: procedure PREDICT(int counter, int threshold)
35:     if counter > threshold then return true
36:     else return false

37: procedure VICTIMENTRY(Set set)
38:     for int i = 1 to associativity  do
39:         entry ←  set.entries[i]
40:         if entry.isDead = true then return entry
        return LRUentry()

41: procedure UPDATEPREDTABLE(int index, bool Dead)
42:     if Dead = true then
43:         predTable[index]++
44:     else
45:         predTable[index]−−
```

Fig. 5. CHIRP algorithm.

take the modulo of the table size to generate the hash table index(Figure 5, line 6).

Note that the signature relies on bits from the branch PC, not conditional branch outcomes or bits from branch targets.

### C. CHIRP Prediction Table

CHIRP stores metadata for each L2 TLB entry, consisting of 3 LRU stack position bits, a valid bit, a 16-bit signature and a prediction bit (See Figure 4, Updating TLB Metadata). CHIRP uses a table of saturating counters to provide a prediction. The table is indexed by a hash function of the signature. The corresponding counter is thresholded, and if the counter exceeds the threshold, the entry is predicted as dead.

### D. CHIRP Operations

In contrast to SHiP and GHRP that require updating the prediction table on each TLB access, the bulk of CHIRP operations occurs off the TLB critical path, with minimal impact to TLB latency. In particular, CHIRP updates its prediction table on a TLB miss only if the selected victim is LRU (i.e. no dead entry is found).

The operations pertaining to a TLB miss involve (1) selecting a victim, (2) updating the victim's reuse history in the prediction table if the victim is LRU, and (3) updating the prediction metadata for the new TLB entry.

*a) Victim selection*

On a TLB miss, CHIRP first attempts to select a victim among the entries predicted as dead. If no such entry is found, CHIRP evicts the LRU entry (Figure 5, line 37).

*b) Prediction table update*

Because CHIRP updates its prediction table only if the victim is LRU (Figure 5, line 10 – 12), evicting the LRU entry effectively makes it a dead candidate the next time around. This justifies why the prediction table needs be updated. The signature of the victim entry is used to index the prediction table and the corresponding counter is incremented, since the entry was just shown to be dead (Figure 5, line 41).

*c) Prediction metadata update*

After the new entry is inserted into the TLB, its CHIRP metadata is updated to inform CHIRP of the next replacement decision. First, the signature of the new entry is used to index the prediction table and then the corresponding counter is read out (Figure 5, line 6). The counter value is thresholded and used to decide if the incoming entry should be predicted dead or live in the future. The resulting prediction status is then used to update the prediction bit in the CHIRP metadata.

On a TLB hit CHIRP updates its prediction table only if the current access is the first hit to the TLB entry line 16). These optimizations improve both performance and energy as they reduce the frequency of access to the CHIRP prediction table to only 10.14% of all TLB accesses (Figure 11). In addition, for smaller prediction tables, they prove very effective at improving MPKI by reducing aliasing (Figure 9). These optimizations and results are discussed in detail in Section VI).

A TLB hit (Figure 5, line 14) involves the following operations:

*d) Prediction table update*

On a hit the prediction table is accessed only on the first access or reuse (Figure 5, line 16). The old signature in the entry (Figure 5, line 17) is used to index the prediction table and the corresponding counter is then decremented to assure this entry will be predicted as live under the same conditions in the future (Figure 5, lines 18 and 41). Then the old signature is replaced with the new one.

*e) Prediction metadata update*

The new signature of the hitting entry is used to index the prediction table and then the corresponding counter is read out. The counter value is thresholded and used to decide if that entry should be predicted dead or live in the future. The resulting prediction status is then used to update the prediction bit in the CHIRP metadata. Figure 5 summarizes the steps taken during a TLB hit.

*E. Adapting Training Algorithm for TLB*

Access to a TLB reuse predictor must be fast and energy efficient, as the TLB is on the critical path to accessing memory. Thus, we are motivated to minimize the number of updates made to prediction structures. We find that two specific events are sufficient for an accurate training update:

- The first hit of an entry.
- A miss in a set with no dead entry (this leads the algorithm to choose an entry to evict based on LRU.)

With this technique, CHIRP reduces the access ratio to the prediction tables by 90% compared to SHiP and GHRP (see Figure 11), which must access tables on *every* access to the TLB.

| Component | Size |
|---|---|
| Prediction bits | 1 bit × 1024 = 128B |
| FirstHit bits | 1 bit × 1024 = 128B |
| Signature bits | 16 bits × 1024 = 2KB |
| Path history register | 64 bit × 1 = 8B |
| Cond. history register | 64 bit × 1 = 8B |
| Uncond. history register | 64 bit × 1 = 8B |
| Counters | 128B .. 8KB |
| Total | 2.775KB .. 8.265KB |

TABLE I
STORAGE OVERHEAD OF CHIRP FOR A 1024 ENTRY, 8-WAY L2 TLB WITH
4KB PAGES.

| Processor | Parameter |
|---|---|
| L1 i-Cache | 64KB, 8 way, 4 cycles |
| L1 d-Cache | 64KB, 8 way, 4 cycles |
| L2 Unified Cache | 256KB, 16 way, 12 cycles |
| L3 Unified Cache | 8MB, 16 way, 42 cycles |
| DRAM | 240 cycles |
| Branch Predictor | Hashed perceptron, 4K entry BTB, 20 cycle miss penalty |
| L1 i-TLB | 64 entry, 8 way, 1 cycle |
| L1 d-TLB | 64 entry, 8 way, 1 cycle |
| L2 Unified TLB | 1024 entries, 8 way, 8 cycle hit latency, 20 to 360 cycle miss penalty |

TABLE II
SIMULATION PARAMETERS

## V. METHODOLOGY

To implement and test CHIRP, we use the simulator and traces released for the recent Championship Value Prediction Competition (CVP1) [69]. There are hundreds of traces available (of which we use 870), coming from a variety of workload categories of interest to Qualcomm who provided them. In particular, the workloads come from the team working on their (now defunct) server project. The traces contain SPEC, database, crypto, scientific, web, "big data" and other applications, many of which exhibit interesting address translation behavior. The traces contain very detailed information such as instruction type, register values, effective addresses of loads and stores, and data values, making them suitable to drive a performance simulator. Short traces are simulated completely, while long traces are allowed to run for 100 million instructions. To measure the performance numbers

we built a timing-approximate performance model similar to previous work [37].

Our model simulates first-order sources of processor latency such as the memory hierarchy composed of L1 i-TLB and L1 d-TLB, L1 i-cache, L1 d-cache, L2 and L3 unified caches, DRAM, a branch prediction unit that includes an indirect branch predictor, a conditional branch predictor with branch target buffer, and an in-order pipeline model. We use a hashed perceptron predictor as the branch direction predictor [73].

We measure misses per 1000 instructions (MPKI) as well as instructions per cycle (IPC) based on the simulated microarchitecture across a variety of page table walk latencies derived from previous work. A recent reverse engineering study on TLB [27] reported a range of L2 TLB miss penalties for Intel microarchitecture: 230 cycles for Coffeelake, 272 cycles for BroadwellXeon, 212 cycles for Skylake and 18 cycles for Haswell. A related study on TLBs, Li *et al.* [12], uses 150 cycles for L2 TLB miss penalties. We measure speedup for a range of 20 to 340 cycles page walk latencies, shown in Figure 10.

We model static re-reference interval prediction (SRRIP), signature-based hit prediction (SHiP), global history reuse prediction (GHRP) and control-flow history reuse prediction (CHiRP). CHiRP keeps metadata for each L2 TLB entry. CHiRP also uses one prediction table. Each of the entries in the table contains a two bit counter. The additional metadata for each entry consists of 1 prediction bit, 3 bits to maintain LRU positions, and 16 bits of signature. Table-I summarizes the storage requirements for CHiRP for a 1024-entry 4KB page size L2 TLB with 8-way associativity.

We assume a 4KB page size similar to previous work [12]. Large pages are supported in current microarchitectures, *e.g.* Intel's Skylake supports page sizes of 4KB, 2MB, 4MB, and 1GB. Large pages can reduce capacity misses in TLBs when program behavior exhibits high locality. However, 4KB pages are still the norm for most mobile and desktop operating systems, providing a good balance between impact of page faults for workloads with good locality and impact of fragmentation for workloads with poor locality. It would be easy to say, "just use large pages" but the performance of legacy systems, mobile apps, cloud computing workloads, etc. that continue to use 4KB pages matters to users of those systems.

The complexity of variable-sized TLB entries (as compared to fixed-sized lines in cache replacement) further complicates efforts to improve TLB replacement. Entries for different sized pages share the L2 TLB; as the L2 TLB is built for capacity, it is not partitioned among page sizes.

Reasoning about how to do replacement with a mix of page sizes is an interesting problem we plan to tackle in future work; imagine, when one entry covers 4KB and another covers 2MB, which one is is more important to keep? It is no longer a matter of pure replacement in the sense of trying to achieve Bélády's optimal result [74], but now requires taking into account the different costs of replacing different sized entries [75], [76], [77], [78]. This question is beyond the scope of this initial study. In addition, large pages' susceptibility to memory

fragmentation requires simulating traces with varying levels of known fragmentation behavior, complicating an already complex issue. We hope this initial work invites the community to consider and tackle the problem of TLB replacement further from the surface work seen so far. Thus, in this initial study of predictive replacement policies for TLBs, we focus on the standard 4KB page size.

## VI. RESULTS

In this section, we describe the results of experiments simulating the CHiRP policy and demonstrate its superior over policies used in previous work. Results with a range of hardware budgets are presented. In the absence of public data about L2 TLB size, our calculation accounts only for tag, physical page number, replacement metadata, protection bits, valid bit, and ASID, estimating 118 bits for a TLB entry, giving 14.75KB for a 1024-entry TLB. A 6% TLB overhead places CHiRP overhead of 1KB, which still offers a 28% MPKI improvement (see Figure 9).

### A. MPKI Results

Figure 7 shows an S-curve of MPKIs for 870 benchmarks. The $x$-axis shows the benchmarks in order of sorted MPKI for LRU and is compared with other policies. Insets highlight key areas of the graph.

LRU and Random yield an average 1.51 and 1.47 MPKI, respectively. SRRIP, which uses a simple static prediction on each TLB entry placement and has a lower cost than LRU, yields 1.35 MPKI, a 10.36% improvement over LRU. SHiP, a PC-based reuse predictor, gives an average 1.50 MPKI, performing almost the same as LRU with 0.88% improvement. GHRP, which uses a more detailed prediction signature, yields an average 1.37 MPKI, or a 9.03% reduction in misses over LRU. CHiRP, with a signature specially designed for TLB replacement, gives an average MPKI of 1.08% an improvement of 28.21% over LRU. Nearly all of the tested benchmarks exhibit considerable MPKI reduction under CHiRP, achieving an improvement of 58.93% in some cases.

These results demonstrate that the case for LRU as a TLB replacement policy is weak, as even Random replacement slightly outperforms it. SRRIP, which is a simpler and low-overhead policy, could more conveniently be deployed in current processors, yielding better performance. CHiRP is somewhat more complex but yields the best improvement, more than double the improvement provided by SRRIP.

### B. Accesses to Prediction Table

Predictive replacement policies often access tables of counters to make a prediction. Previous work on cache and BTB replacement policies read out from the tables on every access to the cache/BTB to make a prediction for the next access. The tables are also modified frequently as counters are updated.

CHiRP only accesses the prediction table on a TLB miss or on a hit to a TLB set different than the one accessed last, following our selective hit update policy. It follows from this that consecutive hits to the same set do not result in writing or reading from the prediction table but only updates to the

Fig. 6. Effect of correlating features, transforming input, shifting PC bits and scaling, signature formula and prediction table update policies on reducing misses in the L2 TLB. The $x$-axis is the reduction rate of average MPKI over 870 traces over a baseline LRU. Previous predictive replacement policies need specific optimizations to work for L2 TLB. Results show the advantage of CHⅰRP.
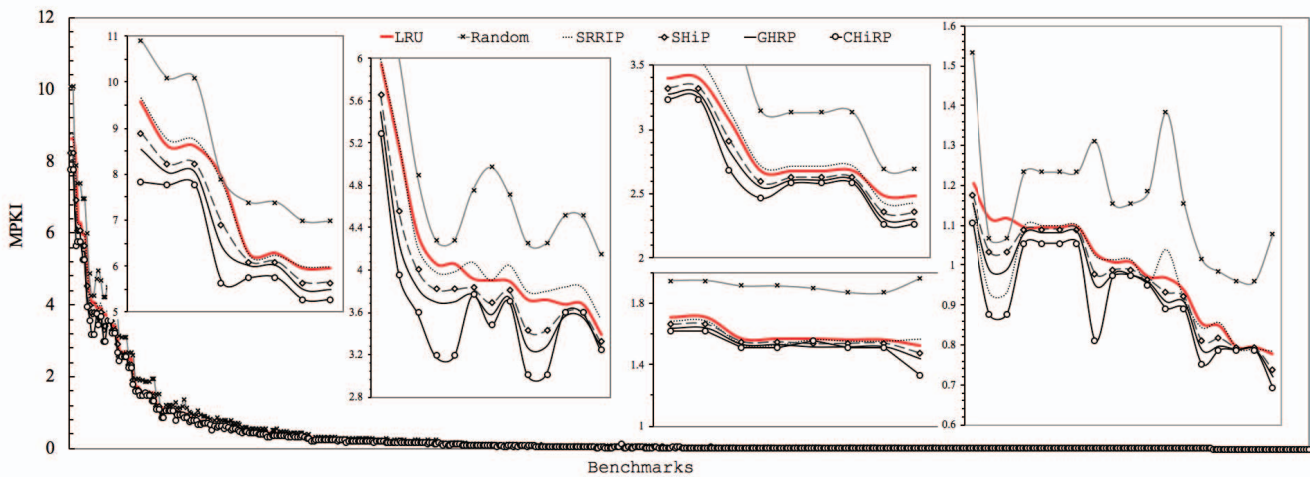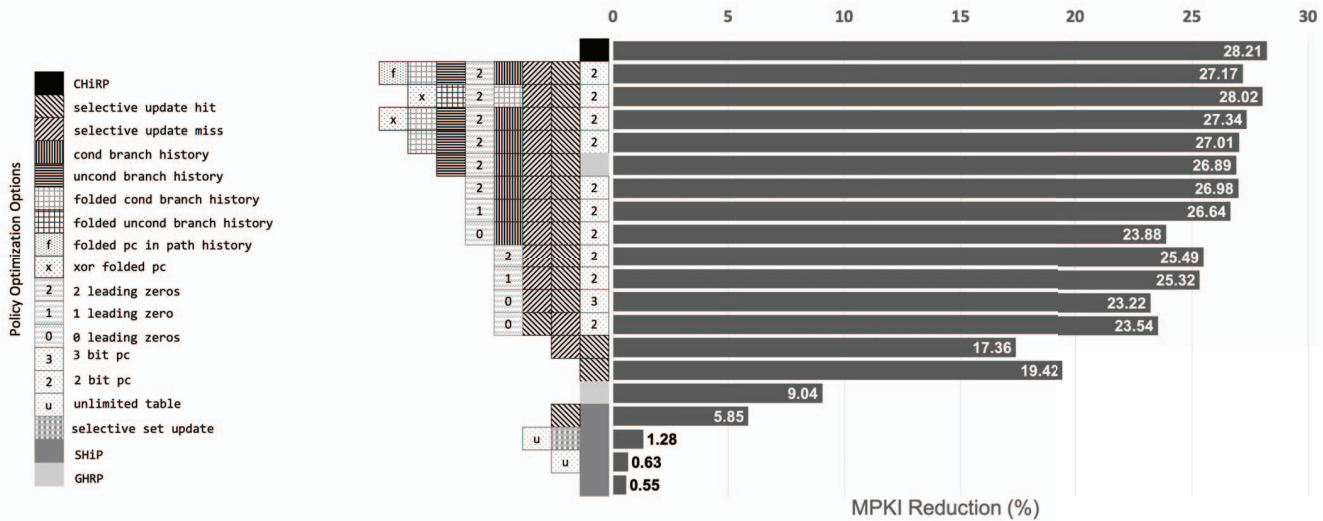


Fig. 7. MPKI comparison of various policies. The horizontal axis shows the benchmarks in the order of sorted MPKI for LRU. Multiple zoomed-in areas of the graph are shown in insets.

signature in the TLB entry. Updating the signature bits in the TLB entry has the same overhead as an LRU stack update. Thus, the energy and timing properties of CHⅰRP are considerably more favorable to implementation than techniques based on previous predictive replacement policies.

Figure 11 shows a density plot of the rate of the number of accesses to prediction table over accesses to the TLB for SHⅰP, GHRP and CHⅰRP. The plot shows the distribution of access rates over all the benchmarks. For SHⅰP and GHRP, the access rate has a very high variance, reaching over 100% in many cases. The rate can exceed 100% because, for every TLB access, there could be two accesses to the prediction table: one to read out the prediction, and another to update the table for training. For CHⅰRP the access rate is quite low, and

has low variance, making for a far more practical policy for implementation. On average CHⅰRP accesses the prediction table for 10.14% of the accesses to L2 TLB.

### C. Speedup

Figure 8 shows speedup for various policies with TLB miss penalty of 150 cycles. Page walk latency depends on several microarchitectural and software parameters. Thus, we explore a range of L2 TLB miss penalties to provide an estimate of performance under different assumptions. With TLB miss penalty of 150 cycles, CHⅰRP improves performance by 4.80% compared to 0.42% for Random, 1.65% for SRRIP, 0.13% for SHⅰP, and 0.94% for GHRP. At higher latencies, the advantage of predictive policies grows. With a penalty of 320 cycles, representing more memory intensive behaviors, CHⅰRP

provides a speedup of over 10%. Other predictive replacement policies do not provide significant speedup. Clearly, CHiRP provides significant improvement to performance over all other replacement policies.
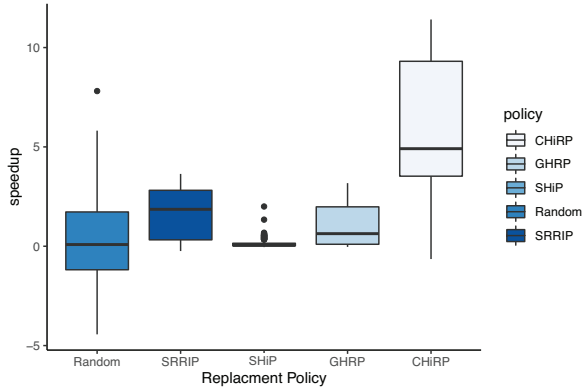


Fig. 8. Speedup for 870 traces.

### D. TLB Efficiency

Cache efficiency is the average amount of time in which a block was live in the cache. We calculate cache efficiency [79] for TLB entries instead of cache blocks. Figure 1 depicts cache efficiency for the L2 TLB for 870 benchmarks. Each row is the cache efficiency of one benchmark for various policies scaled by LRU. Benchmarks are sorted from low to high cache efficiency from down to up respectively. Figure 1 shows Random helps improve the cache efficiency of some benchmarks (most which had high efficiency already) but CHiRP removes most of the dead entries in TLB for all 870 benchmarks. CHiRP improves average cache efficiency over 870 traces by 8.07% compared to LRU. This number is 2.92% for GHRP, 1.85% for SHiP, 2.84% for SRRIP, and 3.10% for Random. Thus, the MPKI improvement in CHiRP comes from reducing dead entries and increasing live entries in the TLB.

### E. Complexity and Efficiency

Note that CHiRP is more complex than simple replacement policies such as LRU and RRIP. However, it is far less complex than, for example, branch prediction techniques such as TAGE [80] and perceptron [81] that have been implemented in recent processors. These predictors require far more logic, dynamic energy, and state than CHiRP and have tighter timing constraints. Thus, we believe the complexity of CHiRP is very manageable given its benefits to front-end performance.

Consistent with branch predictor implementation, CHiRP only updates the tables of counters at commit with right-path branches to prevent pollution of the tables. For misprediction recovery, CHiRP maintains two path histories: the speculative history updated using the outcome of the branch predictor, and a non-speculative history updated when a branch commits.

The energy overhead of predictive policies results from accesses to the prediction table and updating respective metadata in the TLB entry. Because CHiRP reduces the number

of accesses to the prediction table by 90% compared to previous predictive policies (*e.g.* SHiP and GHRP), energy consumption related to accessing the tables should be less of a concern (Figure 7).

CHiRP requires more accesses to the prediction metadata in the TLB compared to SHiP. While SHiP only updates metadata at insertion time, CHiRP updates the signature in the TLB during both hit and insertion. CHiRP updates the metadata in the same manner as updating LRU bits. While SHiP cannot perform better than LRU for the L2 TLB, updating the metadata on every access is the cost for an accurate predictor.
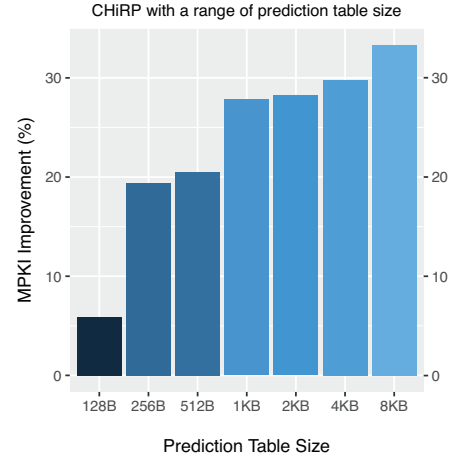


Fig. 9. MPKI improvement over LRU for CHiRP with a range of prediction table sizes.

### F. Impact of Predictor Size

Figure 9 shows MPKI improvement over LRU for CHiRP with a range of prediction table sizes. The size of the prediction table has an impact on area, energy, and timing, so we would like to choose a size that yields a good improvement while maintaining a reasonable cost. At a very small hardware budget of 128B, we note that even though we may experience higher conflicts rate in the prediction table, CHiRP still yields up to 7% MPKI improvement over LRU. As we double the prediction table size to 256B and 512B, the MPKI improvement increases to up to 20% and 22%, respectively. What this shows is that even with a small hardware budget size of 256B, CHiRP doubles the MPKI improvement realized by an 8K GHRP (9%). Table sizes of 1K and 2K yield similar improvement: about 28% MPKI reduction; our main results are presented with a 1K budget. Gains realized by larger table sizes are higher, but come with larger area overhead.

### G. Performance Gain

This first study focuses on single core. IPC gains in each generation are usually within 10-15%, of which 20-40% might come from the front-end (2-4% overall), which is considered aggressive. A 4.8% improvement over LRU is a significant milestone in this case. Figure 11 shows the speedup for CHiRP is statistically significant over 870 workloads assuming a TLB miss penalty of 150 cycles.

## H. Area Overhead vs. Performance

CHiRP reduces hardware overhead by two-thirds compared to GHRP because the signature formula enables CHiRP to use one prediction table rather than the three needed by GHRP. The predictor cost was evaluated for a range of extra overhead. Figure 9 shows even a small 256B predictor leads to a 20% MPKI reduction. As a matter of comparison, a recent study from Intel [82] demonstrates a branch prediction technique that costs 64KB hardware overhead improves IPC by 2.7%. CHiRP for a TLB with 1KB overhead and 4.80% speedup is 13× more efficient in terms of speedup-per-KB overhead. This is due to the high TLB page walk latency compared to other miss penalties in the pipeline.
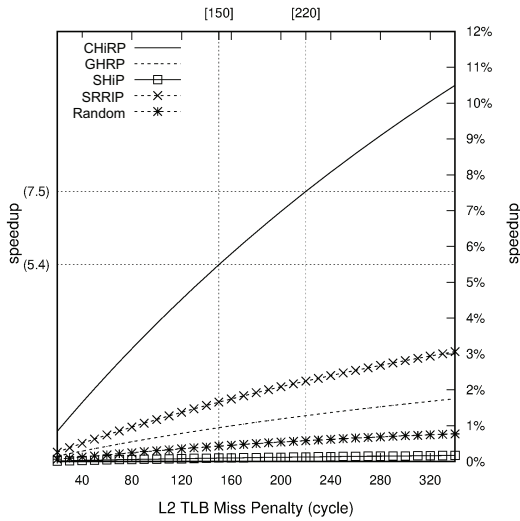
Fig. 10. Average Speedup for a range of L2 TLB miss penalties over 870 traces.

## VII. RELATED WORK

### A. Reuse Prediction

This paper proposes reuse prediction for TLB replacement. However, reuse prediction has a long history in the literature. Caches often retain *dead blocks*, *i.e.* blocks in the cache that will not be used again until they are evicted [60]. Dead blocks waste space and energy in the cache. Lai *et al.* initially proposed dead block prediction [60] to prefetch data into predicted dead blocks. Kharbutli *et al.* propose a counter-based dead block prediction approach [83] for replacement and bypass. Liu *et al.* [84] propose a predictor leveraging the burst-like nature of accesses to the L1 cache. Teran *et al.* propose using perceptron learning for reuse prediction [54], [63].

Dead block prediction has been evaluated in the context of making replacement decisions in the L1 data cache [3], [53], [54], [60], [63], [84], last-level cache [3], [84], [85], prefetching [60], [86], bypassing [87], [88], [89], [90], [91], [92], [93], [94], [95], [96], [97], [98], [99], power reduction [100], [101], and cache coherence protocol optimization [102], [103], [104]. However, no replacement policy has been proposed for the TLB based on dead block prediction.
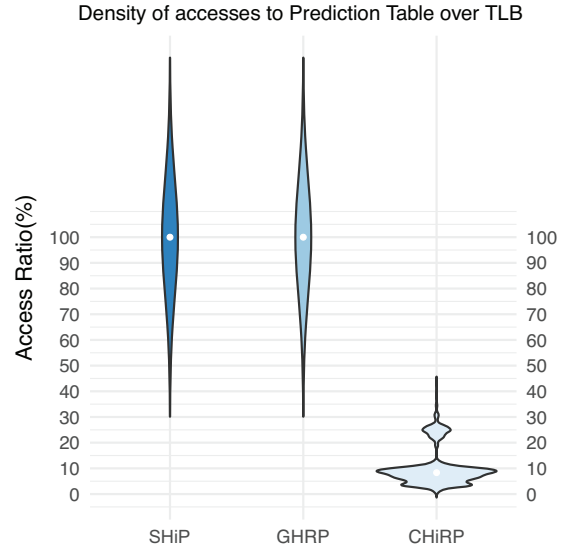
Fig. 11. Density plot for rate of accesses to prediction table over accesses to TLB for SHiP, GHRP and CHiRP with mean values.

### B. Translation Lookaside Buffers

Previous work shows that superpages, *i.e.* any page size larger than the default, can increase TLB reach and reduce TLB misses. Large pages are especially beneficial for applications with a hard upper bound of memory usage in terms of maximal heap size [43], [44], [45], [47], [48], [105], [106]. Still, Peng *et al.* [43] show that while superpages can remove nearly all the TLB miss overhead of some benchmarks, an increased page size of 1MB cannot cover the working set of some benchmarks due to unpredictable memory access patterns. If memory access patterns are predictable, TLB misses can be reduced through prefetching and speculation [10], [34], [49], [50], [51].

Conversely, using superpages may unnecessarily increase the memory footprint of an application, resulting in elevated, but useless, paging traffic and memory allocation. Additionally, handling multiple page sizes increases complexity in the operating system [44], [45], [46], [47], [107], [108]. Algorithms to evaluate the need for larger pages based on applications' behavior are essential for choosing the appropriate page size. Techniques for mapping multiple smaller pages into a single superpage TLB entry [37], [45], [56], [57], [109] reduce splintering and make superpage usage more efficient, but require deep OS-hardware co-design.

With the prevalence of chip multiprocessors (CMPs) and parallel workloads, recent TLB work has focused on distributed TLBs in architectures. Cooperative TLB [36], [110] and shared last-level TLB [39], [51], [55] schemes have been proposed.

## VIII. Conclusions and Future Work

This paper extensively investigated the replacement policy of TLBs, which has been rarely studied in previous work. In the past, the only way to provide a predictive policy for larger cache structures was to use a sampling method. We show that sampling does not work in the L2 TLB. The idea of sampling is to generalize learning over sets; we used the granularity of L2 TLB entries to generalize learning instead of sampling. Prior work does not recognize the effect of the granularity of a structure on sampling and dead block prediction. The signatures of previous policies do not detect dead blocks in the L2 TLB. Because they do not follow control flow, it was impossible for them to learn the reuse patterns in the L2 TLB properly. They end up averaging over traces, whereas we present a specific signature that tracks the trace of dead blocks in a large granularity environment while minimizing the prediction counters' fluctuations. That allows CHiRP to use a small table with fast convergence, providing a predictive replacement policy that fits into constraints of the L2 TLB for the first time. In future work we plan to extend CHiRP to TLBs with mixed page sizes.

## IX. Acknowledgement

## 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. ACM, 2010, pp. 60–71.

[2] 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

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

[4] T. Juan, T. Lang, and J. J. Navarro, "Reducing tlb power requirements," in *Proceedings of 1997 International Symposium on Low Power Electronics and Design*, Aug 1997, pp. 196–201.

[5] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen, "Generating physical addresses directly for saving instruction tlb energy," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, Nov 2002, pp. 185–196.

[6] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam, "Compiler-directed physical address generation for reducing dtlb power," in *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*, March 2004, pp. 161–168.

[7] D. Fan, Z. Tang, H. Huang, and G. R. Gao, "An energy efficient tlb design methodology," in *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ser. ISLPED '05. New York, NY, USA: ACM, 2005, pp. 351–356. [Online]. Available: http://doi.acm.org/10.1145/1077603.1077688

[8] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating two-dimensional page walks for virtualized systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346286

[9] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: Skip, don't walk (the page table)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 48–59, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1815970

[10] ——, "Spectlb: A mechanism for speculative address translation," *SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 307–318, June 2011. [Online]. Available: http://doi.acm.org/10.1145/2024723.2000101

[11] A. Sodani, "Race to exascale: Opportunities and challenges," in *Keynote at the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[12] Y. Li, R. G. Melhem, and A. K. Jones, "Ps-tlb: Leveraging page classification information for fast, scalable and efficient translation for future cmps," *TACO*, vol. 9, pp. 28:1–28:21, 2013.

[13] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 178–189. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.37

[14] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large pages and lightweight memory management in virtualized environments: Can you have it both ways?" in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830773

[15] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 631–643.

[16] J. F. Couleur and E. L. Glaser, "Shared-access data processing system," November 19 1968, uS Patent 3,412,382.

[17] D. W. Clark and J. S. Emer, "Performance of the vax-11/780 translation buffer: Simulation and measurement," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 31–62, February 1985. [Online]. Available: http://doi.acm.org/10.1145/214451.214455

[18] S. Kanev, J. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2014, pp. 158–169.

[19] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 462–473. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322234

[20] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "Softsku: Optimizing server architectures for microservice diversity @scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 513–526. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322227

[21] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[22] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 30–42. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173178

[23] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2018, pp. 643–656.

[24] S. Srinivas, U. Pawar, D. Aribuki, C. Manciu, and G. Schulhof, "Runtime performance optimization blueprint: Intel architecture

optimization with large code pages," Intel, Tech. Rep. Intel White Paper, https://software.intel.com/sites/default/files/managed/a0/0e/RuntimePerformanceOptimizationBlueprint_LargeCodePages.pdf, 2019.

[25] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[26] J. H. Ryoo, N. Gulur, S. Song, and L. K. John, "Rethinking tlb designs in virtualized environments: A very large part-of-memory tlb," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 469–480. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080210

[27] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security*, August 2018. [Online]. Available: https://www.vusec.net/download/?t=papers/tlbleed_sec18.pdf

[28] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, "Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1093–1108.

[29] D. Skarlatos, U. Darbaz, B. Gopireddy, N. S. Kim, and J. Torrellas, "Babelfish: Fusing address translations for containers."

[30] Intel Corporation, "Intel 64 and ia-32 architectures optimization reference manual," Intel Corporation, Tech. Rep. Order Number: 248966-033, 2016.

[31] "5-level paging and 5-level ept white paper," http://software.intel.com/content/www/us/en/develop/download/5-level-paging-and-5-level-ept-white-paper.html, May 2018.

[32] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[33] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, pp. 1–134, 2011.

[34] A. Bhattacharjee, "Translation-triggered prefetching," 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. 63–76. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037705

[35] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee, "Hardware translation coherence for virtualized systems," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 430–443.

[36] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative tlb for chip multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 359–370. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736060

[37] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," 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. 435–448. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037704

[38] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 383–394. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540741

[39] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared last-level tlbs for chip multiprocessors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 62–63. [Online]. Available: http://dl.acm.org/citation.cfm?id=2014698.2014896

[40] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant memory mappings for fast access to large memories," *SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 66–78, June 2015. [Online]. Available: http://doi.acm.org/10.1145/2872887.2749471

[41] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 237–248, June 2013. [Online]. Available: http://doi.acm.org/10.1145/2508148.2485943

[42] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos, "Prediction-based superpage-friendly tlb designs," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 210–222.

[43] J. Peng, G.-Y. Lueh, G. Wu, X. Gou, and R. Rakvic, "A comprehensive study of hardware/software approaches to improve tlb performance for java applications on embedded systems," in *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, ser. MSPC '06. New York, NY, USA: ACM, 2006, pp. 102–111. [Online]. Available: http://doi.acm.org/10.1145/1178597.1178614

[44] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in supporting two page sizes," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992, pp. 415–424.

[45] M. Talluri and M. D. Hill, "Surpassing the tlb performance of superpages with less operating system support," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: ACM, 1994, pp. 171–182. [Online]. Available: http://doi.acm.org/10.1145/195473.195531

[46] J. Navarro, S. Iyer, P. Druschel, and A. Cox, "Practical, transparent operating system support for superpages," in *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, ser. OSDI '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 89–104. [Online]. Available: http://dl.acm.org/citation.cfm?id=1060289.1060299

[47] N. Ganapathy and C. Schimmel, "General purpose operating system support for multiple page sizes," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '98. Berkeley, CA, USA: USENIX Association, 1998, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268256.1268264

[48] M. Swanson, L. Stoller, and J. Carter, "Increasing tlb reach using superpages backed by shadow memory," in *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, July 1998, pp. 204–213.

[49] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based tlb preloading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 117–127. [Online]. Available: http://doi.acm.org/10.1145/339647.339666

[50] G. B. Kandiraju and A. Sivasubramaniam, "Going the distance for tlb prefetching: an application-driven study," in *Proceedings 29th Annual International Symposium on Computer Architecture*, May 2002, pp. 195–206.

[51] A. Bhattacharjee and M. Martonosi, "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, Sept 2009, pp. 29–40.

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

[53] 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.

[54] 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

[55] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee, "Scalable distributed last-level tlbs using low-latency interconnects," in *Proceedings of the 51st International Symposium on Microarchitecture*, ser. MICRO-51. New York, NY, USA: ACM, 2018.

[56] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing tlb reach by exploiting clustering in page translations," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 558–567.

[57] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 258–269.

[58] M. Parasar, A. Bhattacharjee, and T. Krishna, "Seesaw: Using superpages to improve vipt caches," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, vol. 00, Jun 2018, pp. 193–206. [Online]. Available: doi.ieeecomputersociety.org/10.1109/ISCA.2018.00026

[59] A. Bhattacharjee and D. Lustig, *Architectural and Operating System Support for Virtual Memory*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017. [Online]. Available: https://doi.org/10.2200/S00795ED1V01Y201708CAC042

[60] 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.

[61] 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*, 2008, pp. 222–232.

[62] 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.

[63] 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

[64] S. McFarling, "Combining branch predictors," Digital Western Research Laboratory, Tech. Rep. TN-36m, June 1993.

[65] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 436–448, 2017.

[66] 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

[67] B. Widrow and M. Hoff Jr., "Adaptive switching circuits," in *IRE WESCON Convention Record, part 4*, 1960, pp. 96–104.

[68] B. Widrow and M. Lehr, "30 years of adaptive neural networks: Perceptron, MADALINE, and backpropagation," *Proceedings of IEEE*, vol. 78, no. 9, pp. 1415–1442, September 1990.

[69] *The 1st Championship Value Prediction Competition (CVP-1), http://www.microarch.org/cvp1.* International Symposium on Computer Architecture, June 2018.

[70] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[71] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent."

[72] "Integer hash function," https://gist.github.com/badboy/6267743, Mar 2007.

[73] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 3, pp. 280–300, September 2005. [Online]. Available: http://doi.acm.org/10.1145/1089008.1089011

[74] L. A. Bélády, "A Study of Replacement Algorithms for a Virtual-storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[75] S. Albers and S. Arora, "Page replacement for general caching problems." Citeseer.

[76] N. E. Young, "On-line file caching," *Algorithmica*, vol. 33, no. 3, pp. 371–383, 2002.

[77] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu, "Caching is hard—even in the fault model," *Algorithmica*, vol. 63, no. 4, pp. 781–794, 2012.

[78] D. S. Berger, N. Beckmann, and M. Harchol-Balter, "Practical bounds on optimal caching with variable object sizes," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–38, 2018.

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

[80] A. Seznec, "Storage free confidence estimation for the tage branch predictor," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, feb. 2011, pp. 443–454.

[81] 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)*, January 2001, pp. 197–206.

[82] N. Soundararajan, S. Gupta, R. Natarajan, J. Stark, R. Pal, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Towards the adoption of local branch predictors in modern out-of-order superscalar processors," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 519–530. [Online]. Available: http://doi.acm.org/10.1145/3352460.3358315

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

[84] 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.

[85] 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.

[86] 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.

[87] 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. IEEE, 1989, pp. 277–285.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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*, 1996, pp. 72–78.

[95] 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.

[96] 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.

[97] 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*. IEEE, 2002, pp. 131–140.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] 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.

[104] 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: ACM, 2004, pp. 37–45.

[105] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. McKee, "Reevaluating online superpage promotion with hardware support," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, ser. HPCA '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 63–. [Online]. Available: http://dl.acm.org/citation.cfm?id=580550.876428

[106] *Characterizing the d-TLB Behavior of SPEC CPU2000 Benchmarks*, ser. SIGMETRICS '02. New York, NY, USA: ACM, 2002. [Online]. Available: http://doi.acm.org/10.1145/511334.511351

[107] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad, "Reducing tlb and memory overhead using online superpage promotion," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995, pp. 176–187. [Online]. Available: http://doi.acm.org/10.1145/223982.224419

[108] "Transparent huge pages in 2.6.38," http://lwn.net/Articles/423584/, January 2011.

[109] C. H. Park, T. Heo, J. Jeong, and J. Huh, "Hybrid tlb coalescing: Improving tlb translation coverage under diverse fragmented memory allocations," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 444–456.

[110] S. Srikantaiah and M. Kandemir, "Synergistic tlbs for high performance address translation in chip multiprocessors," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010, pp. 313–324.