View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Explorer

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches

Citation for published version:
Faldu, P & Grot, B 2017, Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches. in 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 180-193, 26th International Conference on Parallel Architectures and Compilation Techniques, Portland, United States, 9-13 September. DOI: 10.1109/PACT.2017.32

Digital Object Identifier (DOI):
10.1109/PACT.2017.32

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Jun. 2018

In Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17)

Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches

Priyank Faldu, Boris Grot

Institute for Computing Systems Architecture (ICSA)
School of Informatics, University of Edinburgh
{*priyank.faldu, boris.grot*}*@ed.ac.uk*

***Abstract*— The looming breakdown of Moore's Law and the end of voltage scaling are ushering a new era where neither transistors nor the energy to operate them is free. This calls for a new regime in computer systems, one in which every transistor counts. Caches are essential for processor performance and represent the bulk of modern processor's transistor budget. To get more performance out of the cache hierarchy, future processors will rely on effective cache management policies.**

This paper identifies *variability* in generational behavior of cache blocks as a key challenge for cache management policies that aim to identify dead blocks as early and as accurately as possible to maximize cache efficiency. We show that existing management policies are limited by the metrics they use to identify dead blocks, leading to low coverage and/or low accuracy in the face of variability. In response, we introduce a new metric – Live Distance – that uses the stack distance to learn the temporal reuse characteristics of cache blocks, thus enabling a dead block predictor that is robust to variability in generational behavior. Based on the reuse characteristics of an application's cache blocks, our predictor – Leeway – classifies application's behavior as *streaming-oriented* or *reuse-oriented* and dynamically selects an appropriate cache management policy. By leveraging live distance for LLC management, Leeway outperforms state-of-the-art approaches on single- and multi-core SPEC and manycore CloudSuite workloads.

***Keywords*-variability; last-level cache; dead-block prediction;**

I. Introduction

The microprocessor industry has enjoyed four decades of exponentially growing transistor budgets, enabling complex core microarchitectures, multicore processors, and cache capacities reaching into tens of MBs. The looming reality, however, is that Moore's law is nearing its limits both in terms of physics and economics. Combined with the end of voltage scaling, the semiconductor industry is about to enter a new phase where transistors become a limited resource and a new technology generation cannot be counted on to double them.

Caches are an essential feature of modern processors. As out-of-order cores have reached the limits of complexity scaling due to a combination of wire delays and power density limitations, caches have been instrumental in providing performance gains across processor generations via ever-larger capacities. In the future, however, further increases in cache capacity may become a difficult proposition for reasons above.

Dead Block Predictors (DBPs) have been shown to be effective in improving cache performance through better utilization of existing capacity. These schemes all rely on some metric of temporal reuse to make their decisions regarding the end of a given block's useful life. Previous work has suggested hit count [1], last-touch PC [2], and number of references to the block's set since the last reference [3], among others, as metrics for determining whether the block is dead at a given point in time. By identifying and evicting dead blocks in a timely and accurate manner, these schemes allow other blocks (that have not exhausted their useful life) to persist in the cache and see further hits.

The task of a DBP is complicated by the fact that applications exhibit *variability* in the re-reference patterns of cache blocks touched by them. The sources of variability are numerous, stemming from microarchitectural noise (e.g., speculation), control-flow variation, cache pressure from other threads, etc. The variability manifests itself as an inconsistent behavior of the individual cache blocks from one cache lifetime, or generation, to the next. This inconsistency challenges DBPs in reliably identifying the end of a block's useful lifetime, thus resulting in lower prediction accuracy, coverage, or both.

The thesis of this paper is that DBPs require metrics and policies that can tolerate inconsistencies. To that end, we propose *live distance* – a metric of temporal reuse based on stack distance. For a given cache block, live distance is the largest observed stack distance in a generation (from allocation to eviction). Live distance is an efficient way to represent a block's range of temporal use and, as we argue in Sec. II-C, has a number of useful properties that make it attractive for dead block prediction in the face of variability.

We introduce Leeway, a new DBP that uses live distance as a metric for prediction. Leeway uses code-data correlation to associate live distance for a group of blocks with a PC that brings the block into the cache. While live distance as a metric provides a high degree of resilience to variability, the per-PC live distance values themselves may fluctuate across generations. To correctly train live distance values in the face of fluctuation, we observe that individual applications' cache behavior tends to fall in one of two categories: streaming (most allocated blocks see no hits) and reuse (most allocated blocks see one or more hits). Based on this simple insight,

we design a pair of corresponding policies that steer updates in live distance values either toward zero (for bypassing) or toward the maximum recently-observed value (to maximize reuse). For each application, Leeway picks the best policy dynamically based on the observed cache reuse behavior.

To avoid the need to access specialized external structures (e.g, prediction tables) upon each LLC access, Leeway embeds its prediction metadata (i.e., live distance) directly with cache blocks. This is in contrast with prior predictors [2], [4]–[6], which need to access a dedicated predictor table upon every single LLC access. Because modern multicore processors feature distributed last-level caches, accesses to dedicated prediction tables introduce detrimental latency and energy overheads in traversing the on-chip interconnect to query such structures.

We study cache management policies on traditional singlethread and multiprogrammed SPEC workloads as well as manycore scale-out server workloads, and make the following contributions:

- We propose Live Distance as a metric to track a block's useful lifetime in a cache, which enables dead block prediction with both high coverage *and* high accuracy even in presence of variability.
- We introduce Leeway, an adaptive DBP that leverages Live Distance for predictions. To further increase prediction accuracy and coverage under variability, Leeway deploys novel reuse-aware update policies that steer live distance values to maximize either bypass or reuse opportunities based on application preference.
- We compare Leeway to prior cache management techniques for LLC, demonstrating that Leeway consistently provides good performance that generally matches or exceeds that of state-of-the-art approaches. In the presence of a data prefetcher, Leeway achieves a geomean speed-up of 5.1% (up to 45%) on singlethread applications versus 3% for prior techniques. It also achieves geomean weighted speed-up of 5% (up to 19%) in 100 multiprogrammed mixes versus 3.3% for prior techniques.

II. Motivation

A. Variability in Block Reuse Behavior

DBPs aim to improve cache behavior by identifying dead blocks and discarding them shortly after their last use, thereby providing an opportunity for blocks with long temporal reuse distances to persist. Effectiveness of dead block prediction hinges on stability of application behavior with respect to the metric used for determining whether the block is dead. Naturally, the more consistent the reuse behavior across the block's lifetimes (also called *generations*) in the cache, the more accurate the predictions.

In practice, there are many reasons for why a block's live time may vary across generations, including:

Control flow variation: When the memory reference instruction is predicated on a condition whose behavior varies at runtime, the corresponding cache block might be referenced a different number of times across generations based on the predicate.

Microarchitectural noise: This includes references on a mispredicted control flow path and hits in lower-level caches due to conflicts in higher cache levels.

Shared data: When a block is shared by multiple threads, it might see different reference patterns due to runtime dynamics and scheduler decisions.

Cache pressure: Application behavior may be consistent but due to cache pressure in the presence of co-running applications, a block may be prematurely evicted. As a result, the block would observe fewer references in a prematurely terminated generation than it would otherwise.

Our insight is that the ability of a DBP to tolerate *inconsistency across generations* hinges on the choice of the metric used for making the prediction. Spurred by the observation, we next use a simple taxonomy to understand the space of metrics.

B. Metrics for Dead Block Prediction

Fundamentally, all DBPs require a metric for determining when a block has reached the end of its useful life. Existing metrics can be classified broadly into two categories: *direct* and *indirect*.

Direct metrics: Also known as *event*-based metrics, these relies on monitoring accesses to the block in order to detect the final access based on previously observed behavior. Reference count [1], trace signature of instructions referencing a block [7], and last PC [2] are all examples of direct metrics used by previously proposed DBPs. An advantage of direct metrics is that a block's fate is determined exclusively by accesses to itself, thereby shielding the decision-making mechanism from noise due to accesses to other blocks.

The downside of direct metrics is their inflexibility in the face of *inconsistent* behavior, which we define as any variation from one lifetime of a block to the next. Consider a simple code snippet below, which shows a reference to a cache block holding the variable X, followed by a predicated second reference to X.

```
PC_i: Ld X
. . .
PC_v: Beq cond, SKIP
PC_w: Ld X
SKIP:
```

Assuming that the second reference occurs only a fraction of the time due to the data-dependent nature of the predicate, predictors that rely on direct metrics are faced with three choices: 1) predict the block dead after the first reference, incurring a miss if the predicate resolves to False; 2) predict the block dead after the second reference, which may never

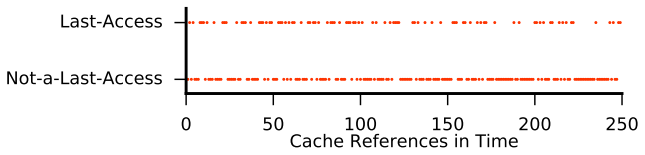

Figure 1: Variability for a PC being the last touch or not in *h264ref*

occur; or 3) not make a prediction. Alas, none of the options are satisfying, as they reduce either coverage or accuracy.

Fig. 1 demonstrates such behavior for the last-PC metric [2] in *h264ref* for a PC responsible for 37% of the misses. The behavior captured in the figure is representative of the entire execution; for clarity, however, the figure shows only a sample of 250 consecutive cache references by that PC (X axis). For each reference, the Y axis shows whether the reference is, indeed, the last access to the block or not under an LRU replacement policy. For the last-PC metric to be useful in identifying dead blocks upon a last access to them, this behavior should be consistent, with all points falling on either the Last-Access (indicating dead blocks) or Not-a-Last-Access (indicating live blocks) line. Meanwhile, the fluctuation shown in the figure indicates that the predictor using last-PC metric may struggle to accurately determine the end of the useful lifetime for blocks touched by this PC.

Indirect metrics: Also known as *age*-based metrics, these rely on an external reference signal to inform the prediction mechanism of the block's age. When the block's age matches the learned value, the block is considered dead, while hits to the block will reset its age. The age can be computed in cycles, number of accesses to the cache [8], or number of accesses to the set [1], [3].

A major advantage of indirect metrics is their inherent ability to tolerate uncertainty in a block's behavior. Coming back to the code snippet above, a carefully chosen age may allow the block to stay in the cache long enough to see the second hit, if any, while ensuring that the block won't greatly overstay its likely useful lifetime.

The drawback of indirect metrics is their imprecision and susceptibility to noise. Because the prediction is made based on events unrelated to the block itself (e.g., the count of all accesses to the block's set), the age used for deciding whether the block is dead must have some tolerance to fluctuation built into it. This tolerance inevitably increases the block's dead time, even for highly predictable blocks, potentially causing the block to stay in the cache long after its last access while waiting for the age to reach the previously learned value.

C. Toward a Better Metric

Stack distance is defined as the number of accesses to unique blocks made since the last reference to a given block [9]. Stack distance provides a useful way to reason about a block's reuse behavior: blocks that have short reuse intervals will have short stack distances, while blocks with long reuse intervals will see larger stack distances over their lifetime in the cache. In practice, a short stack distance means that a block is likely to experience a hit when it is near the top of the LRU stack (i.e., close to the MRU position). Conversely, a long stack distance means that a hit may come near the LRU position, or – if the stack distance exceeds the associativity of a cache – will result in a miss to the block. By predicting dead blocks early, DBPs aim to keep blocks with long stack distances in the cache long enough for them to see a hit.

We make the observation that stack distance can be turned into a powerful metric for dead block prediction. Fig. 2 provides the intuition. The figure shows the observed stack distances for a sample of 250 cache references for all blocks allocated by a single PC which is responsible for highest number of LLC misses in *GemsFDTD*. Blocks that do not see any hits are shown having stack distance of zero. The key take-away is that despite significant variability across references, the stack distance is largely confined to 4.

Based on this insight, we define *live distance* as the maximum observed stack distance during a block's residency in the cache. Live distance is a good indicator of the block's temporal reuse limit, so when the block's position within an LRU stack exceeds its known live distance, the block is unlikely to be referenced and can be predicted dead. To obtain stack distance values, we exploit the fact that LRU-based policies implicitly track stack distances of cache-resident blocks. In true LRU, when a block hits, its current LRU stack position corresponds to its stack distance. For policies that deviate from true LRU, such as multi-bit NRU (see Sec. III-C for details), a block's stack position upon a hit only approximates the true stack distance. Nevertheless, it provides an efficient heuristic to approximate stack distance and, correspondingly, live distance.

Table I demonstrates how stack and live distance is determined for a block `X` for various reference patterns in a 4-way set. In this example, the largest observed stack distance is 3, yielding a live distance of 3 and indicating that `X` can be predicted dead after the reference to `C` in ref pattern #5.

Live distance combines the best properties of both direct and indirect metrics, making it more effective than "pure" approaches. Specifically, to determine if a block is dead, live distance uses an indirect signal, which is the block's place within an LRU chain. This signal is indirect, since the block ages as a result of hits to other blocks within the set. Crucially, however, live distance for a block X is trained only upon hits to X (same as direct metrics), which demarcate the range of the block's temporal reuse within the LRU stack. Because of this combination, live distance can naturally tolerate variability across generations as long as the reuse interval for the block falls within a previously observed range. At the same time, live distance provides an efficient mechanism for rapidly identifying blocks that have

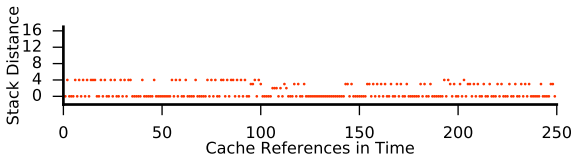

Figure 2: Stack Distances for one PC in *GemsFDTD*

Ref #	Reference pattern	Stack distance	Live distance	Cache event
1	X A X	2	2	Hit
2	X A B X	3	3	Hit
3	X A B B B A X	3	3	Hit
4	X F X	2	3	Hit
5	X A B C F ..	∞ (>4)	3	Miss

Table I: Stack Distance & Live Distance for block X in 4-way set

exceeded their typical reuse window and can therefore be predicted dead.

Compared to other indirect metrics, live distance has an additional attractive property. By relying on stack distance, which only grows as a result of hits to *unique* blocks, live distance provides a degree of dampening to noise resulting from variability in access patterns to recently-accessed blocks. Because most-recently accessed blocks are the ones likely to receive future hits, suppressing variability in these hit counts is beneficial [10]. For instance, consider reference patterns #2 and #3 in Table I. When trying to learn the reuse distance for X, counting the number of accesses to the set between references to X as proposed in prior work [3] produces an inconsistent distance. In contrast, the stack distance for X in both reference patterns is unaffected by variability in the number of accesses to blocks A and B, resulting in a consistent live distance.

III. Leeway Design

We introduce Leeway, a DBP that uses live distance as its underlying metric. We first explain the Leeway basics and features that make it robust against variability in the context of LLC. We then show how Leeway works with a low-cost 2-bit NRU replacement policy. We then discuss microarchitectural details and compare its cost and complexity with prior techniques. Later we extend Leeway to a multicore setup.

A. Overview

The baseline Leeway policy uses a full LRU stack and records the maximum observed hit position (i.e., live distance) during a block's residency in the cache. At eviction time, the live distance is recorded in a separate structure, *Live Distance Predictor Table (LDPT)*, for subsequent recall when the block is allocated again. Leeway uses the live distance learned in the block's previous generations to infer when the block may have exceeded its useful lifetime and predicts it dead. To avoid the prohibitive storage costs of tracking individual cache blocks in the LDPT, Leeway

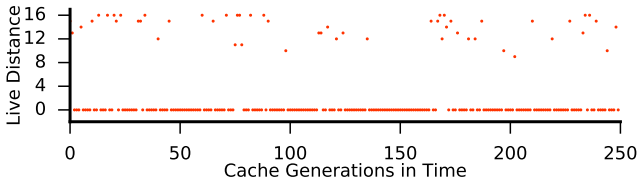

Figure 3: Variability in live distance with a bias of streaming for a PC in *mcf*. A Live Distance of 0 indicates a bypass opportunity.

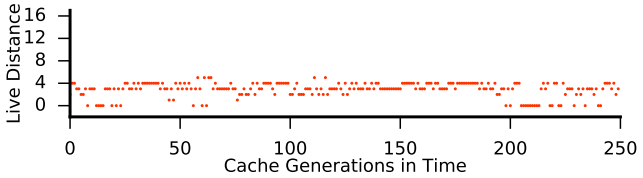

Figure 4: Variability in live distance with a bias of reuse for a PC in *calculix*

exploits code-data correlation and associates all cache blocks allocated by a given PC with one PC-indexed LDPT entry.

The functionality of Leeway can be divided into three categories - *Learning, Prediction* and *Update*. Learning is a continuous process for cache-resident blocks that involves checking a block's position in the LRU stack upon each hit and, if the current position exceeds the past maximum, updating the live distance. Prediction is triggered during victim selection on a miss to a set. Any block that has moved past its predicted live distance in an LRU stack is predicted dead. Update occurs upon a block's eviction from the cache, propagating the new live distance to the LDPT. To effectively handle variability in live distance across generations of a given block and across blocks tracked by a single PC-indexed LDPT entry, the update process is conditional as explained in the next section.

Leeway implements set-sampling, similar to [11], to learn the blocks' live distances by observing their behavior in a small number of sample sets. The reason for sampling is two-fold: 1) it helps filter out some of the noise in observed live distances; 2) it significantly reduces Leeway's storage requirement as only blocks belonging to the sample sets need to be augmented with storage and logic needed for learning.

B. Adapting to Variability

As explained in Sec. II-A, a block's observed reuse behavior may fluctuate in time even if its fundamental reuse characteristics are not changing. While the live distance metric provides a degree of protection from intra-generation noise, Leeway must contend with inevitable fluctuation in live distance across generations and across different blocks allocated by the same PC. In particular, it must separate unrepresentative live distance values from actual shifts in reuse behavior. This observation points to the need for an intelligent update policy for Leeway's live distance values.

To design a variability-tolerant update policy, we study both SPEC and scale-out server workloads (CloudSuite) to understand their reuse behavior. Our workload analysis

reveals that applications tend to fall in one of two categories in terms of their reuse behavior affecting LLC management.

The first category is dominated by streaming accesses that do not observe any LLC hits and should be bypassed. For example, in *mcf*, over 90% of cache blocks are not reused after allocation in LLC under LRU. In many cases, however, we find that blocks allocated by certain streaming PCs will occasionally observe one or more hits. Fig. 3 shows one such PC responsible for 21% of the misses in *mcf*. Moreover, such behavior sometimes occurs in clusters, forcing a shift in cache management policy from bypassing to keeping blocks on chip. Such a shift is generally undesirable, as the behavior tends to quickly revert back to streaming. A multi-bit hysteresis threshold may be effective in delaying a shift in policy; however, the high threshold is counter-productive when the behavior reverts back to streaming as it will lead to blocks being allocated in LLC rather than be bypassed.

The second category of applications is dominated by blocks that do see reuse prior to being evicted from the LLC. For example, in *calculix* more than 60% blocks are reused at least once after their allocation in LLC under LRU. We observe considerable variability in live distance for many PCs that allocate blocks exhibiting reuse. Fig. 4 shows one such PC responsible for 29% of the misses in *calculix*. This observation is consistent with prior work that observed that blocks with reuse are more prone to variability in inter-generational behavior than streaming blocks, thus posing a challenge for DBPs [12]. Given the uncertainty in reuse behavior, such blocks should be kept longer to maximize opportunity for reuse.

The two types of behavior naturally lead to a pair of policies designed to maximize bypass opportunities for streaming workloads and reuse for workloads that exhibit it.

Bypass-Oriented: This policy seeks to maximize opportunities for bypass by being slow to increase the live distance and fast in dropping it back towards 0. An incoming block with a predicted live distance of 0 is bypassed, unless it maps to a sampler set (see Sec. III-D2 for details).

Reuse-Oriented: To maximize reuse opportunities for allocated blocks when there is fluctuation in live distance values, this policy is quick to increase the live distance and slow to decrease it. Since Leeway does not evict blocks that have not reached their live distance value in the LRU or multi-bit NRU stack, a larger live distance enables a longer temporal window for uncovering reuse.

Enabling the policies: The two policies call for diametrically opposite behavior: whereas the Bypass-Oriented policy is slow to increase the live distance values in LDPT but fast to decrease them, the Reuse-Oriented policy is fast to increase live distance values but slow to decrease them. To satisfy the demand for separate policies in increasing and decreasing live distance in the LDPT, Leeway deploys two *Variability Tolerance Thresholds (VTTs)* that control the rate

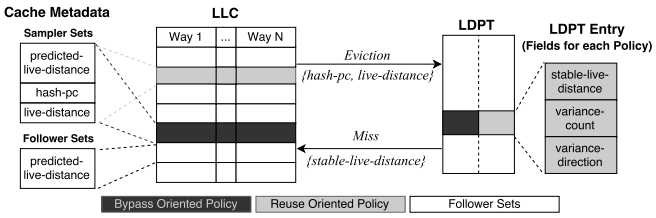

Figure 5: Schematic of Leeway for LLC

at which live distance values are adjusted based on workload behavior and the direction of change in live distance.

In order to choose the preferred policy for a running application, Leeway leverages Set Dueling [11] and implements both policies (Bypass- and Reuse-Oriented) simultaneously on separate sampler sets. The rest of the cache follows the policy that minimizes the misses.

C. Leeway with Cost-Efficient NRU

So far, we have considered Leeway on top of true LRU, which may be unattractive for highly-associative caches. In this section, we explain the minimal modifications required to make Leeway work with a low-cost multi-bit *Not Recently Used (NRU)* family of policies.

Multi-bit NRU uses two or more bits per cache block to indicate a partial relative order of LRU stack positions. For instance, a 2-bit NRU policy keeps blocks in a set in one of four equivalence classes as a function of their relative stack positions, with class 1 for MRU blocks and class 4 for LRU ones. During victim selection, a block in class 4 is evicted (ties are broken through random selection). If no block is found in class 4, every block is moved to the next class and the process is repeated. Both RRIP [13] and SHiP [14] use 2-bit NRU.

Leeway implementation over multi-bit NRU, *Leeway-NRU*, relies on the partial relative order maintained by NRU to make dead block predictions. It uses a block's NRU value to approximate its stack distance, and in turn, live distance. It cannot differentiate between the relative order of blocks in the same recency class.

In general, Leeway can be implemented with any base policy which maintains 1) a partial relative order of blocks based on their relative reference time and 2) a monotonically non-decreasing order for a given block's position between re-references or until eviction.

D. Microarchitecture

1) Physical Fields and Structures: Fig. 5 summarizes key elements of the design.

LDPT: Each PC-indexed LDPT entry contains a *stable-live-distance* field that indicates the current live distance based on most recent history. Updates to stable-live-distance are controlled by VTTs and two additional LDPT fields: 1) *variance-count* is a counter for tracking the number of consecutively evicted cache lines whose live distance differs

from the stored value, and 2) *variance-direction* is a bit indicating the direction of the difference. Once the count matches the value of a VTT for a given direction, the value of *stable-live-distance* is updated. To avoid additional storage for transient live distance values, the new *stable-live-distance* value is taken from the evicted block that triggers the update.

VTTs: To enable Bypass- and Reuse-Oriented policies, Leeway uses a pair of Variability Tolerance Thresholds that control the rate at which *stable-live-distance* values are updated (Sec. III-B). Empirically, we find that a 3-bit VTT is sufficient, and use the maximum value for the slow update (i.e., requiring 7 consecutive evictions with a live distance different, and in the same direction, from the *stable-live-distance*) and a value of 1 for the aggressive threshold. Thus, the two valid VTT configurations are either $\{7,1\}$ (for the Bypass-Oriented policy, with a slow increase and fast decrease) and $\{1,7\}$ (for the Reuse-Oriented policy with a fast increase and slow decrease).

LLC: Leeway requires all LLC blocks to carry a field, *predicted-live-distance*, which is read from the LDPT at block allocation time and is subsequently used for dead block prediction. Sampler sets carry two additional fields: *live-distance* & *hash-pc*. These are used for learning, allowing evicted blocks to index the LDPT and, if necessary, update its fields as explained above.

2) Leeway in Action:

Cache Miss: On an LLC miss, the LDPT is indexed using a hash of the miss PC to recall the *stable-live-distance*, which is then transferred to the incoming block's *predicted-live-distance* field. If *predicted-live-distance* is 0, the block is expected to have no reuse and is bypassed to higher level cache. Since bypassed blocks have no opportunity to retrain, Leeway inserts them into the sampler sets with a small probability (1-3%) to enhance learning.

Cache Hit (Learning): On a hit to a sampler set, the block's live-distance is updated if its stack position is greater than the value of the *live-distance* field. No action for other cases.

Eviction (Prediction and Update): To find victim, Leeway searches for a dead block by comparing each block's LRU or NRU position to its *predicted-live-distance* field. If more than one dead block is found, a victim is picked at random. If no block is predicted dead, the LRU block is evicted. If the evicted block resides in the sampler set, its *live-distance* and *hash-pc* is forwarded to the LDPT for a potential update.

3) Mechanism for Policy Selection: To dynamically choose between Bypass- and Reuse-Oriented policies, Leeway relies on set dueling [11]. Thus, two separate groups of sampler sets are used, with each group implementing one of the two policies. To support simultaneous implementation of policies, the LDPT must be extended to support two sets of {*stable-live-distance, variance-count, variance-direction*} fields per entry. While the sampler sets always access their dedicated fields based on a static mapping, the rest of the sets read the *stable-live-distance* from the winning policy.

To determine the winning policy, Leeway maintains two saturating miss counters, one for each policy. The counters are incremented on a miss and decremented on a hit to a sampler set of a respective policy. Periodically, the miss counters are sampled and the winning policy is selected based on the counter with the lowest value.

Often, the winning policy remains the same throughout the application's execution. In some cases, however, the winning policy may change due to changes in the application's phase or its co-runner(s). In theory, a policy change requires reloading *predicted-live-distance* for all cache blocks using the *stable-live-distance* of the new winning policy in LDPT. In practice, we find that policy change is infrequent, indicating that the simplest way to deal with it is to leave existing blocks untouched, potentially incurring a handful of poor decisions but minimizing microarchitectural complexity.

Technique	Base (KB)	Overhead (KB)	Total (KB)	When is Predictor Table accessed?
SDBP	16	22.75	38.75	Hits + Misses
SHiP	8	9.75	17.75	Misses
Hawkeye	12	19	31	Hits + Misses
Leeway	16	52.5	68.5	Misses
Leeway-NRU	8	36	44	Misses

Table II: Cost for 16-way 2MB LLC and 16K-entry Predictor Table

E. Cost and Complexity Analysis

Cost: We analyze storage requirements for a 16-way 2MB LLC with 64B blocks. We find that a 16K-entry LDPT per core is sufficient and is not affected by destructive aliasing, thus affording a tagless design. For each of two Leeway policies, each LDPT entry has 8 bits: 4 for *stable-live-distance*, 3 for *variance-count* and 1 for *variance-direction*. The resulting cost of LDPT is thus 32KB.

We use a 64-set sampler per policy. Each block in the sampler carries a 4-bit *live-distance* and 14-bit *hash-pc* fields, requiring 4.5KB of storage in total. All cache blocks, including the sampler, include a 4-bit *predicted-live-distance*, totaling 16KB. The total storage overhead of Leeway is thus 52.5 KB, or $\sim$1.8% of the LLC storage. Using 2-bit NRU instead of LRU further reduces the overhead by $\sim$31% to 36KB by lowering live distance storage costs from 4 to 2 bits.

Table II compares the storage requirements of Leeway to those of prior techniques. SHiP [14], an insertion policy, has the lowest storage cost at the expense of not predicting blocks that are reused. Among DBPs that also predict reused blocks, the preferred Leeway NRU configuration requires 44KB of storage in total (including NRU bits), compared to 38.75KB for Hawkeye [5] and 31KB for SDBP [2], considering same number of sample sets and predictor table entries for all techniques. While Leeway is slightly more

expensive, we observe that the storage requirements for all techniques are in a similar range of several tens of KBs. Such modest storage requirements are dwarfed by the size of the LLC.

Complexity: Operations performed by Leeway at various stages are limited to simple addition and comparisons, which are quite hardware friendly. Additionally, Leeway embeds the metadata necessary for the prediction (i.e., *live distance*) with the cache blocks. As a result, LLC hits and replacement decisions never access remote metadata. The only time Leeway accesses its prediction table (LDPT) is upon cache misses, when *stable-live-distance* is read and possibly updated. These accesses are entirely off the critical path, since they do not involve state updates to a live cache block.

In contrast, state-of-the-art DBPs, such as SDBP [2] and Hawkeye [5], use a PC-indexed prediction table that is probed on every LLC access (including hits) to inform the block's eviction priority. For example, Hawkeye incurs $\sim$2.3x more accesses to its prediction table when compared to Leeway (SPEC average). Such frequent accesses to the prediction table are particularly undesirable in a modern multicore CPU with a NUCA LLC, as each LLC hit requires state-of-the-art DBPs to access the PC-indexed prediction table located elsewhere on a chip, incurring latency, energy, and traffic overheads due to the need to traverse the on-chip network.

F. Leeway for Multicore

Leeway can be naturally extended to multicore deployments. The only notable difference is in determining the winning policy for each individual core. When extended to multicore, the sampler sets for a given core, referred to as the *owner core*, are shared with other *follower cores* that will use them as followers of their respective (and potentially different) policies. Because the choice of a policy used by each core affects other cores, we study trade-offs in policy selection through three different strategies – *Greedy*, *Cumulative* and *Democratic*.

The Greedy selection strategy tries to minimize the misses for an application executing on a given core without regard to performance of other applications sharing the cache. The core may select a winning strategy which works better for its own application but may hurt applications on other cores.

In the Cumulative strategy, the cache policy for each core seeks to minimize the *total misses across all applications.* A core may select a policy which may not work best for its own application but reduces overall misses.

Finally, the Democratic selection strategy seeks to benefit the highest *number of applications* regardless of their contribution to total misses. Thus, applications with high number of misses do not single-handedly control the outcome. Voting ties are resolved via the Cumulative strategy.

Core Model	OoO: 4-wide pipeline, 128-entry ROB
L1 Caches	Private, Split, 8-ways 32KB
L2 Cache	Private, Unified, 8-ways 256KB
L3 Cache	Shared, Unified, 16-ways 2MB per core
Memory	200-cycle access latency

Table III: System parameters for SPEC simulations

Core Model	UltraSPARC III ISA, 16 cores OoO: 4-wide pipeline, 128-entry ROB
L1 Caches	Private, Split, 8-ways 32KB
L2 NUCA	Shared, Unified, 16-way, 256KB/core (4MB total)
Interconnect	4x4 2D mesh, 3 cycles/hop
Memory	200-cycle access latency

Table IV: System parameters for CloudSuite simulations

Microarchitectural extensions: For all multicore strategies, LDPT is implemented as a single logical structure which is dynamically shared by all cores. LDPT entry is indexed by a combination of PC and CoreID. Additionally, the various strategies require a set of saturating counters for tracking misses, as follows. Both Greedy and Cumulative strategies require two saturating counters (one each for Bypass- and Reuse-Oriented policies) for each core to count misses in a sampling interval. These counters are updated only by the owner core for the Greedy strategy and by all cores (including follower cores) for the Cumulative strategy. In the Democratic strategy with N cores, each core requires a pair (one each for Bypass- and Reuse-Oriented policies) of N saturating counters, one for each core. These counters are updated by their respective cores exclusively. At the end of a sampling interval, values of these counters reflect how the implementation of both policies by the owner core affects all cores.

IV. Methodology

A. Workloads and Simulation Infrastructure

SPEC CPU 2006: We evaluate the performance of SPEC CPU 2006 benchmarks using a modified version of CMP$im [15] provided with the JILP Cache Replacement Championship [16] and used in prior research in dead block prediction [2], [4], [5], [14]. Table III summarizes the features of the simulated processor.

We use *SimPoint* [17] to identify up to six simpoints of one billion instructions each representing a different phase of a workload. Note that prior work in this space has used only a single simpoint in their evaluation [2], [5], [14]. We use SimPoint tool to generate the weights for each simpoint that are then used to calculate the overall performance. Each program is run with the first *ref* input provided by *runspec* command. For each run, the simpoint is used to warm microarchitectural structures for 500M instructions, then it measures and reports the result for the subsequent one billion instructions. The result reported for each benchmark is the weighted average of the results for the individual simpoints.

For multicore workloads, we use ten workload mixes used in prior work [2] on dead block prediction, which

allows us to compare the effectiveness of various techniques on each mix. To generalize the results, we also evaluate 100 randomly-generated multiprogrammed workloads. For each workload in the mix, we use the highest weighted simpoint. Each mix is run on quad-core system for 1 billion instructions following a warmup of 500 million instructions. Workloads which finish before others are restarted to maintain the cache pressure until the slowest one has finished. We report the weighted speed-up over LRU. To compute it, we run every thread in isolation with 8MB LLC with LRU replacement to calculate $SingleIPC_i$. We then calculate Weighted IPC as $\sum_{i=1}^{N} IPC_i$ / $SingleIPC_i$, where IPC_i is the workload's IPC in the presence of co-runners.

CloudSuite [18] is a collection of contemporary scale-out server workloads, which have been shown to have fundamentally different characteristics than traditional desktop and parallel applications [19]. We evaluate CloudSuite on Flexus [20], a Simics-based, full-system multiprocessor simulator. We model a 16-core CMP with core models loosely based on ARM-Cortex A72 [21]. Table IV lists the system parameters.

For performance evaluation, we use the SimFlex multiprocessor sampling methodology [20], which extends the SMARTS sampling framework [22]. Our samples are collected over 10-30 seconds of workload execution. For each measurement point, we start the cycle-accurate simulation from checkpoints with warmed architectural state and run 100K cycles of cycle-accurate simulation to warm up the queues and the interconnect state, then collect measurements from the subsequent 200K cycles. We use ratio of the number of application instructions to the total number of cycles (including the cycles spent executing operating system code) to measure performance. This metric has been shown to accurately reflect the overall system throughput [20].

B. Evaluated Prediction Schemes

Sampling Dead Block Predictor (SDBP) [2] is a dead block predictor that correlates "last touch" to the block with the PC of the memory instruction making the touch. We use source code from the Championship website for SDBP. We use default settings provided for SPEC workloads except for increasing the number of sampler sets from 32 to 128.

Signature based Hit Predictor (SHiP) [14] is an insertion policy which builds on RRIP [13]. It learns and records whether block is re-referenced after insertion and uses this information to guide insertion placement. We implement SHiP with 2-bit RRIP as a baseline policy and 14-bit PC signature. Each predictor table entry contains a 3-bit saturating counter which is updated by the 128 sampled sets.

Hawkeye [5] learns a block's behavior by simulating Belady's optimal algorithm [23] and trains the predictor that, on each cache access, updates the block's eviction priority. The authors kindly provided the source code of their technique, which we use for evaluation of SPEC workloads.

Leeway: We evaluate Leeway with LRU and NRU. For learning, we use 64 sets per core for each policy. We use set dueling to find the preferred policy (Sec. III-D3). Miss counters for both polices are sampled every 100M cycles. The LDPT has 16K entries for singlecore and 64K shared entries for multicore.

V. Evaluation

A. Singlethreaded SPEC Applications

To better understand the effects of all cache policies, we classify SPEC applications into three categories: 1) *High opportunity*, if performance improves by at least 10% over LRU with any one policy; 2) *No opportunity* if performance doesn't vary by more than 0.5% for all policies; 3) *Mix opportunity* for the rest. We do not show individual results for the No opportunity applications (*gamess, namd, gobmk, dealII, povray, sjeng* and *lbm*) and do not include them in the geomean.

1) Miss Reduction: Fig. 6 (top) shows the reduction in LLC misses compared to the default LRU for High and Mix opportunity applications. All prior techniques are very effective on High opportunity workloads, with miss reduction ranging from 22.2% (SDBP) to 26.9% (SHiP). Leeway is the most effective technique, reducing misses by 29.7%. Leeway achieves the highest miss reduction on five out of six High opportunity workloads coming in second only on *cactusADM* (32% vs 32.7% for Hawkeye). Leeway-NRU is the second-best policy (after Leeway) with an average miss reduction of 28%.

On Mix opportunity workloads, Hawkeye and SHiP are the most effective prior techniques, reducing misses by an average of 7.6% and 10%, respectively. Interestingly, SDBP *increases* average misses by 8.4%, mainly due to extremely poor performance on *calculix*. Leeway and Leeway-NRU outperform prior techniques, averaging 10.6% and 9.8% respective miss reduction. While Leeway increases misses in *gcc*, *zeusmp* and *calculix*, the increase is generally smaller than with prior techniques. For instance, *zeusmp* suffers an increase in misses of 24.8% with SHiP, compared to less than 1% with Leeway. On *calculix*, all predictors increase misses by 29% to 283% as compared to under 11% in Leeway. On both, *omnetpp* and *tonto*, Leeway and Leeway-NRU are able to reduce misses (by 4-9%), whereas SDBP and SHiP increase misses by 1-5%.

We note that on some of the Mix opportunity workloads (e.g., *leslie3d, GemsFDTD*), Leeway's miss reduction trails Hawkeye despite achieving similar prediction coverage and accuracy for both techniques. The reason is that Leeway does not predict blocks dead until they have passed their live distance, which in some cases lead to an increased dead time in the cache. The upside of Leeway's conservative strategy is that it minimizes premature evictions under high variability in reuse behavior. For instance, on *calculix*, all techniques increase misses over LRU. However, Leeway's increase is

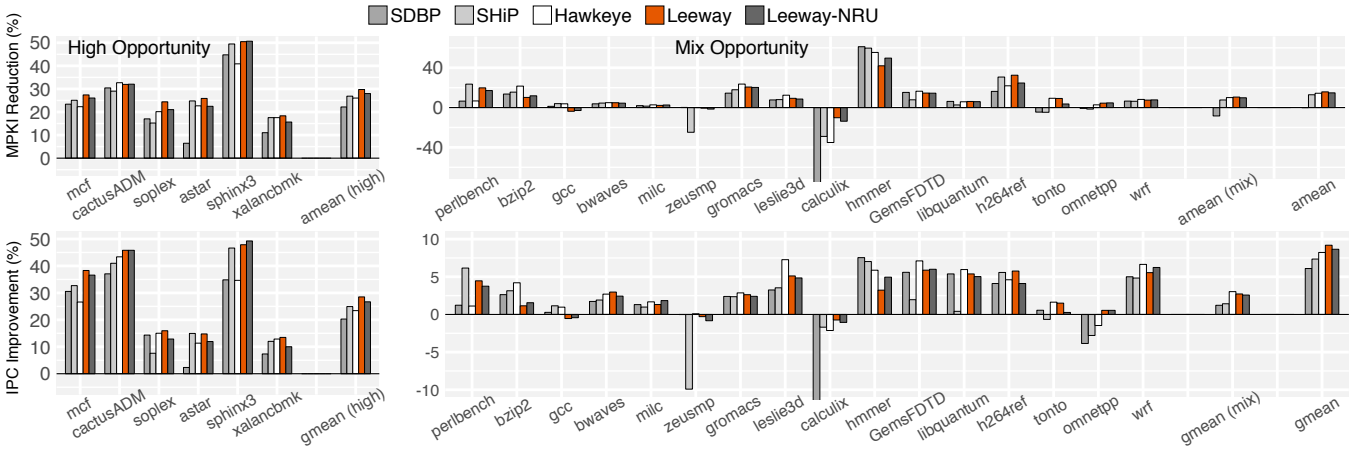

Figure 6: MPKI reduction (top) and IPC improvement (bottom) over unmanaged cache on SPEC applications

the smallest thanks to its higher accuracy of 71.1% vs 50% in Hawkeye and less for others.

2) Performance Improvement: Fig. 6 (bottom) shows the impact of cache management policies on performance with respect to an unmanaged cache. The performance of all techniques generally correlates well with miss reduction.[1]

Performance improvement for all High opportunity applications is substantial. SDBP, Hawkeye & SHiP improve performance by 20.3%, 23.4% & 24.9% respectively. Leeway & Leeway-NRU deliver the highest IPC improvement of 28.5% & 26.7% respectively with Leeway significantly outperforming five out of six applications coming in second only in *astar* (14.8 vs 15% in SHiP). On Mix opportunity workloads performance gains are more modest as expected. Performance improvement is less than 1.4% for SHiP and SDBP. Hawkeye improves performance by 3% as compared to 2.7% & 2.6% for Leeway & Leeway-NRU, respectively. However, Hawkeye's maximum slowdown is 2.1%. In contrast, Leeway and Leeway-NRU limit performance degradation to 0.7% and 1.1% in the worst case, demonstrating their ability to effectively adapt to adversarial scenarios.

Across all of SPEC, Leeway and Leeway-NRU deliver a geomean speed-up of 9.2% and 8.7%, respectively, versus 8.2% and 7.4% for the top performing prior proposals (Hawkeye, SHiP). Both Leeway variants not only outperform prior schemes, but also slowdown the fewest applications.

[1]It should be noted that prior works [2], [5], [14] used just one simpoint per application for their evaluations versus up to *six* in ours. A single simpoint cannot capture different phases in programs whose LLC behavior varies considerably across phases. For instance, Leeway achieves a 17% speed-up over LRU for one simpoint of *xalancbmk* but less than 5% for other simpoints. Based on the weight of each simpoint, normalized improvement is 13.5%. Because of this difference in methodology, performance numbers reported in this paper are different (often, smaller) as compared to those reported elsewhere. For example, authors of Hawkeye report a speed-up of over 25% for *GemsFDTD* using a single-simpoint methodology [5]. Meanwhile, using the same source code provided to us by the authors and the same infrastructure, we achieve a more modest speed-up of 7.1% using the multi-simpoint methodology. Overall, we believe that by evaluating multiple simpoints for each applications, our simulations more closely tracks realistic behavior.

Performance with Prefetcher: We also evaluate all predictors in the presence of a stream prefetcher. By itself, the prefetcher improves the performance of SPEC applications by 44.2%, on average. Improvement over the prefetcher is shown in Fig. 7. In general, prefetching reduces the opportunity for all techniques. Among prior techniques, SHiP provides the maximum improvement with geomean speed-up of 3%. Surprisingly, Hawkeye performs relatively poorly (2% speed-up), in stark contrast to its performance without the prefetcher.[2] Leeway & Leeway-NRU deliver the highest gain of 5.1% and 5%, respectively. On High opportunity workloads, both Leeway variants improve performance by over 14%, compared to 10.2% for SHiP and less for others.

To understand the drop in performance for Hawkeye, we analyzed the prediction coverage and accuracy for all techniques and compared them with the corresponding data without prefetch. We found that, unsurprisingly, accuracy drops for all techniques, including Leeway, by 3-5% when the data prefetcher is enabled. In the case of Hawkeye, coverage also drops significantly (average 69.9% with versus 80.8% without prefetch), indicating that Hawkeye identifies fewer lines as cache averse in the presence of prefetcher-induced variability.

B. Multiprogrammed SPEC

Fig. 8 illustrates weighted IPC improvement for various policies on SPEC application mixes. We use the Democratic selection strategy for Leeway, as we found it to be the best-performing policy achieving weighted speed-up of 15.1% when compared to Greedy(12%) and Cumulative(12.8%). This is expected as both Greedy and Cumulative strategies make decisions that favor individual workloads but may be detrimental to others. In contrast, the Democratic strategy ensures that the majority of applications benefit, which improves both fairness and performance.

Among prior techniques, Hawkeye delivers the highest performance improvement of 13.1%, matching the single-core trend. Leeway & Leeway-NRU outperform all prior

[2]Hawkeye's evaluation [5] did not consider a data prefetcher.

techniques, improving performance by 15.1% & 15.0%, respectively. In general, performance trends across techniques vary for different mixes based on the cache access patterns generated by their complex interactions in the shared LLC. To provide some insight into the trends, we explain performance of Leeway and Hawkeye for two mixes in detail.

In mix1, three applications are sensitive to cache management, out of which Leeway outperforms Hawkeye on two (*mcf* & *hmmer*) whereas Hawkeye outperforms Leeway on *omnetpp*. Leeway's improvement on *mcf* is significant due to a massive reduction in misses (48.5% vs 23.6% for Hawkeye), which drives the weighted IPC in favor of Leeway. Leeway correctly selects the Bypass-Oriented Policy for *mcf*, which reduces pollution by bypassing many streaming blocks directly to higher level cache. Overall, Leeway achieves prediction coverage of 95.6% for the whole mix vs 87.6% for Hawkeye with roughly similar accuracy.

In mix4, two applications are sensitive to cache management and both Hawkeye and Leeway are effective in reducing misses and improving performance when compared to baseline LRU. Whereas Leeway reduces more misses than Hawkeye in *soplex*, Hawkeye reduces more misses in *cactusADM*. Overall, the total number of misses reduced by Leeway is higher than that by Hawkeye; however, the miss reduction in *soplex* translates to a more modest improvement in performance as compared to that on *cactusADM* by Hawkeye. Because of this, Hawkeye outperforms Leeway.

Performance with Prefetcher: In a multicore setup with prefetching, Leeway and Leeway-NRU deliver 7.9% and 7.4% weighted speed-up, compared to 6.1% or less for prior techniques (Fig. 7). We note that the opportunity for cache management increases with multiple cores due to increased pressure on the LLC. We also observe that prefetching and cache pressure due to multiple cores introduce further variability to the LLC's access and eviction stream, which impedes learning. As in the singlecore case, Hawkeye continues to perform poorly under the prefetch induced variability.

To generalize the results, we evaluated these techniques for 100 randomly chosen mixes generated from all 29 SPEC applications (Fig. 9). We observe that the relative trends in the presence of prefetcher hold, with SHiP outperforming Hawkeye and achieving a geomean weighted speed-up of 3.3%. While Hawkeye is generally superior to LRU, it struggles in the face of variability as noted above. Leeway and Leeway-NRU deliver the highest performance of 5% and 4.4%, respectively. Both Leeway variants reduce most overall misses, with 17% and 13.9% reduction, on average, as compared to 12.9% reduction for SHiP and less than 6% for others.

C. Understanding Leeway's Performance

1) Coverage and Accuracy: In this section, we analyze the effectiveness of Leeway and other predictors by means

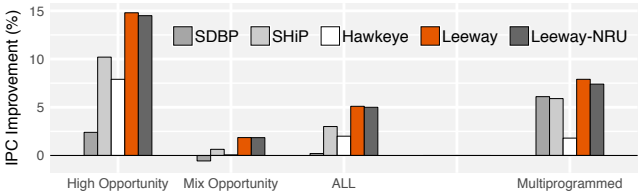

Figure 7: Speed-up with data prefetcher on SPEC applications

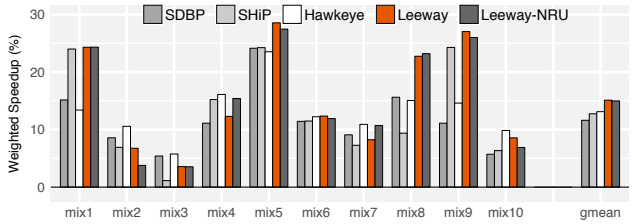

Figure 8: Weighted speed-up on multi-programmed SPEC mixes

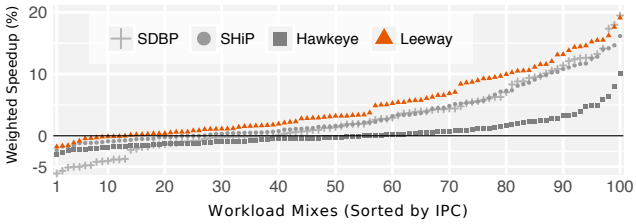

Figure 9: Speed-up with data prefetcher on SPEC mixes

of their prediction coverage and accuracy. The *coverage* of a DBP is defined as the number of dead blocks predicted as a fraction of total evictions, while *accuracy* is the number of correct predictions as a fraction of successful predictions. Fig. 10 shows the standard box-and-whiskers plot for coverage and accuracy for SDBP, SHiP, Hawkeye and Leeway (Leeway-NRU is not shown for brevity). The box is plotted from 75th to 25th percentile (y-axis), forming a core range representing 50% of the distribution. The whiskers are extended up to 1.5 times of this range in each direction. The points not covered by the range of whiskers are plotted individually as circles.

Coverage data shows that SHiP has the lowest coverage overall, which is not surprising as SHiP limits predictions only to the time of insertion. Leeway has higher overall coverage than SDBP and Hawkeye. For instance, Leeway's box, which captures 50% of the distribution, shows coverage ranging from 94.7% to 85.3% versus 92.4% to 73.4% for Hawkeye. The higher coverage in Leeway can be explained as follows: 1) In face of variability, other predictors fall back to not making predictions (Sec. II-B) while Leeway can continue making predictions by adjusting live distance to match the highest observed stack distance; 2) On workloads characterized by low reuse in the LLC, Leeway achieves much higher coverage than prior techniques thanks to its Bypass-Oriented policy that drives live distance quickly towards zero in face of variability.

Accuracy for prior predictors fall in a similar range,

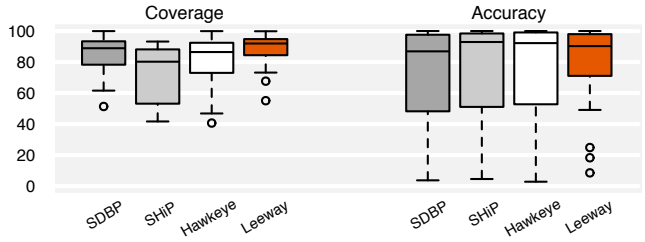

Figure 10: Coverage(left) and Accuracy(right) of various predictors

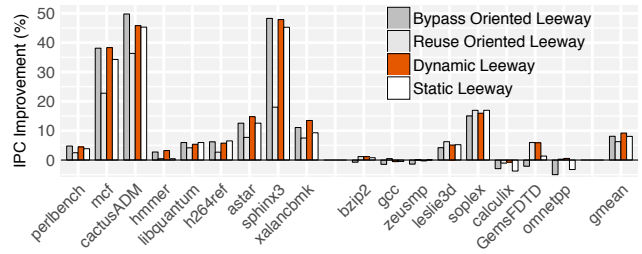

Figure 11: Effect of reuse-aware policies on Leeway's performance

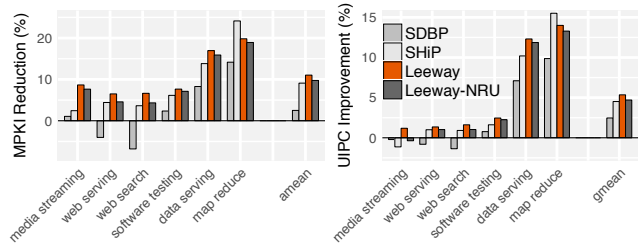

Figure 12: Miss-reduction and Speed-up on CloudSuite workloads

with Hawkeye slightly better than SDBP or SHiP. The box for Hawkeye ranges from 99.1% to 59.8%. In comparison, Leeway has a much tighter accuracy range of 98% to 73.4%. For workloads preferring the Bypass-Oriented policy, Leeway's prediction accuracy is comparable to Hawkeye (recall from above that Leeway's coverage is higher for these workloads). For workloads that prefer the Reuse-Orientated policy, Leeway has superior accuracy stemming from it quickly increasing the live distance under variability, hence maximizing opportunities for additional hits via prolonged live time.

2) Adaptivity in Leeway:

Reuse-Aware Update Policies: To understand the effect of Leeway's policy choice, we compare the performance of individual static policies (Bypass- and Reuse-Oriented) with an adaptive scheme (Dynamic Leeway or simply Leeway) that dynamically chooses a policy at runtime (Sec. III-B). Dynamic Leeway was used throughout the evaluation. Fig. 11 presents the results. The applications whose performance does not depend on the choice of policy are not shown for clarity.

Applications benefiting more from Bypass-Oriented policy are shown on the left group of Fig. 11. Such applications include all high opportunity applications (except *soplex*) and several mixed opportunity ones. For these applications, the Reuse-Oriented policy conservatively increases the live distance in face of variability. However, the access pattern is dominated by bypassable blocks and predicting higher live distance for such blocks only contributes to higher dead time and in turn, lower cache efficiency.

Right side of Fig. 11 shows applications benefiting more from the Reuse-Oriented policy. On most of these applications, none of the techniques are very effective. The culprit is high incidence of blocks with reuse and inter-generational variability. In the case of Leeway, the Reuse-Oriented policy generally proves beneficial by steering the live distance toward the recently-observed maximum in order to boost opportunity for reuse. For instance, this proves particularly beneficial on *omnetpp*, on which Leeway and Leeway-NRU are the only techniques to avoid a slowdown (see Fig. 6).

Finally, Fig. 11 shows that by choosing the preferred policy at runtime, Leeway can effectively adapt to all applications. Moreover, Leeway can adapt to phase behavior within a single application, as demonstrated on *astar* and *xalancbmk*, both of which have distinct cache behavior across phases. On these applications, dynamic Leeway outperforms the best static policy by over 2%.

Reuse-Unaware Static Leeway: To isolate the performance due to only dead block predictions using live distance from the choice of dynamic policy, we evaluate *Static Leeway*, which employs a static VTT value of 7 in both directions and thus does not require set-dueling for policy selection.

Fig. 11 shows the effectiveness of Static Leeway on SPEC applications. Overall, Static Leeway provides a geomean speed-up of 8.1%; however, due to its reuse-unaware design, it underperforms the preferred dynamic policy (Bypass- or Reuse-Oriented) for almost all applications, thus justifying the adaptivity of the reuse-aware Dynamic Leeway design.

D. CloudSuite Applications

Figure 12 compares the performance of both Leeway variants against SDBP and SHiP on scale-out multithreaded CloudSuite workloads. Overall, we find that these workloads have low sensitivity to LLC size, corroborating prior work [19], [24]. While cache management delivers only modest benefits, the relative performance trends across techniques follow those on SPEC. SHiP outperforms SDBP (4.5% vs 2.5%). Leeway & Leeway-NRU outperform both, with a speed-up of 5.4% and 4.7% respectively. To put this result in perspective, doubling the LLC size with the baseline policy improves performance by 7.3%. The largest gains are recorded by Leeway on *data serving* and *map reduce* applications, with a performance improvement of 12.3% and 14%, respectively. For other applications, Leeway reduces misses by 6 to 9%, yielding a modest performance gain of 1.2% to 2.5%.

The reason for the modest speed-ups on these workloads is their low MPKI. While *data serving* and *map reduce* have

Rank	1	2	3	4
SPEC	Leeway	Leeway-NRU	Hawkeye	SHiP
SPEC-MC	Leeway	Leeway-NRU	Hawkeye	SHiP
SPEC(P)	Leeway	Leeway-NRU	SHiP	Hawkeye
SPEC-MC(P)	Leeway	Leeway-NRU	SDBP	SHiP
Cloudsuite(LLC)	Leeway	Leeway-NRU	*SHiP	SDBP

Table V: Top performers for various configurations.
Legends *MC: multicore, P: prefetch*; *Hawkeye not evaluated;

MPKI of 4.8 and 5.0, thus benefitting the most from cache management, others have MPKI in the range of 1.4 to 3.7.

E. Summary

Table V summarizes the effectiveness of various techniques under different setups. Leeway and Leeway-NRU consistently provide the best performance due to robustness stemming from their use of live distance as a metric and a reuse-aware variability-tolerant update policies. Both Leeway variants also enjoy lower implementation complexity, thanks to embedded prediction metadata, as compared to state-of-the-art dead block predictors (Sec. III-E).

VI. Related Work

Duong et. al introduced a DBP based on the notion of Protected Distance (PD) [3]. PD leverages reuse distance, an indirect metric that counts non-unique references to a set. A single PD is used for an entire application. If a block is not referenced beyond the application's PD, it is predicted dead. While conceptually PD sounds similar to Leeway, Leeway has two key advantages over PD. First, PD maintains a single Protected Distance for an entire application, whereas Leeway maintains a Live Distance per PC that is continuously trained throughout the application's execution. This maximizes Leeway's adaptivity while minimizing dead time of blocks prior to prediction. Secondly, Live Distance relies on stack distance, and such naturally "filters" non-unique references to the set. In contrast, PD counts all references to the set, which can inflate PD values and lead to increased dead time for cache blocks. Indeed, our evaluation of PD shows that it is generally inferior to both Leeway and other recent cache management schemes. On SPEC, Leeway's average performance improvement is 9.2% versus 6.2% for PD without the data prefetcher, and 5.1% versus 1.6% (in favor of Leeway) with the prefetcher.

Others have also suggested using stack distance or reuse distance for cache replacement or modeling [3], [25]–[29]. Doing so requires maintaining a Reuse Distance Distribution (RDD) for an application, which itself can be storage intensive as it involves keeping separate counter for different reuse distances maintained. Further, turning this RDD into a useful metric is challenging and computationally intensive. For example, [3] proposes dedicated compute logic while [27] relies on a software framework that runs on a core. In contrast, Leeway monitors the readily-available stack position within a set, which is already maintained by the base replacement policy. Deriving a blocks live distance is then as simple as taking the max of observed stack positions upon hits in its lifetime. Thus, live distance fundamentally enables a very efficient hardware implementation within this general class of metrics.

Teran et. al [6] proposed perceptron learning based predictor for LLC. Instead of correlating cache block behavior with just a single feature like load-PC, it proposes to combine multiple features for predicting block's reuse behavior. To do so, it maintains a separate predictor table for each feature, for a total of six tables. Each of these predictor tables need to be accessed on every cache access (including hits) which makes this design difficult to scale for multicore processors as explained in Sec. III-E.

Contrary to the traditional recency stack, *Pseudo-LIFO* [30] manages the LLC as a fill stack. The approach dynamically learns the preferred eviction positions within the fill stack, and prioritizes the blocks close to the top of the stack for eviction. It learns the preferred positions for an application based on the combined behavior of all the cache blocks, lacking fine-granularity adaptation that state-of-the-art approaches, including Leeway, use.

Finally, Albericio et. al [31] proposed the *reuse cache* that allocates data only for reused lines to reduce the effective capacity of the LLC data array. The design is driven by the observation that most blocks in the LLC are dead and thus not storing data for such blocks will not hurt performance.

VII. Conclusion

In the absence of exponential growth in transistor count, future microprocessors will rely on cache management strategies to improve performance. Complicating the task is variability in reuse characteristics in the applications' working sets. This paper argues for variability-tolerant metrics and strategies for cache management. As a step in that direction, we introduce Leeway, a dead block prediction scheme leveraging live distance - a new metric for capturing temporal reuse behavior using stack distance. By augmenting live distance with a reuse-aware update policies, Leeway achieves good performance across a variety of workloads and deployment scenarios.

Acknowledgment

We thank Daniel Jimnez for providing us with simpoint traces of SPEC'06 applications. We thank the PARSA group at EPFL for providing us with disk images of CloudSuite applications and Onur Koberber & Javier Picorel for their help in setting up these images on Flexus. We thank Akanksha Jain for providing us with the source code of Hawkeye for evaluation. We thank Artemiy Margaritov, Amna Shahab, Arpit Joshi, Cheng-Chieh Huang, Mainak Chaudhuri, Vijay Nagarajan and the anonymous reviewers for their valuable feedback on earlier drafts of this work. Finally, we thank Ron K. Cytron for shepherding this paper.

REFERENCES

[1] M. Kharbutli and Y. Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, April 2008.

[2] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling Dead Block Prediction for Last-Level Caches," in *Proceedings of the International Symposium on Microarchitecture*, December 2010, pp. 175–186.

[3] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving Cache Management Policies Using Dynamic Reuse Distances," in *Proceedings of the International Symposium on Microarchitecture*, December 2012, pp. 389–400.

[4] E. Teran, Y. Tian, Z. Wang, and D. A. Jimenez, "Minimal Disturbance Placement and Promotion," in *Proceedings of the International Symposium on High Performance Computer Architecture*, March 2016, pp. 201–211.

[5] A. Jain and C. Lin, "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement," in *Proceedings of the International Symposium on Computer Architecture*, June 2016, pp. 78–89.

[6] E. Teran, Z. Wang, and D. A. Jimenez, "Perceptron Learning for Reuse Prediction," in *Proceedings of the International Symposium on Microarchitecture*, October 2016, pp. 1–12.

[7] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block Prediction & Dead-block Correlating Prefetchers," in *Proceedings of the International Symposium on Computer Architecture*, June 2001, pp. 144–154.

[8] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior," in *Proceedings of the International Symposium on Computer Architecture*, May 2002, pp. 209–220.

[9] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, June 1970.

[10] 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 International Symposium on Microarchitecture*, November 2008, pp. 222–233.

[11] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A Case for MLP-Aware Cache Replacement," in *Proceedings of the International Symposium on Computer Architecture*, June 2006, pp. 167–178.

[12] P. Faldu and B. Grot, "LLC Dead Block Prediction Considered Not Useful," in *International Workshop on Duplicating, Deconstructing and Debunking*, June 2016.

[13] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP)," in *Proceedings of the International Symposium on Computer Architecture*, June 2010, pp. 60–71.

[14] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "SHiP: Signature-based Hit Predictor for High Performance Caching," in *Proceedings of the International Symposium on Microarchitecture*, December 2011, pp. 430–441.

[15] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator," in *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, June 2008.

[16] A. R. Alameldeen, A. Jaleel, M. Qureshi, and J. Emer, *JILP Workshop on Computer Architecture Competitions: Cache Replacement Championship*, June 2010, http://www.jilp.org/jwac-1.

[17] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for Accurate and Efficient Simulation," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2003, pp. 318–319.

[18] *CloudSuite: The Benchmark Suite of Cloud Services*, Parallel Systems Architecture Lab, EPFL, http://cloudsuite.ch/.

[19] 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 International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2012, pp. 37–48.

[20] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, July 2006.

[21] M. Demler, "Cortex-A72 Takes Big Step Forward," *Microprocessor Report*, February 2015.

[22] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *Proceedings of the International Symposium on Computer Architecture*, June 2003, pp. 84–97.

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

[24] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out Processors," in *Proceedings of the International Symposium on Computer Architecture*, June 2012, pp. 500–511.

[25] R. Sen and D. A. Wood, "Reuse-based Online Models for Caches," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 2013, pp. 279–292.

[26] C. Ding and Y. Zhong, "Predicting Whole-program Locality Through Reuse Distance Analysis," in *Proceedings of the Conference on Programming Language Design and Implementation*, May 2003, pp. 245–257.

[27] N. Beckmann and D. Sanchez, "Modeling Cache Performance Beyond LRU," in *International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 225–236.

[28] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache Replacement Based on Reuse-Distance Prediction," in *International Conference on Computer Design*, October 2007, pp. 245–250.

[29] S. Das, T. M. Aamodt, and W. J. Dally, "Reuse Distance-Based Probabilistic Cache Replacement," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 4, pp. 33:1–33:22, October 2015.

[30] M. Chaudhuri, "Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches," in *Proceedings of the International Symposium on Microarchitecture*, December 2009, pp. 401–412.

[31] J. Albericio, P. Ibanez, V. Vinals, and J. M. Llaberia, "The Reuse Cache: Downsizing the Shared Last-level Cache," in *Proceedings of the International Symposium on Microarchitecture*, December 2013, pp. 310–321.