PRINCIPLED APPROACHES TO LAST-LEVEL CACHE MANAGEMENT

A Dissertation

by

ELVIRA TERAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Daniel A. Jiménez
Committee Members,	Nancy M. Amato
	Eun Jung Kim
	Paul Gratz
Head of Department,	Dilma Da Silva

August 2017

Major Subject: Computer Science

ABSTRACT

Memory is a critical component of all computing systems. It represents a fundamental performance and energy bottleneck. Ideally, memory aspects such as energy cost, performance, and the cost of implementing management techniques would scale together with the size of all different computing systems; unfortunately this is not the case. With the upcoming trends in applications, new memory technologies, etc., scaling becomes a bigger a problem, aggravating the performance bottleneck that memory represents. n A memory hierarchy was proposed to alleviate the problem. Each level in the hierarchy tends to have a decreasing cost per bit, an increased capacity, and a higher access time compared to its previous level. Preferably all data will be stored in the fastest level of memory, unfortunately, faster memory technologies tend to be associated with a higher manufacturing cost, which often limits their capacity. The design challenge is, to determine which is the frequently used data, and store it in the faster levels of memory.

A cache is a small, fast, on-chip chunk of memory. Any data stored in main memory can be stored in the cache. For many programs, a typical behavior is to access data that has been accessed previously. Taking advantage of this behavior, a copy of frequently accessed data is kept in the cache, in order to provide a faster access time next time is requested. Due to capacity constrains, it is likely that all of the frequently reused data can not fit in the cache, because of this, cache management policies decide which data is to be kept in the cache, and which in other levels of the memory hierarchy. Under an efficient cache management policy, a encouraging amount of memory requests will be serviced from a fast on-chip cache.

The disparity in access latency between the last-level cache and main memory motivates the search for efficient cache management policies. There is a great amount of recently proposed work that strives to utilize cache capacity in the most favorable to performance way possible. Related work focus on optimizing the performance of caches focusing on different possible solutions, *e.g.* reduce miss rate, consume less power, reducing storage overhead, reduce access latency, etc.

Our work focus on improving the performance of last-level caches by designing policies based on principles adapted from other areas of interest. In this dissertation, we focus on several aspects of cache management policies, we first introduce a space-efficient placement and promotion policy which goal is to minimize the updates to the replacement policy state on each cache access. We further introduce a mechanism that predicts whether a block in the cache will be reused, it feeds different features from a block to the predictor in order to increase the correlation of a previous access to a future access. We later introduce a technique that tweaks traditional cache indexing, providing fast accesses to a vast majority of requests in the presence of a slow access memory technology such as DRAM.

DEDICATION

A mi madre, mi padre, mi hermana y mi hermano, que por los mil caminos que he tomado en esta vida, en todos me han seguido y apoyado.

ACKNOWLEDGMENTS

I would first like to express my deepest gratitude to my doctoral advisor Dr. Daniel A. Jiménez, not only for convincing me to pursue a graduate degree, but also guiding me through the process, until the very end. All of the work achieved during these years would have not been possible without your help, guidance, and indescribable amount of patience towards me. You believed in me in times when I did not believed in myself, and for that I will always be grateful.

I would like to thank my co-authors, Yingying Tian, Zhe Wang, Jinchun Kim, Dr. Paul Gratz, Chris Wilkerson and Zeshan Chishti, for all their help in the collaboration of several research projects and the knowledge they shared with me.

To my family, I have no words to thank you for all the support you provided me. To the friends I gained through these years at school, thank you for always backing me up when times got tough.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Daniel A. Jiménez, Nancy M. Amato, Eun Jung Kim of the Department of Computer Science and Engineering and Professor Paul Gratz of the Department of Electrical and Computer Engineering.

Chapter 3 and 4 were a collaboration with Yingying Tian and Zhe Wang while they were students at Texas A&M University. Chapter 5 was a collaboration with Chris Wilkerson, Zeshan Chishti and Zhe Wang of Intel Labs.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

This work was funded in part by National Science Foundation grants CCF-1216604/1332598 and CCF-1649242, as well as a gift from Intel Corporation.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

To this day, memory latency continues to be a critical performance bottleneck. For years, processor speeds continued to constantly increase, while memory speeds failed to keep up. In order to alleviate the cost of memory latency, computer architects proposed the idea of a multi-level memory hierarchy, where each level design is chosen by comparing the trade-offs between speed, capacity, manufacturing costs, and others.

1.1 Multi-level Cache Hierarchy

Starting from the memory hierarchy level that is physically the closest to the core, the trend appears to be, to sacrifice capacity for better access latency compared to the next level in the hierarchy. On-chip memory is referred to as caches. A cache can be define as a small fast memory, which goal is, to store only data that is frequently accessed.

The most significant gap in memory speeds between two consecutive levels in the memory hierarchy appears in between the last level cache (LLC) located on-chip and main memory located off-chip. For many years, the rule of thumb was, where this scenario consisted of an SRAM LLC, and DRAM technology for main memory, however, with emerging Non-Volatile Memory (NVM) technologies, architects have proposed adding an on-chip DRAM LLC to the hierarchy, as a cost effective way to replace DRAM as main memory. Each of the different technologies ingrain in different challenges, SRAM is the most access-latency efficient technology, unfortunately it's manufacturing cost, and density makes it unfeasible to satisfy these days huge capacity requirements. DRAM has been the one solution for capacity problems for years, but DRAM access-latency is higher than desired. New trends in technology appeal to be a manufacturing cost-efficient solution for main memory, but the savings come with the price of paying an even higher access latency than that of DRAM. Due to the attractive potential benefits that can be obtained from LLCs, many researches in the field have proposed a wide variety of policies to obtain optimize the performance of LLCs, all with the solely purpose of managing cache content in most favorable to performance way possible [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27]. Despite the amount of efforts in the research, today there is still room to bridge the performance of the state-of-the-art policy and that of an unfeasible optimal technique. The very best performance of a cache will happen when every single memory request can be served from an on-chip cache, and accessing the slow main memory is avoided at all cost. However, in order for this to happen, data would have to be fetch to the cache before the core requests it, and in order to make this possible, knowledge of the future is required, making this approach unfeasible. The goal of architects is to design better management policies such that their performance come as close as possible to the performance of an optimal policy.

1.2 Last-level Cache Management Policies

The effectiveness of the management policy translates directly into an overall better system performance. Because of the limited capacity of the LLC, and the constant growth of memory requests from current applications, very often, it becomes not possible to fit the application's working set in the LLC. Policies must make the decision to evict some of the residing data in the LLC, in order to make room for the new requested data. There are several other management decisions policies cover, in these dissertation, we present several policies that focus on tackling different challenges that are present in LLCs.

1.3 Dissertation Statement

The performance and efficiency of last-level caches can be improved by applying principles of machine learning and other areas of research to intelligent last-level cache management policies.

1.4 Dissertation Organization

In this dissertation we present three novel management policies that are designed to overcome some difficulties that are present in current memory systems, as well as preparing for the challenges that upcoming memory technologies represent. Our proposed work is characterized by the idea of following principles found in machine learning and other areas of interest. We demonstrate that by adapting this principles to cache management, we are able boost performance, save storage overhead, and further reduce access latency.

For the following chapters, in order to put our work into perspective, we will first introduce some of the most relevant related work. Following, each of the proposed mechanism will be introduced, first stating the background and motivation that lead up to this work. Each technique will then be introduced in detail, followed by articulating the methodology and results we obtained. In Chapter 3, our Minimal Disturbance Placement and Promotion policy is introduced, following the same intuition as that of the minimal disturbance principle found in neural networks, we were able to optimize the compelling storage overhead saving tree-based PseudoLRU technique, yielding the same performance as the state-of-the-art technique, while requiring a fraction of the storage overhead. Chapter 4 proposes an accurate reuse predictor policy, it is able to highly accurately predict whether a block in the LLC will be reused again before it is evicted. The predictor is inspired by neural networks, and trained following the perceptron learning rule. It provides the predictor with an extra kick of accuracy and a reduces the training period for application with changing behavior phases. Chapter 5 is used to describe a policy that provides affordable access-latency in the presence of a set-associative DRAM cache, with essentially the high hit-rate associativity delivers. Finally, Chapter 6 gives a conclusion of the work done and presented in this dissertation.

2. RELATED WORK

Our work builds on the contributions of many other researchers to the field. To place our work in the context of other research, we now review some of the most recent related work.

2.1 Placement and Promotion Decision

Each reference block has a last reference during the execution of an application, for several of the blocks the first reference might be the last reference. The placement and promotion of a block should be accordingly to their reuse.

2.1.1 Dynamic Placement

The Least Recently Used (LRU) policy places blocks into the Most Recently Used (MRU) position. Qureshi *et al.* observe that changing the placement position to LRU sometimes results in an improvement as blocks that are no re-referenced after being brought to the cache, are quickly eliminated avoiding these blocks to pollute the cache [28]. The authors of this work propose dynamic insertion policy (DIP) that determines at run-time whether LRU or MRU placement works best for a particular workload. Jaleel *et al.* propose a thread-aware version of DIP, TADIP [13] that improves over DIP as well as previous thread-aware cache management policies. Khan and Jiménez propose decision tree analysis for choosing from a wider range of placement positions [14].

2.1.2 Placement and Promotion Vectors

Genetic Insertion and Promotion for PseudoLRU Replacement (GIPPR) uses genetic algorithms to find *insertion and placement vectors* (IPVs) for tree-based PseudoLRU [22]. The vectors assign the placement position for new blocks as well as the position to which a block in a given original position should be promoted. A single vector does not generalize well to different workloads, so in this work the authors propose to supply the policy with four vectors from which the best performing one is chosen at run-time using set-dueling. GIPPR begins with a blank slate on which the trace-driven genetic algorithm writes a novel placement and promotion drawn from an enormous search space. GIPPR work attempts to find the most favorable positions for blocks to be placed and promoted to.

2.2 Reuse Prediction

Much recent work on last-level cache replacement has focused on predicting whether and when a block will be re-referenced.

2.2.1 Re-reference Interval Prediction

Re-reference Interval Prediction (RRIP) categorizes blocks as near-immediate re-reference interval, intermediate re-reference intervals, and distant re-reference interval [12], associating these positions with cache blocks in a way roughly analogous to the positions in other replacement policies. Static RRIP (SRRIP) always places into the same position (*e.g.* distant re-reference interval) while Dynamic RRIP (DRRIP) uses set-dueling [29] to adapt the placement position to the particular workload. RRIP is a simple policy, requiring little overhead (2 or 3 bits per block) and no complex prediction structures, while resulting in significant improvement in performance. The intuition behind this policy is to allow incoming blocks to the cache to remain just the enough amount of time to be re-referenced again and promoted to a more favorable position, or in the opposite case, to be evicted quickly and allow the allocation of other blocks.

2.2.2 Dead Block Prediction

Dead block predictors predict whether a block will be used again before it is evicted. There are numerous dead block predictors applied to a variety of applications in the literature [30, 31, 32, 33, 34, 35, 6, 16]. In some of our work, we take inspiration from the

Sampling Dead Block Prediction (SDBP) of Khan *et al.* [16]. In this work, a *sampler* keeps partial tags of sampled sets. Three tables of two-bit saturating counters are accessed using a technique similar to a skewed branch predictor [36]. For each hit in the sampled set, the program counter (PC) of the relevant memory instruction is hashed into the three tables and the corresponding counters are decremented. For each eviction from a sampled set, the counters corresponding to the PC of the last instruction to access the victim block are incremented. For any access to the LLC, the predictor may be consulted by hashing the PC of the memory access instruction into the three tables and taking the sum of the indexed counters. When the sum exceeds some threshold, the accessed block is predicted to be dead. Tags in sampled sets are managed with true LRU and a reduced associativity, but the LLC may be managed by any policy. The paper applies dead block prediction to replacement and bypass, but in this work we use the predictor for determining placement position and bypass.

2.2.3 Signature-Based Hit Prediction

Wu *et al.* [37] approach re-reference prediction from the opposite perspective of dead block prediction: rather than predict whether a block is dead, their signature-based hit predictor (SHiP) predicts the likelihood that a block will be hit again. This technique uses a table of 3-bit saturating counters indexed by a signature of the block, *e.g.* the memory instruction PC that caused the initial fill of the block. When a block with a given signature is hit, the counter associated with that signature is incremented. When a block with a given signature is evicted, the counter is decremented. The hit predictor is used to determine placement position for RRIP. When the counter corresponding to the signature of an incoming block exceeds some threshold, the block is predicted to have a "distant re-reference interval," otherwise it is predicted to have an "intermediate re-reference interval." That work proposes an efficient sampling-based predictor where a small subset of cache sets keep metadata such as signatures, and the predictor generalizes to the whole cache.

2.3 Machine Learning for Cache Management

2.3.1 Perceptrons in Microarchitecture

Perceptron learning was proposed for predicting conditional branches [38]. In this work, a perceptron weights vector is selected by a hash of the branch address and its dot product with an input vector of branch history outcomes. If the product is at least 0, the branch is predicted as taken, otherwise it is predicted not taken. The weights are updated with the perceptron update rule: if the prediction is incorrect, or the dot product does not exceed some threshold magnitude, then the weights are incremented or decremented based on their correlation to the corresponding history bits.

Subsequent work on predicting branches with perceptron-like algorithms improved the latency of the operation by using history bits to generate indices into weights tables rather than as input to the dot product computation [39, 40, 41]. In some of our work we take this approach as well, using input features to index weights tables and updating the weights with perceptron learning.

2.4 DRAM Last-Level Caches

With the increasing demands for memory capacity, on-chip stacked DRAM caches have been proposed, as the new last level cache that comes right before main memory, in order to improve the hierarchy performance [42, 43, 44, 45]. DRAM technology represent challenges that are often not an issue with SRAM, some proposed optimizations include: reducing the overhead of supporting tags, improving hit rate, and optimizing access latency, etc.

2.4.1 Tag Optimizations for DRAM LLCs

Because of the large magnitude of DRAM caches, storing each corresponding tag from each of the cache blocks.

The Loh-Hill cache [47] was proposed to co-locate the tags and data in the same DRAM cache row. The request will read the tag first, then determine in which column the data resides and access it all with a row buffer hit access timing. Although the Loh-Hill cache reduces access latency compared to a naive scheme that places the tag and the data into different rows, accessing the tag still adds significant access latency to the critical path. The DRAM cache data may be accessed only after obtaining the tag and finding out the location of the data. Huang and Nagarajan[26] propose prefetching tags into a small on-chip SRAM tag cache. Requests with spatial locality are able to read tags from the fast on-chip tag cache. However, incorrectly prefetched tags to the tag cache incur in a significant bandwidth overhead. The Buffered Way Predictor (BWP) technique [48] has been proposed to predict the way number of the cache set where the data being requested is located. The data can be fetched from the DRAM cache according to the predicted way number, avoiding having to read in all tag from the set first. However, the key drawback of BWP is that it requires 6% of the SRAM cache capacity to store the way number of the DRAM cache data.

2.4.2 Access Latency Optimization for Direct-Map DRAM Caches

Qureshi and Loh[46] propose the Alloy Cache, a direct-mapped DRAM cache design. This work explores the trade-off between access latency and miss rate for DRAM caches. The Alloy cache proposed to prioritize access latency over miss rate for a DRAM cache. The technique co-locates tags and data in the same DRAM cache block so the tag and data can be streamed out in two back-to-back data bursts. It incorporates a Memory Access Predictor (MAP) to predict whether a request will hit in the DRAM cache or not. The prediction is used to guide whether to do a parallel or sequential access between the DRAM cache and main memory. The Alloy Cache sacrifices DRAM cache hit rate for a lower access latency, however, for workloads that significantly benefit from a set-associative cache design, the Alloy cache might degrade their performance.

3. MINIMAL DISTURBANCE PLACEMENT AND PROMOTION [*]

In this chapter, we introduce our work Minimal Disturbance Placement and Promotion (MDPP), a space-efficient replacement algorithm. MDPP builds upon a tree-based PseudoLRU technique, and guides the placement and promotion decisions following the *minimal disturbance* principle. MDPP is presented in a static and dynamic version. Static MDPP can match the state-of-the-art policies' performance, while dynamic MDPP outperforms previously proposed work.

3.1 Introduction

With the disparity between cache latency and memory latency, a replacement policy that reduces misses can significantly improve performance. Many replacement policies order blocks into distinct positions, choosing to replace the block in a least favorable position when there is a miss to a set. Positions are manipulated by the placement and promotion policies. The placement policy decides what the initial position of a block should be, while the promotion policy decides what the new position of a re-referenced block should be. To accommodate a block in a new position, often other blocks must change their positions. These changes can disturb non-referenced blocks, causing them to move closer to the least favorable position by an amount that may be out of proportion to their merit. Previous work focuses on what happens to the just-referenced block without regard to the the effect on the other, non-referenced blocks.

Figure 3.1 shows the average change in position of non-referenced blocks for two common replacement policies: least-recently-used (LRU) replacement policy as well as tree-based PseudoLRU over a set of 100 multi-core workloads described in Section 3.4.

[*]2016 IEEE. Reprinted, with permission, from E. Teran, Y.Tian, Z. Wang and D. Jiménez, "Minimal disturbance placement and promotion", April 2016

An evicted block travels one position beyond the maximum value, *i.e.* it moves out of the set. The LRU policy is relatively well-behaved, moving non-referenced blocks no more than one position and often less than one position when a block is promoted or placed. Tree-based PseudoLRU, a widely-used space efficient policy, moves blocks' positions much further than LRU, and the movement varies wildly with the non-referenced block's original position.

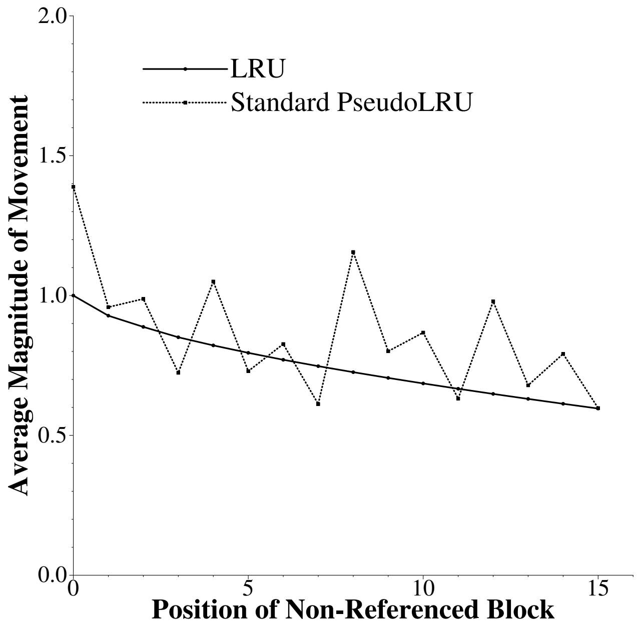


Figure 3.1: Average change in position caused by a promotion for the non-promoted blocks

In this work, we develop a replacement policy that reserves promotion privileges to blocks with an actual risk of being evicted, we found that promoting blocks that hold higher positions in the recency stack introduce unnecessary changes to all blocks' ordering, something we would like to avoid.

Widrow and Lehr introduced the *minimal disturbance principle* applied to neural network learning that says: "Adapt to reduce the output error for the current training pattern, with minimal disturbance to responses already learned". Guided by this principle, we present a technique that only promotes referenced blocks when unprotected, meaning they are at risk of being evicted, it does not promote blocks protected from eviction, doing this we attempt to introduce the minimal disturbance to blocks in the set.

We apply this approach to tree-based PseudoLRU. The results are a simple static policy that significantly outperforms true LRU with less complexity, and a dynamic policy that outperforms the state-of-the-art on multi-core workloads. The dynamic MDPP Cache achieves a normalized weighted speedup of 11.0% on multi-core workloads over LRU while SHiP [37] achieves an 9.1% speedup.

This chapter describes the following contributions:

1. We show that tree-based PseudoLRU is a disturbing policy. That is, it causes changes to the positions of non-referenced blocks that are out of proportion to those blocks' likelihood to be referenced again.

2. We introduce a static minimal disturbance placement and promotion policy for PseudoLRU. We describe the policy and show that it outperforms other static policies for single-threaded workloads and is competitive with other static policies for multi-core workloads. Because it builds on the space-efficient PseudoLRU policy, the MDPP has lower space overhead than previous recent work on replacement.

3. We introduce a minimal disturbance dynamic placement and promotion policy that adapts the placement policy to changing program behavior using dead block prediction resulting in dynamic MDPP. We show that this policy matches the performance of the state-of-the-art replacement policy single-threaded workloads with a

speedup of 5.9% over LRU, and exceeds the state-of-the-art performance for multi-core workloads with a normalized weighted speedup of 11.0% over LRU.

3.2 Motivation

3.2.1 Policy Based on Binary Trees

PseudoLRU policies are based on providing an effect similar to the least-recently-used (LRU) policy, but with a more efficient representation. LRU orders blocks in a recency stack from most-recently-used (MRU) to least-recently-used, placing new blocks into the MRU position and evicting blocks from the LRU position. As blocks are reused, they are moved to the MRU position with all blocks previously ahead of the reused block shifted down the recency stack. At any point in the algorithm, each block has a distinct position in the recency stack. For an n-way set-associative cache, the positions are numbered 0 (MRU) through $n-1$ (LRU).

The tree-based PseudoLRU replacement policy uses a binary tree to prioritize cache blocks [49]. The leaves of the tree are the indices of blocks (*i.e.* ways) in a set. The internal nodes are labeled with bits: 0 or 1. The internal nodes are represented as an array of bits with an implicit tree structure. There are $n-1$ internal nodes in a binary tree with n leaves. Thus, tree-based PseudoLRU is space efficient since it requires only $n-1$ bits per set compared with $n \log_2 n$ bits per set for LRU.

PseudoLRU Replacement

On replacement, the victim block is chosen by tracing a path from the root of the tree to a leaf block. At each internal node, if the label is one, then the next node on the path is the right child of the current node; otherwise, it is the left child.

PseudoLRU Placement and Promotion

When a block is placed or re-referenced, it is promoted. The bits along the path from the root to the associated leaf are modified such that the replacement algorithm would

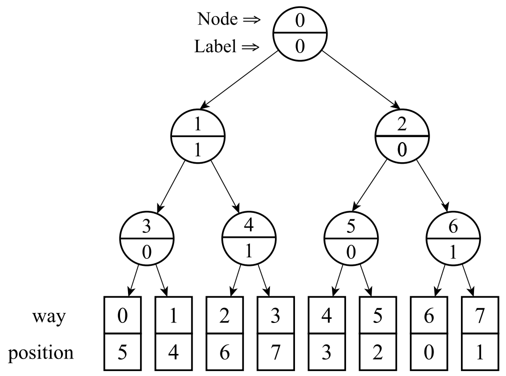


Figure 3.2: The nodes in an example PseudoLRU tree arranged in ascending order of node number showing the labels guiding replacement policy

choose to go in the opposite direction from the leaf. That is, for all the nodes along a path from root to leaf, if the leaf is in the left subtree of the current node, then the bit in that node is changed to 1, otherwise the bit is changed to 0. Thus, among all blocks in the set, the promoted block receives the most protection from being evicted.

Protection

We can formalize the notion of protection in a PseudoLRU tree: a leaf is *protected* at tree level l if the l^{th} node on a path from root to leaf is labeled 0 when the leaf is in the right subtree of that node, or labeled 1 when the leaf is in the left subtree of that node; otherwise, the leaf is *unprotected*. A promoted block is protected at all levels, while the block chosen as the victim is unprotected at all levels. We also define the *protected subtree* of a node to be the right subtree if the node is labeled 0, or the left subtree if the node is labeled 1.

PseudoLRU Ordering

The leaves of the PseudoLRU tree may be ordered from 0 to $n - 1$ in much the same way as the positions in the LRU recency stack [22]. The most protected leaf occupies position 0, *i.e.* the "PseudoMRU" block. The most unprotected leaf occupies position

Old Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
New Position	0	1	2	3	4	5	6	7	0	1	2	3	0	1	0	0

Table 3.1: Minimal Disturbance promotion for 16-way associativity.

$n-1$, *i.e.* it is the PseudoLRU block. The other blocks occupy positions in between. For example, the sibling of the PseudoMRU block occupies position 1, and the first cousins of the PseudoMRU block are in positions 2 and 3. Let us think of the position of a leaf as a binary integer whose least significant bit is associated with the leaf. At each level of the tree, the relevant bit in the position is 1 if the leaf is unprotected at that level, or 0 if it is protected at that level.

One way to see the ordering is by considering a simple transformation on the binary tree: for each internal node labeled with 0, swap the left and right subtrees and change the label to 1. This procedure produces a canonical form in which the leaf nodes are ordered from left to right in ascending recency position: the rightmost leaf is the PseudoLRU way and the leftmost leaf is the PseudoMRU way. For example, Figure 3.2 shows an example 8-way associative PseudoLRU tree representing internal nodes numbers 0 through 6 labeled with randomly chosen binary values. After the transformation, nodes 0, 2, 3, and 5 would have their left and right subtrees swapped, placing the leaves into ascending order of position.

Promoting a block has the effect of moving other blocks into less fortunate positions, sometimes dramatically so. For instance, promoting a block such that the bit in the root node is toggled may move the previous PseudoLRU block down by $n/2$ positions.

Only one block at a time is in the PseudoLRU position, but the position of each block is roughly proportional to its vulnerability to being moved to the PseudoLRU position in the near future.

3.2.2 PseudoLRU Disturbs Non-Promoted Blocks

Let us consider the placement/promotion algorithm for tree-based PseudoLRU. At each level in the tree, the algorithm alters a bit to protect the referenced block without regard for the disturbance caused to other blocks. We consider the disturbance to a block to be the number of positions it is moved as a result of an placement or promotion to another block. Disturbance typically can be defined as the act of interfering with the arrangement, order, or harmony of, in this case the promotion of referenced blocks alters the order of non-referenced blocks in the set , that can lead to consequences such as making non-referenced blocks become vulnerable to eviction.

PseudoLRU is a disturbing policy because it changes in the positions of the non-referenced blocks out of proportion to the analogous changes in standard LRU. Figure 3.1 shows the average magnitude of the change in position (the y-axis) for non-referenced blocks when a block at a given position (the x-axis) is placed or promoted for LRU and PseudoLRU. The average is for a 16-way set-associative 16MB cache taken over the 100 8-core workloads described in Section 3.4. For LRU, the maximum change in the position of a non-referenced block is 1 when the formerly MRU block is not referenced, and goes down to 0.6 for the LRU block. For tree-based PseudoLRU, the non-referenced blocks' positions are changed much more. The formerly PseudoMRU block travels on average two positions away, the block in position 12 moves 1.39 positions away, and at most block positions the distance traveled is significantly higher than the corresponding position in true LRU.

Thus, tree-based PseudoLRU is more disturbing to blocks' positions than true LRU, so the blocks' PseudoLRU recency positions are inaccurate reflections of their true recency positions in true LRU. That is, blocks that might have locality may needlessly be moved too close to LRU, exposing them to the risk of eviction. To the extent that imitating true

LRU is a worthy goal, PseudoLRU somewhat fails. This is not really surprising as we expect to give up some performance when gaining a more space-efficient policy. However, what is surprising is that *this level of disturbance is totally unnecessary*, and removing it results in a replacement policy superior to LRU.

3.3 Design

We present a different placement and promotion policy for tree-based PseudoLRU: place and promote blocks to positions that cause the minimal amount of disturbance among all blocks. The positions of blocks in the tree can be directly manipulated with simple algorithms [22], but we must still choose what the new positions should be to minimize disturbance.

3.3.1 Minimal Disturbance Placement

When an incoming block is placed, we must balance the potential disturbance caused to existing blocks with the potential risk of eviction to the placed block if it is left completely unprotected from eviction. What is the choice of placement position to achieve minimal disturbance?

From this, we decide that the logical choice of placement is the one that affords the most protection to the incoming block while still allowing for a majority of the existing blocks to remain in the same position, while disturbing a minority of blocks .

Based on this premise, for an n-way set-associative cache, a MDPP Cache would place incoming blocks in position $3n/4$. Since the previous position of a newly evicted block was $n-1$, placing the block into position $3n/4$ insures that the labels of the root node and the second-level nodes remain the same so there are no wide swings in the positions of the $3/4$ of blocks closest to PseudoMRU. Changing bits any closer to the root risks affecting the positions of a majority of other blocks. The newly placed block remains unprotected at the root level and second level, but fully protected beyond that so it has a good opportunity to be referenced and thus promoted.

3.3.2 Minimal Disturbance Promotion

Table 3.1 gives the old and new positions for promotions for 16-way associativity. Let us recursively subdivide the PseudoLRU tree into $1 + \log_2 n$ regions of greater and lesser protection, starting with the protected subtree of the root, then the protected subtree of the unprotected subtree of the root, and so on, ending with singleton subtrees at positions $n-2$ and $n-1$. Figure 3.3 illustrates these regions of protection. When a block *B* is promoted, a minimal number of bits in the tree are changed such that the first block in the smallest unprotected region that contains *B* is moved to the MRU position. As a result, *B* is moved along with the first block to a position relatively close to MRU. This strategy gives due protection to the promoted block while minimizing movements among the other blocks. Thus, for $n = 16$, *i.e.* 16-way associativity, the blocks in positions 0 through 7 remain in the same positions because they are already in the protected subtree. Position 8 moves to position 0, position 9 moves to position 1, and so forth as in Table 3.1.

Note that implementing our modified promotion and placement policies incurs virtually no extra energy or area penalty; the replacement status vector is updated using the same sort of logic as tree-based PseudoLRU, but with a different set of state transitions. Indeed, dynamic energy is reduced because for a great many accesses no update needs to be made to the status vector.

Effect of Minimal Disturbance on Block Positions

Figure 3.4 shows the effect of minimal disturbance placement and promotion on the average change of non-referenced blocks. The magnitude of changes in position is far less than the average for PseudoLRU or even true LRU. The block in position 0 moves the most, an average of 0.79 positions. Movements of subsequent blocks steadily decrease until block 8, which moves an average of 0.40 positions. The block in the PseudoLRU

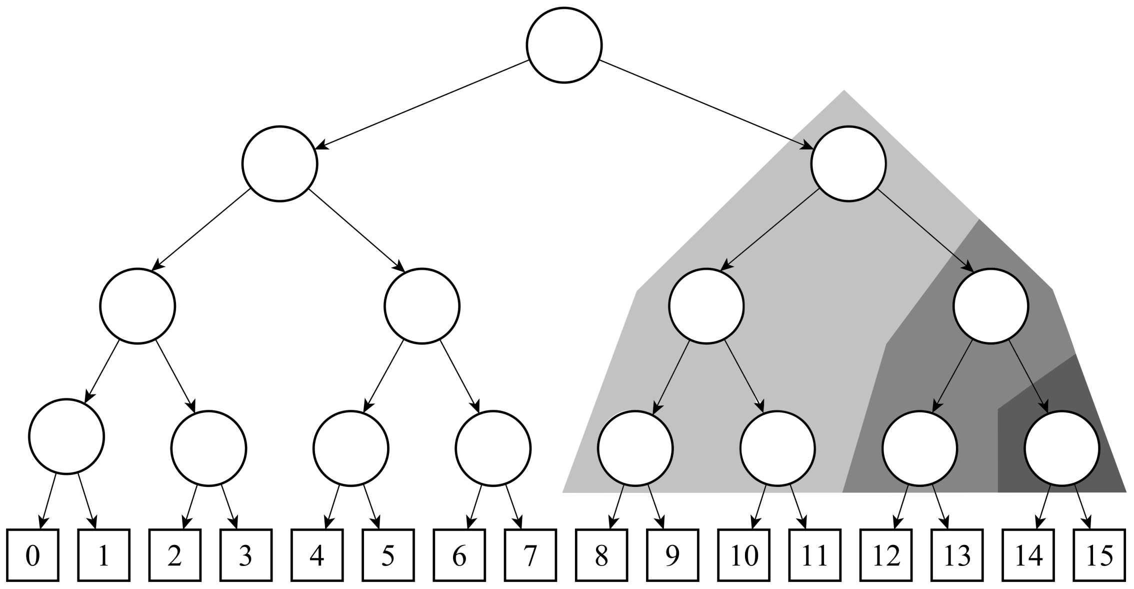


Figure 3.3: Regions of Protection in the PseudoLRU Tree

position moves an average of 0.27 positions. We can quantify the "total disturbance" of a placement and promotion policy as the area under the curve in Figure 3.4. Integrating the curves for the three replacement policies, we see that the total disturbance caused by LRU is 12.1, by PseudoLRU is 13.8, and by minimal disturbance PseudoLRU is 5.2.

Quantifying disturbance for RRIP is problematic for two reasons: 1) block positions are not distinct, as in LRU and PLRU; and 2) promotion of a block in RRIP is decoupled from changes in other block positions; blocks are only moved away from MRU on evictions so RRIP is not directly comparable with policies where promotions necessarily disturb some non-promoted blocks.

Disturbance to the Evicted Block

Figure 3.4 assumes that the disturbance of moving from one position to the next is constant. However, evicting a block, *i.e.* moving it from LRU to a position out of the cache, incurs far more disturbance to a block than simply moving it to another position. Thus, for

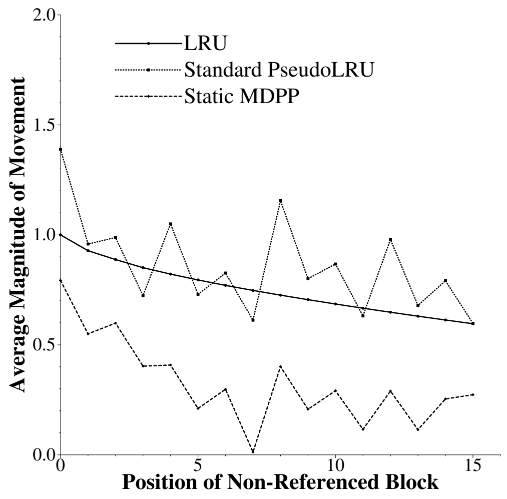


Figure 3.4: Average change in recency stack position for LRU, PseudoLRU, and MDPP Cache

example, an absurd policy of inserting only into the LRU position and never promoting any blocks would have a very high miss rate but relatively cause minimal disturbance. A more nuanced formulation of the disturbance metric would assign a larger distance to evicting a block than to simply changing its position. This distance would take into account latency to satisfy a cache miss as well as the probability that the block will be referenced again. There is no general way to know that probability *a priori* but, as we will see in Section 3.3.3 we can estimate it dynamically with dead block prediction.

3.3.3 Dynamic MDPP

The minimal disturbance MDPP Cache policy as presented so far is a *static* policy. That is, the policy remains the same regardless of the run-time features of the workload. However, a great deal of recent work has focused on *dynamic* policies that change in response to workload characteristics. Thus, we extend the minimal disturbance idea to a dynamic policy. In particular, we allow the placement position to be adapted through dead block prediction.

Some recent dynamic policies adapt the placement position using techniques such as prediction [37] and set-dueling [28, 12]. Other policies implement selective *bypass*, *i.e.* when an incoming block is predicted to not be used before it is evicted, it is not placed in the cache but rather bypassed to the core [16, 19]. Both of these ideas are compatible with the minimal disturbance idea of causing as little disturbance as possible. Clearly, selective bypass works along provoking minimal disturbance: a block that will not be re-referenced will only pollute the cache and confuse the replacement policy. Such a block cannot be disturbed by bypass because it is already a dead block. Adapting the placement position can also achieve the minimal disturbance for PseudoLRU caches as long as the placement position is beyond the halfway point in the PseudoLRU recency stack so as not to make wide swings in block positions.

We use the sampling-based dead block prediction algorithm (SDBP) [16] to drive block placement and bypass. A dead block predictor predicts whether a block that has been placed or referenced will be referenced again before it is evicted. The SDBP paper uses the program counter (PC) of a memory instruction to index a skewed predictor that provides a number between 0 and 9. If that number exceeds a threshold, the referenced block is predicted to be dead. If the block is referenced again, the corresponding counters in the prediction tables are decremented. If the block is evicted before being referenced again, the counters are incremented. The predictor learns from a *sampler*, a small set of partial tags held separately from the cache and managed with the LRU policy. In the original paper, each cache block has a prediction bit associated with it to drive the replacement policy. Our technique does not need this extra bit since the PseudoLRU block is always evicted. The technique only consults the predictor when a block is placed.

We modify the MDPP Cache policy to incorporate dynamism for a 16-way set-associative cache as follows: on a cache miss, the PC of the offending memory instruction is used to consult the dead block predictor. The predictor returns a confidence value c between 0 and 9. If $c = 0$, the block is placed in position 8, which is the closest placement to PseudoMRU that does not disturb the 8 blocks in the protected subtree of the root. If $0 < c < 6$ then the block is placed in position 14. This choice of placement position swaps the previous PseudoLRU block with the next-to-PseudoLRU block, having no effect on the positions of any other block in the set. If $c \geq 6$ then the block is bypassed, and the replacement state for the set remains unchanged. The cutoff for bypass of 6 or greater was determined empirically.

3.4 Evaluation

3.4.1 Methodology

We model performance with two simulators: one for single-threaded workloads and one for eight-core multi-programmed workloads. Both simulators use the following memory hierarchy parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way, 16-way, DRAM latency: 200 cycles. The single-thread simulator uses a 4MB L3 cache while the multi-core simulator uses a 16MB L3 cache. The single-thread simulator is a modified version of CMP$im [50]. The version we used was provided with the JILP Cache Replacement Championship [51]. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. This infrastructure enables collecting instructions-per-cycle (IPC) figures as well as misses per kilo-instruction and dead block predictor accuracy. For multi-core workloads, we found that CMP$im was quite slow and was unable to complete correctly beyond six cores. Thus, we adapted an in-house simulator to interoperate with CMP$im's cache replacement code to model eight core workloads.

Workloads

We use the 29 SPEC CPU 2006 benchmarks. Each benchmark is compiled for the 64-bit X86 instruction set. The programs are compiled with the GCC 4.1.0 compilers for C,

Technique	**Overhead for 4MB cache**	**Overhead for 16MB cache**
LRU	4 bits $\times$ 2^{16} blocks = 32KB	4 bits $\times$ 2^{18} blocks = 128KB
PseudoLRU	15 bits $\times$ 2^{12} sets = 7.5KB	15 bits $\times$ 2^{14} sets = 30KB
Static MDPP Cache	same as PseudoLRU: 7.5KB	same as PseudoLRU: 30KB
SRRIP/DRRIP	2 bits $\times$ 2^{16} blocks = 16KB	2 bits $\times$ 2^{18} blocks = 64KB
SHiP	16KB for RRIP + 11KB predictor = 27KB	64KB for RRIP + 11KB predictor = 75KB
Dynamic MDPP Cache	7.5KB + 11KB predictor = 18.5KB	30KB + 11KB predictor = 41KB

Table 3.2: Overhead required by the various techniques

C++, and FORTRAN. We use SimPoint [52] to identify up to 6 segments (i.e. *simpoints*) of one billion instructions each characteristic of the different program phases for each workload.

Single-Threaded Workloads

For single-threaded workloads, the results reported per benchmark are the weighted average of the results for the individual simpoints. The weights are generated by the SimPoint tool and represent the portion of all executed instructions for which a given simpoint is responsible. Each program is run with the first `ref` input provided by the `runspec` command. For each run, the simpoint is used to warm microarchitectural structures for 500 million instructions, then measures and reports results for the subsequent one billion instructions.

Multi-Core Workloads

For eight-core multi-programmed workloads, we generated 100 workloads consisting of mixes from the SPEC CPU 2006 simpoints described above. Each workload is a mix of eight simpoints chosen uniformly randomly without replacement. For each workload the simulator warms microarchitectural structures until 500 million total instructions have been executed, then measures results until at least one benchmark reaches one billion instructions. Thus, all 8 cores are active during the entire measurement period.

Replacement Policies

We compare our proposed technique against several related techniques: static and dynamic versions of re-reference interval prediction (SRRIP/DRRIP)[12], signature-based hit predictor for RRIP (SHiP)[37], and static and dynamic versions of genetic placement and promotion for tree-based PseudoLRU replacement (GIPPR/DGIPPR) [22]. For both simulators, we directly use code provided by the respective authors either through the World Wide Web or through personal communication. Our dynamic policy uses SDBP [16]. For that predictor, we used the code that is available from the cache replacement championship website[51].

The SHiP authors also graciously provided code for their hit predictor in a form that was readily adaptable to the RRIP code. We modified this code to implement the sampling-based policy described in their paper and use the program counter (PC) as the signature for the hit predictor. Thus, what we hereafter refer to as SHiP in this paper is called SHiP-PC-S in that paper's nomenclature[37]. This modification allows us to control the overhead consumed by SHiP's structures. Thus, we may allocate the same amount of storage to both SHiP and SDBP for a fair comparison of the dynamic policies. To be clear, we implement sampling-based SHiP using the PC as the signature and deciding the placement position for a baseline RRIP policy with a maximum re-reference prediction value of three.

The results we present in this paper compare our technique against SRRIP, DRRIP, SHiP, and GIPPR. In preparing this work, we also compared our technique against SDBP-based replacement and bypass [16] and protecting distance based policy [19]. We find that SHiP is superior in performance to these other policies, so we choose not to present those results and are content to compare with SHiP. We compare with SRRIP as an excellent static policy. We compare with DRRIP as a relatively simple dynamic policy with good performance. We compare with GIPPR/DGIPPR as the most closely related previous work.

3.4.2 Overhead Analysis

Overhead for Predictors

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

Overhead for Techniques

Each replacement policy we study require some overhead. Table 3.2 summarizes the overhead of the various polices. Note that DRRIP and DGIPPR require a few extra bits for set dueling counters. We ignore this tiny extra storage overhead. The multi-core MDPP Cache saves area over SHiP due to the fact that it requires only one bit of replacement state per block rather than two. This savings translates to a fraction of a percent of the bits allocated to the cache so we do not wish to overstate this advantage. Nevertheless, we note that the replacement state saved is typically implemented using larger 8T register file cells rather than 6T SRAM cells, and the savings in terms of cells per core of the dynamic MDPP Cache over SHiP is approximately 8KB, which is roughly equivalent to the size of a moderate dual-ported branch direction predictor or branch target buffer.

Impact of Storage Overhead

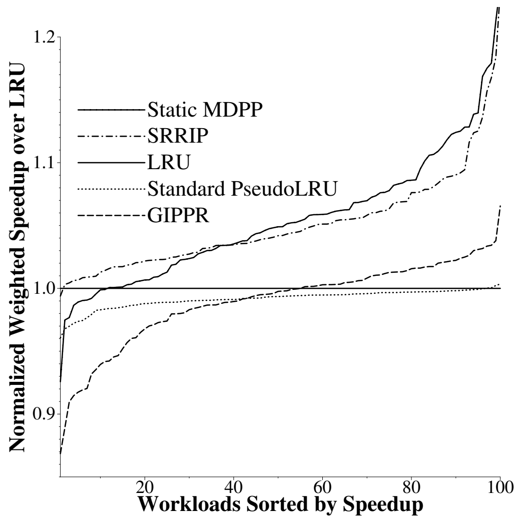
In the main results, we model SHiP and the dynamic MDPP Cache with the same storage overhead for the predictors, including tables of counters and set sampling overhead.

However, SHiP consumes 32 extra bits per set for storing the 16 2-bit re-reference interval prediction values, while the MDPP Cache requires only an additional 15 bits per set to store the PseudoLRU tree.

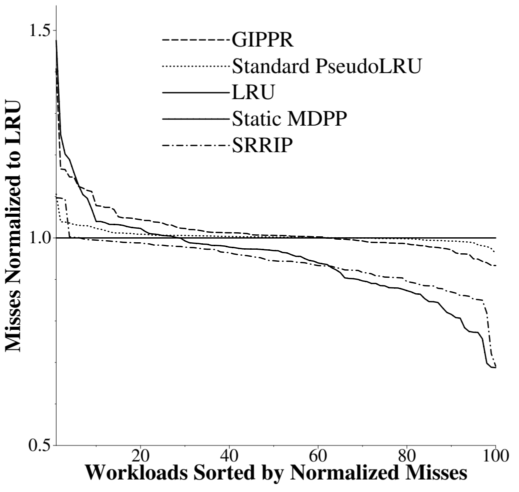
In another area study (not illustrated for space reasons), we performed three iso-area comparisons of SHiP with dynamic MDPP Cache to study the impact of this additional storage overhead. In the first experiment, we modeled SHiP with a smaller 2.5KB predictor, thus reducing the total overhead of SHiP to roughly the same as the dynamic MDPP Cache for the single-core 4MB cache configuration. This change had negligible impact on SHiP's performance. In the second experiment, we modified SHiP to use one bit for RRIP states rather than two bits, so that SHiP and the dynamic MDPP Cache would have equivalent storage requirements for the larger 16MB cache. This change resulted in a reduction of SHiP's speedup from 9.1% to 6.9%. In the third experiment, we allowed the 16MB dynamic MDPP Cache to expand the number of sampled sets such that SHiP and the MDPP Cache consume the same number of bits overall, including the per-set bits in the hardware budget. This allows the MDPP Cache to use 966 sampled sets instead of 96, resulting in an increase in the speedup to 11.5% from 11.0%. Thus, while the size of the SHiP predictor does not seem to have a large impact on performance, the size of the per-block state, which is the majority of the cache management metadata storage in the multi-core cache, has a significant impact on performance. Moreover, replacing SHiP with the dynamic MDPP Cache and keeping total area constant results in an 11.5% speedup.

3.4.3 Results for Static Policies

This section gives results for static policies. These policies do not adapt to changing workload characteristics, but are relatively simple to implement and verify. For instance, the static MDPP Cache policy could be plugged in easily to an existing PseudoLRU-replaced cache by simply modifying the algorithm or lookup table that implements placement and promotion.



(a)



(b)

Figure 3.5: (a) Normalized Weighted Speedup over LRU for Static Policies on 8-core Workloads, and (b) Misses Normalized to LRU for Static Policies

Multi-Core Workloads

Figure 3.5 shows weighted speedup normalized to LRU for SRRIP, static MDPP Cache, and standard PseudoLRU with a 16MB last-level cache (see Section 4.3.1 for our definition of speedup). The figure shows the speedups for each workload in ascending sorted order to yield S-curves. Standard PseudoLRU performs somewhat worse than LRU, giving a geometric mean speedup of 0.99. SRRIP gives a 5.2% improvement over LRU. Static MDPP Cache gives a 2.8% improvement over LRU. The best speedup of the static MDPP Cache over SRRIP is 10%, while the best speedup of SRRIP over the static MDPP Cache is 16%. SRRIP performs better, but at a cost of 56% more storage overhead. GIPPR with a single vector (not shown) performs quite poorly, so much so that it is impractical to

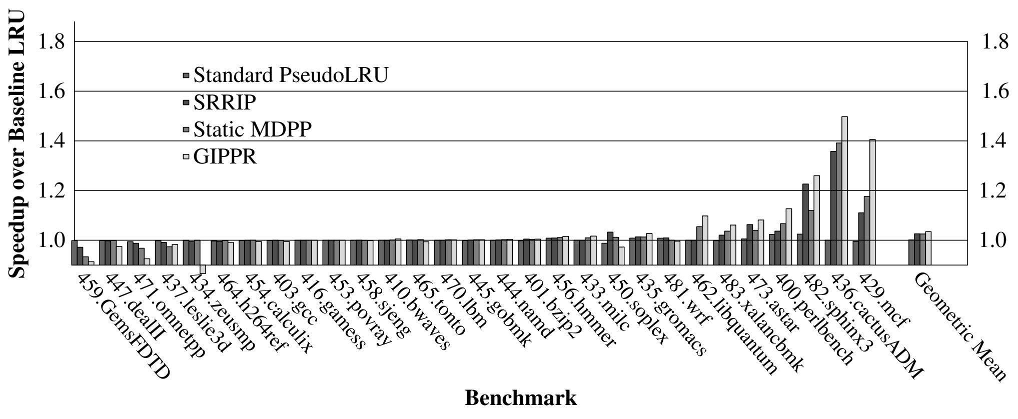


Figure 3.6: Speedup over LRU for Static Policies on Single-Threaded Workloads

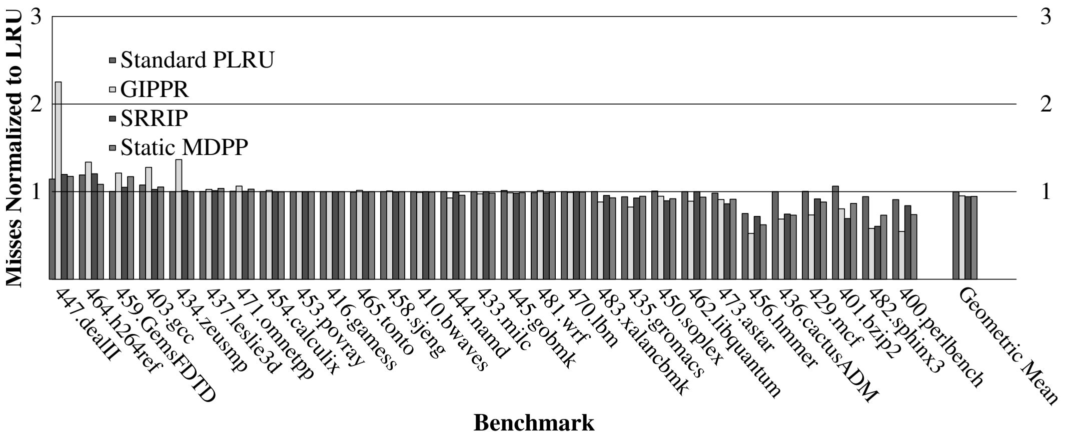


Figure 3.7: Misses Normalized to LRU

illustrate the performance graphically. Its average speedup over LRU is 0.87, *i.e.* it delivers a 13% slowdown. We have probed deeply into the GIPPR algorithm with the author of the original work to try to determine why it performs so poorly on multi-core workloads. GIPPR was designed by a genetic algorithm using single-threaded workloads. We conclude that its parameters work well in that context, but are not suitable for multi-core workloads. Single-core workloads are characterized by a few distinct patterns of behavior that can be captured by a handful of GIPPR vectors, but multi-core workloads combine characteristics resulting in a wide range of behaviors that cannot be distilled down to a small number of vectors.

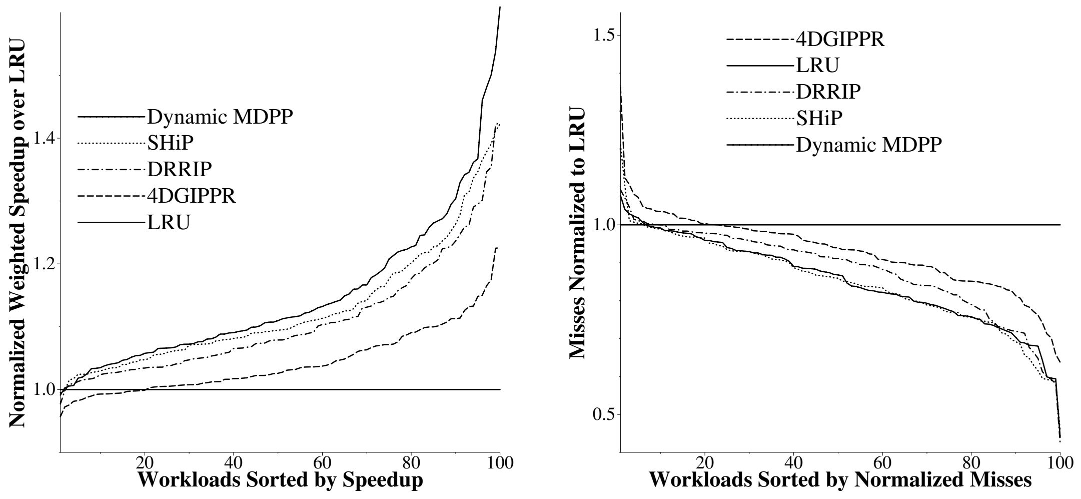
Misses

Figure 3.5(b) illustrates misses for the static policies on multi-core workloads. The graph shows misses normalized to LRU in descending order, *i.e.* best-to-worst from left-to-right, to yield S-curves. GIPPR delivers the fewest misses for approximately 20% of the workloads, but is worse than LRU for a majority of workloads, giving on average 5.7% more misses than LRU. SRRIP and static MDPP Cache perform closely, with SR-RIP giving 94.5% of the misses of LRU and MDPP minimal disturbance PseudoLRU giving 94.8% of LRU. minimal Disturbance PseudoLRU performs significantly worse than LRU for about 20% of the workloads. Several of those workloads are quite sensitive to misses, explaining SRRIP's significant performance advantage over minimal disturbance PseudoLRU. Standard PseudoLRU performs similarly to LRU.

Single Threaded Workloads

Figure 3.6 gives the speedup over LRU for several static techniques on single-threaded workloads with a 4MB last-level cache. The benchmarks on the y-axis are sorted based on their performance on the DRRIP policy. We use the same ordering in Section 3.4.4 for the bar chart there. Most of the benchmarks are not helped or significantly hurt by most of the polices. The aggregate speedup attributable to the polices are mostly due to the memory intensive benchmarks listed to the right of `401.bzip2`. However, some benchmarks such as `459.GemsFDTD` and `471.omnetpp` experience some slowdown from non-LRU policies.

Standard PseudoLRU gives no speedup or slowdown over LRU on average. SRRIP gives a geometric mean 2.6% speedup over LRU. GIPPR with a single vector optimized for the SPEC CPU 2006 benchmarks gives a speedup of 3.5%. minimal Disturbance



(a) (b)

Figure 3.8: (a) Normalized Weighted Speedup over LRU for Dynamic Policies on 8-core Workloads, and (b) Misses Normalized to LRU for Static Policies

PseudoLRU gives a speedup of 2.5%. Thus, the static MDPP Cache gives comparable performance to SRRIP. While GIPPR is slightly better than the static MDPP Cache on single-threaded workloads, we shall see that GIPPR performs poorly on multi-core workloads.

Misses

Figure 3.7 illustrates misses normalized to LRU for the single-threaded workloads. The benchmarks on the y-axis are sorted in descending order of misses for DRRIP normalized to LRU. Although GIPPR gives the best speedup, it only gives a geometric mean 95% of the misses of LRU. This counterintuitive result is largely due to the contribution of `447.dealII`, for which GIPPR more than doubles the misses of LRU, but whose change in performance is smaller than the changes for benchmarks such as `436.cactusADM` and `429.mcf`. The geometric mean of normalized misses for standard PseudoLRU is approximately the same as LRU. SRRIP and static MDPP Cache both give an arithmetic

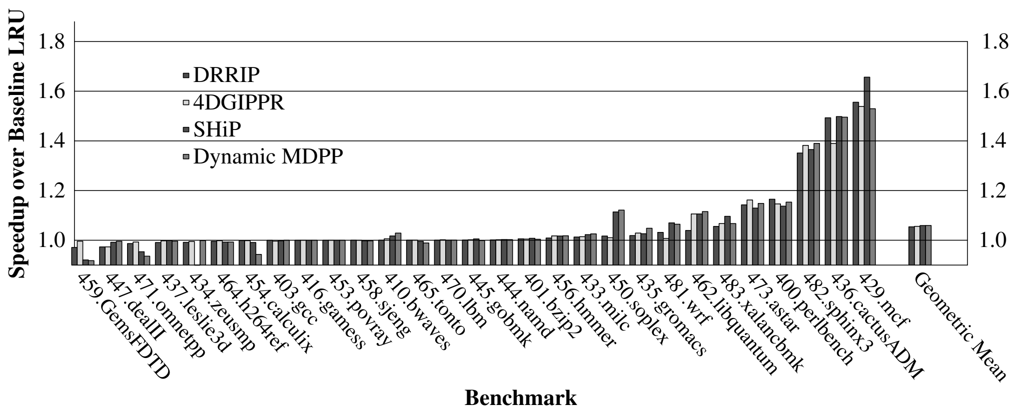


Figure 3.9: Speedup over LRU for Dynamic Policies on Single-Threaded Workloads

mean 94% of the misses of LRU.

3.4.4 Results for Dynamic Policies

This section gives results for dynamic policies. These policies adapt to different workload characteristics using set-dueling and/or prediction. Dynamic policies deliver the best performance in the literature. In particular, SHiP represents the current state-of-the-art cache management policy.

Multi-Core Workloads

Figure 3.8(a) shows the normalized weighted speedup for the various dynamic techniques. The speedups for the 100 workloads are shown as S-curves, sorted in ascending order. DRRIP yields an 8.4% speedup. The state-of-the-art SHiP policy gives a 9.1% speedup. The dynamic MDPP Cache achieves an 11.0% speedup. The best speedup of the dynamic MDPP Cache over SHiP is 25%, while the best speedup of SHiP over the dynamic MDPP Cache is 18%. Thus, the MDPP Cache improves performance over previous techniques without increasing complexity.

Misses

Figure 3.8(b) shows the misses normalized to LRU for the various dynamic techniques sorted in descending order, *i.e.* best-to-worst from left-to-right, to yield S-curves. 4DGIPPR is by far the worst, giving an average 4.2% more misses than LRU. SHiP and dynamic MDPP Cache are evenly matched, both delivering 85% of the misses of LRU. The advantage of MDPP Cache over SHiP on speedup is likely due to the effects of memory-level parallelism and other second-order effects in the workloads. DRRIP gives 87% of the misses of LRU. All of the policies except for 4DGIPPR give fewer misses than LRU for 85% – 90% of the workloads.

Single-Threaded Workloads

Figure 3.9 illustrates the speedup over LRU of the various dynamic techniques. The benchmarks on the y-axis are sorted based on their speedup over LRU for the DRRIP policy. DRRIP achieves a speedup of 5.4% over LRU. Dynamic MDPP Cache yields a 5.9% speedup. 4DGIPPR, the 4-vector dynamic version of GIPPR, gives a 5.6% speedup. SHiP gives a 5.9% speedup. Thus, dynamic MDPP Cache matches the performance of the state-of-the-art policy SHiP and exceeds the performance of 4DGIPPR whose 4 vectors are specially designed to work well with these specific single-threaded workloads.

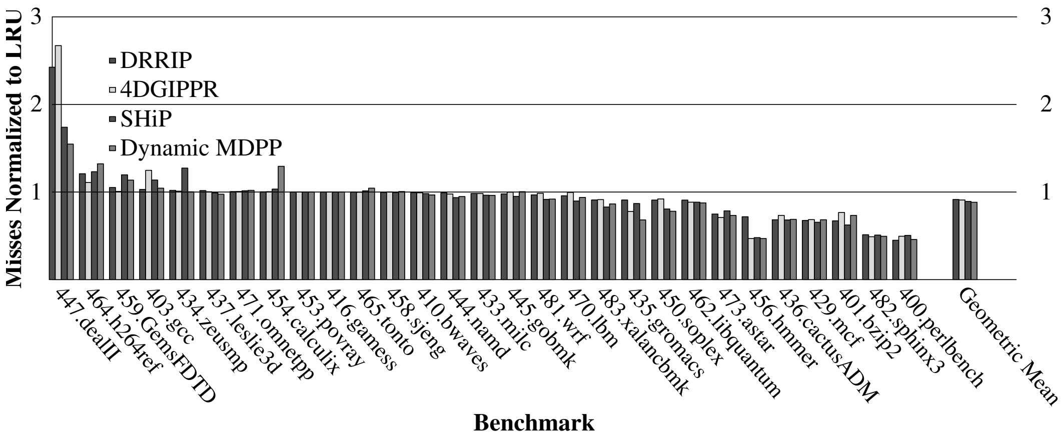


Figure 3.10: Misses for the Dynamic Policies Normalized to LRU

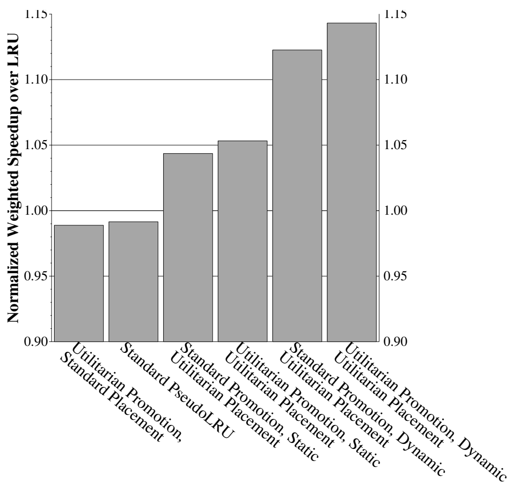


Figure 3.11: Contributions of minimal disturbance promotion and static/dynamic minimal disturbance placement

Misses

Figure 3.10 gives the normalized misses for the dynamic policies. The benchmarks on the y-axis are sorted in descending order of their misses on DRRIP normalized to LRU. DRRIP and 4DGIPPR both give geometric mean normalized misses of 91% of LRU. SHiP delivers a geometric mean normalized misses of 89% of LRU. Dynamic minimal disturbance PseudoLRU yields 88% of the misses of LRU. Curiously, all of the dynamic policies seem to give more misses than the static policies on `447.dealII`, although this fact does not have a large impact on performance because `447.dealII` is not as memory-bound as some of the other benchmarks.

3.4.5 Analysis

Figure 3.11 quantifies the contributions of the individual placement and promotion policies to improving tree-based PseudoLRU. All combinations of standard PseudoLRU placement and promotion, minimal disturbance promotion, and static and dynamic minimal disturbance placement are considered. The normalized weighted speedups are aggregated over the 100 multi-core workloads. The bar chart is ordered in ascending geometric mean speedup over true LRU.

Minimal disturbance promotion with standard placement at position 0 gives a geometric mean of 98.6% of the performance of LRU, very slightly worse than normal PseudoLRU at a 99.0%. Standard promotion with minimal disturbance placement yields a 2.7% speedup over LRU. Minimal disturbance promotion and static minimal disturbance placement together (*i.e.* the static MDPP Cache) give a 2.8% improvement over LRU. At this point, it is clear that the placement position is more important for performance than the promotion policy, but both are needed for the best static policy.

The contribution of minimal disturbance promotion is more apparent for the dynamic policies. Dynamic minimal disturbance placement with standard PseudoLRU promotion yields a geometric mean speedup of 9.3%, which is slightly better than the performance of SHiP (not illustrated) at 9.1%. With both minimal disturbance promotion and dynamic minimal disturbance placement (*i.e.* the dynamic MDPP Cache), the geometric mean speedup is boosted to 11.0%.

3.5 Summary

In this chapter, we have considered the principle of minimal disturbance as applied to the changes in positions of cache blocks. We have shown that tree-based PseudoLRU is disturbing, and that mending it significantly improves the policy. We present results to prove that the static minimal disturbance policy as well as a dynamic version using dead block prediction provide performance competitive with state-of-the-art policies at a significantly lower cost in terms of storage overhead.

4. PERCEPTRON LEARNING FOR REUSE PREDICTION *

We have seen how machine learning principles can give rise to a simple and efficient replacement policy. In this chapter, we will explore machine learning further to develop a highly accurate reuse predictor.

Reuse predictors determine which blocks in the cache are most likely to be frequently reference. However, the accuracy of the prediction mechanisms limits the scope of optimization. We present Perceptron Learning for Reuse Prediction. Our predictor yields higher accuracy than previously proposed work, which translates into better system performance.

4.1 Introduction

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

*2016 IEEE. Reprinted, with permission of E.Teran, Z. Wang, D. Jiménez, "Perceptron learning for reuse prediction", December 2016

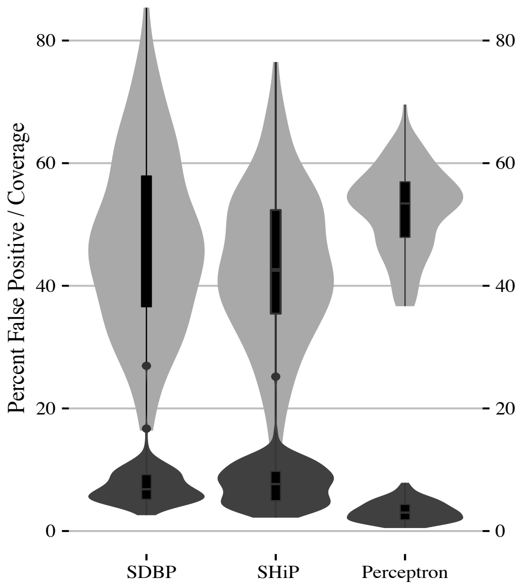


Figure 4.1: Violin plots for false positive (darker) and coverage (lighter) rates. Perceptron learning is significantly more robust, with higher coverage, fewer false positives, and less variance in accuracy across workloads.

4.1.1 Better Accuracy with Perceptron Learning

Figure 4.1 shows violin plots for the accuracy of three reuse predictors: sampling dead block prediction (SDBP) [16],

signature-based hit prediction (SHiP) [37], and our proposed perceptron-based reuse predictor over multi-programmed workloads. The coverage rate (shaded darker) is the percentage of all predictions for which a block is predicted not to be reused. The false positive rate (shaded lighter) is the percentage of all predictions for which a block was incorrectly predicted as not reused. The plots show the mean as a bar in the center of the plot along with the probability density of the data. The coverage for perceptron-based reuse prediction is higher than that for SDBP and SHiP (52.4% versus 47.2% and 43.2%, respectively) while the false positive rate is lower (3.2% versus 7.4% and 7.7%, respectively). Coverage represents the opportunity for optimization given by the predictor, while false positives represent the potential cache misses caused when an incorrect prediction leads to a live block being replaced. Thus, perceptron learning leads to increased opportunity for optimization while reducing false positives by a factor of 2. For more on the violin plots, see Section 4.3.6.

Perceptron Learning

This chapter describes an algorithm that uses perceptron learning for reuse prediction. Perceptrons are a simple model of neurons in neural networks [54, 55] modeled by vectors of signed weights learned through online training. The output of a perceptron is the dot product of the weights and a vector of inputs. In this work, we do not actually use perceptrons, but we make use of the perceptron learning algorithm. There are two components to the abstract perceptron learning algorithm:

1. To predict true or false, a vector of signed weights is chosen according to some criteria. The dot product of the weights and input vectors, called y_{out}, is computed. If y_{out} exceeds some threshold, the prediction is true, otherwise it is false.

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

In the original paper describing branch prediction with perceptrons, the input vector was the global history of branch outcomes [38]. In this work, as in some subsequent branch

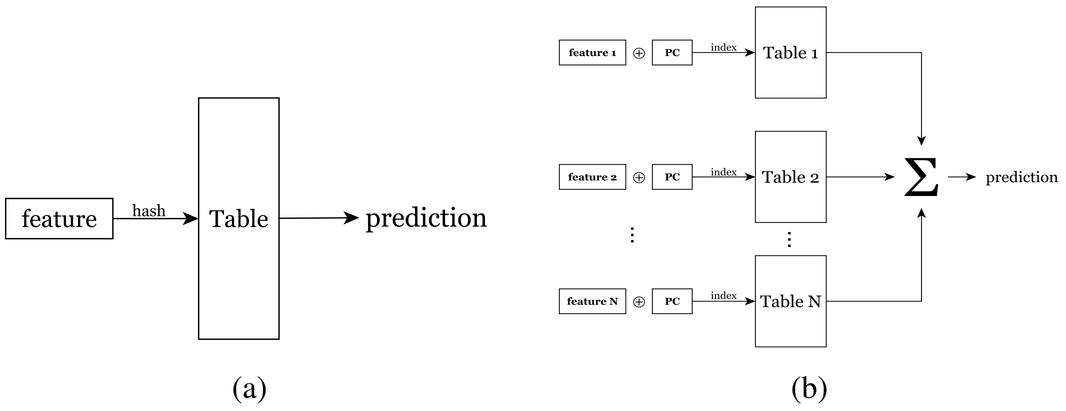


Figure 4.2: (a) Previous PC-based Reuse Predictors and (b) Perceptron-based Reuse Predictor

prediction work [41, 40], the weights in the vectors are chosen by indexing independent tables using indices computed as hashes of features such as branch pattern and path history. Such *hashed perceptron* predictors do not use a vector of binary weights. Rather, instead of a dot product, they simply compute a sum which can be thought of as a dot product of the weights vector with $\vec{1}$. In this work, we use features relevant to cache management such as the address of memory instructions and bits from the block address, tag, or page number. Thus, although the technique does not use classical perceptrons, it uses perceptron learning to adjust the weights.

Contributions

The work presented in this chapter makes the following contributions:

1. It shows that perceptron learning can be used to effectively combine multiple inputs features to greatly improve the accuracy of reuse prediction.

2. It applies perceptron-learning-based reuse prediction to a replacement and bypass optimization, outperforming the state-of-the-art policies for both single and multi-programmed workloads. For 1000 multi- programmed workloads, perceptron learn-

ing gives a geometric mean normalized weighted speedup of 8.3%. For single-threaded workloads, perceptron learning gives a geometric mean speedup of 6.2% over LRU, with a speedup of 18.3% on a memory-intensive subset of the benchmarks.

3. It shows that cache management based on perceptron learning more than doubles cache efficiency over the LRU policy.

4. It evaluates out Perceptron predictor as well as previous work in the presence of a stream prefetcher. Previous work had been evaluated without considering prefetching.

Features Correlated with Last Access to a Block

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

1. **The trace of memory access instructions** (PCs) beginning when the block is placed and ending at the current access [31, 35]. This sequence of addresses can be concisely summarized as a truncated sum of those PCs, generating an integer signature to be used to index a prediction table. In an L1 cache, this signature is correlated with the last access to a block. However, in an LLC, particularly in the presence of a middle-level cache, many PCs are filtered, leaving a porous signature that is unsuitable for prediction [16]. In this work, we keep track of recent PCs accessing the LLC, but we treat each PC as a separate feature.

2. **The PC of the memory instruction** that caused the access. This feature is a succinct proxy for the program behavior that led to the ultimate eviction of the block. Unlike the instruction trace, it has reasonable accuracy in the last-level cache [16]. However, its accuracy is limited because a single PC can exhibit multiple behaviors

depending on the control-flow context, *e.g.*, an access in a given subroutine may or may not be the last access to a block depending on the caller.

3. **Bits from the memory address**, *e.g.*, the page number or tag. Data in the same memory region are often treated similarly by programs, so the last access (or the only access) to one block may correlate with the last access to a neighboring block. There are many more memory addresses than memory access PCs, so using bits from the memory address as an input feature may lead to destructive interference in small prediction tables. Liu *et al.* combine the trace signature with 3 bits from the block address [35] to form a single index. Our work treats bits from the memory addresses separately, isolating the effects of interference.

4. **A compressed representation** of the data itself [25]. Data that are similar in blocks may lead to similar behavior.

5. **Time/reference count**. Some reuse predictors work on the idea that a block may be accessed a certain number of times before it is evicted, and that this count may be predicted. This technique requires keeping a count for each block in the cache, an idea we dismiss as too complex for implementation in a large last-level cache.

4.2 Design

4.2.1 Perceptron Learning for Reuse Prediction

In this section we present the prediction algorithm. The structures are similar to those of sampling-based dead block prediction [16], but the algorithm and inputs are altered to provide superior accuracy.

The Main Idea

Our predictor combines multiple features. To make a prediction, each feature is used to index a distinct table of saturating counters (hereafter *weights*) that are then summed.

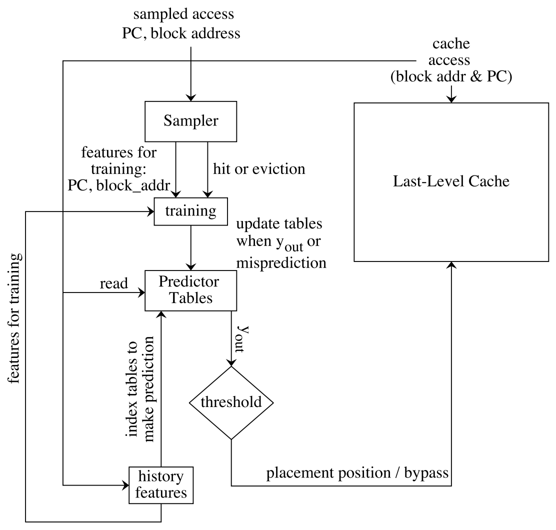


Figure 4.3: Datapath from extracting features from an access, to making a prediction and acting on it

If the sum exceeds some threshold, then the accessed block is predicted not to be reused. A small fraction of the accesses are sampled to update the predictor using the perceptron update rule: if the prediction is incorrect, or the sum fails to exceed some magnitude, then the weights are decremented on an access or incremented on an eviction.

Combining features by summing weights avoids the destructive interference and exponential blowup that can be caused by combining them into a single index. That is, if we combine PC bits and memory address bits by, say, summing or XORing them directly into an index, the predictor must learn all the likely patterns that lead to different behavior before it can predict well. Using the perceptron update rule allows the predictor to adapt to changes in program behavior. In previous work, both summing and perceptron update have contributed to significant gains in branch prediction accuracy, motivating this work [38, 40].

Figure 4.2(a) shows the organization of recent reuse predictors SDBP and SHiP. A single feature is used to index a table of weights to make a prediction. SHiP and SDBP use as the feature a hash of the PC of the memory instruction that causes the reference. Other dead block predictors use a signature formed by truncated addition of all the PCs from the placement of the block to the current reference [31, 35], as well as some bits from the address of the references block [31], but these features are all combined into a single index into the table. It has been shown that the truncated addition idea works poorly for a LLC [16] due to the fact that first- and middle-level caches filter out many of the useful PCs, leaving a noisy signal.

Figure 4.2(b) shows the organization of the perceptron-based reuse predictor. Each feature is hashed, then XORed with the PC to index a separate table. The corresponding weights are summed and thresholded to get an aggregate prediction. This scheme works well because, at any given time, one or another feature may carry correlation with reuse behavior, while others do not. The perceptron algorithm discovers correlation and ignores the lack of correlation, to give a more accurate prediction.

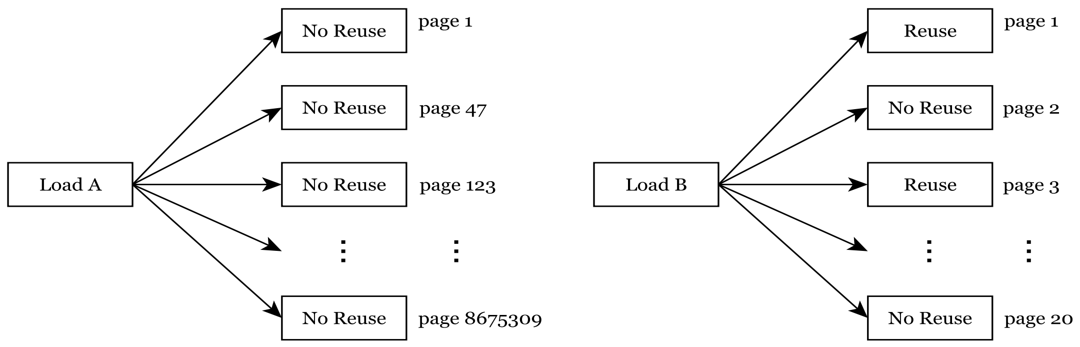


Figure 4.4: Two different kinds of correlation. The PC of one load (left) always correlates with no reuse on thousands of pages, while another load (right) has no correlation through the PC but does have correlation through page number.

Example

To illustrate why perceptron learning yields superior accuracy to previous reuse predictors, let us consider a simple example. Suppose we must design a reuse predictor indexed by either the PC of the memory instruction, the page number of the memory access, or a combination of both. Figure 4.4 illustrates two instructions: load A and load B. Load A randomly accesses cache blocks across thousands of pages, but these blocks experience little to no reuse after being accessed by load A. (Perhaps load A is in a destructor in a pointer-chasing object-oriented program.) Load B accesses blocks with good spatial locality over a small number of pages. Cache blocks accessed by load B on a certain set of pages experience reuse, but on another set of pages blocks experience little reuse. (Perhaps load B is a factory method handing out objects to various callers with different access patterns.) Load A would be predicted well by a PC-based predictor, but load B would be predicted poorly because of the variable nature of subsequent accesses. Load A would be predicted poorly by a page-based predictor because thousands of page references would cause aliasing (*i.e.,* collisions) in the prediction table, while load B might be better predicted by a page-based predictor since its behavior is page-based. Neither one would be predicted by indexing a table by a combination (*e.g.*, XOR) of PC and page number: load A continues to have the problem with aliasing, and the prediction table for load B now has more aliasing pressure because we are multiplying the number of indices into the table by the number of other PCs in the working set of load and store instructions.

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

Note that SDBP uses three separate tables indexed by different hashes of the same PC [16]. This slightly reduces the impact of aliasing, but does not help in situations like load B where there is page-based correlation that is not discovered by a PC-based predictor.

Training with a Sampler

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

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

2. A small integer sum (y_{out}) representing the most recent prediction computation. y_{out} is used to drive the *threshold rule* of the perceptron learning algorithm: if the magnitude of y_{out} exceeds some threshold and the prediction is correct, the predictor does not need to be updated. As this strategy prevents the weights from all saturating at their extreme values, it improves prediction accuracy by ensuring that the predictor can respond quickly to changes in program behavior.

3. A sequence of hashes of the input features used to index the prediction tables. In this sequence, each feature is hashed, then XORed with the lower-order bits of the PC of the memory instruction that last accessed this entry. These hashes will be used to index the predictor's tables.

4. LRU bits for replacement within the sampler.

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

4.2.2 Predictor Organization

The predictor is organized as a set of tables, one per input feature. Each table has a small number of weights. In our experiments, we find that providing each table with 256 entries of 6-bit signed weights (ranging in value from -32 to +31) is sufficient. A sequence of hashed features are kept in per-core vectors that are updated on every memory access. We have listed several possible features in Section 4.1.1. We find that using the last few PCs to access the cache as well as bits extracted from the address of the currently accessed block yield the best accuracy. Figure 4.3 shows the datapath from extracting features from memory accesses to making a prediction.

Making a Prediction

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

Training the Predictor

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

When there is a replacement in the sampler: a miss in the sampler requires eviction of the LRU entry. The fact that this block is evicted shows it is unlikely to be reused, so the predictor learns from this evidence. If the value of y_{out} for the victim entry is less than a threshold θ, or if the prediction was incorrect, then the predictor table entries indexed by the corresponding hashes in the victim entry are incremented with saturating arithmetic. Thus, future similar sequences of features for this instruction are more likely to be predicted as not reused. (Note that the threshold θ for invoking training may be different from the threshold τ for predicting reuse; see Section 4.2.2).

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

4.2.3 Replacement and Bypass Optimization

The predictor is applied to the optimization of dead block replacement and bypass. As in previous work [16], the LLC is replaced with a default replacement policy such as LRU, but a block predicted with no reuse (*i.e.,* a dead block) will be replaced before the candidate given by the default policy. In this work, the LLC uses tree-based PseudoLRU as the default replacement policy [22]. Each cache block is associated with a prediction bit that holds the most recent reuse prediction for that block.

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

Tree-based PseudoLRU requires $n - 1$ bits per set for an n-way set associative cache. Thus, including the prediction bit, each block requires about two bits of replacement state, which is equivalent to the per-block overheads for DRRIP [12] and SHiP [37].

Prefetches

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

4.3 Evaluation

4.3.1 Methodology

We model performance with an in-house simulator using the following memory hierarchy parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way, DRAM latency: 200 cycles. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. The single-thread simulations use a 4MB L3 cache while the multi-core simulations use a 16MB L3 cache. The simulator models a stream prefetcher.

It starts a stream on a L1 cache miss and waits for at most two misses to decide on the direction of the stream. After that it starts to generate and send prefetch requests. It can track 16 separate streams. The replacement policy for the streams is LRU. This infrastructure enables collecting instructions-per-cycle figures as well as misses per kilo-instruction and dead block predictor accuracy.

Workloads

We use the 29 SPEC CPU 2006 benchmarks. Each benchmark is compiled using a configuration file for the X86 instruction set distributed with the SPEC CPU 2006 benchmarks. We use SimPoint [52] to identify up to 6 segments (*i.e., simpoints*) of one billion instructions each characteristic of the different program phases for each workload.

Single-Threaded Workloads

For single-threaded workloads, the results reported per benchmark are the weighted average of the results for the individual simpoints. The weights are generated by the SimPoint tool and represent the portion of all executed instructions for which a given simpoint is responsible. Each program is run with the first `ref` input provided by the `runspec` command. For each run, the 500 million instructions previous to the simpoint are used to warm microarchitectural structures, then the subsequent one billion instructions are used to measure and report results.

Multi-Core Workloads

For 8-core multi-programmed workloads, we generated 1000 workloads consisting of mixes from the SPEC CPU 2006 simpoints described above. We follow the sample-balanced methodology of FIESTA [56]. We begin by selecting regions of equal standalone running time for each simpoint. Each region begins at the start of the simpoint and ends when the number of cycles in a standalone simulation reaches one billion cycles. Each workload is a mix of 8 of these regions chosen uniformly randomly without replacement. For each workload the simulator warms microarchitectural structures until 500 million total instructions have been executed, then measures results until each benchmark has executed for at least one billion additional cycles. When a thread reaches the end of its one billion cycle region, it starts over at the beginning. Thus, all 8 cores are active during the entire measurement period.

Avoiding Overfitting of the Parameters

Perceptron has several parameters whose values affect accuracy and performance. We took steps to avoid overfitting the parameters to the workloads.

For single-threaded workloads, we use a leave-out-one cross-validation methodology. That is, for each of the 29 SPEC CPU 2006 benchmarks, we explore the space of parameters that give the best aggregate performance for the other 28 benchmarks, then use those parameters for the benchmark in question. The parameters varied were the values of θ, τ_{bypass}, τ_{replace}, the bit width of the weights, and parameters related to selecting the features listed in Section 4.3.1. Thus, for each of the benchmarks, we report results based on parameters that were not tuned for that benchmark. We find remarkable consistency in the parameters chosen for each benchmark. For instance, when $|y_{\text{out}}| < \theta$, training is invoked. We find that for every combination of 28 benchmarks but one, the best value is $\theta = 74$. When the benchmark sphinx3 is held out the best value is $\theta = 71$.

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

Replacement Policies

We compare Perceptron against two closely related techniques: sampling based dead block prediction driving replacement and bypass (SDBP) [16] and signature-based hit prediction (SHiP) [37]. We use SDBP code provided by the original authors. The SHiP

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

Table 4.1: Overhead required by the various techniques is approximately the same in our experiments. See text for details of sampler entry overheads.

authors provided code for their hit predictor in a form that was readily adaptable to RRIP code they also provided. We modified this code to implement the sampling-based policy described in their paper and use the program counter (PC) as the signature for the hit predictor. Thus, what we hereafter refer to as SHiP in this chapter is called SHiP-PC-S in that paper's nomenclature [37]. This modification allows us to control the overhead consumed by SHiP's structures. Thus, we may allocate the same amount of storage to SHiP, SDBP, and perceptron-based reuse prediction for comparison. To be clear, we implement sampling-based SHiP using the PC as the signature and deciding the insertion position for a baseline RRIP policy with a maximum re-reference prediction value of three. In the following text and figures, *Perceptron* with a capital P refers to the replacement and bypass optimization that uses perceptron learning for reuse prediction.

We evaluate SDBP because its structure is similar to that of our predictor and it also uses summation and thresholding to make a prediction. We evaluate SHiP because it provides the best speedup in the literature for cache management based on reuse prediction. There is significant other work in reuse prediction [31, 33, 34, 35, 6, 12, 19, 22] but because of SHiP's superior performance and for space reasons we do not report results for those techniques.

SHiP and Bypass

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

SDBP and SHiP with Prefetching

In the original papers, SDBP and SHiP were evaluated without modeling prefetching. To provide a realistic evaluation, we model a stream prefetcher. We found that the unmodified SDBP and SHiP algorithms perform poorly in the presence of prefetching because the prefetcher sometimes issues prefetches that hit in the cache. These hits artificially boost the apparent locality of the blocks, causing SDBP and SHiP to promote blocks that should remain near LRU. We modify the code for both SDBP and SHiP to ignore hitting prefetches and observe that the change restores the good performance of these algorithms. In Section 4.3.6 we describe results without prefetching to evaluate our work in the same methodological context as the previous work.

Measuring Performance

In Section 4.3.4 we report performance relative to LRU for the various techniques tested. For single-threaded workloads, we report the speedup over LRU, *i.e.,* the instructions-per-cycle (IPC) of a technique divided by the IPC given by LRU. For the multi-core workloads, we report the weighted speedup normalized to LRU. That is, for each thread i sharing the 16MB cache, we compute IPC_i. Then we find SingleIPC_i as the IPC of the same program running in isolation with a 16MB cache with LRU replacement. Then we compute the weighted IPC as $\sum \text{IPC}_i/\text{SingleIPC}_i$. We then normalize this weighted IPC with the weighted IPC using the LRU replacement policy.

4.3.2 Overhead Analysis

SHiP, SDBP, and perceptron-based reuse prediction require extra state for their prediction structures. Table 4.1 summarizes the overheads for the various techniques. SHiP uses a 14-bit signature derived as a hash of the PC indexing a table of 16,384 three-bit saturating counters. It also keeps 14-bit signatures and one reuse bit for each block in the sampled sets. SDBP keeps three tables of 8,192 2-bit saturating counters each indexed by a different 13-bit hash of the PC. It also keeps a *sampler*, an array of cache metadata kept separately from the main cache. Each block in the sampler includes a 15-bit partial tag, a prediction bit, a valid bit, a 4-bit LRU stack position and a 15-bit signature. The associativity of the sampler is 12. To keep the same amount of state for each predictor, we use 192 sampled sets for SHiP and 96 sampled sets for SDBP. Both predictors consume approximately 11KB.

The state required for perceptron-based reuse prediction is proportional to the number of features used. We empirically determined that six features provided the best trade-off between storage and accuracy (see Section 4.3.3). We find that using an associativity of 16 for the sampler provides the best accuracy, as opposed to the associativity of 12 that was used in the SDBP work [16]. The prediction tables for the perceptron sampler have 6-bit signed weights, and there are six such tables. We find that using 256-entry tables provides a good trade-off between accuracy and storage budget. Thus, the prediction tables consume 1.125KB.

Each perceptron sampler entry contains six 8-bit hashes of the features, a valid bit, a 9-

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

Each of the three policies also keeps storage associated with each cache block. SHiP is built on RRIP [12] requiring two bits per cache block. The perceptron-based reuse predictor uses tree-based PseudoLRU as its default replacement policy using one bit per cache block. It adds another bit for each block to store the most recent reuse prediction for that block. Thus, SHiP and perceptron-based reuse prediction keep the same number of bits associated with each block. SDBP uses true LRU instead of PseudoLRU, so it requires 4 LRU bits plus one prediction bit for each block in the cache. We use the unmodified SDBP code rather than hack PseudoLRU into it. We stipulate that PseudoLRU and true LRU ought to behave similarly so the 5-bit overhead of SDBP should not be counted against SDBP.

The implementation complexity of the perceptron-based reuse predictor is slightly higher than that of SHiP or SDBP, but it is still reasonable. Conditional branch predictors based on perceptron learning have been implemented in Oracle and AMD processors [57, 58]. Branch predictors must operate under very tight timing constraints. Predictors in the last-level cache have a higher tolerance for latency, so we are confident perceptron learning can be easily adapted to reuse predictors.

4.3.3 Features and Parameters

Features for the Predictor

We determined that the following six features provide a good trade-off between hardware overhead and prediction accuracy. Suppose PC$_i$ is the address of the i^{th} most recent instruction accessing the LLC from the current thread, where PC$_0$ is the PC of the current memory access instruction. Then the features are:

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

We choose not to use features such as the number of accesses to a block [6] or other history of the block itself because of the overhead required in storing such information per block.

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

The next three features are the most recent PCs of memory access instructions that accessed the LLC, each shifted right by an increasing number. These features give the predictor an idea of the "neighborhood" the instruction was in when it made the access. Shifting by increasing amounts allows a wider perspective for events further in the past, *i.e.,* rather than specifying the instruction at PC$_3$, the predictor learns from the 8-byte region around that instruction.

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

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

Other Parameters

The parameter τ is the threshold below which a block is predicted to be reused. We

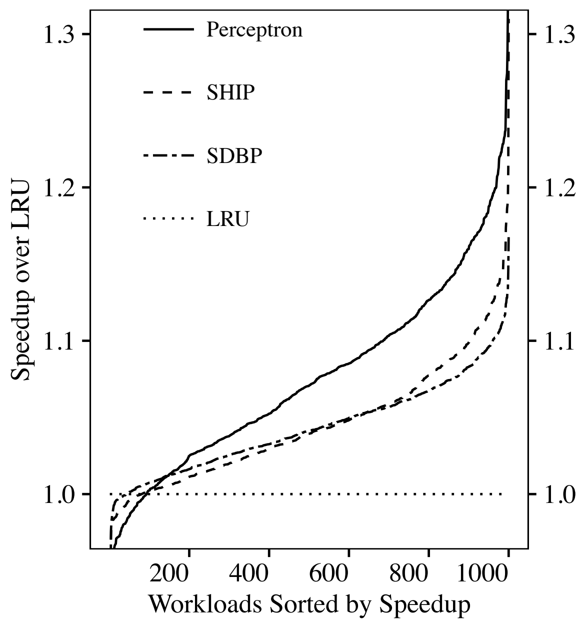


Figure 4.5: Normalized Weighted Speedup over LRU for 8-Core Multi-Programmed Workloads

chose two different values for τ based on the purpose of the prediction: τ_{bypass} for predicting whether a block should be bypassed, and τ_{replace} for predicting whether a block may be replaced after a hit. On 100 multi-programmed workloads (as described in Section 4.3.1), we empirically found the best values were $\tau_{\text{bypass}} = 3$ and $\tau_{\text{replace}} = 124$. The best value for θ, for which training is triggered when the magnitude of a correct prediction is below θ, was 68. Recall that, even though $\theta = 68$, training will still be invoked if the prediction is incorrect. These parameters were used to provide the results for the 1000 multi-programmed workloads (that exclude the 100 used for training). For single-thread workloads, the parameters were chosen according to the cross-validation methodology described in Section 4.3.1 to avoid overfitting.

4.3.4 Results for Multi-Core

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

Performance

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

Figure 4.5 shows weighted speedup normalized to LRU for SDBP, SHiP, and Perceptron with a 16MB last-level cache. See 4.3.1 for the definition of weighted speedup. The figure shows the speedups for each workload in ascending sorted order to yield S-curves. SDBP yields a geometric mean 4.3% speedup and SHiP yields a 4.4% speedup. Perceptron gives a geometric mean speedup of 7.4%. The superior accuracy of Perceptron gives a significant boost in performance over the other two reuse predictors.

Misses

Figure 5.11 shows misses per 1000 instructions (MPKI) various techniques sorted in descending order, *i.e.,* worst-to-best from left-to-right, to yield S-curves with a log scale

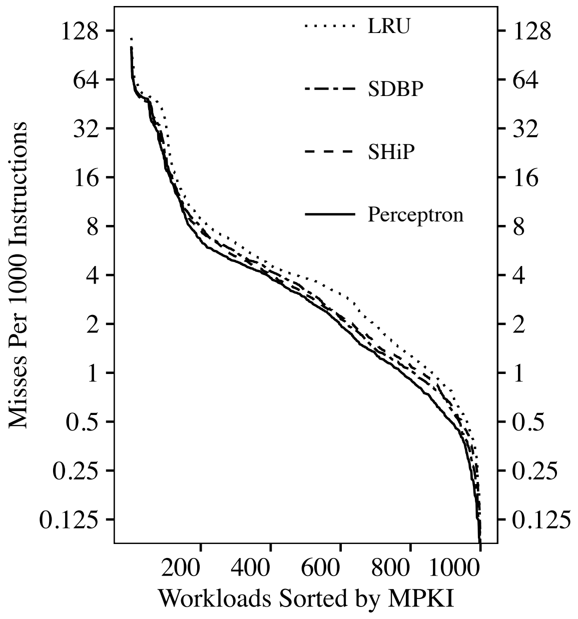


Figure 4.6: Misses per 1000 Instructions for 8-Core Multi-Programmed Workloads

y-axis. Perceptron, at an arithmetic mean 7.2 MPKI, delivers fewer misses than the other techniques. LRU, SDBP, and SHiP yield 9.4 MPKI, 7.8 MPKI, and 7.6 MPKI, respectively.

4.3.5 Results for Single-Core

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

Performance

Figure 4.7 shows single-thread speedup of the techniques over LRU. The benchmarks are sorted by speedup with Perceptron. SDBP and SHiP achieve a geometric mean 3.5% and 3.8% speedup over LRU, respectively. Perceptron yields a 6.1% geometric mean speedup. For 25 out of the 29 benchmarks, Perceptron matches or exceeds the performance of LRU, and for most workloads it exceeds the performance of the other two techniques.

Many of the benchmarks experience few misses on a 4MB cache. We define a memory-intensive subset of SPEC as benchmarks with an MPKI of at least 1.0 under the LRU replacement policy. These benchmarks are GemsFDTD, sphinx3, xalancbmk, cactusADM, lbm, astar, soplex, omnetpp, milc and mcf. On this subset (not separately illustrated), perceptron learning yields a geometric mean 18.3% speedup, versus 10.5% for SHiP and 7.7% for SDBP.

Misses

Figure 4.8 gives the MPKI for the 29 benchmarks Note the y-axis is a log scale. SDBP and SHiP have 2.3 and 2.2 MPKI, respectively. Perceptron gives 2.0 MPKI. The average MPKI figures are low because most of the benchmarks fit a large part of their working sets into a 4MB cache. For omnetpp, a benchmark with considerable misses, SDBP gives 8.3 MPKI, SHiP yields 9.2 MPKI, and Perceptron gives 7.7 MPKI.

4.3.6 Alternative Metrics

The original SDBP and SHiP studies were presented in simulation infrastructures that did not include prefetching. Prefetching has a significant effect on cache management studies because many more references to memory are made. We believe cache management is best studied alongside prefetching since modern high performance processors include hardware prefetchers. As stated in Section 4.3.1, we modify SDBP and SHiP to remove the confusing influence of prefetching, boosting their performance. Nevertheless, to put our work in the proper context of previous work, we run experiments with prefetching disabled for all techniques including the baseline LRU policy, SDBP, SHiP, and Perceptron. We do not illustrate the results with graphs for space reasons. The geometric mean normalized weighted speedups for SDBP, SHiP, and Perceptron are 7.5%, 8.3%, and 11.2%, respectively. The first two numbers are consistent with the conclusions of the previous work: SHiP has an advantage over SDBP. The third number shows that

Perceptron continues to yield significant speedup over SHiP and SDBP. Note that all of the speedups are higher than the speedups obtained with prefetching. Prefetching eliminates some of the opportunity for optimization by replacement policy since it turns many potential misses into hits. Our LRU baseline with prefetching achieves a geometric mean 37% speedup over no prefetching, so obviously we are satisfied to accept a slight decrease in the advantage provided by an improved replacement policy.

Accuracy

Figure 4.1 (from Section 4.1, the introduction) illustrates the accuracy of the three predictors, giving violin plots for the false positive and coverage rates of each predictor. Violin plots show the mean as a bar in the center of the plot along with the probability density of the data. The coverage rate is the percentage of all predictions for which a block is predicted not to be reused. The false positive rate is the percentage of all predictions for which reuse was incorrectly predicted.

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

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

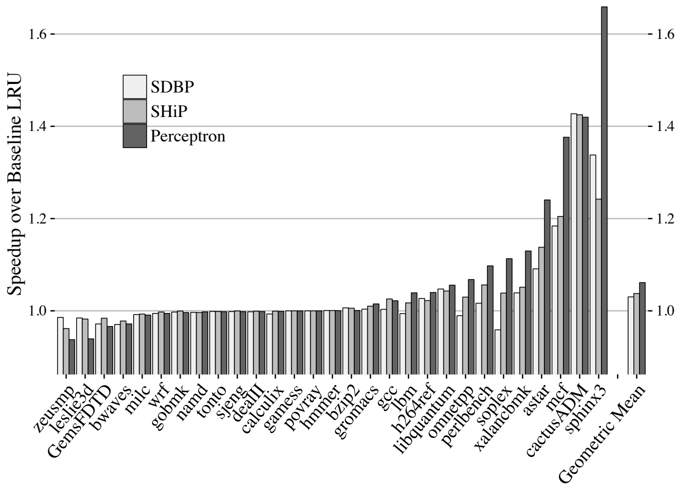


Figure 4.7: Speedup over LRU for Single-Thread Workloads.

Cache Efficiency

Cache management with reuse prediction improves cache efficiency. Cache efficiency is defined as the fraction of time that a given cache block is live, *i.e.*,, the number of cycles that a block contains data that will be referenced again divided by the total number of cycles the block contains valid data [59]. A more efficient cache makes better use of the available space and wastes less energy storing dead blocks. Figure 4.9 illustrates cache efficiency for the various techniques on a typical multi-programmed workload. In the four 128×128 heat maps, each pixel represents the average efficiency of one set in the 16MB

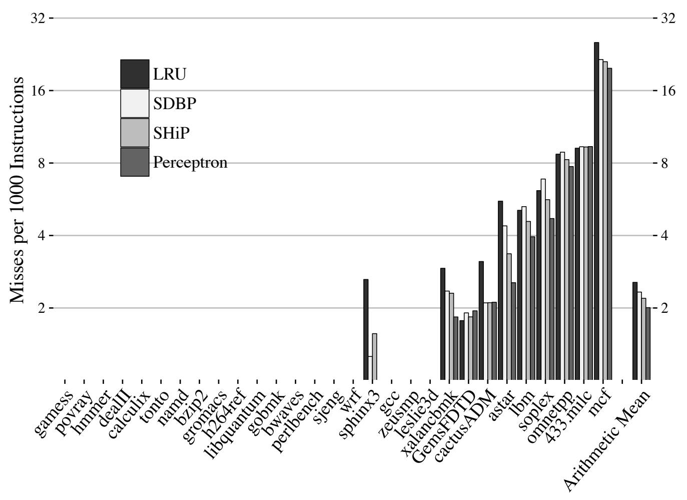


Figure 4.8: Misses per 1000 Instructions for Single-Thread Workloads.

cache, with higher efficiency represented by lighter values. The LRU cache is quite dark. SDBP and SHiP are clearly more efficient, but Perceptron is the lightest, thus it is the most efficient. On average over all blocks and 1000 multi-programmed workloads, LRU yields an efficiency of 21%, *i.e.,* at any given time 21% of blocks are live and 79% are dead. SDBP gives an average efficiency of 47% and SHiP gives an efficiency of 43%. Although SHiP outperforms SDBP, its efficiency is somewhat lower because it does not bypass, so some dead blocks inevitably enter the cache and depress efficiency. Perceptron has 54% efficiency. Thus, Perceptron more than doubles the efficiency of the cache over LRU.

4.3.7 Analysis

Perceptron learning improves accuracy over previous techniques by being able to incorporate more input features such that highly correlated features make a large contribution to the prediction while uncorrelated features make little contribution. Wider weights allow more discrimination between levels of correlation. More features allow more correlations to be found. Figure 4.10 shows the results of experiments using different weight widths

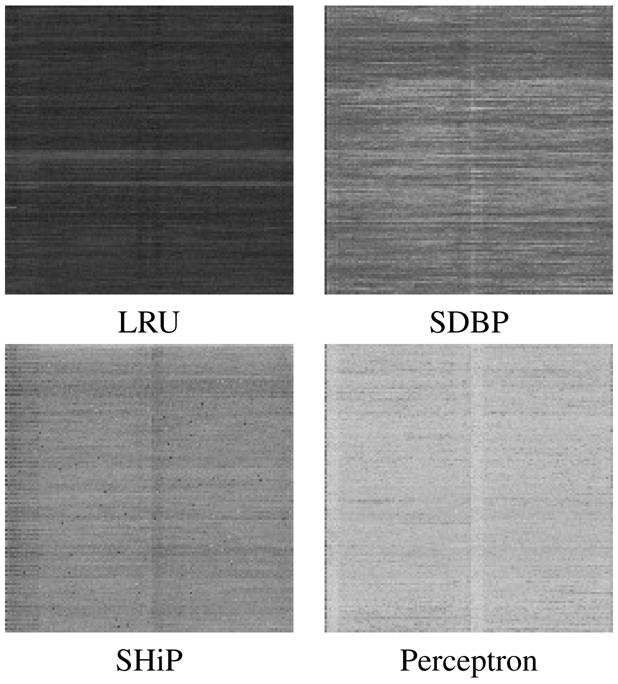


Figure 4.9: Heat map of cache efficiency in cache sets for the various techniques on a typical multi-programmed workload. Darker signifies poor efficiency while lighter means better efficiency.

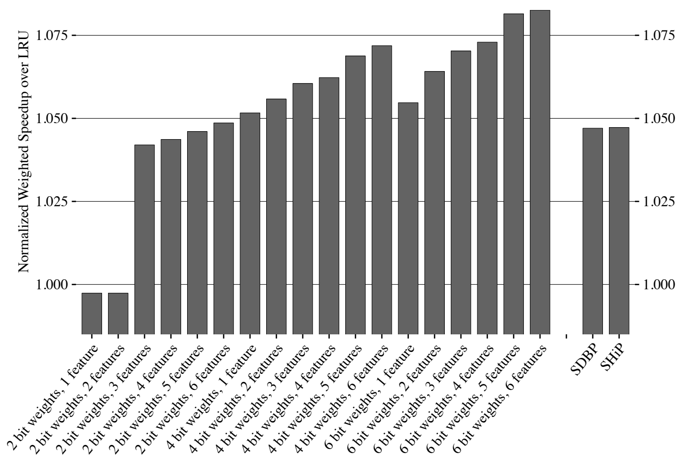


Figure 4.10: Cumulative impact of increasing numbers of features and weight widths.

and numbers of features, as well as for SDBP and SHiP for reference. The graph shows the geometric mean normalized weighted speedup over the multi-programmed workloads with a 16MB LLC. Features are added in the order they appear in Section 4.3.3. Clearly, wider weights and more features contribute to improved performance. Considering the global history of PCs and memory addresses individually allows finding correlations not apparent from any one feature or from combining features into a single index as in previous work.

4.4 Summary

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

5. LOW-COST ASSOCIATIVITY FOR DRAM CACHES

As the memory capacity needs continue to increase, on-chip DRAM caches have been proposed as a solution to mitigate the high latency of off-chip memory. With DRAM as the LLC, a whole set of new challenges arise from those of SRAM caches. For this work, we first provide studied in detail those challenges that are to be overcome, and later we revisit the idea of associativity for DRAM caches. In this chapter we present Just-in-Time Associativity (JITA), a low-cost technique to makes associativity affordable in DRAM caches.

5.1 Introduction

As the demand for main memory capacity continues to increase, memory systems consisting of multiple levels of memory are becoming increasingly popular for satisfying memory capacity needs, while at the same time providing high memory bandwidth and low memory latency in the common case. These memory systems typically consist of fast memory, possibly high bandwidth stacked DRAM, and slow memory, perhaps commodity DRAM or an alternative memory technology like Intel's 3DXpoint. The performance of these systems is highly sensitive to the organization of the cache. In cases where slow memory consists of commodity DRAM, the latency difference between slow and fast memory is likely to be small ($\sim 2X$). Here, the latency added by associativity to DRAM cache hits may outweigh the benefits of incremental miss-rate reduction, motivating the use of a direct-mapped cache organization. In contrast, where slow memory consists of alternative memory technologies such as 3DXpoint with much higher latencies ($\sim 8X$) than stacked DRAM the tradeoffs shift [60]. In this case, the high latency of DRAM cache misses outweigh the additional costs of associativity, rebalancing the tradeoffs in favor of associativity. In this chapter, we examine the role of associative DRAM caches in multi-level memories incorporating alternative memory technologies such as 3DXpoint.

Associativity has a large impact on cache performance by avoiding conflict misses. With no associativity, direct map caches suffer the most degradation in performance due to conflict misses. Section 5.2 gives data emphasizing the seriousness of this problem. Nevertheless, direct map DRAM caches give a lower access latency, as each chunk of data can only be located in one location in the cache so only that location must be probed for a tag match.

Alternatively, in a set associative design, blocks can be mapped any of W blocks in a set, where W is the associativity. Set associativity provides flexibility to reduce conflict misses. The key drawback to associativity is that, when searching through the cache for requested data, each possible location in the set must be searched to determine whether the data is allocated in the cache.

The impact of associativity on performance is not always clear. The performance of a cache is usually specified by the following equation:

$$P = (hit\_ratio * hit\_latency) + ((1 - hit\_ratio) * miss\_latency)$$

Since a hit in a direct-mapped cache is usually faster than in a set-associative cache, it is possible that the average memory access time may be lower even if the miss ratio is higher. This is often the case for DRAM caches [46]. The main source of high access latency for set-associative DRAM caches is the serialization of tag and data access. When there is an access to the cache, all tags for each of the blocks in the corresponding set must be read and compare to the tag of the requested data. If there is a match, then the location of the data is determined and data is read. In the case of set associative SRAM caches, the latency of tags and data serialization is reasonable. The performance gains obtained from significantly higher hit ratio from a direct map cache outweigh the higher hit latency.

However, this is not be the case for DRAM caches. Design decisions in DRAM caches can quickly aggravate the already high access latency of a DRAM. Consequently, prior work has prioritized optimizing access latency rather than hit ratios for DRAM caches [46].

For a DRAM cache, both access latency and hit ratio are strongly influenced by the cache organization, specifically by the following two factors: *tag placement*, *associativity*

From the access latency perspective, ideally all the DRAM cache tags (and other metadata) would be kept on-chip in SRAM arrays. On a DRAM cache access, the SRAM store could quickly provide information about whether and where the requested data is available on the DRAM cache. However, since DRAM caches are expected to have capacities in the range of multiple GBs, the area and latency overhead of an on-die metadata store is too high. For example, for a 2GB DRAM cache with 64-byte cache blocks and assuming 52-bit physical addresses, the metadata store would require at least 96 MB. Consequently, most prior DRAM cache work assumes that metadata is stored in the DRAM cache itself. For example, in the previously proposed Loh-Hill cache [47], tags and data for the same cache set are co-located in a DRAM row. Specifically, a 2KB DRAM cache row contains 29 64-byte data blocks and 3 64-byte tag blocks for one 29-way DRAM cache set. On a DRAM cache hit, the row is opened to read the tags first and then the tag comparison indicates the column number for the matching data way. The key drawback of this organization is that the tag and data accesses are serialized, resulting in long latency for cache hits.

To address the hit latency problems of set-associative DRAM caches, Qureshi *et al.* proposed Alloy Cache[46] that uses a direct-mapped organization and co-locates the tag and data for each set in a contiguous 72B chunk within a row. On each cache access, the metadata and data are read together in one DRAM access. Since this is a direct-mapped cache, one access would suffice to determine the cache hit/miss status and to read the data for a hit. Furthermore, unlike the Loh-Hill cache, since multiple cache sets are mapped to the same DRAM row, streaming accesses to consecutive sets result in row hits. Both of these factors contribute to a lower hit latency relative to Loh-Hill cache. However, the direct-mapped design of Alloy cache can result in a large number of conflict misses.

In light of the above two extreme design choices, we observe that there is a trade-off between the DRAM cache hit rate and the row buffer hit rate. Increasing the associativity increases the cache hit ratio but reduces the number of sets mapped to the same DRAM row, potentially reducing the row buffer hit rate. Conversely, a lower associativity may increase the row buffer hit rate but may also end up increasing the number of cache misses. We illustrate this trade-off, and show the impact of changing associativity on cache hit ratio (Figure 5.1) and row buffer hit ratio (Figure 5.2). Based on this data, we conclude that an associativity of 4 to 8 provides a good balance between achieving a high cache hit ratio and a high row buffer hit rate.

In this chapter, we analyze the trade-offs between set-associative and direct-mapped DRAM cache organizations. While prior work has focused on choosing one of the above and exploiting its advantages, we propose a low-cost novel technique, Just-In-Time Associativity (JITA) achieves the best of both alternatives. Just as in a direct-mapped cache, JITA attempts to map every incoming block to a distinct position in the cache, enabling low hit latency for subsequent accesses. We refer to these positions as "direct-mapped" or *static* positions. However, when there is a possible conflict miss, JITA dynamically predicts whether it is advantageous to continue to allocate the new block into its static position, potentially causing an eviction to a "hot" (repeatedly referenced) block currently residing in that position; or to steer the incoming block to a new location as in a traditional set-associative cache and thus preserving multiple useful blocks and reducing the miss ratio. This re-mapping of blocks away from their static positions affects the access latency of subsequent references to this data, but this will only be done when a conflict miss would otherwise occur. Avoiding such a miss is more important than optimizing for

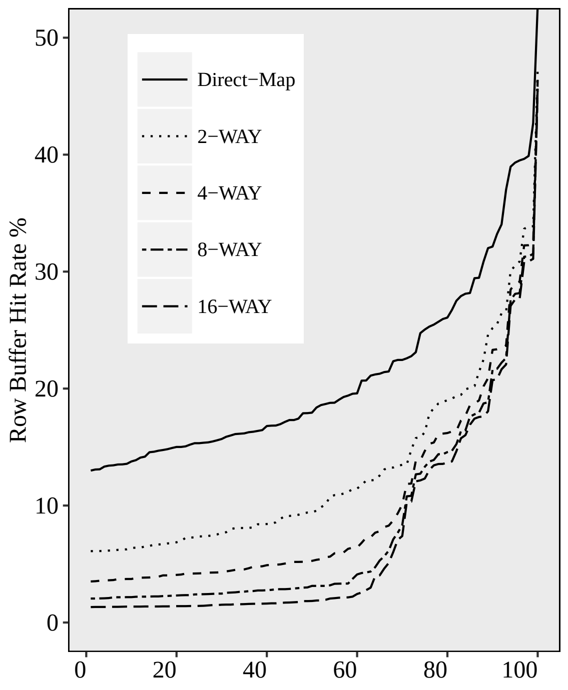


Figure 5.1: Row-buffer hit rates for one hundred randomly chosen eight-core mixes. The mixes on the x-axis are sorted in increasing order of row-buffer hit rate

low access latency. Our evaluation shows that, when the avoided conflict misses are not the common case, JITA delivers the low access latency benefit of a direct-mapped cache with a negligible performance loss.

5.2 Motivation

5.2.1 Importance of Associativity

The role of associativity in DRAM caches continues to be a subject of debate, particularly in light of the high cost and complexity of implementing associativity with commodity DRAMs. In this section we provide some intuition behind the importance of associativity. We illustrate this in the figure below. We simulate a 64MB cache with a variety of associativities when running a workload that repeatedly scans a 32MB (2^{19} cache lines)

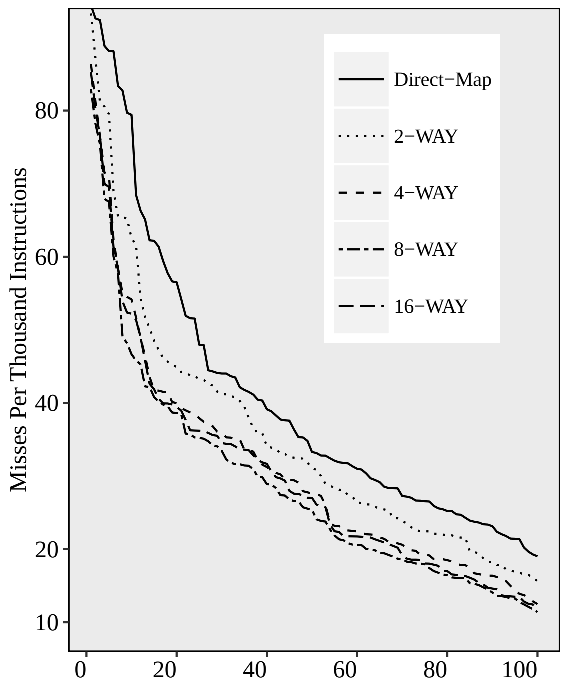


Figure 5.2: Misses per thousand instructions for one hundred randomly chosen eight-core mixes. The mixes on the x-axis are sorted in increasing order of MPKI

working set. Each of the blocks referenced by the workload is essentially random, exhibiting no spatial locality. However, since the 32MB data structure is scanned repeatedly, and the 32MB data structure is smaller than the 64MB capacity of the cache there is a great deal of temporal locality in this reference stream. In Figure 5.3, when the associativity is 1 (direct-mapped) the miss-rate of the cache (solid line) is 40%, despite residing in a cache with twice the capacity of the working set. We see the reason behind this high miss-rate in the total percentage of lines used (dotted line), which shows that only 4% of the cache lines in the cache are used. This is in contrast to the 50% that would be used if an ideal mapping function could be achieved. As associativity increases, the percentage of used cache lines increases and the miss rate decreases until an associativity of 32 completely eliminates the conflict misses, eliminating cache misses and increasing the number of used cache lines to 50%. Figure 5.4 illustrates the same trend, this time on a much larger, 256MB cache, 8X the size of the working set. Here, the miss rate for the direct-mapped cache is much lower but still fairly high at around 12%.

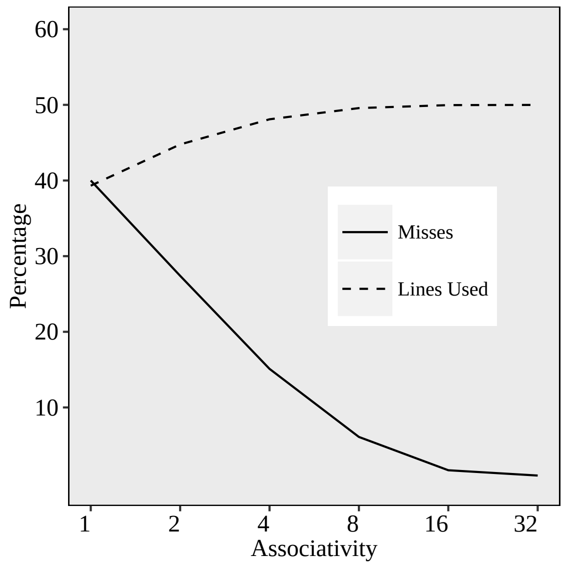


Figure 5.3: Percentage of accesses that miss in the cache, as well as the percentage of total lines in the cache that were used for a 64MB cache running a workload that repeatedly scans a 32MB working set.

We gain two insights from considering this experiment. First, evenly distributing cache lines across the cache sets is difficulty even when the addresses are randomly distributed and the working set is significantly smaller than the cache capacity. Increasing capacity is not a cost effective way of reducing misses caused by conflicts. Second, if hot sets can occur when addresses are random, improved indexing functions are also unlikely to solve this problem. In fact, associativity seems to be the only effective mechanism to solve this problem. However, most sets require no additional associativity. In fact, we have found that adding an additional way to 20% of the sets in a direct-map cache is sufficient to improve reduce the miss-rate to that of a 2-way set associative cache, we illustrate this in figure 5.5. This is the general motivation behind our proposed JITA cache design.

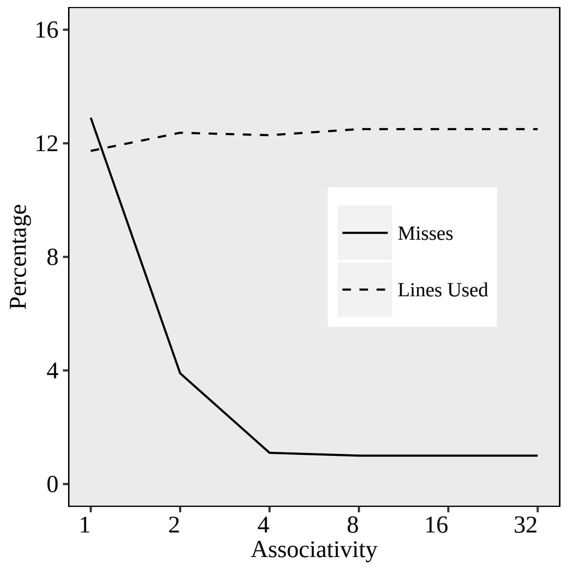


Figure 5.4: Percentage of accesses that miss in the cache, as well as the percentage of total lines in the cache that were used for a 256MB cache running a workload that repeatedly scans a 32MB working set.

5.3 Design

5.3.1 Just-In-Time Associativity

Our proposal, the Just-In-Time-Associativity (JITA) cache provides the flexibility of acting as a direct mapped cache when conflict misses are rare and as a set associative cache when they are not. The JITA cache achieves this by exploiting the flexibility available in modern cache insertion policies. In a traditional Least Recently Used (LRU) replacement algorithm, only a single cache block (the LRU block) is a candidate for replacement. How-

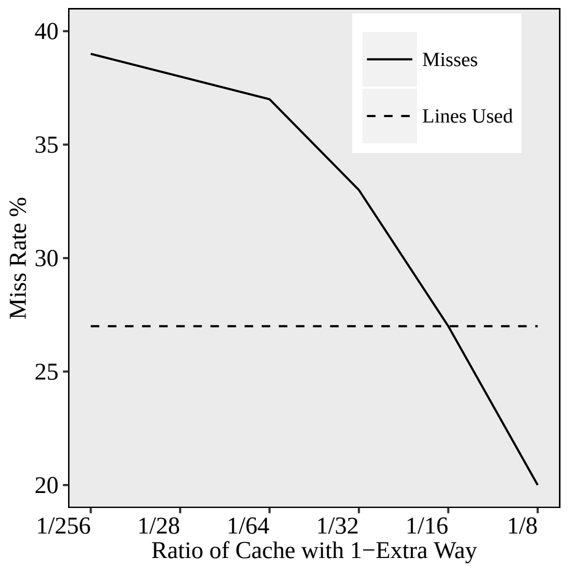


Figure 5.5: Percentage of the cache where we need associativity.

ever, most modern replacement algorithms implement an approximation to LRU, where they group several cache blocks in a set in a recency category. For these policies, many cache blocks belong in an LRU equivalence class. They might not be the last block that was referenced, but instead they are one of the many blocks that have not been referenced recently. All of these blocks serve as potential replacement candidates. The JITA cache exploits the flexibility to steer newly allocated lines to their *static* locations. The goal is that under a workload behavior that experiences rare conflict misses JITA would map most cache lines to their one-to-one mapping, just as it would in a low access latency direct map cache design.

The Main Idea

The DRAM cache controller is responsible for choosing where to place the data requested from main memory. The flexibility that the controller has in choosing this mapping is determined by the associativity of the DRAM cache. A fully-associative cache allows newly allocated data to be placed anywhere in the cache, an 8-way set associative cache may use 1 of 8 locations, and on a direct-mapped cache the hardware may use only one location to place this data. When looking for a previously allocated data in the cache, each of the possible mapping locations for the data must be examined to determine whether the data resides in the cache. This is done by performing a tag-search in which the address of the requested data is compared with the tag of the data stored in each possible mapping, to determine if and where the data is located. Generally, high associativities reduce the probability of conflicts misses, which translates into lower miss ratios, hence better performance.

However, the pre-requisite to perform a tag-search before accessing the data entails a significant increase in cache access latency. An ideal DRAM cache would act as a direct-mapped cache all the times associativity is not required, and revert to a set-associative cache when appropriate. Our proposal, Just-In-Time-Associativity (JITA) cache provides this flexibility. JITA achieves this by exploiting flexibility available in modern cache insertion policies. The JITA cache exploits the flexibility to preferentially insert newly allocated cache blocks into locations that correspond to their *static* locations. When these lines are subsequently requested by the processor, we are able to quickly locate them by inferring their *static* position from the requested address. Requests that reside in their *static* position are consider low access latency accesses, hereafter *fast hits.* Their locations can be successfully predicted prior to a tag access. When running workloads with relatively smaller and regular working sets in which conflicts are rare, JITA will succeed in mapping a vast majority of the cache lines to their *static* locations. However, when running workloads with larger and more irregular working sets, allocating all the cache lines in their direct-mapped position would suffer from a large number of conflict misses. In such scenarios, JITA exploits the flexibility provided by associativity to map lines to other available locations, thereby minimizing misses. If any of these re-mapped blocks is subsequently accessed, their location within a set cannot be inferred directly from the request address and would require a tag search. We refer to these request as *slow hits*. Thus, when compared to a direct mapped cache, such as Alloy cache, a JITA cache trades off expensive conflict misses for relatively cheaper *slow hits*. The relative latencies of *slow hit vs.* a cache miss depends on the main memory technology. For a DRAM-based main memory like the one assumed in Alloy cache work[46], the miss latency is similar to the hit latency. However, a main memory solution built with emerging memory technologies, such as PCM or 3DXpoint may be 4X to 10X slower than the DRAM cache hit latency [60]

5.3.2 JITA Cache Organization

The JITA cache is organized like a conventional set-associative DRAM cache. Similar to the Loh-Hill cache, all tags and data associated with each way in a set are contained within the same DRAM row. However, unlike the Loh-Hill cache, a single row contains multiple cache sets. The number of cache sets mapped to a single row is determined by the associativity, cache block size and the DRAM row size. For example, Figure 5.7 shows a JITA cache organization for an 8-way DRAM cache with 64B blocks and a DRAM row size of 2KB. In this case, four consecutive cache sets are mapped to the same DRAM row. For each set, there is a 64B tag block that contains the tags for all the ways followed by 7 contiguous 64B. In this organization, the location of tag blocks for each set can be statically determined from the set number. Furthermore, given a way number within a set, the location of the data block can also be statically determined.

5.3.3 Accessing the JITA Cache

When a request arrives at the DRAM cache controller, a subset of the address bits is first used to find the set number corresponding to this address. This set number is used to identify the exact DRAM row mapped to this address. Moreover, another subset

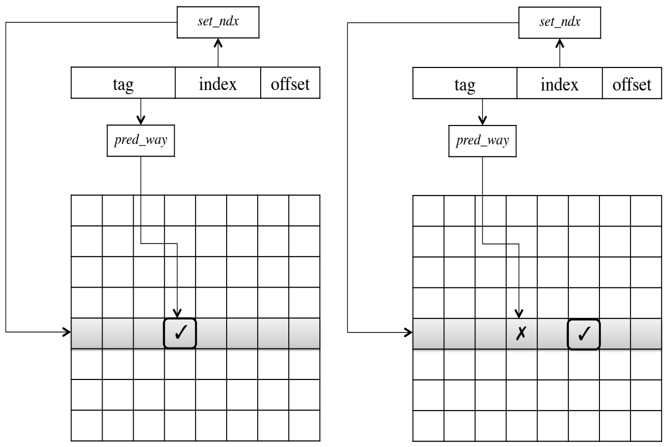


Figure 5.6: On the left, we illustrate a fast hit, the requested data is in fact is in the predicted way. To right of the figure we can observe a slow hit, the requested data is no located on its corresponding static position, but does reside and in the cache and is accessed paying a somewhat more costly access time.

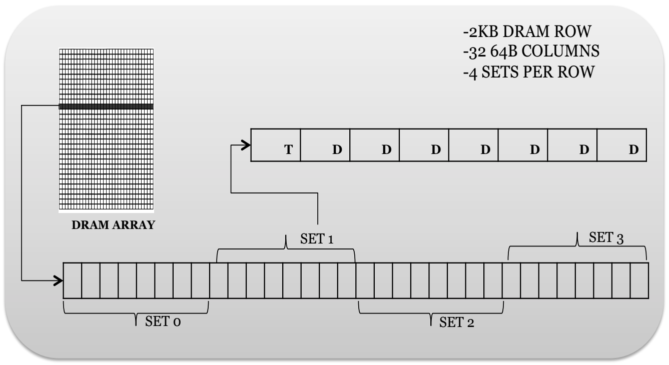


Figure 5.7: 2KB DRAM row containing four cache sets, for each set, we allocate its corresponding tags in a 64B block, followed by seven 64B blocks filled with data

of address bits derived from the tag bits is used to predict the way number within the set which contains the requested cache line. We call this the *static* mapping for the address. As a simple example, for an 8-way set-associative cache, the static way number may simply be determined by the three least significant bits in the tag. This process is illustrated in Figure 5.6, where $pred\_way$ refers to the *static* mapping:

Once the set number set_ndx and the static way number $pred\_way$ are known, the DRAM cache controller generates corresponding row and column addresses and sends two back-to-back read requests to the DRAM cache. The first request is for the tag block which contains all the tags in the set and the second request is for the data block residing in the statically predicted way. Since the exact column numbers for both these requests are already known, the CAS command for the second request can be sent in parallel with the data burst for the first request, resulting in consecutive back-to-back bursts of the tag and data blocks from DRAM cache. Such back-to-back accesses would not have been possible in a traditional set-associative DRAM cache (such as the Loh-Hill cache), since the generation of the second CAS request would first require the tag block to be read from the DRAM and compared with the tag for the requested address. After receiving the tag block, the JITA controller compares the tag for the statically predicted way number with the tag obtained from the requested address. The outcome of this comparison can lead to one of the following three cases: (i) *Fast Hits*, (ii) *Slow Hits*, (iii) *Misses*.

Fast hits: If the tag in pred_way matches the tag from the requested address, then this is a fast cache hit, illustrated in Figure 5.6. The data is retrieved from the cache much faster than in a traditional set-associative cache.

Slow Hits: When a block resides in the cache, but is not located on its static position, we refer to this, as a *slow hit*. To pinpoint the location of the block, the JITA controller compares the address tag with the tags stored for all the other ways in the set, similar to what a traditional set-associative cache does for each cache access. The comparison can be done in parallel with when the DRAM cache is sending the data burst for the $pred\_way$ data block. If the JITA controller finds out that the requested address tag matches the stored tags for one of the ways (different from $pred\_way$), then it calculates the column number for the data block in that way and sends a request to read that data from the cache. We classify this request as a slow hit since it takes longer than the previously described fast hit. However, it is important to note that according to this definition, all the hits in a traditional set-associative DRAM cache would be characterized as slow hits.

Misses: If neither the statically predicted tag, nor any of the other tags within the set match the tag from the requested address, then this request is a cache miss. In that case, the cache eviction algorithm finds a victim block to be replaced by the new block.

Insertion Policy

Consider a block B_{new} that needs to be allocated into the cache on a cache miss. Assume that B_{new} maps to set i and static way number j. In order to make the allocation decision for B_{new}, JITA always prioritizes mapping B_{new} to its static position. The first step is to check the availability of position i. If the position is available (*i.e.* current block contains no valid data), B_{new} is allocated in position i, and consequently access to B_{new} will result in fast hit type of access. If position i is already occupied with valid data, JITA's replacement algorithm is consulted to determine whether B_{old}, the current residing block is likely to be referenced again. As JITA prefers to place blocks in their static position, it acts more aggressively when deciding what is consider to ready for eviction than it would in a different scenario. If, according to the JITA's algorithm B_{old} shows that it will most likely be referenced again in the near future, JITA takes advantage of the flexibility of being a set-associative cache and will re-direct B_{new} to a different available position in the cache. Figure 5.8 illustrates JITA's insertion mechanishm.

In order to avoid performance degradation by having most accesses result in slow hits, JITA has a maximum number of allowed re-map blocks per set τ_r . We first determine whether the number of currently re-mapped blocks has already reached τ_r. If this is the case, JITA searches among blocks that are not residing in their static position, *i.e. re-mapped blocks*, and as a replacement candidate the block that shows little promise of being re-referenced again before being evicted from the cache. When all blocks show future reuse, one block among all the re-mapped blocks is chosen at random.

Otherwise, if there is room for more re-mappings within the set, a replacement candidate in the set is chose by relying of JITA's replacement policy. However, this re-mapping would subsequently result in slow hits upon later accesses to B_{new}.

JITA's replacement algorithm is based on the static RRIP eviction algorithm [12], which uses a two bit re-reference interval prediction value (RRPV) to categorize a block in a reuse-distance category. When JITA makes the decision whether to evict B_{old}, the policy is less forgiving and decide to evict B_{old} if the corresponding RRPV is at least two. In other cases, JITA follows the original technique where it does only evict once the block's RRPV has saturated.

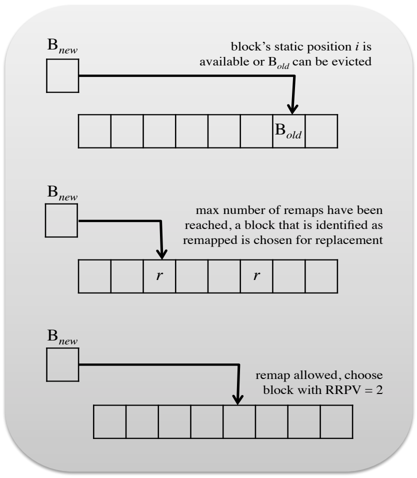

B_{new}
block's static position i is available or B_{old} can be evicted
B_{old}
B_{new}
max number of remaps have been reached, a block that is identified as remapped is chosen for replacement
r
r
B_{new}
remap allowed, choose block with RRPV = 2

Figure 5.8: JITA's insertion policy.

Summary of Latencies

To summarize the different latencies for each type of access, Figure 5.9 shows the comparison of JITA latencies to Alloy cache and the Loh-Hill cache. The fast hit latency for JITA is close to the hit latency for Alloy cache and significantly faster than that of the Loh-Hill cache. The slow hit latency for JITA is similar to the hit latency for Loh-hill cache. Note that the figure shows only the case of a row buffer hit. In the case of a row buffer miss, the activate command (ACT) latency is added to these latency numbers.

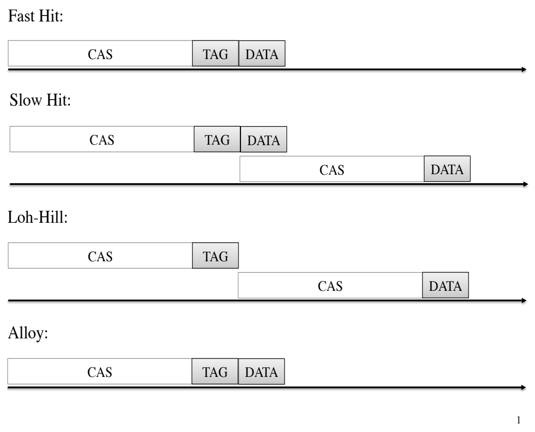


Figure 5.9: Different access latencies for each policy.

5.4 Evaluation

In this section, we evaluate the JITA cache. First, we present the evaluation methodology and compare the performance of JITA with the most closely related to our technique DRAM cache designs.

5.4.1 Methodology

We compare JITA to other proposed designs for DRAM caches, using an in-house simulator with the following memory hierarchy parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-ways. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. We model a 256MB DRAM cache. The latencies we model are the same as those for previously proposed work [46, 47]. For both the Loh-Hill cache and JITA we used an SRRIP [12] based policy, with some modifications for the JITA cache, as part of our optimization. Because of the simplicity of our proposed technique (*i.e.* does not require any extra SRAM storage, etc) we decided to compare JITA to the more related previously proposed work, the Alloy cache as a direct-map design, and the Loh-Hill as a set associative cache. We evaluated JITA as an 8-way set associative cache, where in each set, on of the ways is used to store the corresponding tags for the set.

Workloads

We use the 29 SPEC CPU 2006 benchmarks [61] as well as three server workloads from CloudSuite[62] and a machine learning workload from mlpack [63]. We use SimPoint [64] to identify up to 6 segments (i.e. simpoints) of one billion instructions each characteristic of the different program phases for the SPEC and mlpack workloads. For CloudSuite, we fast-forward at least 30 billion instructions to get past the initialization phases. In total, we use 99 segments representing 33 benchmarks.

Multi-Core Workloads

For 8-core multi-programmed workloads, we generated 100 distinct workloads consisting of mixes from the 99 segments described above. We follow the sample-balanced methodology of FIESTA [56]. We begin by selecting regions of equal standalone running time for each simpoint. Each region begins at the start of the simpoint and ends when the number of cycles in a standalone simulation reaches one billion cycles. Each work-

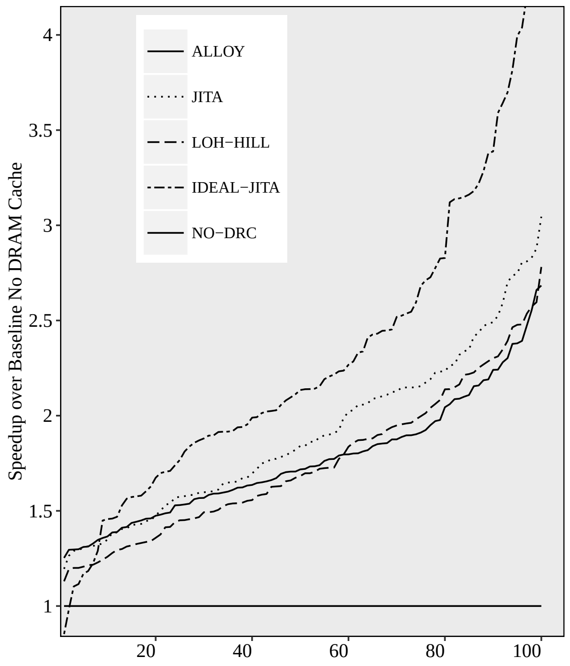


Figure 5.10: Normalized Weighted Speedup over baseline no DRAM cache for 8-Core MultiProgrammed Workloads

load is a mix of 8 of these regions chosen uniformly randomly without replacement. For each workload the simulator warms microarchitectural structures until 100 million total instructions have been executed, then measures results until each benchmark has executed for at least one billion additional cycles. When a thread reaches the end of its one billion cycle region, it starts over at the beginning. Thus, all 8 cores are active during the entire measurement period.

5.4.2 Results

Figure 5.10 shows weighted speedup for Alloy, Loh-Hill and JITA caches with 256MB normalized to no DRAM cache. A solid line across the figure represents our baseline, a memory hierarchy with no DRAM cache where every request that misses in the SRAM

LLC would have to access main memory. We propose JITA in the context of a system with a main memory solution built with emerging memory technologies, where the access latency is significantly higher. In this context, a DRAM cache is crucial to preserve system performance. An extra line in the figure was added to demonstrate the upper bound gain in performance we could obtain if we could achieve a complete set-associative DRAM cache with a like access latency as a direct map cache, we refer to this as ideal JITA. Figure 5.10 shows the speedups for each workload in ascending sorted order to yield S-curves. Loh-Hill yields a geometric mean 74% speedup and the Alloy cache yields a 77% speedup. JITA gives a geometric mean speedup of 87%. The superior hit-rate and the low-cost access latency of JITA gives a significant boost in performance over the other two other techniques. Note that the Loh-Hill cache achieves similar performance to the performance of the Alloy cache in the presence of a significantly more latency expensive main memory technology, such as 3DXpoint. In such scenario, we believe the DRAM cache management policy should strive to optimize hit rate, while yielding the lowest average access latency possible.

Misses

Figure 5.11 shows misses per 1000 instructions (MPKI) for various techniques sorted in descending order, i.e., worst-to-best from left-to-right, to yield S-curves. The Loh-Hill cache yields the fewest misses at an arithmetic mean 27.8 an expected results giving that associativy gets rid of most conflict misses, JITA follows very closely at an arithmetic mean of 28.4 MPKI. As discussed throughout the chapter, JITA's ability to behave like both a direct-map cache and a set-associative one, deliver a fast access latency for vast majority of the access to DRAM cache and places the technique in ahead in the competition for performance, even if Loh-Hill achieves a slightly lower MPKI ratio. As we expected, Alloy cache delivers the most misses, with an arithmetic mean of 40.7, the Alloy cache main goal was not designed to reduce number of misses, rather they focus on reducing the average access latency.

5.4.3 Analysis

Fast Hits

Fast access hits are crucial for JITA to improve performance. With our experiments, we learned that caches might achieve the full benefits associativity can provide by only having a small number of sets behave as a set associative cache. For the vast majority of applications, we learned that if we provide a cache with a 25% associativity, meaning 25% of the cache follows a set associative design, it would be sufficient to enjoy the benefits in performance coming from the high reduction in misses, without having to pay the significant cost as high access latency. Figure 5.12 illustrates the fast hits obtained for each of the 100 eight-core mixes evaluated. For a most mixes the number of fast hits is close to the total of hits. This represents those mixes that already perform well in a direct-map cache design in which conflict misses are rare. This shows that JITA does not hurt the performance of already well-performing benchmarks. For the mixes where the fast hit ratio is less than optimal, this represents the necessity for the presence of associativity. We must remember that most of the not achieved fast hits (*i.e.* slow hits) would have been a miss in a direct-map configuration.

Sensitivity to Cache Size

Figure 5.13 shows the average IPC for the 100 mixes of eight-core workloads, for both the Alloy and the JITA cache, evaluated using different cache sizes. As we increase the cache size, we learn that misses are less likely to increase.

Sequential or Parallel Access

The Alloy cache [46] proposes a simple yet effective mechanism to decide, whether to make accesses to the DRAM cache in a sequential or parallel fashion. For the scenario where the Alloy cache was proposed for, a DRAM cache backed up by DRAM, a sequen-

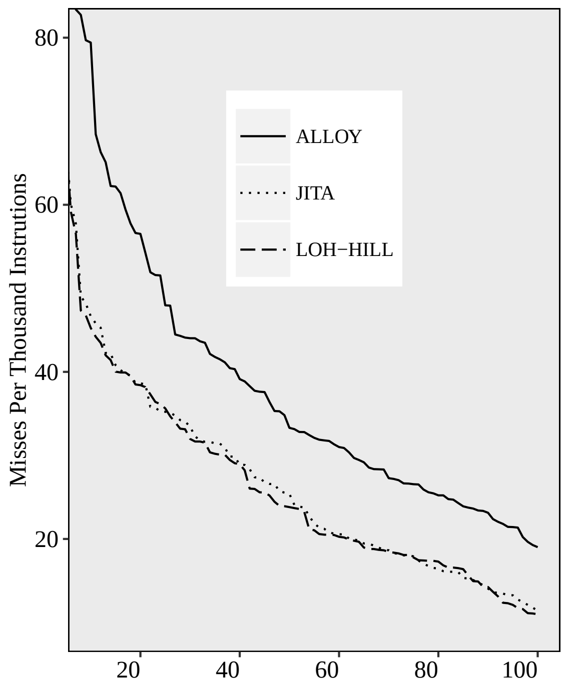


Figure 5.11: Misses per 1000 Instructions for 8-Core Multi-Programmed Workloads

tial access to the DRAM cache that ends up missing on the cache, would be be more costly than the access had been sent directly to main memory. For this, they make use of a Memory Access Predictor, to determine whether the data will be in the cache or not, and they use this prediction to decide how to access main memory. To keep the simplistic trend of this work, the results are shown with the use of no predictor, but for a fair comparison, we evaluated both the Alloy cache and the JITA cache using the MAP predictor, it boosted performance of both techniques, the trend remain the same JITA outperforming the Alloy cache by a 13% on average.

Multi-Level Cache Policy

In this work we have focused on evaluating JITA in the context of managing a DRAM cache. Efficiently implementing associativity in DRAM caches continues to be challenge

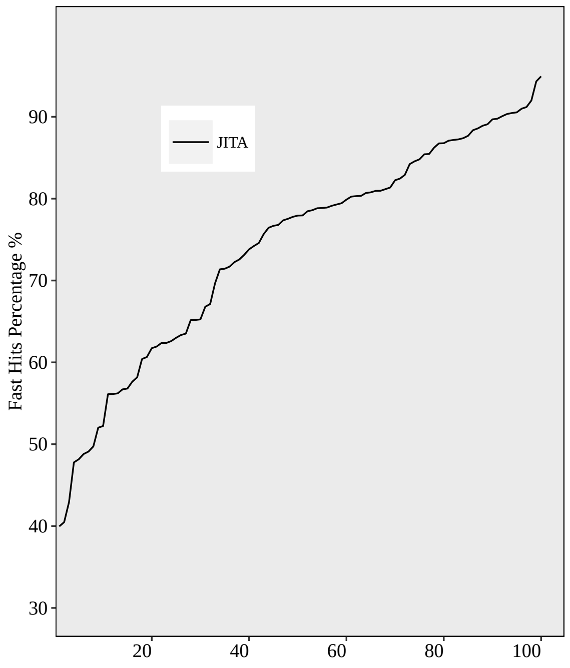


Figure 5.12: Fast hits percentage for each of the 100 eight core mixes. Percentages are sorted in an increasingly order on the x-axis

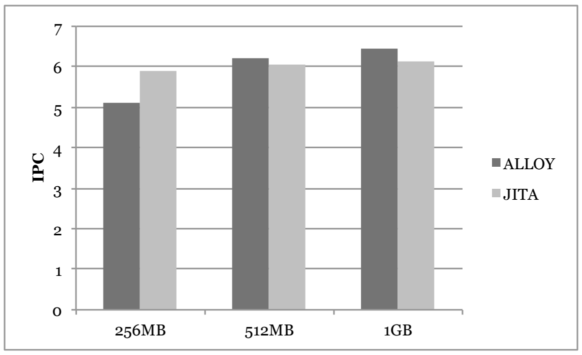


Figure 5.13: Average IPC for 100 eight-core mixes for different cache sizes.

and makes this a natural area of focus. However, in today's power sensitive CPUs, large associative L2 and L3 caches typically require the tag and data lookup to be performed sequentially. Only after the tag lookup identifies which way holds the data is the data array accessed. The concepts, described here are applicable to these caches as well and could be used to predict which way contains the desired data (way-prediction). Here, JITA would allow the data and tag lookup to proceed in parallel, eliminating the latency of the tag lookup from the overall cache latency.

5.5 Summary

This chapter discusses the necessity of achieving high hit rates as well as mitigating the high access latency DRAM caches. We present our technique JITA, a low-cost approach to associativity, that slightly modifies the mapping of data to cache locations in a set-associative cache configuration, resulting in a DRAM cache that behaves as a low access latency direct map cache for most accesses, and defaults to a set associative cache when needed to avoid conflict missed that would come at a cost with a high miss penalty. With minimal changes to the hardware, JITA is able to provide the much larger associativity for DRAM caches, with almost the same low access time of that of a truly direct map cache. JITA outperforms the state-of-the-art technique by a 13% on average.

6. SUMMARY AND CONCLUSIONS

In this chapter, we review the contributions of this dissertation.

6.1 Contributions

Recall the thesis statement from the introduction:

The performance and efficiency of last-level caches can be improved by applying principles of machine learning and other areas of research to intelligent last-level cache management policies.

Cache management has been extensively studied. With our work, we have proven that by applying some principles from other areas of research, the performance of the last-level cache can be improved. As new memory technologies are emerging, future research should focus on designing techniques that, will scale well to these new technologies, and will also efficiently manage last-level caches under the presence future applications with significantly higher memory capacity demands.

In this dissertation we have examined a number of different last-level cache organizations, and evaluated them in the context of both current and future process technologies. We have proposed a low-storage overhead policy that will gracefully scale with future increasing memory capacity such as MDPP, as well as enabling more complex policies such as Perceptron to maximize the efficiency of more critical capacity memories. Following the future trends in technology, we also presented JITA, a technique to enable associativity in higher access latency memories at an accesible latency cost.

We first introduce MDPP, a low-storage overhead replacement policy. MDPP follows the minimal disturbance principle found in neural networks research, which helps solve the problem that the highly cost effective tree-based PseudoLRU represents, when a block is promoted, positions for all non-referenced blocks are significantly altered, and very often are placed in non-favorable positions even when showing high locality. MDDP performs as well as the state-of-the-art policies found in the literature at a fraction of the cost.

We have introduced a reuse predictor that uses neural networks —the perceptron in particular —as the basic prediction mechanism. Perceptrons are attractive because they can use long history lengths without requiring exponential resources. A potential weakness of perceptrons is their increased computational complexity when compared with other simpler schemes, but we have shown how a perceptron predictor can be implemented efficiently. The perceptron predictor achieves a lower misprediction rate than other similar proposed work.

Finally, we introduce a technique to be able to choose a set-associative design in terms of a DRAM cache, at an affordable latency-access cost. First we studied the tradeoff between performance benefits and cost of different degrees of associativity. With then add some simple extra steps to the traditional way of mapping blocks to the cache, to allow the DRAM cache to act as a low-access latency direct-map cache when possible, and default to a set-associative cache when necessary to optimize for hit rate. We evaluated our work in respect to new emerging technologies such as 3DXpoint, which creates a new set of challenges for DRAM caches, and ends the assumption that it is okay not to optimize for hit rate for such scenarios.

In summary, in this dissertation, we have improved last-level cache performance, by rethinking current cache management technique, and incorporating machine learning ideas to improve on such techniques effectiveness, accuracy, and overall performance. Our design philosophy is extensible to current memory system, and it also presents better opportunities for future memory technologies.

REFERENCES

[1] A. chow Lai, C. Fide, and B. Falsafi, “Dead-block prediction and dead-block correlating prefetchers,” in *In Proceedings of the 28th International Symposium on Computer Architecture*, pp. 144–154, 2001.

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

[3] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, “Reducing set-associative cache energy via way-prediction and selective direct-mapping,” in *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pp. 54–65, 2001.

[4] 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.

[5] A. Seznec, “A case for two-way skewed-associative caches,” in *ISCA ’93: Proceedings of the 20th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 169–178, 1993.

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

[7] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 364–373, 1990.

[8] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.

[9] D. A. Jiménez, “Dead block replacement and bypass with a sampling predictor,” in *Proceedings of the 1st JILP Workshop on Computer Architecture Competitions (JWAC-1): Cache Replacement Championship (to appear)*, June 2010.

[10] S. M. Khan and D. A. Jiménez, “Insertion policy selection using decision tree analysis,” in *Proceedings of the 1st JILP Workshop on Computer Architecture Competitions (JWAC-1): Cache Replacement Championship (to appear)*, June 2010.

[11] S. M. Khan, D. A. Jiménez, B. Falsafi, and D. Burger, “Using dead blocks as a virtual victim cache,” September 2010.

[12] A. Jaleel, K. Theobald, S. S. Jr., and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA-37)*, June 2010.

[13] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. S. Jr., and J. Emer, “Adaptive insertion policies for managing shared caches,” in *Proceedings of the 2008 International Conference on Parallel Architectures and Compiler Techniques (PACT)*, September 2008.

[14] S. M. Khan and D. A. Jiménez, “Insertion policy selection using decision tree analysis,” in *In Proceedings of the 28th IEEE International Conference on Computer Design (ICCD-2010)*, October 2010.

[15] G. Keramidas, P. Petoumenos, and S. Kaxiras, “Cache replacement based on reuse-distance prediction,” in *In Proceedings of the 25th International Conference on Com-*

puter Design (ICCD-2007), pp. 245–250, 2007.

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

[17] S. M. Khan, Z. Wang, and D. A. Jiménez, “Decoupled dynamic cache segmentation,” in *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA-18)*, February 2011.

[18] J. D. Kron, B. Prumo, and G. H. Loh, “Double-dip: Augmenting dip with adaptive promotion policies to manage shared l2 caches,” in *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, 2008.

[19] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, “Improving cache management policies using dynamic reuse distances,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’12, (Washington, DC, USA), pp. 389–400, IEEE Computer Society, 2012.

[20] D. Sanchez and C. Kozyrakis, “The zcache: Decoupling ways and associativity,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’43, (Washington, DC, USA), pp. 187–198, IEEE Computer Society, 2010.

[21] G. H. Loh, “Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy,” in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 201–212, ACM, 2009.

[22] D. A. Jiménez, “Insertion and promotion for tree-based pseudolru last-level caches,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pp. 284–296, 2013.

[23] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 235–246, 2012.

[24] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 247–257, 2012.

[25] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting compressed block size as an indicator of future reuse.," in *HPCA*, pp. 51–63, IEEE, 2015.

[26] C.-C. Huang and V. Nagarajan, "Atcache: reducing DRAM cache latency via a small sram tag cache," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 51–60, ACM, 2014.

[27] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 404–415, ACM, 2013.

[28] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer, "Adaptive insertion policies for high performance caching," in *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, ACM, 2007.

[29] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for mlp-aware cache replacement," in *ISCA '06: Proceedings of the 33rd annual international symposium*

on Computer Architecture, (Washington, DC, USA), pp. 167–178, IEEE Computer Society, 2006.

[30] A.-C. Lai and B. Falsafi, “Selective, accurate, and timely self-invalidation using last-touch prediction,” in *International Symposium on Computer Architecture*, pp. 139 – 148, 2000.

[31] A.-C. Lai, C. Fide, and B. Falsafi, “Dead-block prediction & dead-block correlating prefetchers,” *SIGARCH Comput. Archit. News*, vol. 29, no. 2, pp. 144–154, 2001.

[32] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi, “Memory coherence activity prediction in commercial workloads,” in *WMPI ’04: Proceedings of the 3rd workshop on Memory performance issues*, (New York, NY, USA), pp. 37–45, ACM, 2004.

[33] 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.

[34] J. Abella, A. González, X. Vera, and M. F. P. O’Boy le, “Iatac: a smart predictor to turn-off l2 cache lines,” *ACM Trans. Archit. Code Optim.*, vol. 2, no. 1, pp. 55–77, 2005.

[35] H. Liu, M. Ferdman, J. Huh, and D. Burger, “Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency,” in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, (Los Alamitos, CA, USA), pp. 222–233, IEEE Computer Society, 2008.

[36] P. Michaud, A. Seznec, and R. Uhlig, “Trading conflict and capacity aliasing in conditional branch predictors,” in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 292–303, June 1997.

[37] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, J. Simon C. Steely, and J. Emer, "SHiP: Signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 430–441, ACM, 2011.

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

[39] G. H. Loh and D. A. Jiménez, "Reducing the power and complexity of path-based neural branch prediction," in *Proceedings of the 2005 Workshop on Complexity-Effective Design (WCED'05)*, pp. 28–35, June 2005.

[40] A. Seznec, "Genesis of the o-gehl branch predictor," *Journal of Instruction-Level Parallelism (JILP)*, vol. 7, April 2005.

[41] D. Tarjan, K. Skadron, and M. Stan, "An ahead pipelined alloyed perceptron with single cycle access time," in *Proceedings of the Workshop on Complexity Effective Design (WCED)*, June 2004.

[42] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked dram cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 25–37, IEEE Computer Society, 2014.

[43] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but effective heterogeneous main memory with on-chip memory controller support," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society, 2010.

[44] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 247–257, IEEE Computer Society, 2012.

[45] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "Chop: Integrating dram caches for cmp server platforms," *IEEE micro*, vol. 31, no. 1, pp. 99–108, 2011.

[46] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 235–246, IEEE Computer Society, 2012.

[47] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 454–464, ACM, 2011.

[48] Z. Wang, D. Jimenez, T. Zhang, G. H. Loh, and Y. Xie, "Building a low latency, highly associative dram cache with the buffered way predictor," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 51–60, ACM, 2014.

[49] J. Handy, *The Cache Memory Book.* Academic Press, 1993.

[50] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob, "CMP$im: A pin-based on-the-fly single/multi-core cache simulator," in *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2008)*, June 2008.

[51] A. R. Alameldeen, A. Jaleel, M. Qureshi, and J. Emer, “1st JILP workshop on computer architecture competitions (JWAC-1) cache replacement championship.” http://www.jilp.org/jwac-1/.

[52] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, “Using simpoint for accurate and efficient simulation,” *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 318–319, 2003.

[53] G. H. Loh and M. D. Hill, “Efficiently enabling conventional block sizes for very large die-stacked dram caches,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 454–464, ACM, 2011.

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

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

[56] A. Hilton, N. Eswaran, and A. Roth, “FIESTA: A sample-balanced multi-program workload methodology,” in *Workshop on Modeling, Benchmarking and Simulation (MoBS)*, June 2009.

[57] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja, “Sparc t4: A dynamically threaded server-on-a-chip,” *IEEE Micro*, vol. 32, no. 2, pp. 8–19, 2012.

[58] A. Fog, “The Microarchitecture of Intel, AMD, and VIA CPUs.” http://www.agner.org/optimize/microarchitecture.pdf, 2014.

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

[60] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative,” in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 2–13, ACM, 2009.

[61] “Standard Performance Evaluation Corporation CPU2006 Benchmark Suite..” `http://www.spec.org/cpu2006/`.

[62] 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 *ACM SIGPLAN Notices*, vol. 47, pp. 37–48, ACM, 2012.

[63] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, “MLPACK: A scalable C++ machine learning library,” *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.

[64] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, “Using simpoint for accurate and efficient simulation,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 318–319, ACM, 2003.