INTELLIGENT CACHE MANAGEMENT TECHNIQUES

FOR REDUCING MEMORY SYSTEM WASTE

APPROVED BY SUPERVISING COMMITTEE:

____
Daniel Jiménez, Ph.D., Chair

____
Hugh Maynard, Ph.D.

____
Dakai Zhu, Ph.D.

____
Clint Whaley, Ph.D.

____
Doug Burger, Ph.D.

Accepted: ________________________________
Dean, Graduate School

Copyright 2012 Samira M Khan
All Rights Reserved

DEDICATION

To the most courageous and independent woman in my life, my grandmother.

INTELLIGENT CACHE MANAGEMENT TECHNIQUES FOR REDUCING MEMORY SYSTEM WASTE

by

SAMIRA M. KHAN, B.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
August 2012

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my most sincere gratitude to my advisor Dr. Daniel A. Jiménez. His knowledge, constant guidance, and careful insights have helped me to become an independent researcher. This dissertation would not have been possible without his help, motivation, and support. I would also like to thank Dr. Doug Burger who has inspired and encouraged me from the very first day I met him. I am also thankful to Dr. Babak Falsafi for making me part of his wonderful research group. I would like to thank my mentors at Intel Corporation, Chris Wilkerson and Alaa Alameldeen. They taught me the importance of thinking about the big picture of a research topic. I would also like to thank my dissertation committee members, Dr. Whaley, Dr. Zhu, and Dr. Maynard, for their guidance and comments for improving this dissertation. I am eternally indebted to my parents who have stimulated the love of science in me from the very early stage of my life. I sincerely hope that I can make them proud. I also owe my deepest gratitude to my sister, Fariba, whose footsteps I have always been following. Last but not the least, I would like to thank my husband, Omar, for always being there for me and for inspiring me to pursue my dreams.

August 2012

INTELLIGENT CACHE MANAGEMENT TECHNIQUES FOR REDUCING MEMORY SYSTEM WASTE

Samira M. Khan, Ph.D.
The University of Texas at San Antonio, 2012

Supervising Professor: Daniel A. Jiménez, Ph.D.

The performance gap between modern processors and memory is a primary concern for computer architecture. Caches mitigate the long memory latency that limits the performance of modern processors. However, modern chip multiprocessor performance is sensitive to the last-level cache capacity and miss latency. Unfortunately, caches can be quite inefficient. On average, 86.2% of the blocks in a 2MB last level cache are useless. These blocks are *dead* as they will not be referenced again before eviction. These dead blocks are a waste of valuable cache space that should contain useful blocks that will contribute to the hit rate and improve performance.

This dissertation explores the inefficiencies in the memory system and proposes simple cache management techniques that reduce memory system waste and improve performance. We propose dead block cache management techniques that reduce dead time and improve performance. We introduce a new dead block predictor that can accurately identify dead blocks by sampling only a small fraction of memory references. This predictor learns from a few cache sets, reducing the predictor power and storage overhead. It also decouples the replacement policy from prediction, so it can improve performance even with the inexpensive random cache replacement policy. We propose a new cache management scheme to use dead blocks efficiently. We propose placing victim blocks in the predicted dead blocks of the cache. When the victim blocks are referenced again, they are found in the dead blocks. This "virtual victim cache" improves performance by avoiding misses. We also propose a dynamic cache segmentation technique that reduces dead time of dead-on-arrival blocks. This segmentation attempts to keep the best number of non-referenced and referenced blocks in cache sets. Dynamic cache segmentation even with a default random policy can outperform LRU using half the space overhead.

TABLE OF CONTENTS

Acknowledgements iii

Abstract iv

List of Tables x

List of Figures xi

Chapter 1: Introduction 1

1.1 Focus of the Work 1

1.2 Source of Memory System Waste 2

1.3 Reducing Memory System Waste 4

1.4 Contributions 5

Chapter 2: Related Work 7

2.1 Cache Basics 7

2.2 Dead Block Prediction 8

2.2.1 Trace Based Predictor 8

2.2.2 Time Based Predictor 11

2.2.3 Cache Burst Predictor 12

2.2.4 Counting Based Predictor 12

2.2.5 Other Dead Block Predictors 13

2.3 Sampling for Cache Placement and Replacement Policy 13

2.3.1 Dynamic Insertion Policy 13

2.3.2 Re-reference Interval Prediction 14

2.4 Cache Management Techniques 14

2.5 Cache Bypassing 14

2.6 Segmentation in Disk Cache 15
2.7 Shared Cache Partitioning 15

Chapter 3: Sampling Dead Block Prediction for Last-Level Caches 17

3.1 Motivation 18
3.2 Description 19
3.2.1 A Sampling Dead Block Predictor 19
3.2.2 Advantage Over Previous Predictors 20
3.3 A Dead-Block Driven Replacement and Bypass Policy 22
3.3.1 Dead Block Replacement and Bypassing with Default Random Replacement 22
3.3.2 Predictor Update in the Optimization 23
3.3.3 Multiple Cores 23
3.4 A Comparison of Predictor Storage and Power 23
3.4.1 Reference Trace Predictor 23
3.4.2 Counting Predictor 24
3.4.3 Sampling Predictor 24
3.4.4 Predictor Power 24
3.4.5 Latency 26
3.5 Experimental Methodology 26
3.5.1 Simulation Environment 26
3.5.2 Optimal Replacement and Bypass Policy 29
3.5.3 Measuring Cache Efficiency 29
3.6 Experimental Results 30
3.6.1 LLC Misses 30
3.6.2 Speedup 31
3.6.3 Poor Performance for Trace-Based Predictor 32
3.6.4 Dead Block Replacement with Random Baseline 32

3.6.5 Prediction Accuracy and Coverage . . . 33
3.6.6 Multiple Cores Sharing a Last-Level Cache . . . 34
3.6.7 Improvement in Cache Efficiency . . . 35
3.7 Using Dead Blocks . . . 35

Chapter 4: Using Dead Blocks as a Virtual Victim Cache . . . 37
4.1 Motivation . . . 38
4.2 Description . . . 39
4.2.1 Identifying Potential Receiver Blocks . . . 39
4.2.2 Placing Victim Blocks into the Adjacent Set . . . 39
4.2.3 Block Identification in the VVC . . . 40
4.2.4 Set Dueling for Placement . . . 40
4.2.5 Why Not Just Evict Predicted Dead Blocks? . . . 41
4.2.6 Implementation Issues . . . 42
4.3 Experimental Methodology . . . 43
4.3.1 Simulation Environment . . . 43
4.3.2 Simulating CMP Workloads . . . 44
4.3.3 Dead Block Predictor Details . . . 45
4.3.4 Estimating Dead Block Hit latency . . . 45
4.3.5 Measuring Cache Efficiency . . . 46
4.4 Results . . . 46
4.4.1 Reduction in L2 Misses . . . 47
4.4.2 Single-thread IPC Improvement . . . 48
4.4.3 CMP Throughput Improvement . . . 49
4.4.4 Improvement in Cache Efficiency . . . 50
4.4.5 Reduction in Cache Area . . . 51
4.4.6 Increase in Tag Array Access . . . 52

4.5 Reducing Dead Blocks without a Dead Block Predictor 53

Chapter 5: Dead Time Reduction through Cache Segmentation 54

5.1 Motivation 56

5.1.1 Addressing Workload Behavior 57

5.1.2 A Decoupled Flexible Policy 57

5.1.3 Segmenting Cache Sets with Prediction 58

5.2 Dynamic Segmented Cache 59

5.2.1 Predicting the Best Segmentation 59

5.2.2 Decoupled from Replacement Policy 61

5.2.3 Automated Bypass 62

5.2.4 Ensuring Thrashing Resistant 63

5.2.5 Mutable Policy 64

5.3 DCS with Shared Cache Partitioning 65

5.4 Experimental Methodology 66

5.4.1 Single-Thread Workloads 68

5.4.2 Multi-Core Workloads 68

5.5 Results 68

5.5.1 Effect of Dynamic Segmentation 69

5.5.2 Effect of Segment Predictor 70

5.5.3 Comparison with Other Policies 71

5.5.4 DCS with Random Policy 71

5.5.5 Space Overhead 73

5.5.6 Dynamic Segment Size 74

5.5.7 Cache Sensitivity 75

5.5.8 DCS with Shared Cache Partitioning 75

5.6 Reducing Dead Blocks with Insertion Policy 76

Chapter 6: Dead Time Reduction through Insertion Policy 77
6.1 Motivation 77
6.2 Description 77
6.2.1 Decision Tree Analysis 77
6.2.2 Adaptive policy in Leader Sets for Multi Set Dueling 78
6.2.3 Insertion Policy Selection 80
6.2.4 Storage Requirement 81
6.3 Simulation Methodology 81
6.4 Results 82
6.4.1 Comparison With a Dead Block Predictor Replacement 84

Chapter 7: Conclusion 86
7.1 Summary 86
7.2 Implication 86

Bibliography 88

Vita

LIST OF TABLES

Table 3.1 Storage overhead for the various predictors 25

Table 3.2 Dynamic and leakage power for predictor components. All figures are in Watts. 26

Table 3.3 The 29 SPEC CPU 2006 benchmarks with LLC cache misses per 1000 instructions for LRU and optimal (MIN), instructions-per-cycle for LRU for a 2MB cache, and number of instructions fast-forwarded to reach the simpoint (B = billions). Benchmarks in the subset in boldface. 27

Table 3.4 Multi-core workload mixes with cache sensitivity curves giving LLC misses per 1000 instructions (MPKI) on the y-axis for last-level cache sizes 128KB through 32MB on the x-axis. 28

Table 3.5 Legend for various cache optimization techniques. 30

Table 4.1 (a) Microarchitectural simulator parameters, and (b) dead block predictor parameters. 43

Table 5.1 Comparison among techniques 58

Table 5.2 SPEC CPU 2006 benchmarks with LLC cache misses per 1000 instructions for LRU and optimal (MIN), instructions-per-cycle for LRU for a 2MB cache, and number of instructions fast-forwarded to reach the simpoint (B = billions). 67

Table 5.3 Benchmarks grouping 67

Table 5.4 Multi-core workload mixes with cache sensitivity curves, LLC misses per 1000 instructions (MPKI) on the y-axis for LLC sizes 128KB through 32MB on the x-axis. s=sensitive, i=insensitive and f=LRU friendly 69

Table 5.5 Space overhead in a 2MB 16 way LLC 73

Table 5.6 Comparing space overhead with other policies 73

Table 6.1 Storage for 1MB 16 way cache 80

Table 6.2 Configuration 81

Table 6.3 Insertion Position Selected 82

LIST OF FIGURES

Figure 1.1 Efficiency (i.e. live time ratio) shown as greyscale intensities for 456.hmmer for a 1MB LRU cache. Darker blocks are dead longer. Efficiency is 22% . . . 2

Figure 1.2 Referenced and Non-referenced Blocks in a 1MB 16-way LLC at a certain time . . . 3

Figure 1.3 Percentage of zero reuse blocks . . . 4

Figure 2.1 Reference trace predictor example. The predictor is indexed with the hashed sum of PCs of memory access instructions to block *B* from the time *B* is placed in the L2 to the time it is evicted. . . . 9

Figure 2.2 Fields updated at reftrace predictor . . . 11

Figure 3.1 Dead block replacement and bypass bring the cache to life. Efficiency (i.e. live time ratio) shown as greyscale intensities for 456.hmmer for (a) a 1MB LRU cache and (b) a dead-block-replaced cache using a sampling predictor. Darker blocks are dead longer. Efficiency is 22% for (a) and 87% for (b). . . 17

Figure 3.2 Old dead block predictor (a), and new dead block predictor with sampler tag array (b). The sampler and dead block predictor table are updated for 1.6% of the accesses to the LLC, while the original predictor is updated on every access. . . . 22

Figure 3.3 Reduction in LLC misses for various policies. . . . 31

Figure 3.4 Speedup for various policies . . . 31

Figure 3.5 LLC misses per kilo-instruction for various policies . . . 33

Figure 3.6 Speedup for various replacement policies with a default random cache . . . 33

Figure 3.7 Coverage and false positive rates for the various predictors . . . 34

Figure 3.8 Weighted speedup for multi-core workloads normalized to LRU for a LRU cache . . . 35

Figure 3.9 Cache efficiency for dead-block replacement and bypass combined with LRU replacement. . . . 35

Figure 4.1 Virtual victim cache increases cache efficiency. Block efficiency (i.e., fraction of time block is live) shown as greyscale intensities for `456.hmmer` for (a) a baseline 1MB cache and (b) a VVC-enhanced cache; darker blocks are dead longer. Efficiency is 15% for (a) and 35% for (b). 37
Figure 4.2 (a) Placing evicted block into an adjacent partner set, and (b) hitting in the virtual victim cache 39
Figure 4.3 L2 cache misses per thousand instructions 47
Figure 4.4 IPC improvement 48
Figure 4.5 Speedup 49
Figure 4.6 Normalized throughput IPC for multiple threads and a 2MB cache 49
Figure 4.7 Shared cache contention results in more false positive predictions. 50
Figure 4.8 Cache efficiency 51
Figure 4.9 Reduction in cache area 52
Figure 4.10 Increase in tag array reads due to VVC 52
Figure 5.1 Effect of static segmentation. Row 1 shows the benchmarks with no effect of segmentation. Row 2 shows LRU friendly benchmarks and row 3 shows benchmarks which are effected by segmentation. 56
Figure 5.2 Logical view of the replacement scheme 59
Figure 5.3 Decision Tree Analysis to find the best non-referenced segment size 60
Figure 5.4 3D model of the Optimal Segment Predictor 61
Figure 5.5 Runtime non-referenced segment size for three representative sets from 483.xalancbmk 63
Figure 5.6 Three cases of Dynamic segmentation 64
Figure 5.7 Dynamic Segmentation in each partition of UCP 66
Figure 5.8 Effect of enforced thrash resistance and automated bypassing 70
Figure 5.9 Effect of Segment Predictor 71
Figure 5.10 Effect of Segment Predictor 72
Figure 5.11 Dynamic Segmentation with Random Policy 72
Figure 5.12 Runtime predicted best non-referenced segment size 74
Figure 5.13 Cache size sensitivity of Dynamic Segmentation 75

Figure 5.14 Complementing cache partitioning technique 75
Figure 6.1 Decision Tree Analysis 78
Figure 6.2 Reduction in Leader sets with adaptive policy 79
Figure 6.3 Selecting insertion policy 80
Figure 6.4 Percentage of benchmarks at each insertion position 82
Figure 6.5 MPKI reduction compared to LRU 83
Figure 6.6 Speedup over LRU 83
Figure 6.7 Speedup over LRU replacement policy 84
Figure 6.8 Speedup of over Counting Based Dead Replacement 85

CHAPTER 1: INTRODUCTION

The memory system in microprocessors is a critical component for high performance. With the increasing number of transistors available on a chip in each process generation, the processor speed is getting faster and faster. But though the memory is getting much larger, it is not getting faster. The performance gap between modern processors and memory is a primary concern for computer architecture. This memory wall causes the processor to wait hundreds of cycles for a single memory request. Processors have large on chip caches to mitigate the huge memory latency. Caches are faster but smaller on-chip structure that greatly improve system performance. Caches are an important microarchitectural component that avoids costly off-chip misses by storing blocks that will be referenced again. Caches can access a block in just a few cycles, but a miss that goes all the way to memory incurs hundreds of cycles of delay.

1.1 Focus of the Work

With the increasing number of transistors available on-chip, vendors are manufacturing large on-chip last-level caches that are shared among the cores. There is a large set of memory sensitive applications that are sensitive to last-level cache performance. These applications have large working sets that mostly do not fit in the last-level cache. Large number of memory accesses in these applications make them largely sensitive to the last-level cache capacity and miss latency.

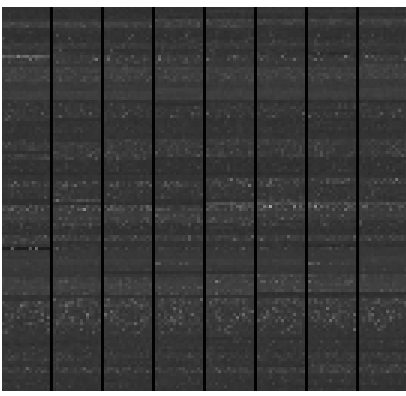
There are other set of applications that are sensitive to bandwidth rather than last-level cache. These applications have working set size that mostly fit in today's large last-level cache. However, they are constrained by the available bandwidth. These applications, along with other non-memory insensitive workloads, are example of applications where the latency of last-level cache do not have a large impact on the performance.

Thus, we focus on applications with *high memory latency sensitivity* that will benefit from intelligent cache management. These applications can gain significant performance if we can eliminate last-level cache misses and increase the number of useful blocks in the cache. For this reason, it is

very important to use the large on-chip cache capacity efficiently so that we can eliminate useless blocks and buffer important blocks in the cache that will be used again in the future.

1.2 Source of Memory System Waste

Caches in modern processors are not utilized well. They contain many blocks that will never be used again. A cache block is live if it will be referenced again before its eviction. From the last reference until the block is evicted the block is *dead* [29]. Cache blocks are dead on average 86.2% of the time over a set of memory-intensive benchmarks in a 2MB last level cache. Dead blocks lead to poor cache efficiency [5, 32] because a block may reside in the cache for a long time between the time it is last accessed and the time it is evicted. In the least-recently-used (LRU) replacement policy, after the last access, a block has to move down from the most-recently-used (MRU) position to the LRU position and then it is evicted. This process takes a long time in a highly associative cache.



Greyscale	Efficiency
	0.0
	0.1
	0.2
	0.3
	0.4
	0.5
	0.6
	0.7
	0.8
	0.9
	1.0

Figure 1.1: Efficiency (i.e. live time ratio) shown as greyscale intensities for 456.hmmer for a 1MB LRU cache. Darker blocks are dead longer. Efficiency is 22%

Figure 1.1 depicts the efficiency of a 1MB 16-way set associative LLC with LRU replacement for the SPEC CPU 2006 benchmark `456.hmmer`. The amount of time each cache block is live is shown as a greyscale intensity. The darkness shows that many blocks remain dead for long stretches of time. The inefficiency in cache wastes space which can be utilized to hold important

blocks in the cache. These dead blocks are a huge waste of space and power. These space should contain useful blocks that will contribute to the hit rate and improve performance.

One special kind of dead block is a zero reuse block. These blocks are dead right after coming to the cache. Memory accesses is filtered by L1 and L2 caches and last-level cache only gets accessed if blocks get reused after getting evicted from the L1 and L2 cache. This filtered temporal locality results in a large number of non-referenced lines in LLC. These blocks get evicted from the cache without being referenced again. Figure 1.2 shows the number of referenced and non-referenced blocks in a LLC at a particular time. It shows that in any instance of time benchmarks brought in large number of blocks that will never used again. This is an opportunity to get rid of these non-referenced blocks earlier to improve cache efficiency. Fig 1.3 shows the percentage of blocks brought to the cache that are never accessed again. On average 79% of the blocks are never used again.

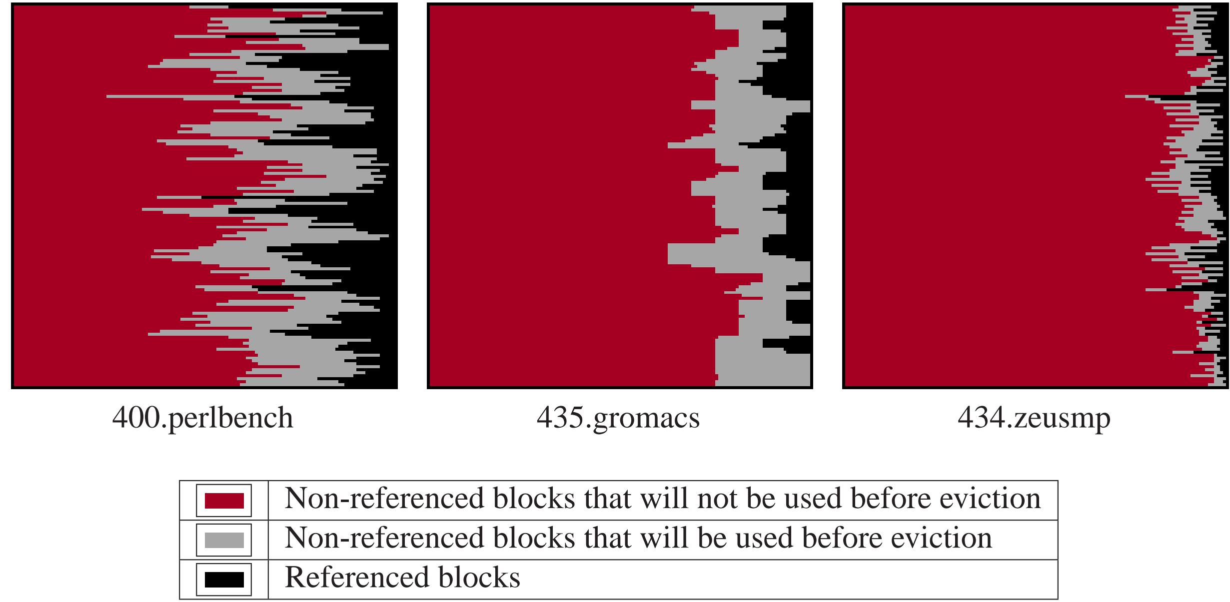


Figure 1.2: Referenced and Non-referenced Blocks in a 1MB 16-way LLC at a certain time

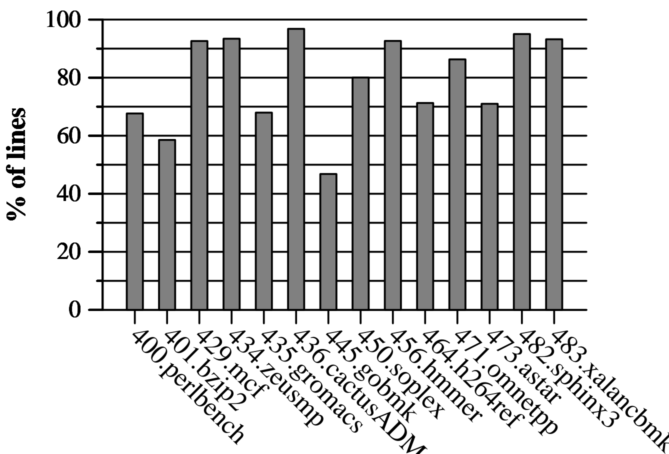


Figure 1.3: Percentage of zero reuse blocks

1.3 Reducing Memory System Waste

This dissertation describes several novel solutions to reduce waste in the last-level cache. We explore the inefficiencies in the memory system and propose simple cache management techniques that reduce memory system waste and improve performance.

Sampling Dead Block Predictor We propose a dead block prediction technique based on *sampling*. This predictor learns from the memory references accessing only a few sets. This technique decouples predictor training from cache replacement policy. We show that our sampling predictor can improve performance even with the inexpensive random cache replacement policy [23].

Dead Blocks as Virtual Victim Cache We propose using dead blocks as extra space to hold victim blocks. In this way cache performance can be improved by using predicted dead blocks to hold victims from cache evictions in other sets. The pool of predicted dead blocks can be thought of as a *virtual victim cache* (VVC) [22].

Dead Time Reduction through Segmentation We introduce an adaptive segmentation technique that dynamically adjusts the number of non-referenced and referenced blocks in the cache. Dynamic Cache Segmentation(DCS) decides whether a victim should be selected from non-

referenced blocks or referenced blocks. This way we can eliminate non-referenced blocks in workloads that do not use them [24].

Dead Time Reduction through Insertion We introduce an insertion policy that evicts the zero reuse lines earlier. This policy dynamically chooses the best the insertion position using decision tree analysis. This way dead-on-arrival blocks get evicted earlier and provides space for other useful blocks [21].

1.4 Contributions

This dissertation makes the following original contributions:

- We show that dead blocks can be predicted by sampling only a fraction of the memory references. Our sampling dead block predictor reduces the predictor power and storage overhead. Our sampling based dead block predictor learns from the LRU replacement policy in the sampler sets while the cache can deploy the inexpensive random replacement policy.

- We introduce the Virtual Victim Cache (VVC). VVC uses predicted dead blocks to hold blocks evicted from other sets. When these evicted blocks are referenced again, the access can be satisfied from the other set, avoiding a costly access to main memory. VVC improves cache performance and efficiency. We also show that VVC is robust to false positive mispredictions.

- We propose a dynamic cache segmentation technique that attempts to keep the best number of non-referenced and referenced blocks in cache sets. We propose a sampling-based scalable low-overhead technique to predict the best segmentation. We show that cache segmentation complements current shared cache partitioning techniques. Dynamic cache partitioning even with a default random policy can outperform LRU using half the space overhead.

- We introduce an insertion policy that reduces the dead time of zero reuse blocks. We propose a new scalable technique that chooses the optimal insertion position adaptively. It evicts the

dead on arrival cache blocks earlier but keeps the other useful blocks long enough to be reused.

CHAPTER 2: RELATED WORK

2.1 Cache Basics

This section summarizes some of the basic concepts of cache used throughout the dissertation.

- Cache Block: A cache block contains the actual data fetched from the main memory. Each cache block holds the tag and some status and replacement bits along with the data. The tag is higher order bits of the memory address used to look up a specific block in the cache. Status bits are used to indicate if a cache block is valid and dirty. The replacement bits are used to make replacement decisions in case of a cache miss.

- Set and Associativity:

  In a fully associative cache, a cache block can be placed in any entry of the cache. While this is a very flexible system, the extra circuitry to achieve full associativity is expensive in terms of latency and power. If a block can be placed only in one exact entry of the cache, it is a direct-mapped cache. A set associative cache divides the cache into sets, where a block mapped to a set can be placed in any entry of that set.

- Cache Misses:

  Compulsory Miss: A compulsory miss occurs when a block is accessed for the first time. A compulsory miss can not be eliminated in any cache organization.

  Capacity Miss : A compulsory miss occurs as the cache size is limited and only limited number of blocks can be buffered in the cache.

  Conflict Miss : In the case of set associative cache, a conflict miss occurs when several blocks are mapped to the same set.

- Block Insertion: When a block is brought to cache, it is placed in an entry of a specific cache set. The block insertion policy determines the replacement state of an incoming block.

- Block Replacement: Cache replacement policy decides which cache block should be evicted from a cache set to make space for the new incoming block. The Least-Recently-Used (LRU) policy replaces the least-recently used block in the set. LRU policy uses the intuition that blocks have temporal locality and the block that has been recently used is more likely to be used again in the future.

2.2 Dead Block Prediction

In this section we describe the work in the literature to identify and use waste in the memory system. Dead block prediction predicts the blocks in the cache that are wasting cache space. Previous work introduced several dead block predictors and applied them to optimizations such as prefetching and block replacement [1, 11, 25, 29, 32, 52].

2.2.1 Trace Based Predictor

Tyson *et al.* [52] first proposed to predict dead-on-arrival blocks by tracking the PCs accessing cache blocks. They track the first PCs of blocks that are never reused. They used a predictor table with saturating counters to learn which PCs result in dead-on-arrival blocks dynamically. However, the term "dead block predictor" was introduced by Lai *et al.* [29]. The predictor proposed in this work is also PC based. However, this predictor keeps track of PC sequence of accesses to a block. It can detect the last access to a block.

This reference trace predictor (hereafter *reftrace*) collects a trace of the instructions used to access a particular block. The theory is that, if a sequence of memory instructions to a block leads to the last access of that block, then the same sequence of instructions should lead to the last access of other blocks. The reftrace predictor encodes the path of memory access instructions leading to a memory reference as the truncated sum of the instructions' addresses. This truncated sum is called a *signature*. Each cache block is associated with a signature that is cleared when that cache block is filled and updated when that block is accessed. The signature is used to access a table of saturating counters. When a block is accessed, the corresponding counter is decremented

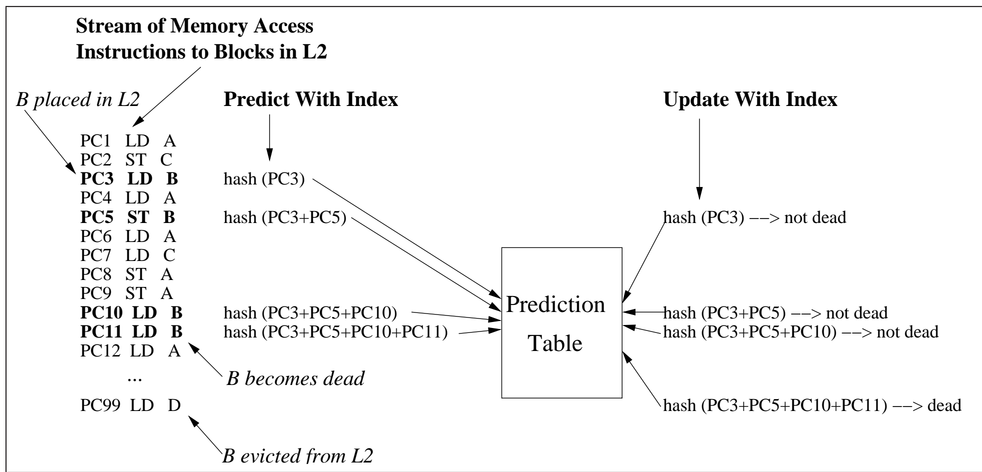


Figure 2.1: Reference trace predictor example. The predictor is indexed with the hashed sum of PCs of memory access instructions to block *B* from the time *B* is placed in the L2 to the time it is evicted.

and then the signature is updated. When a block is evicted, the counter is incremented. Thus, a counter is only incremented by a signature resulting from the last access to a block. When a block is accessed and then the signature is updated, the table of counters is consulted. If the counter exceeds a threshold, then the block is predicted dead. Each cache block stores a single bit prediction.

For comparison, we simulate a reftrace predictor with 15-bit signatures indexing a 32K-entry table of two-bit saturating counters. The predictor exclusive-ORs the first 15 bits of each PC with the next 15 bits and adds this quantity to a running 15-bit signature. We find that a confidence threshold of three gives the best accuracy.

The original reftrace predictor of Lai *et al.* uses data addresses as well as instruction addresses, requiring a large table because of the high number of signatures. Subsequent work uses only instruction addresses and allows smaller tables [32]; thus, we use only instruction addresses for all of the predictors in our work.

2.2.1.1 An Example of Reftrace

Figure 2.1 shows an example of the reftrace predictor. A block *B* is placed into the L2 cache when a load at address *PC3* accesses it. At that time, the dead block predictor is accessed by a hash of *PC3*. Block *B* is again accessed by a store at *PC5*, at which time the predictor is updated with *PC3*, letting it know that the pattern *PC3* tends to lead to the block being accessed again. Then the dead block predictor is accessed with a hash of *PC3+PC5* to provide a new prediction for block *B*. Block accesses, predictions, and predictor updates proceed in this manner. When a load at *PC11* accesses *B*, the dead block predictor, having seen the pattern *PC3+PC5+PC10+PC11* lead to a block eviction before, is likely to predict that the block is dead. Then, a much later load at *PC99* causes block *B* to be evicted, resulting in the dead block predictor being updated with a hash of *PC3+PC5+PC10+PC11*, letting it know that that pattern of references tends to lead to the block being evicted.

The Lai *et al.* predictor is used to prefetch data into predicted dead blocks in the L1 data cache. A trace based predictor is also used to optimize a cache coherence protocol [28, 48]. Dynamic self-invalidation uses traces to detect the last touch and invalidate shared cache blocks to reduce cache coherence overhead [30].

2.2.1.2 Predictor and Cache Metadata Overhead

The trace based dead block predictor used has storage and power overheads. Every cache block must be associated with a significant amount of metadata. The reftrace predictor requires 15 bits of trace signature for each block to be stored with the cacheline. Again at each cache access,a block incurs a read/modify/write cycle for the metadata. The predictor has to go through these steps to update the predictor and the cache metadata.

- Read the trace value associated with the cache line.
- Update the predictor table entry indexed by that trace.
- Calculate and update the new trace metadata.

- Predict the block dead/live with the new trace.

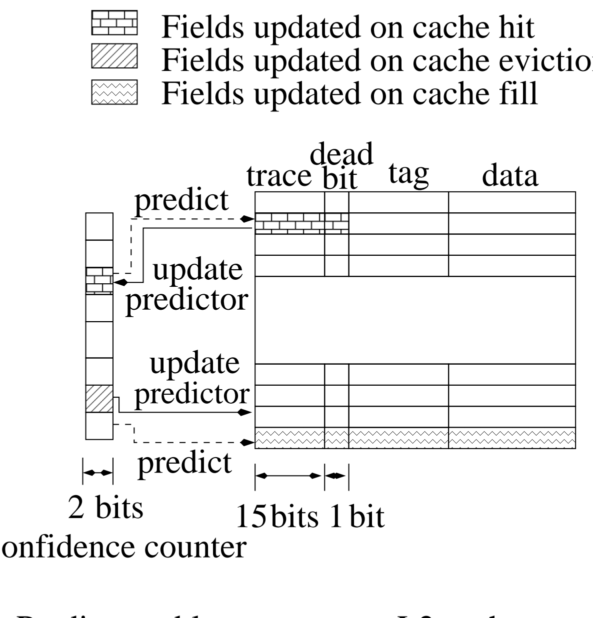


Figure 2.2: Fields updated at reftrace predictor

Figure 2.2 shows the updated fields at each cache access. In our work we propose a sampling based dead block predictor which significantly reduces the number of updates in the predictor by training the predictor only from some sampled cache sets.

2.2.2 Time Based Predictor

Another approach is to predict a block dead when it is not accessed for a certain amount of time. Hu *et al.* propose a time based dead block predictor [11] that learns the number of cycles a block is live and predicts it dead if it is not accessed for more than twice that number of cycles. This predictor is used to prefetch into the L1 cache and filter a victim cache. Abella *et al.* propose a similar predictor [1] based on number of references rather than cycles for reducing cache leakage power by dynamically turning off L2 cache blocks whose content is not likely to be reused without hurting the performance.

2.2.3 Cache Burst Predictor

Cache bursts [32] can be used with trace, counting, or time based dead block predictors. A cache burst consists of all the contiguous accesses that a block receives while in the most-recently-used (MRU) position. A cache burst predictor predicts and updates only on each burst rather than on each reference. Cache bursts predictors can significantly limit the number of accesses and updates to the prediction table for L1 caches, improving power relative to previous work. Our sampling predictor also reduces the number of updates. However, cache bursts have been shown to offer little advantage for higher level caches, since most bursts are filtered out by the L1. Our predictor is specifically targeted to the LLC. Furthermore, cache bursts predictors also require significant additional metadata in the cache, while our predictor requires only one additional bit per cache block.

2.2.4 Counting Based Predictor

Dead blocks can also be predicted depending on how many times a block has been accessed. Kharbutli and Solihin propose counting based predictors for the L2 cache. The Live-time Predictor (LvP) keeps track of the number of accesses to each block. This value is stored in the predictor on eviction. A block is predicted dead if it has been accessed more times than the previous generation. Each predictor entry has a one-bit confidence counter so that a block is predicted dead if it has been accessed the same number of times in the last two generations. The predictor table is a matrix of access and confidence counters. The rows are indexed using the hashed PC that brought the block into the cache and the columns are indexed using the hashed block address. An Access Interval Predictor (AIP) is also described in the same paper, but we focus on LvP as we find it delivers superior accuracy.

The counting based replacement policy chooses the predicted dead block closest to LRU as a victim, or the LRU block if there is no dead block. LvP and AIP are also used to bypass cache blocks which are brought to the cache and never accessed again. These are the blocks that have

zero in their access counter. Bypassing a block whose access counter is mispredicted can result in poor performance as now the block is not present in the cache and has to be brought back from the lowest level of memory hierarchy. That is why the block is bypassed with zero access count only if the set containing only live blocks.

2.2.5 Other Dead Block Predictors

Another kind of dead block prediction involves predicting in software [46, 53]. In this approach the compiler collects dead block information and provides hints to the microarchitecture to make cache decisions. If a cache block is likely to be reused again it hints to keep the block in the cache; otherwise, it hints to evict the block.

2.3 Sampling for Cache Placement and Replacement Policy

2.3.1 Dynamic Insertion Policy

Dynamic insertion policy (DIP) uses *set dueling* to adaptively insert blocks in either the MRU or LRU position depending on which policy gives better performance [38]. Performance is sampled through a small number of dedicated sets, half using MRU placement and the other half using LRU placement. Our predictor also uses sampling to dynamically learn from program behavior, but in addition to sampling the access characteristics of data, it also samples the sequence of instructions that lead to that program behavior. We compare our dead-block-predictor-driven replacement and bypass policy to adaptive insertion in Section 4.4. Thread Aware Dynamic Insertion Policy (TADIP) uses DIP in a multi-core context [13]. It takes into account the memory requirement of each executing program. It has a dedicated leader set for each core for determining the correct insertion position (MRU/LRU). This way thrashing workloads decide to insert in the LRU position while cache-friendly workloads continue to insert in the MRU position. Different insertion positions for multi-threaded workloads has been proposed by [33, 55].

2.3.2 Re-reference Interval Prediction

LRU replacement predicts that a block will be referenced in the near future. Re-reference Interval Prediction (RRIP) categorizes blocks as near re-reference, distant re-reference and long re-reference interval blocks [14]. On a miss the block that is predicted to be referenced most far in the future is replaced. Set-dueling is used where one policy inserts blocks with distant re-reference prediction and other one inserts majority of blocks with distant re-reference prediction and infrequently inserts new blocks with a long re-reference interval. RRIP prevents blocks with distant re-reference interval from evicting blocks that have a near re-reference interval. An extension of RRIP to multi-core shared cache is analogous to DIP for shared cache. Each core selects the best re-reference interval (long or distant) using set dueling.

2.4 Cache Management Techniques

A memory-level parallelism aware cache replacement policy also uses set-dueling to do sampling, relying on the fact that isolated misses are more costly for performance than parallel misses [39]. Keramidas *et al.* [19] proposed a cache replacement policy that uses sampling-based reuse distance prediction. This policy tries to evict cache blocks that will be reused furthest in the future. Mainak [7] proposes to manage cache set as a fill stack as opposed to the traditional access recency stack. [41] proposes a technique to vary the associativity of a cache on a per-set basis in response to the demands of the program. Skewed-associative cache was proposed in [47]. Non-Uniform Cache Architectures (NUCA) provides faster access to cache lines in the portions of the cache that reside closer to the processor [26]. [42] proposes a software managed fully associative memory structure using indirect index cache (IIC).

2.5 Cache Bypassing

Prior work has used bypassing to improve cache efficiency [9, 10, 15, 43, 54]. Tyson et al. proposed bypassing based on the hit rate of the missing load/store instruction [9]. Johnson et al. proposed

bypassing based on the reference frequency of the data being referenced [15] but put bypassed blocks in a separate buffer parallel to the cache. Gonzlez et al. proposed to bypass L1 data cache blocks with low temporal locality [10].

2.6 Segmentation in Disk Cache

There have been many replacement policy proposals for both disk caches and CPU caches. Prior work has proposed different versions of the LRU replacement policy. Segmented LRU [18] was proposed for disk caches. It augments each cache block with a reference bit dividing the traditional LRU stack of cache blocks into two logical sub-lists, the referenced list and the non-referenced list. The replacement policy chooses the LRU block from the non-referenced list. If all cache blocks are in the referenced list then it selects the global LRU cache block from among all the blocks in the set. This policy performs poorly for LRU friendly workloads as the stale blocks in the referenced list are rarely evicted. There are other variations of the LRU replacement policy, such as the LRU-K policy [36]. This policy takes into account the kth access from the last access to determine the victim. The Least Frequently Used (LFU) policy chooses the victim considering access frequency rather than the recency [31]. LFU performs poorly with workloads that have temporal locality, thus much work has been done to develop hybrid policies that take into account both recency and frequency [3, 16, 34]. Adaptive tuning policies like ARC and CAR [3, 34] use segmentation to choose which list to evict from. However, they require a large overhead to determine the best segmentation. ARC requires tracking 2c lines for a cache with size c.

2.7 Shared Cache Partitioning

Utility-based cache partitioning (UCP) [40] proposes cache partitioning in a shared cache depending on the reduction in miss rate for each application. Other work also proposes to divide the LRU blocks into different partitions and replace from different portions of the partitions [16, 33]. Adaptive set pinning [49] tracks the usage of each thread in sets and then allocates some sets to

individual sets where other threads can not evict blocks. PIPP [33] is a pseudo-partitioning cache management technique that uses utility-based cache partitioning (UCP) in conjunction with insertion policy specific to each thread. Vantage [45] is also a shared cache partitioning technique that works in line granularity instead on way granularity. They use [44] technique to consider replacement candidates more than the number of associativity of the set. All these partitioning maintain a static ordering while deciding which partition should be used to choose the victim. They cannot choose the partition adaptively.

CHAPTER 3: SAMPLING DEAD BLOCK PREDICTION FOR LAST-LEVEL CACHES

Dead block predictors learn the PC of the last reference to a block. The idea is if a PC leads to the last access of a block, then the same PC leads to last access for other blocks too [29, 32]. Our sampling based dead block predictor learns only from a few sampler sets as the learning acquired through sampling a few sets generalizes to the entire cache. We keep only a few sampler sets of partial tag array. Predictor is only trained when there is an access or replacement in a cache set with a corresponding sampler set.

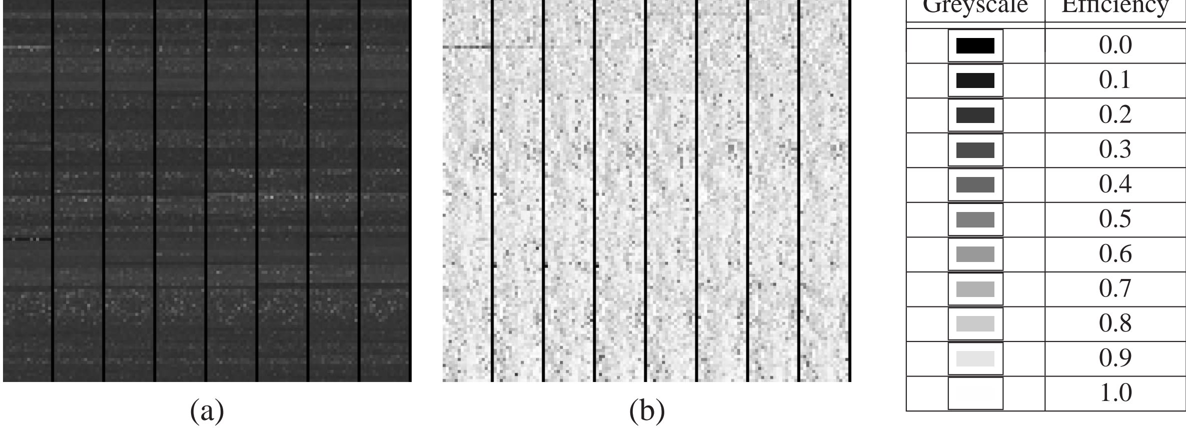


Figure 3.1: Dead block replacement and bypass bring the cache to life. Efficiency (i.e. live time ratio) shown as greyscale intensities for 456.hmmer for (a) a 1MB LRU cache and (b) a dead-block-replaced cache using a sampling predictor. Darker blocks are dead longer. Efficiency is 22% for (a) and 87% for (b).

Figure 3.1 depicts the efficiency of a 1MB 16-way set associative LLC with LRU replacement for the SPEC CPU 2006 benchmark `456.hmmer`. The amount of time each cache block is live is shown as a greyscale intensity. Figure 3.1(a) shows the unoptimized cache. The darkness shows that many blocks remain dead for long stretches of time. Figure 3.1(b) shows improvement in efficiency by driving a replacement and bypass policy with a sampling dead block predictor. We show that replacement policy in the sampler sets do not have to match the replacement policy of the cache. Our sampling based dead block predictor learns from the LRU replacement policy in

the sampler sets while the cache can deploy inexpensive random replacement policy.

3.1 Motivation

Dead block prediction can be used to identify blocks that are likely to be dead to drive optimizations that replace them with live data. Performance improves as more live blocks lead to more cache hits. However, current dead block prediction algorithms have a number of problems that make them unsuitable for the LLC:

- They incur a substantial overhead in terms of prediction structures as well as extra cache metadata. For instance, each cache block must be associated with many extra bits of metadata that can change as often as every access to that block. Thus, the improvement in cache efficiency is paid for with extra power and area requirements.
- They rely on an underlying LRU replacement algorithm. However, LRU is prohibitively expensive to implement in a highly associative LLC. Note that this problem extends to other recently proposed improvements to caches, including adaptive insertion policies [13, 27].
- Due to the large number of memory and instruction references tracked by these predictors, they must be either very large or experience a significant amount of destructive interference in their prediction tables resulting in a negative impact on accuracy.
- Predictors that use instruction traces do not work in a realistic scenario involving L1, L2, and L3 caches because a moderately-sized mid-level cache filters out most of the temporal locality.

We have proposed dead block prediction technique based on *sampling*. Previous dead block predictors find correlations between observed patterns of memory access instructions and cache evictions, learning when program behavior is likely to lead to a block becoming dead. The new prediction technique exploits three key observations, allowing it to use far less power and area while improving accuracy over previous techniques:

- Patterns of memory access instructions are consistent across sets, so it is sufficient to sample a small fraction of references to sets to do accurate prediction. For example, in a 2MB cache with 2,048 sets, sampling references to 1.6% of sets is sufficient to predict with accuracy superior other schemes that learn from every reference. The predictor keeps track of a small number of sets using partial tags replaced by the LRU policy, allowing it to generalize predictions not only to a large LRU cache, but also to a large randomly replaced cache.

- Previous predictors use reference traces or counters for each block. However, simply using the address of the last memory access instruction provides sufficient prediction accuracy while obviating the need to keep track of access patterns for all blocks. Metadata associated with cache blocks is significantly reduced with a proportional reduction in power.

Here sampling prediction is explored in the context of a dead block replacement and bypass policy. That is, the replacement policy will choose a dead block to be replaced before falling back on a default replacement policy such as random or LRU, and a block that is predicted "dead on arrival" will not be placed, i.e., it will bypass the LLC.

3.2 Description

In this section we discuss the design of a new sampling- based dead block predictor.

3.2.1 A Sampling Dead Block Predictor

A Sampling Partial Tag Array The sampling predictor keeps a small partial tag array, or sampler. Each set in the sampler corresponds to a selected set in the cache; e.g. if the original cache has 2,048 sets, the sam- pler could keep 32 sets corresponding to every 2, 048/32 = 64th cache set. Since correctness of matches is not necessary in the sampler tag array, only the lower-order 15 bits of tags are stored to conserve area and energy. (Nevertheless, we observed no incorrect matches in any of the benchmarks; 15-bit tags are quite sufficient for this application.) As with other dead block predictors, each access to the LLC incurs an access to the predictor. However, predictor is

only updated when there is an access or replacement in a cache set with a corresponding sampler set. This strategy works because the learning acquired through sampling a few sets generalizes to the entire cache. We find that predictor accuracy improves slightly as more sets are added to the sampler, although too many sets can increase destructive interference in the prediction tables. We would like to have as few sets as possible to save power. We find that 32 sets provide a good trade-off between accuracy and efficiency.

3.2.1.1 Advantages of a Sampler

A sampler decouples the prediction mechanism from the structure of the cache, offering several advantages over previous predictors:

1. Since the predictor and sampler are only updated on a small fraction of cache accesses and replacements, the power requirement of the predictor is reduced.

2. The replacement policy of the sampler does not have to match that of the cache. For instance, the LLC may use the less costly random replacement policy, but the sampler can still use the deterministic LRU policy. A deterministic policy is easier to learn from because the same sequence of references comes up consistently, uninterrupted by random evictions.

3. The associativity of the sampler does not have to match that of the original cache. We have found that, with a 16-way LLC, a 12-way sampler offers better prediction accuracy than a 16-way sampler since blocks that are likely to be dead are evicted sooner. Also, a 12-way sampler consumes less storage and power.

3.2.2 Advantage Over Previous Predictors

Reftrace keeps a separate signature for every cache block to be used by the predictor when that block is accessed. Counting predictors also store counts with each block. These predictors, as well as cache bursts, have two problems: 1) Every cache block must be associated with a significant

amount of metadata. Reftrace requires keeping the signature for each block, while counting predictors require keeping a count and other data for each block. 2) Every update to a block incurs a read/modify/write cycle for the metadata. Either a count or a signature must be read, modified, Meeting the timing constraints of this sequence of operations could be problematic, especially in a low-power design.

However, the new predictor uses a trace based only on the PC. That is, rather than predicting whether a block is dead based on the trace of instructions that refer to that block, the predictor only uses the PC of the last instruction that accessed a block. Thus, all predictor state can be kept in the predictor, and only a single additional bit of metadata is needed for each cache block – a bit that indicates whether the block has been predicted as dead. Trace metadata is still stored in the sampler, but since the sample tag array is far smaller than the actual LLC tag array, area and timing are not a problem. The predictor state is only modified on the small fraction of accesses to sampler sets.

Figure 3.2 gives a block diagram of the reftrace and sampling predictors, showing the times during which they are accessed and updated. The sampling predictor is accessed as frequently as the reftrace predictor, but it is updated far less often.

3.2.2.0.1 Key Difference with PC-Only Sampler Reftrace could also use just the PC to index its prediction table. Each time a block is accessed, the signature could immediately be used to update the predictor indicating that the block is live. However, reftrace would still need to keep a signature for each cache block for the time between the last access to the block and the eviction of the block to update the predictor on an eviction. In addition to the storage requirement, reftrace would also need a 16-bit channel between the stream of instructions and the LLC.

The sampling predictor does not have this problem. It only needs to keep the PC signature for tags tracked in the sampler. There are far fewer of these sampler signatures, i.e. 1,536 for a 12-way 32-set sampler, compared with the number of signatures for the entire cache with the reftrace predictor, i.e. 32,768. The sampler needs only a one-bit channel to the LLC to provide predictions.

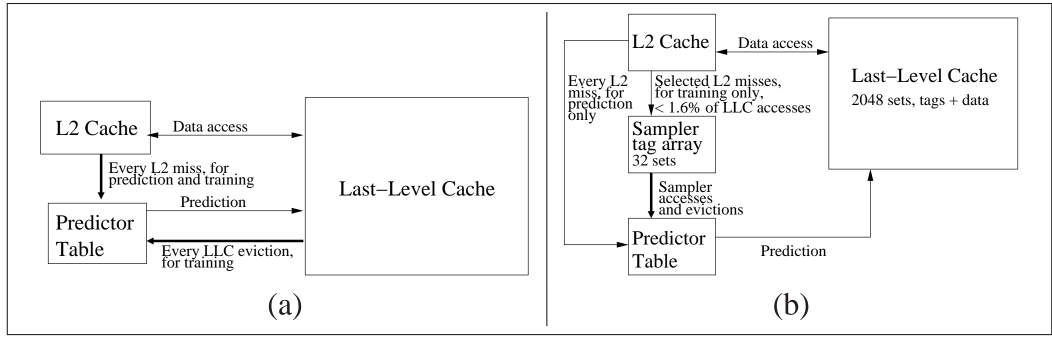


Figure 3.2: Old dead block predictor (a), and new dead block predictor with sampler tag array (b). The sampler and dead block predictor table are updated for 1.6% of the accesses to the LLC, while the original predictor is updated on every access.

3.3 A Dead-Block Driven Replacement and Bypass Policy

We evaluate dead block predictors in the context of a combined dead block replacement and bypassing optimization [25]. When it is time to choose a victim block, a predicted dead block may be chosen instead of a random or LRU block. If there is no predicted dead block, a random or LRU block may be evicted.

If a block to be placed in a set will be used further in the future than any block currently in the set, then it makes sense to decline placing it [35]. That is, the block should bypass the cache. Bypassing can reduce misses in LLCs, especially for programs where most of the temporal locality is captured by the first-level cache. Dead block predictors can be used to implement bypassing: if a block is predicted dead on its first access then it is not placed in the cache.

3.3.1 Dead Block Replacement and Bypassing with Default Random Replacement

In Section 4.4, we show results for dead-block replacement and bypass using a default random replacement policy. We show that this scheme has very low overhead and significant performance and power advantages.

What does it mean for a block to be dead in a randomly replaced cache? The concept of a dead block is well-defined even for a randomly-replaced cache: a block is dead if it will be evicted before it is used again. However, predicting whether a block is dead is now a matter of predicting

the outcome of a random event. The goal is not necessarily to identify with 100% certainty which block will be evicted before it is used next, but to identify a block that has a high probability of not being used again soon.

3.3.2 Predictor Update in the Optimization

One question naturally arises: should the predictor learn from evictions that it caused? We find that for reftrace and for the sampling predictor, allowing the predictor to learn from its own evictions results in slightly improved average miss rates and performance over not doing so. We believe that this feedback allows the predictor to more quickly generalize patterns learned for some sets to other sets. On the other hand, we find no benefit from letting a tag "bypass" the sampler.

3.3.3 Multiple Cores

The sampling predictor described in this paper is used unmodified for both single-thread and multi-core workloads. The same 32-set sampling predictor is used for the 2MB single-core cache as well as the 8MB quad-core cache. There is no special tuning for multi-core workloads.

3.4 A Comparison of Predictor Storage and Power

In this section, we discuss the storage requirements of dead block predictors: the reftrace predictor, the counting predictor, the sampling predictor.

3.4.1 Reference Trace Predictor

For this study, we use a reftrace predictor indexed using 15-bit signature. Thus, the prediction table contains 2^{15} two-bit counters, or 8KB. We find diminishing returns in terms of accuracy from larger tables, so we limit our reftrace predictor to this size. Each cache block is associated with two extra fields: the 15-bit signature arising from the most recent sequence of accesses to that block, and another bit that indicates whether the block has been predicted as dead. With a 2MB cache with 64B blocks, this works out to 64KB of extra metadata in the cache. Thus, the total amount of

state for the reftrace predictor is 72KB, or 3.5% of the data capacity of the LLC.

3.4.2 Counting Predictor

The counting-based Live-time Predictor (LvP) predictor [25] is a 256×256 table of entries, each of which includes the following fields: a four-bit counter keeping track of the number of accesses to a block and a one-bit confidence counter. Thus, the predictor uses 40KB of storage. In addition, each cache block is augmented with the following metadata: an eight-bit hashed PC, a four-bit counter keeping track of the number of times the cache block is accessed, a four-bit number of accesses to the block from the last time the block was in the cache, and a one-bit confidence counter. This works out to approximately 68KB of extra metadata in the cache. Thus, the total amount of state for the counting predictor is 108KB, or 5.3% of the data capacity of the LLC.

3.4.3 Sampling Predictor

The three prediction tables for the skewed dead block predictor are each 4,096 two-bit counters so they consume 3KB of storage.

We model a sampler with 32 sets. Each set has 12 entries consisting of 15-bit partial tags, 15-bit partial PCs, one prediction bit, one valid bit, and four bits to maintain LRU position information, consuming 6.75KB of storage[1]. Each cache line also holds one extra bit of metadata. Thus, the sampling predictor consumes 22.75KB of storage, which is 1.1% of the capacity of a 2MB LLC. Table 3.1 summarizes the storage requirements of each predictor.

3.4.4 Predictor Power

The sampling predictor uses far less storage than the other predictors, but part of its design includes sets of associative tags. Thus, it behooves us to account for the potential impact of this structure on power. Table 3.2 shows the results of CACTI 5.3 simulations [51] to determine the leakage and

[1]For an LRU cache, why do we not simply use the tags already in the cache? Since the sampler is 12-way associative and does not experience bypass, there will not always be a correspondence between tags in a sampler set and the same set in the cache. Also, we would like to access and update the sampler without waiting for the tag array latency.

Table 3.1: Storage overhead for the various predictors

Predictor	Predictor Structures	Cache Metadata	Total Storage
reftrace	8KB table	16 bits × 32K blocks = 64KB	72KB
counting	256 × 256 table, 5-bit entries = 40KB	17 bits × 32K blocks = 68KB	108KB
sampler	3 × 1KB tables + 6.75KB sampler = 9.75	1 bit × 32K blocks = 4KB	13.75KB

dynamic power of the various components of each predictor. The sampler was modeled as the tag array of a cache with as many sets as the sampler, with only the tag power being reported. The predictor tables for the pattern sample predictors was modeled as a tagless RAM with three banks accessed simultaneously, while the predictor table for the reftrace predictor was modeled as a single bank 8KB tagless RAM. The prediction table for the counting predictor was conservatively modeled as a 32KB tagless RAM. To attribute extra power to cache metadata, we modeled the 2MB LLC both with and without the extra metadata, represented as extra bits in the data array, and report the difference between the two.

3.4.4.1 Dynamic Power

When it is being accessed, the dynamic power of the sampling predictor is 57% of the leakage power of the reftrace predictor, and only 28% of the leakage power of the counting predictor. The baseline LLC itself has a dynamic power of 2.75W. Thus, the sampling predictor consumes 3.1% of the power budget of the baseline cache, while the counting predictor consumes 11% of it. Note that CACTI reports peak dynamic power. Since the sampler is accessed only on a small fraction of LLC accesses, the actual dynamic power for the sampling predictor would far lower.

3.4.4.2 Leakage Power

For many programs, dynamic power of the predictors will not be an important issue since the LLC might be accessed infrequently compared with other structures. However, leakage power is always a concern. The sampling predictor has a leakage power that is only 40% of the reftrace

Table 3.2: Dynamic and leakage power for predictor components. All figures are in Watts.

Predictor	**Prediction Structures Power**		**Extra Metadata Power**		**Total Power**	
	leakage	**dynamic**	**leakage**	**dynamic**	**leakage**	**dynamic**
reftrace	0.002	0.030	0.013	0.120	0.015	0.150
counting	0.010	0.175	0.014	0.127	0.024	0.302
sampler	0.005	0.078	0.001	0.008	0.006	0.086

predictor, and only 25% of the counting predictor. This is primarily due to the reduction in cache metadata required by the predictor. As a percentage of the 0.512W total leakage power of the LLC, the sampling predictor uses only 1.2%, while the counting predictor uses 4.7% and the reftrace predictor uses 2.9%.

3.4.5 Latency

CACTI simulations show that the latency reading and writing the structures related to the sampling predictor fit well within the timing constraints of the LLC. It is particularly fast compared with the reftrace predictor since it does not need to read/modify/write metadata in the LLC cache on every access.

3.5 Experimental Methodology

This section outlines the experimental methodology used in this study.

3.5.1 Simulation Environment

The simulator is a modified version of CMP$im, a memory-system simulator that is accurate to within 4% of a detailed cycle-accurate simulator [12]. The version we used was provided with the JILP Cache Replacement Championship [2]. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. This infrastructure enables collecting instructions-per-cycle

Table 3.3: The 29 SPEC CPU 2006 benchmarks with LLC cache misses per 1000 instructions for LRU and optimal (MIN), instructions-per-cycle for LRU for a 2MB cache, and number of instructions fast-forwarded to reach the simpoint (B = billions). Benchmarks in the subset in boldface.

Name	MPKI (LRU)	MPKI (MIN)	IPC (LRU)	FFWD	Name	MPKI (LRU)	MPKI (MIN)	IPC (LRU)	FFWD
astar	2.275	2.062	1.829	185B	bwaves	0.088	0.088	3.918	680B
bzip2	0.836	0.589	2.713	368B	**cactusADM**	13.529	13.348	1.088	81B
calculix	0.006	0.006	3.976	4433B	dealII	0.031	0.031	3.844	1387B
gamess	0.005	0.005	3.888	48B	**gcc**	0.640	0.524	2.879	64B
GemsFDTD	13.208	10.846	0.818	1060B	gobmk	0.121	0.121	3.017	133B
gromacs	0.357	0.336	3.061	1B	h264ref	0.060	0.060	3.699	8B
hmmer	1.032	0.609	3.017	942B	**lbm**	25.189	20.803	0.891	13B
leslie3d	7.231	5.898	0.931	176B	**libquantum**	23.729	22.64	0.558	2666B
mcf	56.755	45.061	0.298	370B	**milc**	15.624	15.392	0.696	272B
namd	0.047	0.047	3.809	1527B	**omnetpp**	13.594	10.470	0.577	477B
perlbench	0.789	0.628	2.175	541B	povray	0.004	0.004	2.908	160B
sjeng	0.318	0.317	3.156	477B	**soplex**	25.242	16.848	0.559	382B
sphinx3	11.586	8.519	0.655	3195B	tonto	0.046	0.046	3.472	44B
wrf	5.040	4.434	0.934	2694B	**xalancbmk**	18.288	10.885	0.311	178B
zeusmp	4.567	3.956	1.230	405B					

figures as well as misses per kilo-instruction and dead block predictor accuracy. The experiments model a 16-way set-associative last-level cache to remain consistent with other previous work [28, 32, 38, 39]. The microarchitectural parameters closely model Intel Core i7 (Nehalem) with the following parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way L3: 2MB/core. Each benchmark is compiled for the x86_64 instruction set. The programs are compiled with the GCC 4.1.0 compilers for C, C++, and FORTRAN.

We use SPEC CPU 2006 benchmarks. We use SimPoint [37] to identify a single one billion instruction characteristic interval (i.e.*simpoint*) of each benchmark. Each benchmark is run with the first `ref` input provided by the `runspec` command.

3.5.1.1 Single-Thread Workloads

For single-core experiments, the infrastructure simulates one billion instructions. We simulate a 2MB LLC for the single-thread workloads. In keeping with the methodology of recent cache

papers [13, 14, 19, 25, 28, 29, 32, 38, 39], we choose a memory-intensive subset of the benchmarks. We use the following criterion: a benchmark is included in the subset if the number of misses in the LLC decreases by at least 1% when using the optimal replacement and bypass policy instead of LRU[2].

Table 3.3 shows each benchmark with the baseline LLC misses per 1000 instructions (MPKI), optimal MPKI, baseline instructions-per-cycle (IPC), and the number of instructions fast-forwarded (FFWD) to reach the interval given by SimPoint.

3.5.1.2 Multi-Core Workloads

Table 3.4 shows ten mixes of SPEC CPU 2006 simpoints chosen four at a time with a variety of memory behaviors characterized in the table by cache sensitivity curves. We use these mixes for quad-core simulations. Each benchmark runs simultaneously with the others, restarting after one billion instructions, until all of the benchmarks have executed at least one billion instructions. We simulate an 8MB shared LLC for the multi-core workloads. For the multi-core workloads, we report the weighted speedup normalized to LRU. That is, for each thread i sharing the 8MB cache, we compute IPC_i. Then we find SingleIPC_i as the IPC of the same program running in isolation with an 8MB 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.

3.5.2 Optimal Replacement and Bypass Policy

For simulating misses, we also compare with an optimal block replacement and bypass policy. That is, we enhance Belady's MIN replacement policy [4] with a bypass policy that refuses to place a block in a set when that block's next access will not occur until after the next accesses to all other blocks in the set. We use trace-based simulation to determine the optimal number of

[2]Ten of the 29 SPEC CPU 2006 benchmarks do not experience significant reduction in misses even with optimal replacement because their working sets fit comfortably into a 2MB LLC and experience only a few compulsory misses. We have tested our technique with all of these benchmarks; it causes no change in performance, neither positive nor negative.

Table 3.4: Multi-core workload mixes with cache sensitivity curves giving LLC misses per 1000 instructions (MPKI) on the y-axis for last-level cache sizes 128KB through 32MB on the x-axis.

Workload Name	**Benchmarks**	**Cache Sensitivity Curve**
mix1	mcf hmmer libquantum omnetpp	43.5 25.5
mix2	gobmk soplex libquantum lbm	22.0 12.6
mix3	zeusmp leslie3d libquantum xalancbmk	14.6 9.4
mix4	gamess cactusADM soplex libquantum	19.0 9.5
mix5	bzip2 gamess mcf sphinx3	36.4 16.6
mix6	gcc calculix libquantum sphinx3	9.7 6.6
mix7	perlbench milc hmmer lbm	10.4 9.6
mix8	bzip2 gcc gobmk lbm	6.7 5.4
mix9	gamess mcf tonto xalancbmk	37.0 16.6
mix10	milc namd sphinx3 xalancbmk	12.4 4.5

misses using the same sequence of memory accesses made by the out-of-order simulator. The out-of-order simulator does not include the optimal replacement and bypass policy so we report optimal numbers only for cache miss reduction and not for speedup.

3.5.3 Measuring Cache Efficiency

Cache efficiency is a statistic defined by Burger *et al.* [5] to quantify the average amount of time blocks in the cache contain live data. Cache efficiency is computed as $E = \frac{\sum_{i=0}^{A \times S-1} U_i}{N \times A \times S}$ where N is the total number of cycles executed, A is the number of blocks per set, S is the number of sets in the cache, and U_i is the total number of cycles for which cache block i contains live data, i.e., data that will be referenced again before it is evicted. Thus, cache efficiency is the average of the live cycles for each cache block. The performance simulation infrastructure collects block live times

Table 3.5: Legend for various cache optimization techniques.

Name	Technique
Sampler	Dead block bypass and replacement with sampling predictor, LRU
TDBP	Dead block bypass and replacement with reftrace, LRU
CDBP	Dead block bypass and replacement with counting predictor, LRU
DIP	Dynamic Insertion Policy, LRU
RRIP	Re-reference interval prediction, NRU
TADIP	Thread-aware DIP, LRU
Random Sampler	Dead block bypass and replacement with sampling predictor, random
Random CDBP	Dead block bypass and replacement with counting predictor, random
Optimal	Optimal replacement and bypass policy as described in Section 3.5.2

and produces cache efficiency as an output.

3.6 Experimental Results

In this section we discuss results of our experiments. In the graphs that follow, several techniques are referred to with abbreviated names. Table 3.5 gives a legend for these names.

For TDBP and CDBP, we simulate a dead block bypass and replacement policy just as described previously, dropping in the reftrace and counting predictors, respectively, in place of our sampling predictor.

3.6.1 LLC Misses

Figure 3.3 shows LLC cache misses normalized to a 2MB LRU cache for each benchmark. On average, dynamic insertion (DIP) reduces cache misses to 93.9% of the baseline LRU, a reduction of by 6.1%. RRIP reduces misses by 8.1%. The reftrace-predictor-driven policy (TDBP) *increases* average misses on average by 8.0% (mostly due to `473.astar`), decreasing misses on only 11 of the 19 benchmarks. CDBP reduces average misses by 4.6%. The sampling predictor reduces average misses by 11.7%. The optimal policy reduces misses by 18.6% over LRU; thus, the sampling predictor achieves 63% of the improvement of the optimal policy.

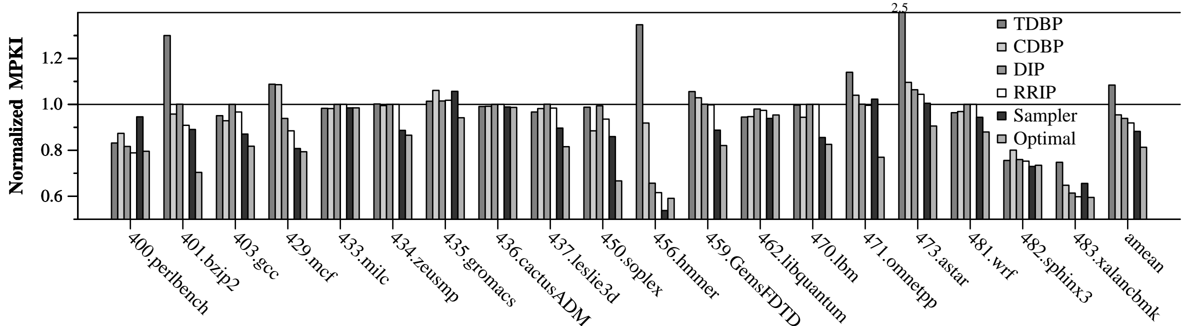


Figure 3.3: Reduction in LLC misses for various policies.

3.6.2 Speedup

Reducing cache misses translates into improved performance. Figure 3.4 shows the speedup (i.e. new IPC divided by old IPC) over LRU for the predictor-driven policies with a default LRU cache.

DIP improves performance by a geometric mean of 3.1%. TDBP provides a speedup on some benchmarks and a slowdown on others, resulting in a geometric mean speedup of approximately 0%. The counting predictor delivers a geometric mean speedup of 2.3%, and does not significantly slow down any benchmarks. RRIP yields an average speedup of 4.1%. The sampling predictor gives a geometric mean speedup of 5.9%. It improves performance by at least 4% for eight of the benchmarks, as opposed to only five benchmarks for RRIP and CDBP and two for TDBP. The sampling predictor delivers performance superior to each of the other techniques tested.

Speedup and cache misses are particularly poor for `473.astar`. As we will see in Section 3.6.5, dead block prediction accuracy is bad for this benchmark. However, the sampling predictor minimizes the damage by making fewer predictions than the other predictors.

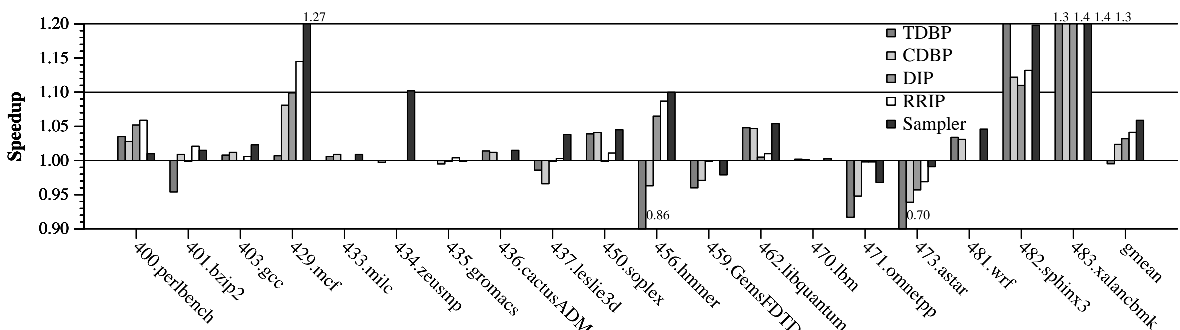


Figure 3.4: Speedup for various policies

3.6.3 Poor Performance for Trace-Based Predictor

Note that the reftrace predictor performs quite poorly compared with its observed behavior in previous work [32]. In that work, reftrace was used for L1 or L2 caches with significant temporal locality in streams of reference reaching the predictor. Reftrace learns from these streams of temporal locality. In this work, the predictor optimizes the LLC in which most temporal locality has been filtered by the 256KB middle-level cache. In this situation, it is easier for the predictor to try to simply learn the last PC to reference a block rather than a sparse reference trace that might not be repeated often enough to learn from[3].

3.6.4 Dead Block Replacement with Random Baseline

In this subsection we explore the use of the sampling predictor to do dead-block replacement and bypass with a baseline randomly-replaced cache. When a block is replaced, a predicted dead block is chosen, or if there is none, a random block is chosen. We compare with the CDBP and random replacement. We do not compare with TDBP, RRIP or DIP because these policies depend on a baseline LRU replacement policy and become meaningless in a randomly-replaced cache.

3.6.4.1 LLC Misses

Figure 3.5 shows the normalized number of cache misses for the various policies with a default random-replacement policy. The figures are normalized to the same baseline LRU cache from the previous graphs, so the numbers are directly comparable. CDBP reduces misses for some benchmarks but increases misses for others, resulting in no net benefit. Random replacement by itself increases misses an average of 2.5% over the LRU baseline. The sampling predictor yields an average normalized MPKI of 0.925, an improvement of 7.5% over the LRU baseline. Note that the sampling predictor with a default random-replacement policy requires only one bit of metadata associated with individual cache lines. Amortizing the cost of the predictor storage over the cache

[3]We have simulated reftrace correctly. In our simulations and in previous work, reftrace works quite well when there is no middle-level cache to filter the temporal locality between the small L1 and large LLC. However, many real systems have middle-level caches.

lines (computed in Section 3.4), the replacement and bypass policy requires only 1.71 bits per cache line to deliver 7.5% fewer misses than the LRU policy.

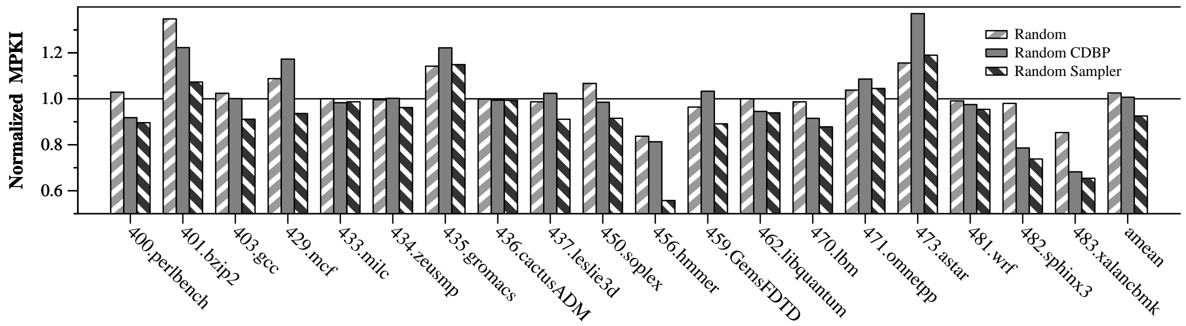


Figure 3.5: LLC misses per kilo-instruction for various policies

3.6.4.2 Speedup

Figure 3.6 shows the speedup for the predictor-driven policies with a default random cache. CDBP yields an almost negligible speedup of 0.1%. Random replacement by itself results in a 1.1% slowdown. The sampling predictor gives a speedup of 3.4% over the LRU baseline. Thus, a sampling predictor can be used to improve performance with a default randomly-replaced cache.

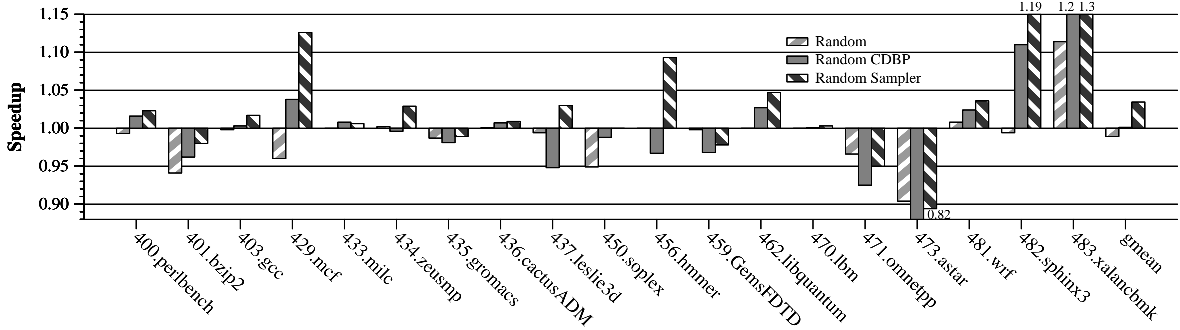


Figure 3.6: Speedup for various replacement policies with a default random cache

3.6.5 Prediction Accuracy and Coverage

Mispredictions come in two varieties: false positives and false negatives. False positives are more harmful because they wrongly allow an optimization to use a live block for some other purpose, causing a miss. The coverage of a predictor is ration of positive predictions to all predictions. If a predictor is consulted on every cache access, then the coverage is the fraction of cache accesses

when the optimization may be applied. Higher coverage means more opportunity for the optimization. Figure 3.7 shows the coverage and false positive rates for the various predictors given a default LRU cache. On average, reftrace predicts that a block is dead for 88% of LLC accesses, and is wrong about that prediction for 19.9% of cache accesses. The counting predictor has a coverage of 67% and is wrong 7.19% of the time. The sampling predictor has a coverage of 59% and a low false positive rate of 3.0%, explaining why it has the highest average speedup among the predictors. The benchmark `473.astar` exhibits apparently unpredictable behavior. No predictor has good accuracy for this benchmark. However, the sampling predictor has very low coverage for this benchmark, minimizing the damage caused by potential false positives.

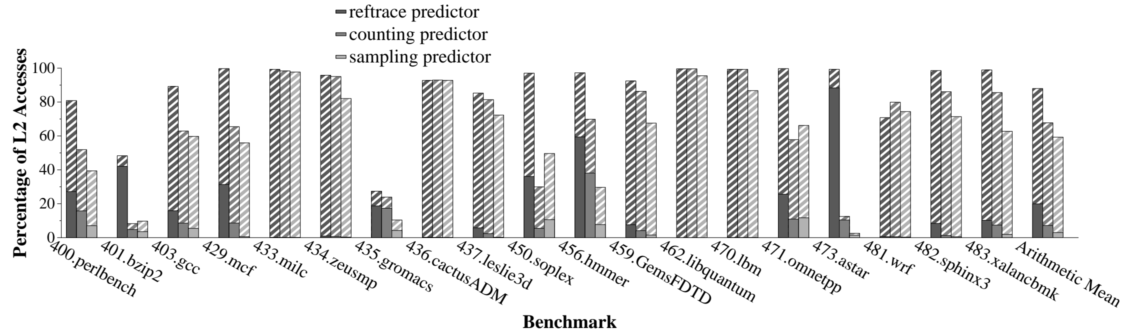


Figure 3.7: Coverage and false positive rates for the various predictors

3.6.6 Multiple Cores Sharing a Last-Level Cache

Figure 3.8(a) shows the normalized weighted speedup achieved by the various techniques on the multi-core workloads with an 8MB last-level cache and a default LRU policy. The normalized weighted speedup over all 10 workloads ranges from 6.4% to 24.2% for the sampler, with a geometric mean of 12.5%, compared with 10% for CDBP, 7.6% for TADIP, 5.6% for TDBP, and 4.5% for the multi-core version of RRIP.

Figure 3.8(b) shows the normalized weighted speedup for techniques with a default random policy. The speedups are still normalized to a default 8MB LRU cache. The average normalized MPKIs are 0.77 for the sampler, 0.79 for CDBP, 0.85 for TADIP, 0.95 for TDBP and 0.93 for multi-core RRIP. The sampler reduces misses by 23% on average.

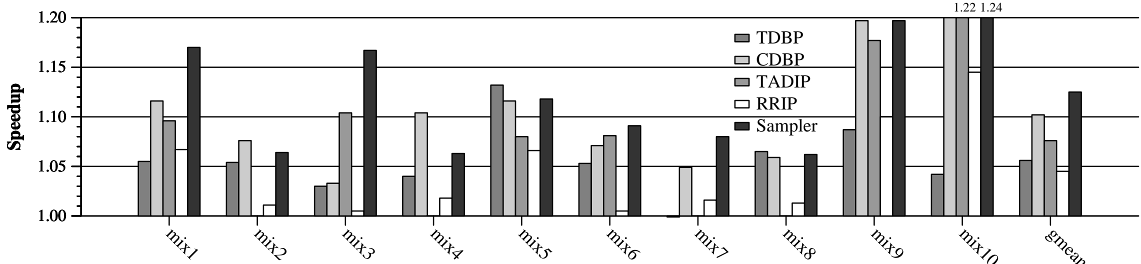


Figure 3.8: Weighted speedup for multi-core workloads normalized to LRU for a LRU cache

3.6.7 Improvement in Cache Efficiency

Dead block replacement and bypass improves cache efficiency by making sure more cache blocks are live. Figure 3.9 quantifies this improvement for a default LRU cache. On average, cache blocks are live only 14% of the time with the LRU replacement policy. The sampling predictor improves average efficiency to 49%. Efficiency is improved for every benchmark. As an extreme example, the cache efficiency for `462.libquantum` is improved from 2.3% to 97%. A more typical example is `456.hmmer`, whose efficiency is improved from 50% to 92%. For the multi-core workloads, the average efficiency is 11% with LRU replacement. The sampling predictor improves average efficiency to 61%.

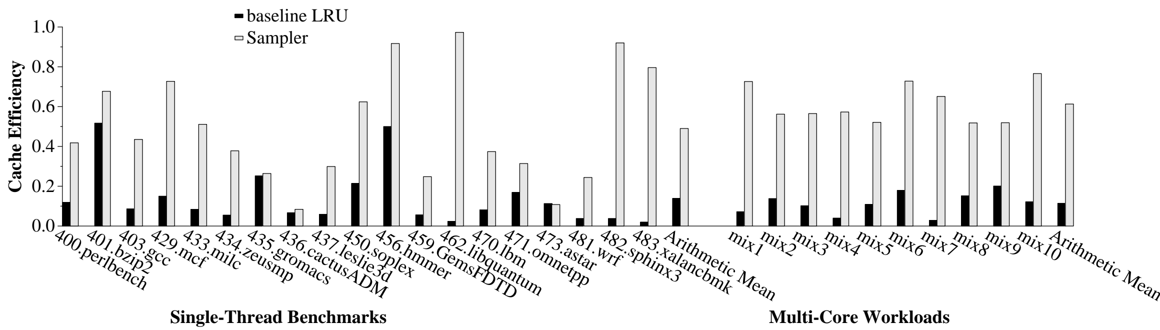


Figure 3.9: Cache efficiency for dead-block replacement and bypass combined with LRU replacement.

3.7 Using Dead Blocks

Dead blocks can be considered as extra space in cache that can be used for useful blocks. Previous works have used dead blocks to hold prefetched blocks [32, 32]. Dead blocks can also be used to reduce wasted cache power. [1] proposed to reduce leakage power by turning off the dead blocks.

Dynamic self-invalidation uses traces to detect the last touch and invalidate shared cache blocks to reduce cache coherence overhead [30]. We propose a new usage for dead blocks. In the next section we propose that dead blocks can be used to hold the evicted blocks. This way dead blocks all over the cache can be considered as a virtual victim cache within the original cache.

CHAPTER 4: USING DEAD BLOCKS AS A VIRTUAL VICTIM CACHE

In this section we introduce a cache management technique to improve cache performance by using predicted dead blocks to hold victims from cache evictions in other sets. The pool of predicted dead blocks can be thought of as a *virtual victim cache* (VVC). In this way when there is a conflict and a block must be evicted from the cache, dead blocks provide extra space to store that victim block. Whenever these blocks get referenced again they can be found in the cache and avoids a costly off-chip miss. Figure 4.1 graphically depicts the efficiency of a 1MB 16-way

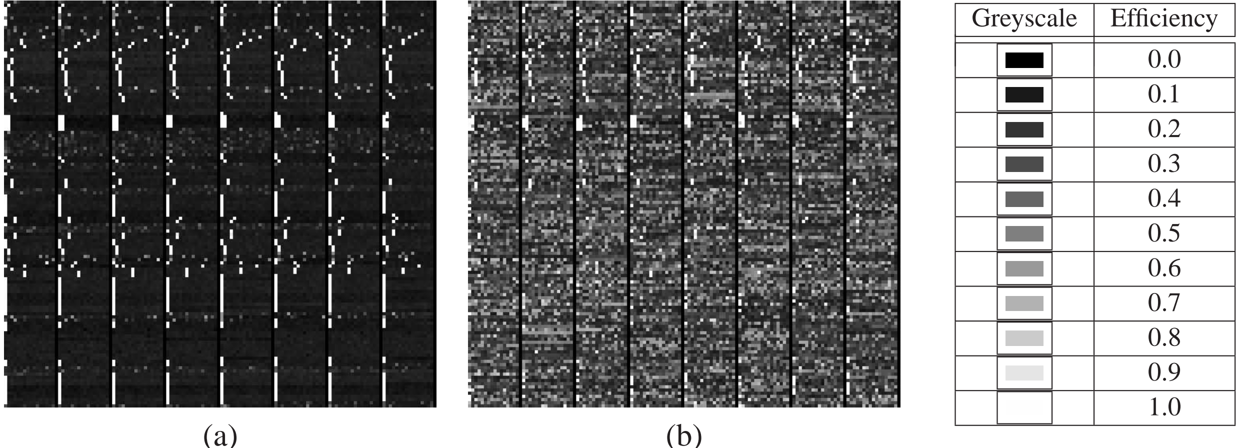


Figure 4.1: Virtual victim cache increases cache efficiency. Block efficiency (i.e., fraction of time block is live) shown as greyscale intensities for `456.hmmer` for (a) a baseline 1MB cache and (b) a VVC-enhanced cache; darker blocks are dead longer. Efficiency is 15% for (a) and 35% for (b).

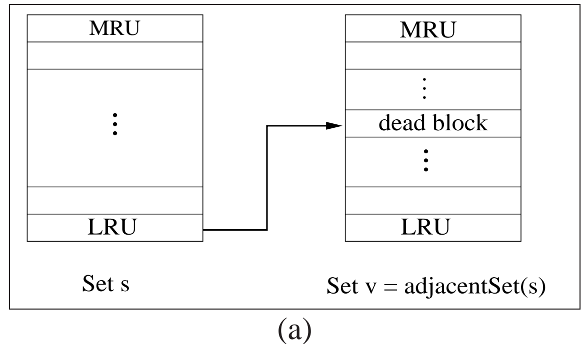
set associative L2 cache with LRU replacement for the SPEC CPU 2006 benchmark `456.hmmer`. The amount of time each cache block is live is shown a as a greyscale intensity. Figure 4.1(a) shows the unoptimized cache. The darkness shows that many blocks remain dead for large stretches of time. Figure 4.1(b) shows the same cache optimized with the VVC idea. Now many blocks have more live time so the cache is more efficient.

4.1 Motivation

The *virtual victim cache* technique stores victim blocks in the dead blocks of other sets and move them back into their original set when they are referenced again. This approach has the potential to reduce both capacity misses and additional conflict misses. Capacity misses are reduced as a dead block gets replaced before the LRU block avoiding a miss that would occur in a fully associativity cache. Conflict misses are also reduced as blocks evicted from the hot sets are placed in dead regions of the cache. This extends the associativity of hot sets and reduces conflict misses.

This cache management technique has a similarity with Victim caches [17]. Victim caches are a separate small fully associative structure that stores the evicted blocks from the cache. If a victim block is referenced again that miss can be satisfies from the victim cache. Victim caches work well because they extend the associativity of the hot sets in the cache. However, victim caches are small full associative structure and blocks in it get flushed very quickly when multiple hot sets are competing for the space in the victim cache. Larger victim caches have not come into wide use because any additional miss reduction benefits are outweighed by the overheads of the large fully associative structure. On the other hand Virtual Victim Cache is not limited by structure size. It can be as large as the number of dead blocks in the cache. It can store victim blocks from multiple hot sets as the dead blocks are distributed all over the cache.

An important question is how the dead blocks outside of a set are found and managed without adding prohibitive overhead. We have proposed to couple sets and store the victim blocks from one set into the predicted dead blocks of its "partner set". This effectively extends the associativity of cache sets while keeping the power overheads very low. As most of the accesses in the cache results in hits, the number of extra lookups at the partner sets are very low. The overheads include one more tag bit to differentiate blocks from original set and the partner set and one extra lookup at the partner set at every cache miss.



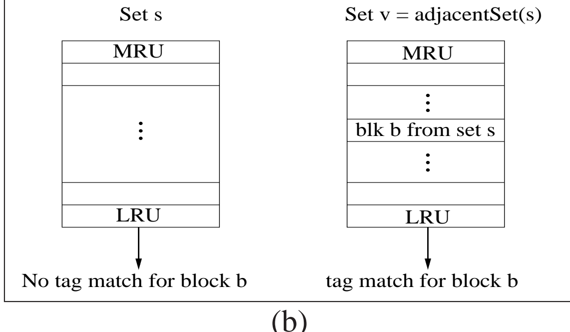


Figure 4.2: (a) Placing evicted block into an adjacent partner set, and (b) hitting in the virtual victim cache

4.2 Description

4.2.1 Identifying Potential Receiver Blocks

A trace based dead block predictor keeps a trace encoding for each cache block. The trace is updated on each use of the block. When a block is evicted from the cache, a saturating counter associated with that block's trace is incremented. When a block is used, the counter is decremented.

Ideally, any victim block could replace any receiver block in the entire cache, resulting in the highest possible usage of the dead blocks as a virtual victim cache. However, this idea would increase the dead block hit latency and energy as every set in the cache would have to be searched for a hit. Thus, there is a trade-off between the number of sets that can store victim blocks from a particular set and the time and energy needed for a hit. We have determined that, for each set, considering only one other partner set to identify a receiver block yields a reasonable balance. Sets are paired into *adjacent sets* that differ in their set indices by one bit.

4.2.2 Placing Victim Blocks into the Adjacent Set

When a victim block is evicted from a set, the adjacent set is searched for invalid or predicted dead receiver blocks. If no such block is found, then the LRU block of the adjacent set is used. Once a receiver block is identified, the victim block replaces it. The victim block is placed into either the most-recently-used or the least-recently-used position in the LRU stack based on set dueling (see

Section 4.2.4).

4.2.3 Block Identification in the VVC

If a previously evicted block is referenced again, the tag match will fail in the original set, the adjacent set will be searched, and if the receiver block has not yet been evicted then the block will be found there. The block will then be refilled in the original set from the adjacent set, and the block in the adjacent set will be marked as invalid. A small penalty for the additional tag match and fill will accrue to this access, but this access is considered a hit in the L2 for purposes of counting hits and misses (analogously, an access to a virtually-addressed cache following a TLB miss may still be considered a hit, albeit with an extra delay).

To distinguish receiver blocks from other blocks, we keep an extra bit with each block that is true if the block is a receiver block, false otherwise. When a set's tags are searched for a normal cache access, receiver blocks from the adjacent set are prevented from matching to maintain correctness. Note that keeping this extra bit is equivalent to keeping an extra tag bit in a higher associativity cache.

Figure 4.2(a) shows what happens when a LRU block is evicted from a set s. If the adjacent set v has any predicted dead or invalid block in it, the victim block replaces that block, otherwise the LRU block of set v is used. Similarly, Figure 4.2(b) depicts a VVC hit. If the access results in a miss in the original set s, that block can be found in a receiver block of the adjacent set v. Algorithm 1 shows the complete algorithm for the VVC.

4.2.4 Set Dueling for Placement

When a block is placed into an adjacent set, we must decide where to place it. Some workloads benefit from placing the receiver of a victim block into the most-recently-used (MRU) position, while others benefit from placement in the LRU position depending on the behavior of the program. For example, benchmarks such as `187.facerec` and `401.bzip2` perform well when evicted blocks are placed in MRU position, but `179.art`, `188.ammp` do not. For some workloads, the

MRU insertion policy might be better because a victim block is not be accessed immediately, but it will be accessed soon so it will still be in the adjacent set making its way down the LRU stack. For other workloads, the LRU insertion policy might be better because 1) when a block will be accessed again, it will be accessed soon, or 2) many victim blocks placed in adjacent sets will not used for a long time and might as well be evicted instead of more useful data; after all, this is the point of the LRU replacement policy.

We used set dueling [38] to determine whether to place victims into the MRU or LRU position. A small number of sets in the cache are dedicated to LRU placement policy and a another small set of sets to MRU placement. An 11-bit counter is incremented whenever there is a miss to one of the dedicated LRU sets, or decremented when there is a miss to one of the dedicated MRU sets. The LRU placement policy is only used if the counter falls below 0; thus, the policy resulting in fewer misses in the dedicated sets wins and rest of the sets in the cache follow that policy. In this work we use 32 dedicated sets for each policy.

4.2.5 Why Not Just Evict Predicted Dead Blocks?

A natural question arises when considering dead block prediction: why not simply evict predicted dead blocks out of the cache instead of evicting LRU blocks into predicted dead blocks? Indeed, this technique has been proposed as an optimization for single-threaded workloads [25]. Note that blocks that are predicted dead and evicted from a set may be cached in the VVC; it seems counterintuitive to replace one dead block with another dead block.

Nevertheless, the VVC does give an advantage over evicting predicted dead blocks for two reasons. The first reason is that the VVC reduces conflict misses by effectively increasing associativity for hot sets. The second reason is because the VVC is robust in the presence of mispredictions. Consider a set S and its adjacent set S'. One of these sets is likely to be more active than the other due to the nonuniform distribution of accesses to sets. The fact that a block b in S is accessed makes it somewhat more likely that S is the more active set, since it was just accessed and S' was

Algorithm 1 Virtual Victim Cache with Trace based Predictor

```
On an access to set s with address a, PC pc
if the access is a hit in block blk then
    blk.trace ← updateTrace(blk.trace, pc)
    isDead ← lookupPredictor(blk.trace)
    if isDead then
        mark blk as dead
        return
else {search adjacent set for a dead block hit}
    v = adjacentSet(s)
    access set v with address a
    if the access is a hit in a dead block dblk then
        bring dblk back into set s
        return
else {this access is a miss}
    repblk ← block chosen by LRU policy
    updatePredictor(repblk.trace)
    v = adjacentSet(s)
    place repblk in an invalid/dead/LRU block in set v
    into position (MRU vs. LRU) determined by set dueling
    place block for address a into repblk
    repblk.trace ← updateTrace(pc)
    return
```

not. Suppose b is incorrectly predicted dead. Caching it in S' would be the right move since there are likely to be true dead blocks in S', while simply discarding b from S would be the wrong move and lead to a miss.

When prediction accuracy is high, replacing dead blocks is a reasonable policy. However, as we will see in Section 4.4.3, when prediction accuracy suffers because of shared cache contention, replacing dead blocks is disastrous for performance while the VVC provides a performance improvement.

4.2.6 Implementation Issues

Adjacent sets differ in one bit, bit k. The set adjacent to set index s is s exclusive-ORed with 2^k. A value of $k = 3$ provides good performance, although performance is largely insensitive to the choice of k. Victims replace receiver blocks in the MRU position of the adjacent set and are

Table 4.1: (a) Microarchitectural simulator parameters, and (b) dead block predictor parameters.

Parameter	Configuration
Issue width	4
Reorder Buffer	128 entry
Load/Store Queue	32 entry
Private L1 I-Cache	64KB, 2 way LRU, 64B blocks, 1 cycle hit
Private L1 D-Cache	64KB, 2 way LRU, 64B blocks, 3 cycle hit
Shared L2 Cache	2 MB, 16 way LRU, 64B blocks, 12 cycle hit
Virtual addresses	64 bits
Main Memory	270 cycle
Number of Cores	4

(a)

Parameter	Configuration
Trace encoding	15 bit
Predictor table entries	32, 768
Predictor entry	2 bit
Predictor overhead	8KB
Cache overhead	64KB
Total overhead	76 KB

(b)

allowed to be evicted just as any other block in the set. Evicted receiver blocks are not allowed to return to their original sets, i.e., evicted blocks may not "ping-pong" back and forth between adjacent sets.

Each cache block keeps the following additional information: whether or not it is a receiver block (1 bit), whether or not the block is predicted dead (1 bit), and the truncated sum representing the trace for this block (14 bits). The dead block predictor additionally keeps two tables of two-bit saturating counters indexed by traces. The predictor tables consume an additional 2^{14} entries $\times 2$ bit counters $\times 2$ tables = 64 kilobits, or 8 kilobytes.

4.3 Experimental Methodology

This section outlines the experimental methodology used in this study.

4.3.1 Simulation Environment

We use SPEC CPU 2000 and SPEC CPU 2006 benchmarks. We use a cycle-accurate microarchitectural multi-core simulator. This simulator is a heavily modified version of SimpleScalar [6]. This infrastructure enables collecting instructions-per-cycle figures as well as misses per kilo-

instruction, dead block predictor accuracy, and cache efficiency.

Table 4.1(a) shows the configuration of the simulated machine. Each benchmark is compiled for the Alpha EV6 instruction set. For SPEC CPU 2000, we use the Alpha executables that were at one time available from `simplescalar.com` compiled with DEC C V5.9008, Compaq C++ V6.2-024, and Compaq FORTRAN V5.3-915. For SPEC CPU 2006, we use binaries compiled with the GCC 4.11 compilers for C, C++, and FORTRAN. For most experiments, we model a 16-way set-associative cache to remain consistent with other previous work [28, 32, 38, 39], but Section 4.4.5 shows that our technique maintains significant improvement at lower associativities.

We use SimPoint [37] to identify a single two billion instruction characteristic region (i.e. *simpoint*) of each benchmark. For single-threaded workloads, the infrastructure simulates two billion instructions, using the first 500 million to warm microarchitectural structures and reporting the results on the next 1.5 billion instructions.

4.3.2 Simulating CMP Workloads

We use a multi-core simulator to measure the performance of the virtual victim cache in the presence of multiple threads. The simulator simulates four cores accessing a shared L2 cache. We choose ten combinations of SPEC CPU benchmarks to run simultaneously to represent chip-multiprocessor (CMP) workloads. These combinations are identified in the x-axis of the relevant graphs in Section 4.4. For the multi-core simulations, we warm the caches for 500 million instructions, then simulate each of the four threads until every thread has committed 250 million instructions. Although some threads continue to execute beyond 250 million instructions to model cache contention for other threads that have not yet reached the limit, we only report statistics based on the first 250 million instructions. We report IPC throughput, i.e., $\sum_{i=1}^{n} \text{IPC}_i$, giving the sum of the individual IPCs for n benchmarks ($n = 4$ in our case), normalized to the baseline LRU cache. Our approach is consistent with methodology from recent papers on caches in multi-core microarchitectures [33, 40].

We choose a memory-intensive subset of the benchmarks based on the following criteria: a

benchmark is used if it (1) does not cause an abnormal termination in the baseline sim-outorder simulator for the chosen simpoint, and (2) if increasing the size of the L2 cache from 1MB to 2MB results in at least a 5% speedup. Benchmarks that experience negligible improvement from a higher capacity cache are unlikely to be affected positively or negatively by our optimization.

4.3.3 Dead Block Predictor Details

4.3.3.1 Accounting for State in the Predictor

The dead block predictor keeps two 16K-entry tables of 2-bit counters. Each cache block includes additional VVC metadata: 1 bit that is true if the block is predicted dead, 1 bit that is true if the block is a receiver block, and 15 bits for the current trace encoding for that block. The overhead of the predictor and VVC metadata is 76KB which is 3.4% of the total 2MB cache space (including both the data and tag arrays). An accounting of the predictor overhead is given in Table 4.1(b).

4.3.4 Estimating Dead Block Hit latency

An L2 cache access takes 12 cycles in the simulated environment. CACTI 5.1 [51] estimates that an additional tag match in the adjacent set, made once the initial tag match in the original set has failed, consumes an extra 2 cycles[1].

An additional sequential tag match latency is simulated for VVC hits. Experiments show that IPC is insensitive to additional L2 hit latency as long as that latency is a small fraction of the miss latency. For instance, negligible change results when pessimistically assuming that VVC hits take double the normal hit latency because a) most accesses hit in the normal L2 cache, and b) a VVC hit at twice the L2 hit latency avoids a much larger L2 miss latency.

[1]We considered doing the two tag matches in parallel, but decided against it because of power and complexity concerns; in essence, this would be no better in terms of power than doubling the associativity of the cache.

4.3.5 Measuring Cache Efficiency

Cache efficiency is a statistic defined by Burger *et al.* [5] to quantify the average amount of time blocks in the cache contain live data. Cache efficiency is computed as:

$$E = \frac{\sum_{i=0}^{A \times S - 1} U_i}{N \times A \times S}$$

where N is the total number of cycles executed, A is the number of blocks per set, S is the number of sets in the cache, and U_i is the total number of cycles for which cache block i contains live data, i.e., data that will be referenced again before it is evicted. Thus, cache efficiency is the average of the live cycles for each cache block. The performance simulation infrastructure collects block live times and produces cache efficiency as an output.

4.4 Results

In this section we discuss results of our experiments. We investigate the virtual victim cache with a skewed dead block predictor in the context of a baseline 2MB 16-way set associative cache as well as a 2MB fully associative cache and a 2MB cache enhanced with a 64KB victim cache. Note that both the fully associative cache and the 64KB victim cache are infeasible in hardware, requiring 32K entry and 1K entry associative memories, respectively. We choose a 64KB victim cache because it requires approximately the same amount of SRAM, including the tag array, as the extra structures of the VVC.

We also compare the virtual victim cache with two other techniques:

1. Dynamic insertion policy (DIP) using set dueling [38]. This technique samples from 32 dedicated sets performing block placement into the MRU position as well as 32 dedicated sets placing blocks into the LRU position. The policy incurring the fewest misses is used for the rest of the cache.

2. A replacement policy based on dead block prediction. We use our skewed dead block pre-

dictor to drive a replacement policy in which a predicted dead block is chosen as a victim, or the LRU block if there is no predicted dead block. This policy is similar to the technique described by Kharbutli and Solihin [25].

4.4.1 Reduction in L2 Misses

Figure 4.3 shows the impact of the VVC on L2 misses per thousand instructions (MPKI). Figure 4.3(a) shows the raw MPKI values for each benchmark and cache configuration. The average MPKIs for the baseline and fully associative LRU caches are approximately 9.7. The real victim cache yields 8.9 MPKI. The VVC provides an average MPKI of 7.2, an improvement over the baseline of 26% and over the real victim cache of 19%. The VVC provides its best reduction in raw MPKI for `181.mcf`, reducing misses by approximately 30 MPKI. DIP achieves 7.5 MPKI while dead block replacement achieves 7.1 MPKI. As a lower limit on MPKI, we include results for Belady's MIN optimal replacement policy [4] which results in an average MPKI of 4.9. The VVC achieves an average MPKI only 46% higher than optimal, while LRU is 96% higher than optimal.

Note that it is not enough to compare MPKI for VVC with the other techniques because the VVC incurs a slight overhead when there is a hit to the adjacent set. However, we also report improved performance in a cycle-accurate simulator taking into account this overhead.

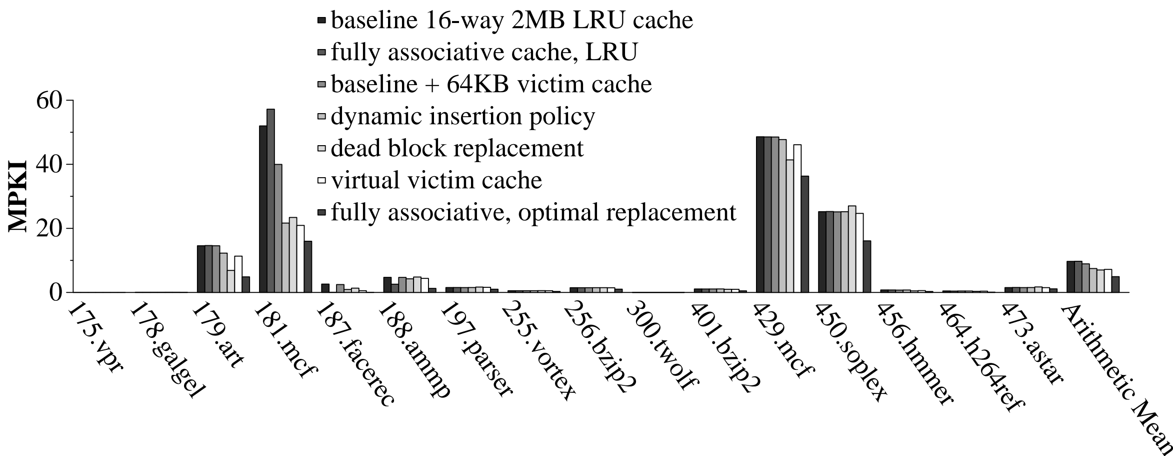


Figure 4.3: L2 cache misses per thousand instructions

4.4.2 Single-thread IPC Improvement

Reducing cache misses translates into improved performance. Figure 4.4 shows the instructions-per-cycle rates given by the VVC as well as other techniques. In 11 of the 16 single-threaded workloads, the virtual victim cache outperforms DIP. In a different 11 of the 16 benchmarks, the virtual victim cache outperforms dead block replacement.

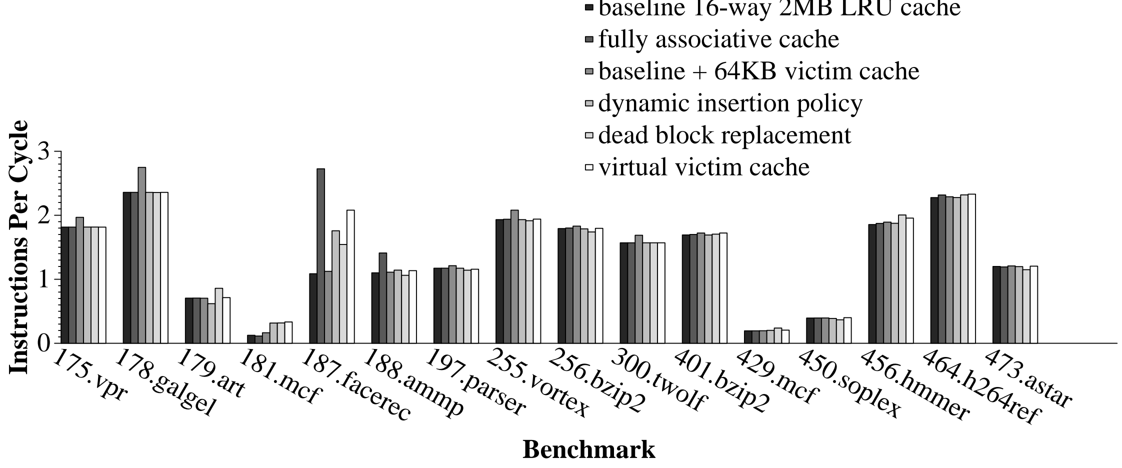


Figure 4.4: IPC improvement

Figure 4.5 shows the speedup computed by dividing the improved IPC by the baseline IPC for each benchmark, as well as the geometric mean speedup. The geometric mean speedup for the VVC is 12.1%, compared with 5.2% for the real victim cache and 7.1% for the fully associative cache, 8.7% for DIP, and 10.3 for dead block replacement. Two benchmarks in particular, `181.mcf` and `187.facerec`, yield remarkable speedups of 167% and 91%, respectively, while other benchmarks show more modest improvements. No benchmark is significantly slowed down by the VVC; `197.parser` is the worst case in terms of slowdown, with a speedup of -1.8%. Note that dead block replacement slows down several benchmarks, the worst of which is `450.soplex` with a speedup of -7.0%.

Although the difference between the performance of the virtual victim cache and dead block replacement is small, there is a very important reason to prefer the virtual victim cache: for a shared multi-core cache, dead block replacement experiences a significant performance degrada-

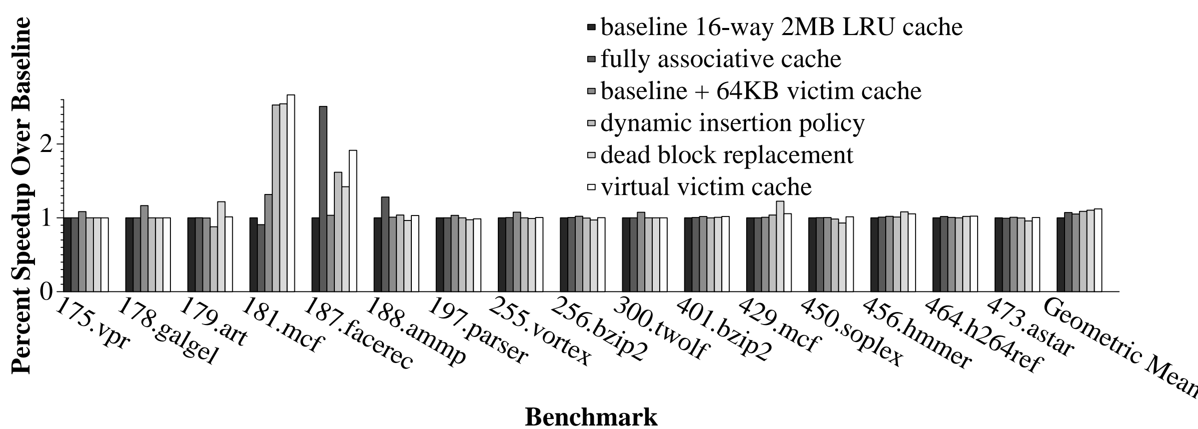


Figure 4.5: Speedup

tion while the virtual victim cache continues to provide a performance improvement as we will see in Section 4.4.3.

4.4.3 CMP Throughput Improvement

We simulate the various cache configurations in a multi-core simulator. Figure 4.6 shows the normalized IPC throughput for ten combinations of benchmarks representing CMP workloads.

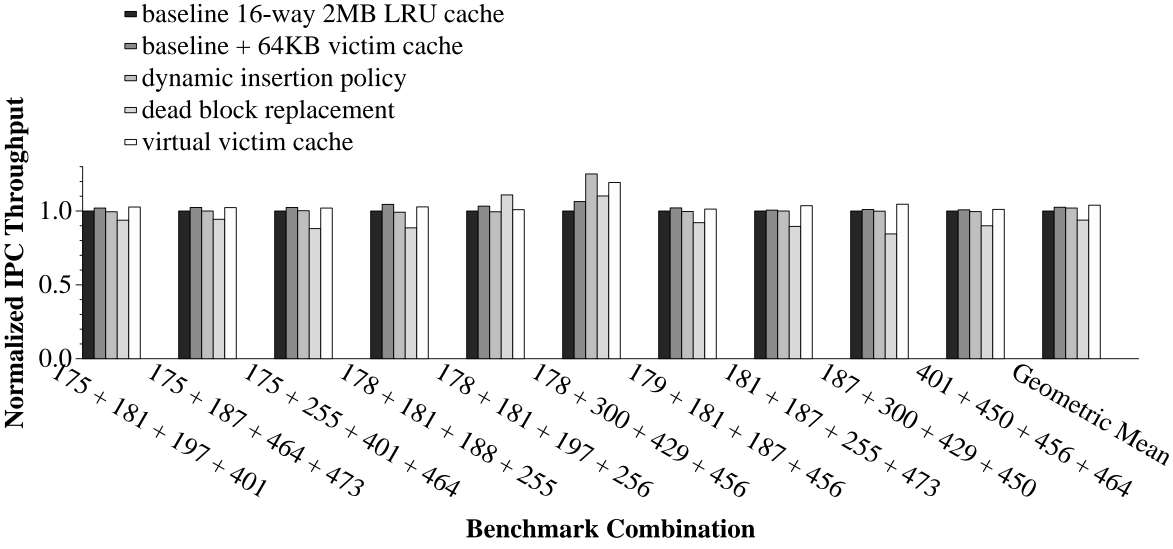


Figure 4.6: Normalized throughput IPC for multiple threads and a 2MB cache

The virtual victim cache achieves a normalized throughput IPC of 1.04, compared with 1.025 for the real victim cache, 1.020 for DIP, and 0.94 for dead block replacement. Although dead block

replacement performs well for single-threaded workloads, it performs quite poorly here while the VVC continues to improve performance. We believe that this is because dead blocks become less predictable as several different program access patterns target the same set. The dead block replacement policy can only consider blocks in one set for eviction. If it often makes the wrong decision for hot sets, performance suffers. However, with the VVC, the adjacent set is somewhat less likely to be hot than the original due to the reasoning in Section 4.2.5. Thus, it is more likely that there will be true dead blocks in the adjacent set than in the original set.

Figure 4.7 illustrates the difference in terms of predictor accuracy for CMP versus single-threaded workloads. The figure shows the percentage of accesses that were incorrectly predicted as the last access to a block, potentially resulting in a live block being replaced. For single-threaded workloads, the average false positive rate is 3.5%, so dead block replacement and the VVC can have similar performance. (A notable exception to this trend is `429.mcf`, which achieves better performance than the VVC despite relatively poor accuracy.) However, for CMP workloads, the false positive rate is 13.0%, resulting in poor performance for the dead block replacement policy but sustaining a benefit for the VVC.

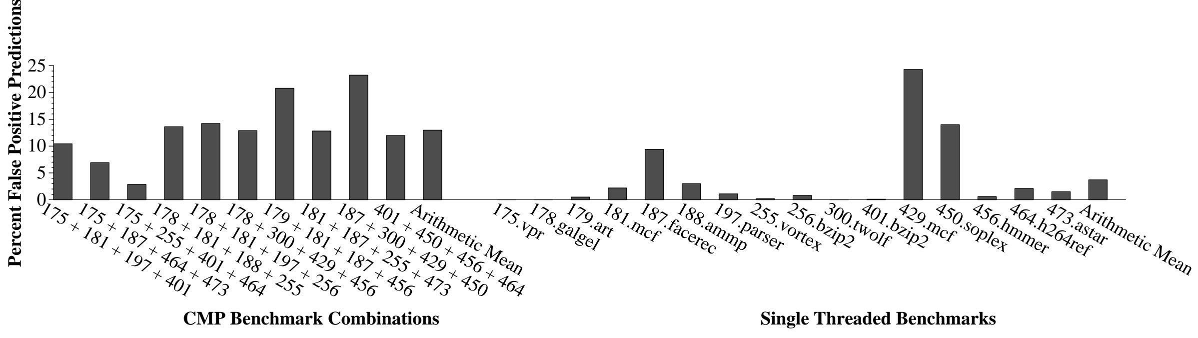


Figure 4.7: Shared cache contention results in more false positive predictions.

4.4.4 Improvement in Cache Efficiency

As stated in the introduction, the VVC improves cache efficiency by making sure more cache blocks are live. Figure 4.8 quantifies this improvement for each CMP benchmark combination and each single-threaded benchmark. The baseline average cache efficiency for the single-threaded

workloads is 0.412, compared with 0.523 for the VVC, an improvement of 27%. For the CMP workloads, the baseline average cache efficiency is 0.243, while the VVC achieves an average efficiency of 0.394, an improvement of 62%.

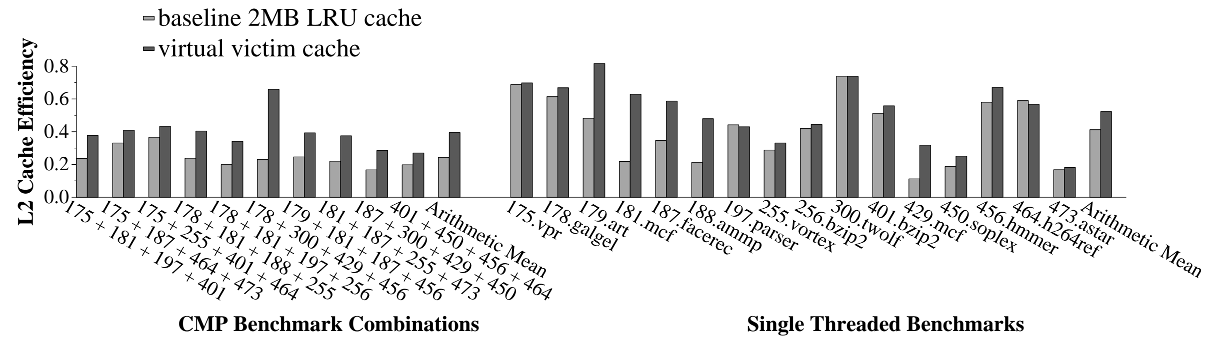


Figure 4.8: Cache efficiency

4.4.5 Reduction in Cache Area

We have shown that the VVC can deliver improved performance by reducing the number of L2 cache misses. Alternately, the VVC can reduce the area consumed by the L2 cache, leading to equivalent performance with reduced power and area requirements. Figure 4.9 illustrates the ability of the VVC to deliver equivalent performance with a reduced capacity cache. Figure 4.9(a) shows the average MPKI for the baseline cache and the VVC for a variety of cache sizes obtained by increasing the associativity of the cache from 8 through 16. At a capacity of 1.7MB representing an associativity of 13, the VVC achieves an average MPKI of 9.9, just above the MPKI of the 2MB baseline cache at 9.7. At a capacity of 1.8MB representing an associativity of 14, the VVC outperforms the baseline with an MPKI of 9.1. Figure 4.9(b) shows how these MPKIs translate into performance. The baseline harmonic mean IPC for a 2MB cache is 0.64, compared with 0.63 for a 1.7MB VVC and 0.68 for a 1.8MB VVC.

At the 1.7MB capacity, the VVC reduces the number of SRAM cells required by 16% including the SRAM cells for data, tags, predictor structures and metadata. At the 1.8MB capacity, the VVC reduces the SRAM cells needed by 9.5%.

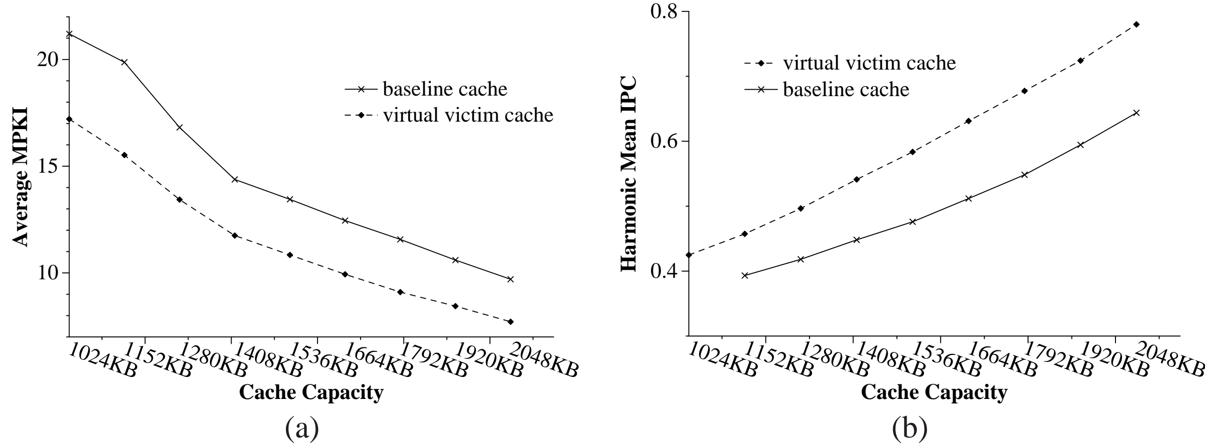


Figure 4.9: Reduction in cache area

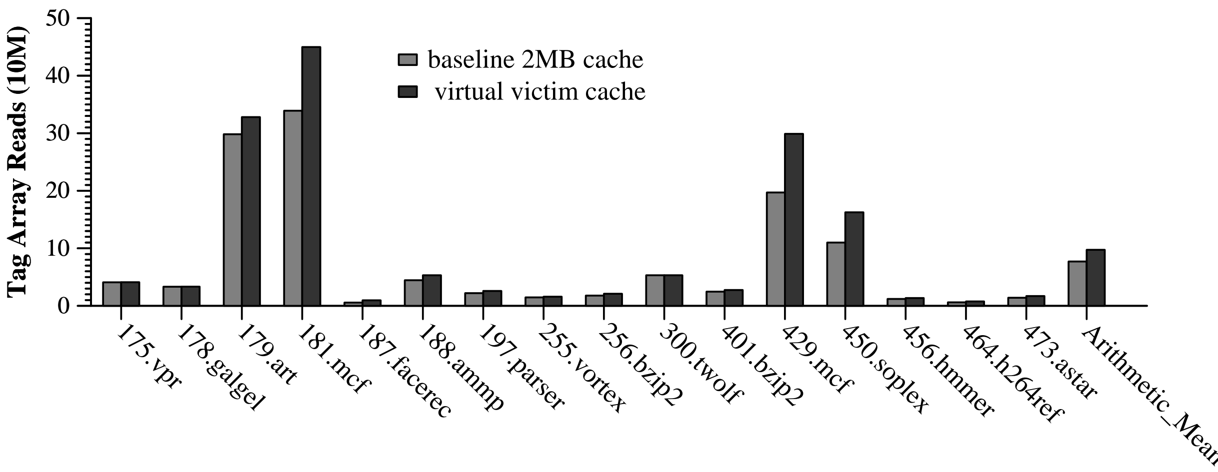


Figure 4.10: Increase in tag array reads due to VVC

4.4.6 Increase in Tag Array Access

The improved performance of the VVC comes at the expense of extra reads to the tag array. Most accesses hit in the cache; however, every time a tag match fails in a set, the adjacent set must also be searched. Figure 4.10 shows the number of reads to the L2 tag array for the baseline 2MB cache as well the VVC. In two billion executed instructions, the L2 tag array is read on average 77 million times in the baseline cache and 97 million times in the VVC, an increase of 26%. To put it another way, the number of tag array reads in the baseline cache is 3.9% of the total number of instructions executed, versus 4.9% for the VVC.

4.5 Reducing Dead Blocks without a Dead Block Predictor

So far we have focused on reducing dead blocks by only accurately predicting the dead blocks with PC-based predictors. Recent works on replacement and insertion policy [14, 38, 55] show that substantial improvement can be achieved by changing the insertion position in the LRU stack. The "dead-on-arrival"blocks can be eliminated using dynamic insertion policy. In the next chapter we show that the dead time of these blocks can be significantly reduced only by segmenting the cache into non-referenced blocks and referenced blocks. By evicting non-referenced blocks that will not be used again in the future earlier, can significantly reduce dead time of these dead-on-arrival blocks.

CHAPTER 5: DEAD TIME REDUCTION THROUGH CACHE SEGMENTATION

Modern processors have large on-chip caches to mitigate off-chip memory latency. The least recently used (LRU) replacement policy represents the cache blocks in a set as a recency stack where the most recently used (MRU) block is at the top of the stack. This policy picks the LRU block as the replacement candidate as blocks recently used are more likely to be accessed in the near future. However, in a three-level cache hierarchy, this temporal locality is filtered by the L1 and L2 caches. Thus, LRU performs poorly in the last-level cache (LLC) [14, 29, 38]. Recent proposals change the insertion policy, placing incoming blocks in different stack positions to adapt to workload characteristics [14, 38, 55]. These proposals are resistant to *scans* and to *thrashing*. A scan is a burst of accesses to data that are never reused before being evicted. Thrashing workloads have a working set size greater than the cache size, causing useful cache blocks to be evicted. Though these proposals take care of scanning and thrashing workloads, they depend on the recency levels the LRU policy and insert blocks adaptively in either recent or non-recent positions. Thus, these techniques cannot be applied to policies with no inherent levels, for example random replacement. In this paper we propose a decoupled technique that is scan and thrash resistant and can be used with any replacement policy. Recent insertion-based proposals require low overhead but cannot detect blocks that will not be reused, i.e. zero-reuse blocks. Replacement policies that can *bypass* zero-reuse blocks [20, 25], i.e., pass them along to the CPU without placing them in the cache, require significant hardware overhead. Our proposed technique is not only scan and thrash resistant, but can automatically detect zero-reuse blocks irrespective of the base replacement policy without significant overhead.

We propose segmenting cache sets into referenced and non-referenced blocks, i.e. blocks that have been referenced at least once since being placed in the cache and blocks that have not yet been referenced since their initial access. This technique attempts to maintain the best number of non-referenced and referenced blocks depending on the workload characteristics. We propose

a scalable low-overhead *segment predictor* based on sampling that can predict the best segmentation at runtime. It adjusts the segmentation based on cache access patterns. It can resist scans and thrashing by evicting non-referenced blocks. This technique is completely decoupled from the replacement policy. Dynamic Cache Segmentation(DCS) decides whether a victim should be selected from non-referenced blocks or referenced blocks. Any replacement policy can be used to choose the victim from that specific list. The technique can improve performance with inexpensive policies like Not Recently Used (NRU) and random. This scan and thrash resistant technique requires far less space than LRU and can still detect zero-reuse blocks. Dynamic segmentation with automated bypass can improve performance with random replacement, requiring only half the space overhead of LRU. It can also be used for multi-core workloads with a conjunction of any cache partitioning techniques. The contributions of this paper are:

- We propose a dynamic cache segmentation technique that attempts to keep the best number of non-referenced and referenced blocks in cache sets. We propose a sampling-based scalable low-overhead technique to predict the best segmentation.
- We show that the segmentation technique can be decoupled from the replacement policy and can work with inexpensive policies like NRU and random. It also automatically detects bypassing opportunities without any extra overhead irrespective of the replacement policy being used.
- We show that cache segmentation complements current shared cache partitioning techniques. Dynamic cache partitioning even with a default random policy can outperform LRU using half the space overhead.

In a single-core processor with 2MB LLC, DCS outperforms LRU policy on average by 5.2% with NRU replacement and on average 2.2% with random replacement, with a memory intensive subset of SPEC CPU 2006 benchmarks. Evaluation with multi-programmed workloads shows that the technique improves the performance of a 8MB LLC in a four-core system on average by 12% with random replacement requiring half the space of LRU.

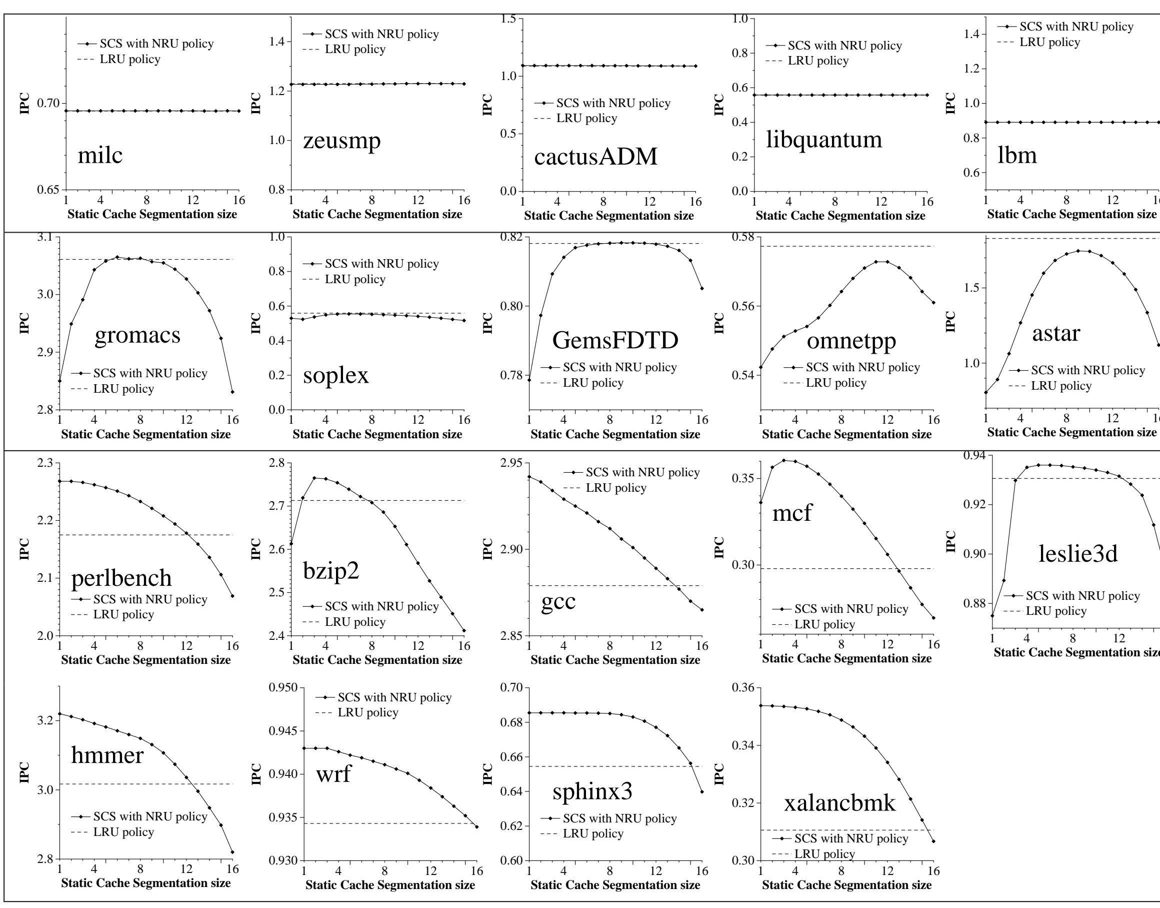


Figure 5.1: Effect of static segmentation. Row 1 shows the benchmarks with no effect of segmentation. Row 2 shows LRU friendly benchmarks and row 3 shows benchmarks which are effected by segmentation.

5.1 Motivation

Recent work has shown that last-level caches (LLCs) perform poorly with the LRU replacement policy [14, 29, 38]. Upper level caches filter out the temporal locality, destroying the property upon which LRU relies, that most recently used block will be used again in the near future. LRU performs poorly in the LLC when the working set is larger than the cache size (thrashing) or bursts of non-temporal references evict the active working set (scanning).

5.1.1 Addressing Workload Behavior

Recent proposals addressing these effects include changing the insertion policy [14, 38] so that thrashing and scanning workloads can perform well in the LLC. Dynamic Insertion Policy (DIP) [38] avoids thrashing by attempting to keep a portion of the workload in the cache by inserting blocks responsible for thrashing into the LRU position. Thus, some part of the working set always remains in the cache and thrashing blocks can not pollute the whole cache. DIP uses LRU for all other non-thrashing workloads, so it performs poorly for workloads where frequent scans discard the working set in the LLC. RRIP [14] is an insertion policy with NRU replacement policy that is both scan and thrash resistant. It inserts thrashing blocks at the end of the NRU stack and other blocks near the end of the NRU stack. Thus RRIP adapts to both thrash resistant DIP and scan resistant LFU. The performance of RRIP depends on the number of NRU levels as non-scanning blocks must be re-referenced before they are evicted.

5.1.2 A Decoupled Flexible Policy

Clearly, it would be best to have a policy that is resistant to both scanning and thrashing but also works with mixed access patterns. Both DIP and RRIP are dependent on a policy that divides the blocks of a set in levels of recency (LRU and NRU) to insert blocks into a specific position. We propose a mutable technique that is decoupled from the choice of replacement policy and works with all access patterns. Additionally, the technique can detect when to bypass blocks without extra overhead. Most blocks brought into the LLC are never used again. Recent work has proposed techniques where zero-reuse blocks are identified and evicted earlier [22, 23, 25]. However they require significant extra overhead to identify blocks that are never reused. We propose a mutable scan and thrash resistant technique with automated bypass that works with any replacement policy without significant overhead.

Table 5.1: Comparison among techniques

Technique	Thrash resistant	Scan resistant	Detect bypass	Decoupled	Overhead
DIP	yes	no	no	no	low
RRIP	yes	yes	no	no	low
DCS	yes	yes	yes	yes	low

5.1.3 Segmenting Cache Sets with Prediction

We propose to segment cache sets into two parts: the *referenced list* and *non-referenced list*. Blocks that have been referenced again after being brought into the cache belong to the referenced list while blocks that have not been reused are in the non-referenced list. A *segment predictor* predicts the best segmentation for the two lists. It predicts the best segment size for the non-referenced list; the resulting size for the reference list is simply the size of the non-referenced list subtracted from the total number of ways. This technique attempts to keep the best number of referenced and non-referenced blocks by choosing replacement victims from the list that has more blocks than the predicted best number of blocks. This technique can address thrashing and scanning blocks by keeping the non-referenced list as a singleton block. It also chooses the best segmentation size for mixed access patterns. Table 5.1 shows the instructions-per-cycle (IPC) with static segment sizes for the SPEC CPU 2006 benchmarks. We show the IPC when each benchmark is run with static segment size 1 to 16. The maximum IPC is achieved at different segment size for different benchmarks. For scanning and thrashing workloads, the best static segment size of the non-referenced list is one, where mixed access pattern workloads perform best with variable segment sizes. For example, perl performs best with segment size 1 but bzip performs best with segment size 4. This clearly shows that we need an adaptive segmentation to determine the best segment size at runtime and unlike previous policies [14, 38], dynamically determining the insertion position is not enough to achieve the best possible performance.

Our technique selects the list from which the victim should be chosen. Any replacement policy

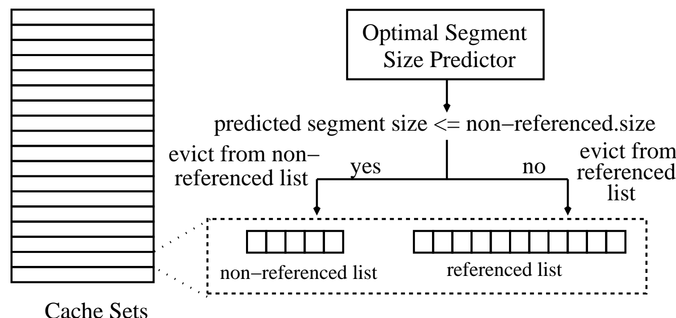


Figure 5.2: Logical view of the replacement scheme

can be used within a list, even random replacement. Thus, the replacement policy is decoupled from the segmentation technique. Another advantage is that segmentation automatically detects when to bypass lines. When the predicted best segmentation size is one, blocks arriving to the LLC cannot be referenced. So segmentation can identify a zero-reuse block without any extra overhead.

5.2 Dynamic Segmented Cache

Dynamic cache segmentation (DCS) partitions cache sets into non-referenced blocks and referenced blocks. It attempts to keep the best partition for all sets. If one of the lists has more blocks than it is suppose to, the next victim is chosen from that list. Figure 5.2 shows the logical view of the segmented cache. The segment predictor predicts the best partition size for a given workload. When a block must be replaced from a set, either list can be chosen depending on the current partition of the set. DCS needs only one bit per cache block to differentiate referenced blocks from the non-referenced blocks.In this section we describe in detail the idea of DCS as well as several versions of the policy based on different underlying replacement policies.

5.2.1 Predicting the Best Segmentation

Here we discuss how the segment predictor predicts the best segment size. We use set-dueling with sampling to determine the best segment size [38]. Set-dueling samples a few “leader” sets from

the cache that always implement one particular policy chosen from the possible policies. Counters are used to keep track of which policy yields the fewest misses and all of the other cache sets, i.e. "follower" sets, follow the best policy. We consider two designs: one uses some sampled sets of partial tags kept externally and the other uses sampled sets in the cache. These randomly chosen sample sets imitate a decision tree used to predict the best segmentation. We consider five segments sizes for the non-referenced list to choose from. They are segment size 1, 4, 8, 12 and 16 for a 16-way set-associative cache. Considering more segment sizes does not significantly improve performance. In the segment predictor, two segment sizes set-duel at each level. The winner of one level determines the next competitor segment size for the next level. Figure 5.3 shows each

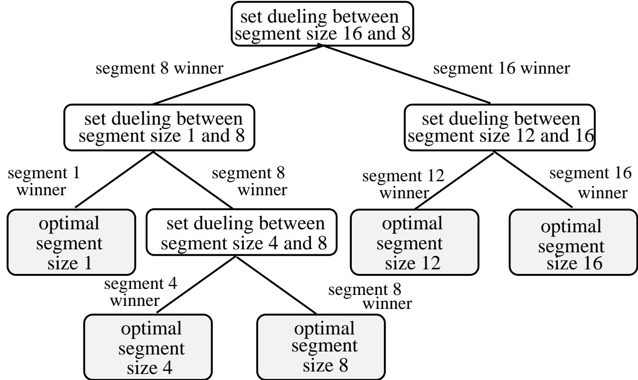


Figure 5.3: Decision Tree Analysis to find the best non-referenced segment size

leaf representing the final best size. Each node in the tree determines the next segment size to be considered. For example, if between segment sizes 8 and 16, segment size 8 is winning, that means that having at most 8 blocks in the non-referenced list will yield the best performance. In order to determine which partition between 1 and 8 is better, the next level duels between segment sizes 1 and 8. This way each node selects the next segment size until it reaches a leaf. We use three kinds of leader sets. Two leader sets are static and always uses segment sizes 8 and 16, ensuring that the predictor will respond to workload phase changes. Depending on which leader set is winning, the adaptive leader picks the segment size following the decision tree. This way the technique can consider variable segment sizes using only three types of leader sets. We can visualize DCS as a

three-dimensional decision making engine while set-dueling is only one-dimensional as depicted in Figure 5.4. Set-dueling can make only one of the two decisions. In a two-dimensional decision engine, we can choose one of four outcomes. We have added another dimension so that we can choose one of eight decisions. However, we find that choosing one of five outcomes is sufficient. This technique is different from multi-set-dueling [33] where each decision has its own leader sets and they simultaneously set-duel at each level in groups of two. Multi-set-dueling is not scalable as the number of leader sets has to grow linearly with the number of outcomes being considered.

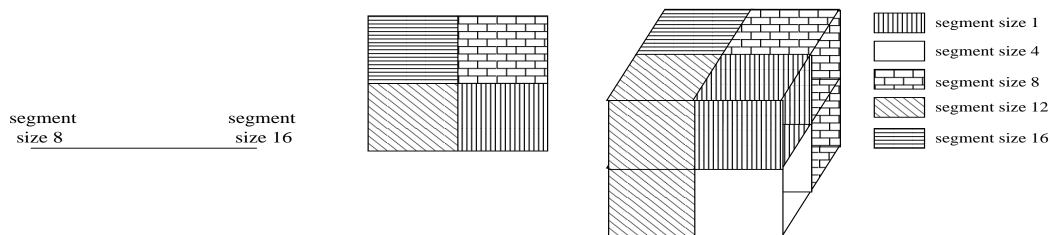


Figure 5.4: 3D model of the Optimal Segment Predictor

5.2.2 Decoupled from Replacement Policy

Partitioning the cache sets into non-referenced and referenced list enables decoupling the replacement policy from the segmentation. DCS only decides which list should be used for choosing replacement victims depending on the current and best list sizes. Each list is free to use any replacement policy to choose the replacement victim. DCS can be used with simple and light replacement policies like Not Recently Used (NRU) and random replacement. In Section 4.4 we show that DCS with simple NRU and random policies outperforms the more costly LRU policy.

Dynamic Segmentation with NRU Policy The NRU policy approximates the recency stack based LRU policy but using only one bit, called the NRU bit, per cache block. The NRU recency stack has only two levels, so each level can have more than one block. By contrast, the LRU policy is organized such that each block will reside in a distinct level. In NRU, when a new line is placed in the cache, its NRU bit is set to zero moving the cache block to the top of the recency stack. On a cache miss a block whose NRU bit is set to one is selected as the victim. Since there can be many

blocks in the lower recency level, the NRU victim search starts from a fixed cache way. If there is no block with its NRU bit set to one, then all of the NRU bits are set to one. We adapt DCS to use the NRU policy for each list to select the victim. Each cache block needs one bit to identify referenced and non-referenced lines and one bit to track the NRU policy within the lists.

Dynamic Segmentation with Random Policy The random replacement policy selects a victim pseudo-randomly from a cache set. It has no information about temporal locality, so it tends to perform poorly for LRU friendly workloads. It also has no information about thrashing/scanning workloads so it performs poorly for them as well. Nevertheless, it is the simplest replacement policy with no cache block overhead. We show that DCS works well even with random replacement. The segment predictor picks the list that from which the victim should be chosen and a block is randomly chosen from that list. This provides the random policy some information about the workload access pattern. In Section 4.4 we show that DCS with random replacement can outperform the LRU policy. Per block overhead in this case is just one bit that differentiates non-referenced and referenced lines.

Ignoring Segmentation Figure 5.1 shows that there are some workloads that will always perform better with LRU than with any segmentation size. In these cases we can choose to ignore the segmentation. We can implement a version of DCS that incorporates non-segmented policies like NRU/LRU. After the predictor chooses the best segmentation, that segmentation set-duels with the non-segmented policy. If the winner of this final stage is non-segmented policy, then the segment predictor concludes that the workload is LRU-friendly and the replacement decision ignores the segmentation. In the case of NRU policy, it picks one of the non-recently used blocks from the whole set.

5.2.3 Automated Bypass

The LLC frequently has a large fraction of zero-reuse blocks [20,25]. Bypassing these blocks to the core can significantly improve LLC efficiency. Bypassing allows the LLC to save space for other useful blocks in the cache by not placing a suspected zero-reuse block in the cache. One advantage

of DCS is that it can inherently detect zero-reuse blocks without any extra overhead. A large number of zero-reuse blocks are referenced when there are thrashing/scanning workloads. DCS can detect these workloads and identify the phases where it can safely bypass an incoming block. When the current partition size is one, DCS can safely bypass incoming blocks. The policy keeps one incoming block in the LLC after every 32 misses to prevent the working set from becoming stale.

5.2.4 Ensuring Thrashing Resistant

DCS does not enforce the partition size in cache sets. Rather, it attempts to gradually converge to the partition size predicted by the segment predictor. When bursts of lines are placed in the cache they all start from the non-referenced list. This initialization can cause the number of lines to far exceed the predicted best size. Figure 5.5 shows the run time non-referenced segment size

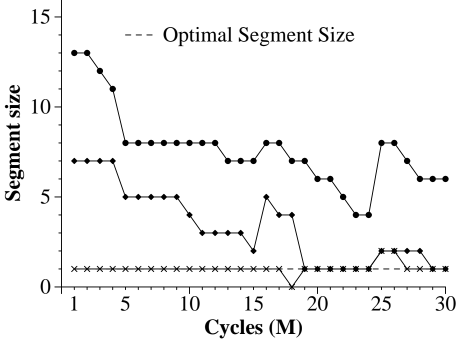


Figure 5.5: Runtime non-referenced segment size for three representative sets from 483.xalancbmk

of three representative sets for the benchmark `483.xalancbmk` chosen randomly. The segment predictor predicts the best non-referenced list size to be one. Some sets quickly converge to the best size while some converge slowly. However, there are also a large number of sets where bursts of accesses always keep the current partition far more than the predicted best size of one. We solve this problem by inserting blocks in the not-recent part of the not-referenced list. This ensures that thrashing blocks are discarded from the cache, while maintaining part of the working set in the

cache.

5.2.5 Mutable Policy

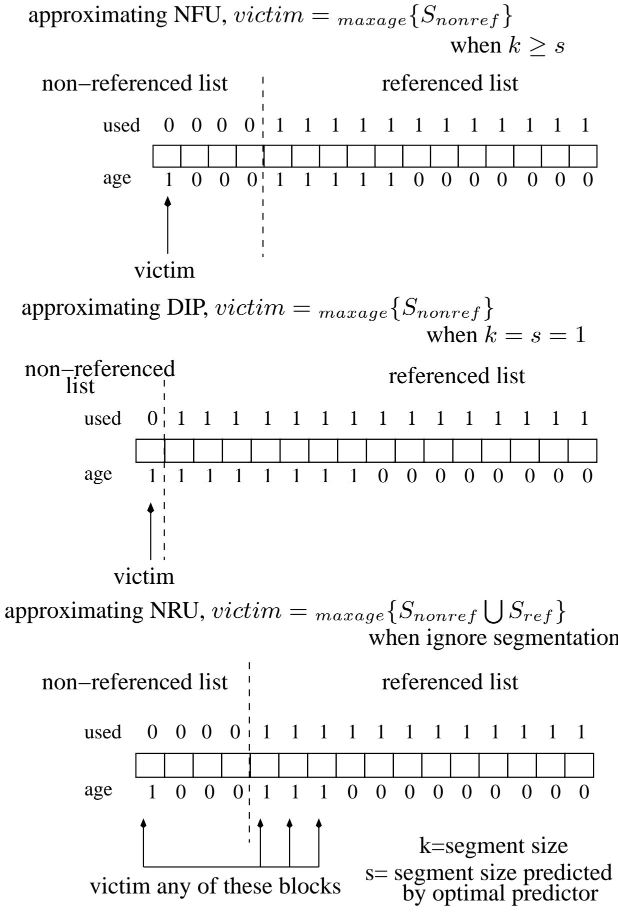


Figure 5.6: Three cases of Dynamic segmentation

Depending on the underlying replacement policies allowed by implementation constraints, DCS can mutate among LRU, LFU, DIP and in between. Using NRU as the baseline policy it mutates between NRU, which approximates LRU, Not Frequently Used (NFU), which approximates LFU, DIP (with underlying NRU policy), and in between. Let S is the set of the blocks in the lists. $S_{nonref} = \{a_i\}_{i=1}^{k}$ and $S_{ref} = \{a_i\}_{i=1}^{assoc-k}$. If k is the size of non-referenced list and s is

the size predicted by the optimal predictor, according to our policy

$$\text{victim} = \begin{cases} {}_{maxage}\{S_{nonref}\}, \text{ if } k \geq s \\ {}_{maxage}\{S_{ref}\}, \text{ if } k < s \\ {}_{maxage}\{S_{ref} \bigcup S_{nonref}\}, \text{ ignore segmentation} \end{cases}$$

The first case in Figure 5.6 shows DCS approximating to LFU. In this case blocks in the non-referenced list have been accessed just once. However all the other blocks in the referenced list have been accessed more than once. If there are no blocks in the non-referenced list, then one block from the referenced list is evicted. In this case that block is a Not Frequently Used block in the whole set. Since we implement a dynamic insertion policy when the segment predictor predicts the non-referenced segment size to be one, DCS falls back to DIP [38] with default replacement policy NRU. With DIP, blocks are inserted at the end of the LRU list. With DCS, when best non-referenced segment size is predicted to be one, incoming blocks are inserted in the NRU position of the non-referenced list. This ensures that thrashing blocks are evicted as soon as possible. This is depicted in the 2nd case in the Figure 5.6. In the last case, the replacement policy ignores the segmentation. In this case since the underlying policy is one-bit NRU, it replaces the Not Recently Used block irrespective of which segment it resides in. Depending on the underlying policy used in the referenced and non-referenced lists, DCS can fall back to other policies as well. For example when FIFO policy is used in the non-referenced list it becomes a 2Q policy [16].

5.3 DCS with Shared Cache Partitioning

DCS partitions cache sets into referenced and non-referenced lists. This technique can partition shared cache sets depending on the mixed access pattern from all the workloads. However, it does not take into account the different behavior of different programs or threads. This can be solved by using DCS in conjunction with any adaptive way-partitioning technique [40, 49, 50]. Our shared-cache-aware DCS technique segments the ways belonging to a specific thread assigned by a cache partitioning technique. Thus, DCS complements way-partitioning. Since cache segmentation is

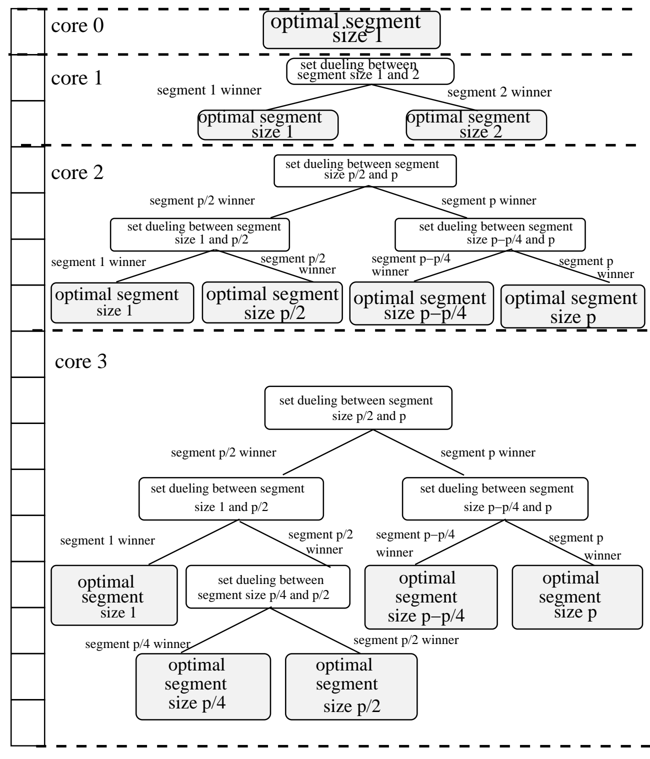


Figure 5.7: Dynamic Segmentation in each partition of UCP

decoupled from replacement policy, segmenting the partitioned cache can work with random replacement. Partitioning the partitions essentially limits the number of replacement candidates such that even choosing a candidate at random is often sufficient. In Figure 5.7 we show how DCS determines the segment size from each UCP partition [40]. If the partition size is p, the segment size is chosen from p, $p - p/4$, $p/2$, $p/4$ and 1.

5.4 Experimental Methodology

This section outlines the experimental methodology used in this study.

We use a memory-intensive subset of 19 SPEC CPU 2006 benchmarks chosen with a methodology outlined below. We have grouped these 19 benchmarks according to their response to static segmentation shown at table 5.1. Benchmarks like *433.milc* experience no effect under segmentation. Some of them are very LRU-friendly. Table 5.3 shows the benchmark grouping. We use a

Table 5.2: SPEC CPU 2006 benchmarks with LLC cache misses per 1000 instructions for LRU and optimal (MIN), instructions-per-cycle for LRU for a 2MB cache, and number of instructions fast-forwarded to reach the simpoint (B = billions).

Name	MPKI (LRU)	MPKI (MIN)	IPC (LRU)	FFWD	Name	MPKI (LRU)	MPKI (MIN)	IPC (LRU)	FFWD
astar	2.275	2.062	1.829	185B	bzip2	0.836	0.589	2.713	368B
cactusADM	13.529	13.348	1.088	81B	gcc	0.640	0.524	2.879	64B
GemsFDTD	13.208	10.846	0.818	1060B	gromacs	0.357	0.336	3.061	1B
hmmer	1.032	0.609	3.017	942B	lbm	25.189	20.803	0.891	13B
leslie3d	7.231	5.898	0.931	176B	libquantum	23.729	22.64	0.558	2666B
mcf	56.755	45.061	0.298	370B	milc	15.624	15.392	0.696	272B
omnetpp	13.594	10.470	0.577	477B	perlbench	0.789	0.628	2.175	541B
soplex	25.242	16.848	0.559	382B	sphinx3	11.586	8.519	0.655	3195B
wrf	5.040	4.434	0.934	2694B	xalancbmk	18.288	10.885	0.311	178B
zeusmp	4.567	3.956	1.230	405B					

Table 5.3: Benchmarks grouping

Insensitive to segmentation	*433.milc 434.zeusmp 436.cactusADM 462.libquantum 470.lbm*
LRU friendly	*435.gromacs 450.soplex 459.GemsFDTD 471.omnetpp 473.astar*
Sensitive to segmentation	*400.perlbench 401.bzip2 403.gcc 429.mcf 437.leslie3d 456.hmmer 481.wrf 482.sphinx3 483.xalancbmk*

modified version of CMP$im, a memory-system simulator [12]. The version we used was provided with the JILP Cache Replacement Championship [2]. It models an out-of-order 4-wide 8-stage pipeline with a 128-entry instruction window. This infrastructure enables collecting instructions-per-cycle figures as well as misses per kilo-instruction. The experiments model a 16-way set-associative last-level cache to remain consistent with other previous work [28, 32, 38, 39]. The microarchitectural parameters closely model Intel Core i7 (Nehalem) with the following parameters: L1 data cache: 32KB 8-way associative, L2 unified cache: 256KB 8-way L3: 2MB/core. Each benchmark is compiled for the x86_64 instruction set. The programs are compiled with the GCC 4.1.0 compilers for C, C++, and FORTRAN. We use SimPoint [37] to identify a single one billion instruction characteristic interval (i.e.*simpoint*) of each benchmark. Each benchmark is run

with the first `ref` input provided by `runspec`.

5.4.1 Single-Thread Workloads

For single-core experiments, the infrastructure simulates one billion instructions. We simulate a 2MB LLC for the single-thread workloads. In keeping with the methodology of recent cache papers [13, 14, 19, 25, 28, 29, 32, 38, 39], we choose a memory-intensive subset of the benchmarks. We use the following criterion: a benchmark is included in the subset if the number of misses in the LLC decreases by at least 1% when using the optimal [4] replacement and bypass policy instead of LRU. Table 3.3 shows memory intensive SPEC CPU 2006 benchmarks with the baseline LLC misses per 1000 instructions (MPKI), optimal MPKI, baseline instructions-per-cycle (IPC), and the number of instructions fast-forwarded (FFWD) to reach the interval given by SimPoint.

5.4.2 Multi-Core Workloads

Table 5.4 shows ten mixes of SPEC CPU 2006 simpoints chosen four at a time with a variety of memory behaviors characterized in the table by cache sensitivity curves. We use these mixes for quad-core simulations. Each benchmark runs simultaneously with the others, restarting after 250 million instructions, until all of the benchmarks have executed at least one billion instructions. For the multi-core workloads, we report the weighted speedup normalized to LRU. That is, for each thread i sharing the 8MB cache, we compute IPC_i. Then we find SingleIPC_i as the IPC of the same program running in isolation with an 8MB 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.

5.5 Results

In this section we present results and analysis of decoupled dynamic cache segmentation (DCS) by itself and with the optimizations described in Section 5.2.

Table 5.4: Multi-core workload mixes with cache sensitivity curves, LLC misses per 1000 instructions (MPKI) on the y-axis for LLC sizes 128KB through 32MB on the x-axis. s=sensitive, i=insensitive and f=LRU friendly

Work -load Name	Benchmarks	Type	Cache Sensitivity Curve
mix1	*mcf hmmer libquantum omnetpp*	ssif	43.5 25.5
mix2	*gobmk soplex libquantum lbm*	sfii	22.0 12.6
mix3	*zeusmp leslie3d libquantum xalancbmk*	isis	14.6 9.4
mix4	*gamess cactusADM soplex libquantum*	iifi	19.0 9.5
mix5	*bzip2 gamess mcf sphinx3*	siss	36.4 16.6
mix6	*gcc calculix libquantum sphinx3*	siis	9.7 6.6
mix7	*perlbench milc hmmer lbm*	sisi	10.4 9.6
mix8	*bzip2 gcc gobmk lbm*	sssi	6.7 5.4
mix9	*gamess mcf tonto xalancbmk*	isss	37.0 16.6
mix10	*milc namd sphinx3 xalancbmk*	ifss	12.4 4.5

5.5.1 Effect of Dynamic Segmentation

In this section we report how DCS performs in presence of enforced partitioning and automated bypassing. Figure 5.8 shows the speedup of DCS compared to LRU replacement. DCS uses NRU as the base line replacement policy. The x-axis represents a memory intensive subset of SPEC CPU 2006 benchmarks grouped according to their behavior with respect to static segmentation as illustrated in Table 5.3. DCS alone achieves a geometric mean 2.4% speedup and reduces misses by 4.3% on average. By inserting the non-referenced lines in the non-recent part of the NRU stack we enforce thrash resistance, improving performance by a geometric mean of 4.9% over LRU and reducing misses by 6.7%. Turning on the automated bypassing yields an average speedup of 5.2% and reduces misses by 7.8%. In general, we can see that DCS can improve performance up to 44% for the sensitive benchmarks. However, it hurts performance for LRU-friendly workloads when

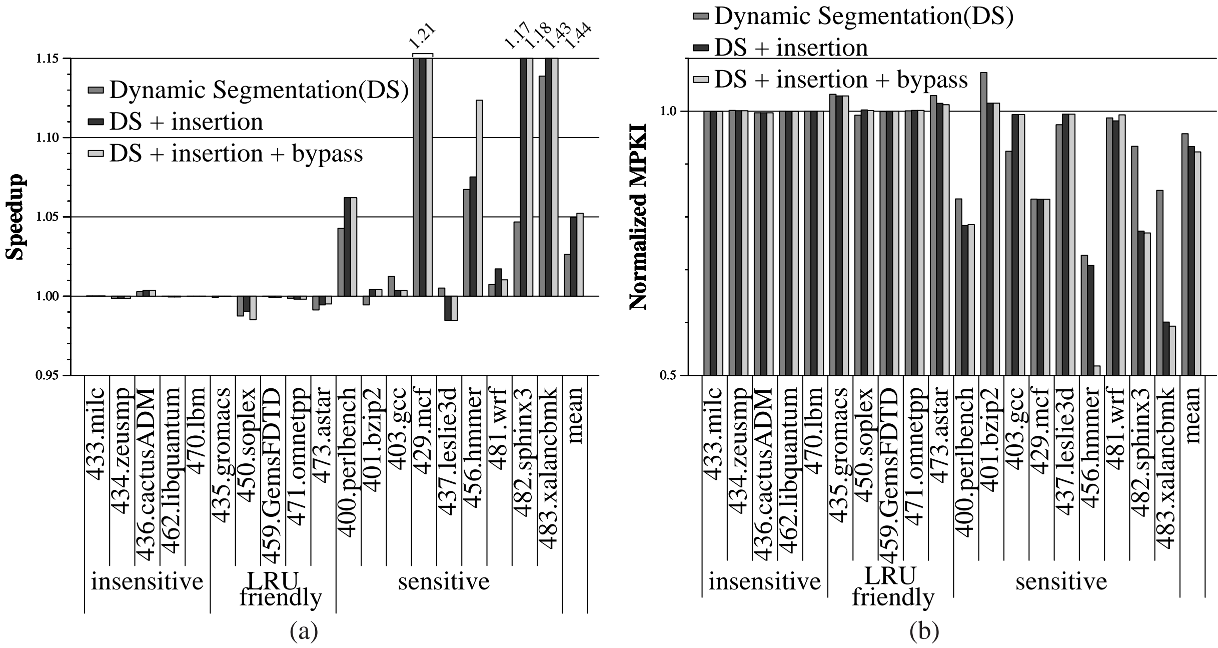


Figure 5.8: Effect of enforced thrash resistance and automated bypassing

the default policy used is NRU. The insensitive benchmarks show no or very minimal effect with DCS.

5.5.2 Effect of Segment Predictor

We randomly choose 16 sampled sets for each of the leader sets. The counter size is 11 bits for a 2MB LLC. The in-cache configuration uses the original cache sets to predict the best segmentation. The out-of-cache configuration uses 15-bit partial tags represented externally to the cache to reproduce the sampling set behavior. The in-cache configuration improves performance by 4.9% and reduces misses by 7% on average over LRU replacement. The out-of-cache configuration improves performance by 5.2% and reduces misses by 7.8% on average over LRU policy. The out-of-cache configuration has the advantage that it avoids cache sets that use the wrong segmentation in the leader sets. This is why LRU-friendly workloads perform worse with the in-cache predictor than the out-of-cache predictor. However, the out-of-cache predictor is limited by the accesses of the original cache. So if the winning non-referenced segment size is one, the predictor decides which segmentation is better given that the LLC access pattern is the dynamic segmentation policy. The

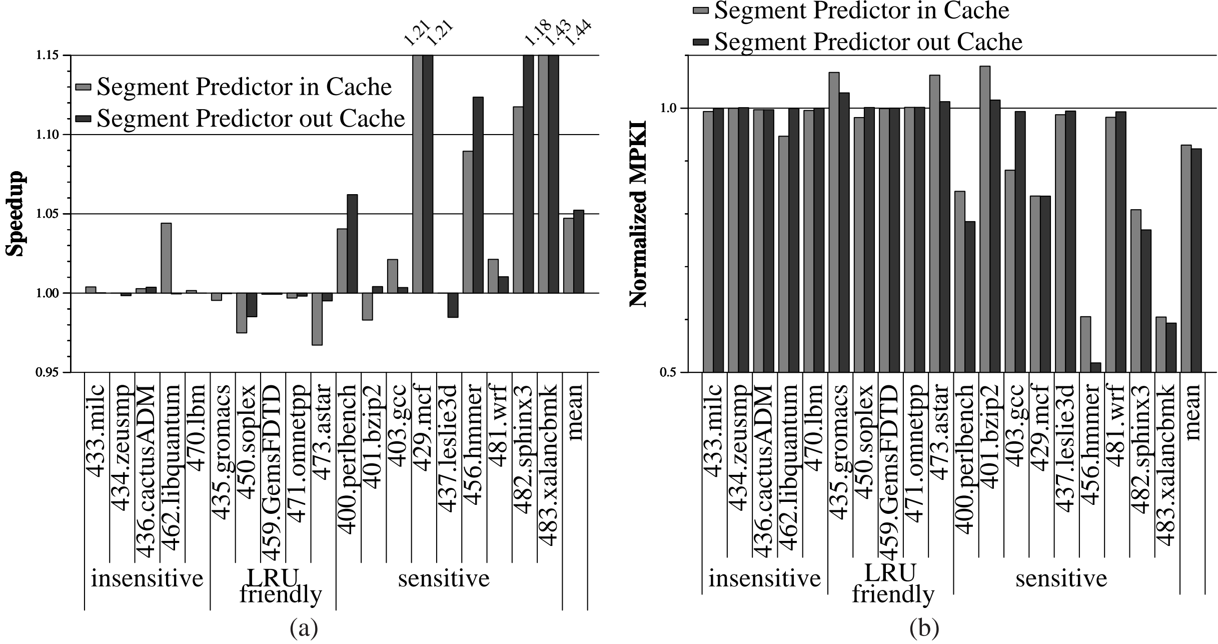


Figure 5.9: Effect of Segment Predictor

in-cache predictor decides which segmentation is best given that LLC access pattern is following the NRU policy. This is why benchmarks *437.leslie* and *481.wrf* perform poorly in the out-of-cache predictor configuration.

5.5.3 Comparison with Other Policies

Figure 5.10 compares DCS with other recent scan and thrash resistant policies. We compare DCS technique with DIP [38] that uses LRU as the baseline policy. We also compare DCS with RRIP [14] that uses two-bit NRU as the baseline policy. DCS uses a one-bit NRU policy. All policies use the same number of leader sets. Figure 5.10 shows that, compared to LRU replacement, DIP and RRIP achieve geometric mean 3.1% and 4.1% speedups. DCS outperforms both of these techniques, achieving a 5.2% geometric mean speedup.

5.5.4 DCS with Random Policy

In Figure 5.11 we show that DCS with random policy outperforms LRU replacement. Random replacement by itself slows down performance on average by 1%. However, DCS with random

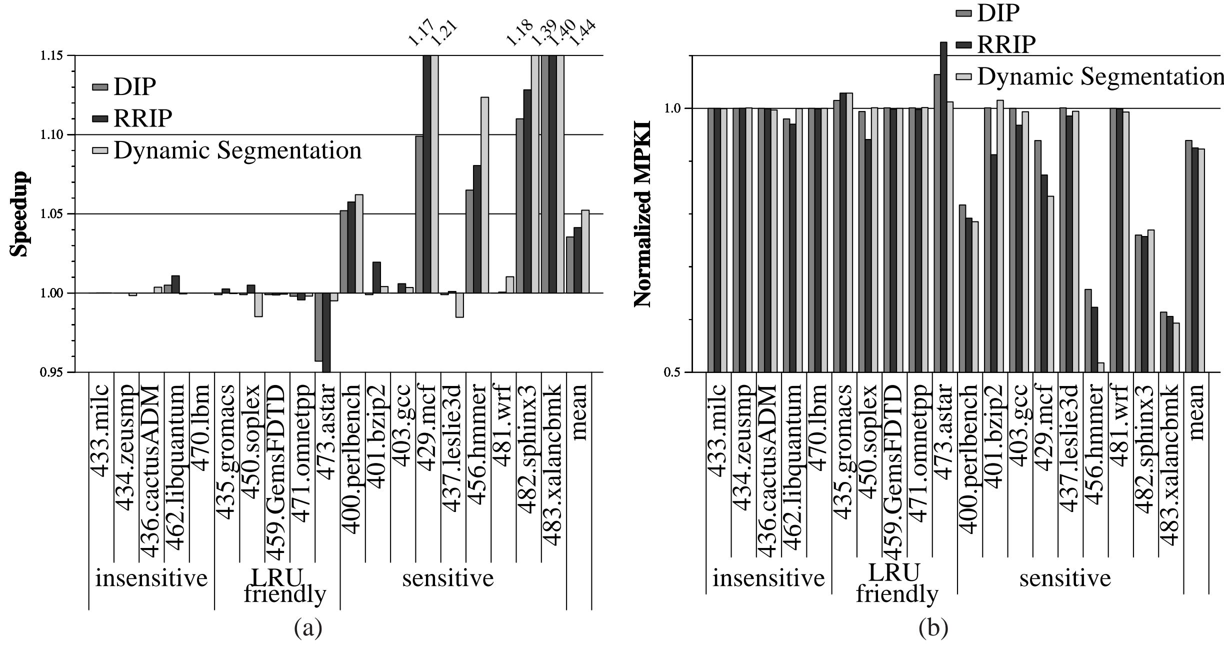


Figure 5.10: Effect of Segment Predictor

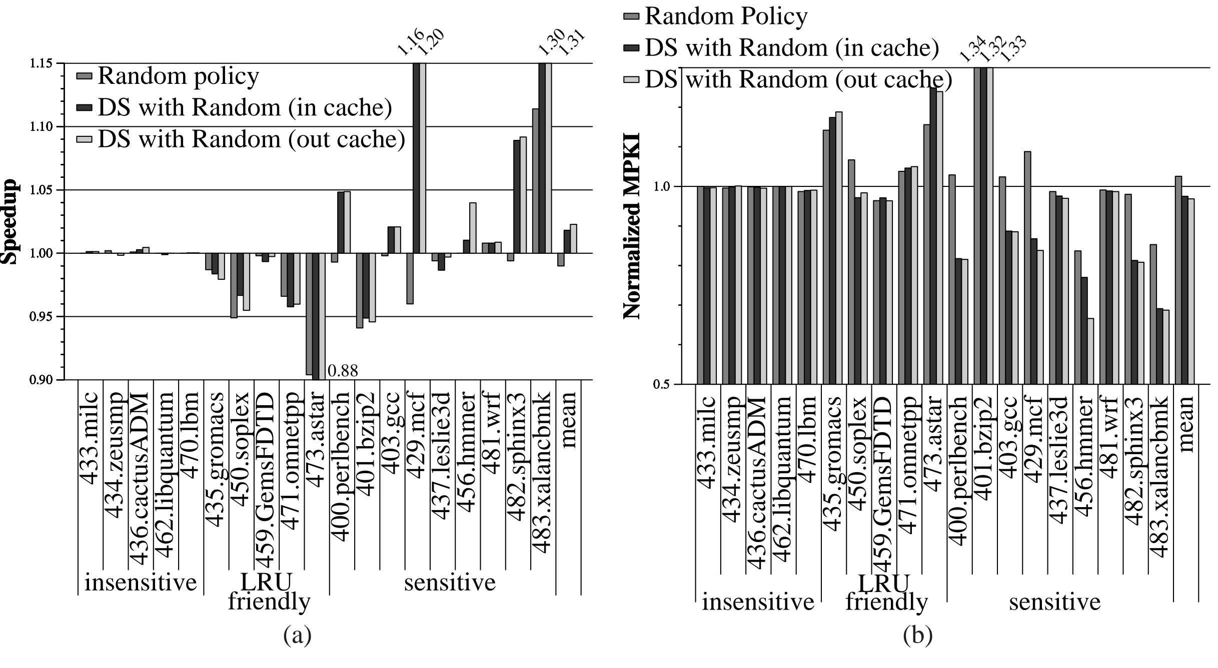


Figure 5.11: Dynamic Segmentation with Random Policy

Table 5.5: Space overhead in a 2MB 16 way LLC

	DCS NRU (in)	DCS NRU (out)	DCS Rand (out)
ref bit, per line	1bit	1bit	1bit
repl state, per line	1 bit	1 bit	0 bit
set type, per set	2 bits	2 bits	2 bits
psel counters	23 bits	23 bits	23 bits
cache overhead ((perline×ways +per set)× sets+psel counters)	8.5KB	8.5KB	4.5KB
sampler set entry (partial tag +ref bit+valid bit+repl state)	0 bit	18 bits	17 bits
number of sampler set	0	16	16
total sampler overhead (4 types ×sampler sets×sampler ways)	0	2.25KB	2.12KB
total	8.5KB	10.75KB	6.62KB
% increase in LLC Data Area	0.41%	0.52%	0.32%

Table 5.6: Comparing space overhead with other policies

	LRU	DIP	RRIP	DCS NRU (in)	DCS NRU (out)	DCS Rand (out)
per line	4b	4b	2b	2b	2b	1b
extra	0B	257.3B	513.3B	514.8B	2.25KB	2.12KB
total	16KB	16.25KB	8.5KB	8.5KB	10.75KB	6.62KB

replacement improves performance by 1.8% and 2.2% respectively for the in-cache and out-of-cache predictors. DCS coupled with random replacement improves performance up to 33% over LRU.

5.5.5 Space Overhead

We compare the space overhead of our dynamic cache segmentation (DCS) with LRU, DIP and RRIP in Table 5.6. LRU and DIP use four bits per cache block for the LRU stack in a 2MB 16 way LLC. RRIP uses two NRU bits per cache block. DCS also uses two bits per cache block.

However, it uses one of the bits for segmentation and the other bit for a one-bit NRU policy. For DIP and RRIP, the extra overhead is the bits required to differentiate the leader sets from the follower sets. Both of them need only one counter to set-duel between two policies. DCS with the in-cache predictor uses only 12 bits more space than RRIP. These 12 bits are two extra counters (1 bit and 11 bits) used to choose among five non-referenced segment sizes. DCS with the out-of-cache predictor uses 15-bit partial tags to obtain the best segmentation. The out-of-cache segment predictor has 15 bits of partial tag, one bit for segmentation and one bit for NRU. The out-of-cache predictor uses 2.12 KB. DCS with either in-cache and out-of-cache predictor uses far less space than the LRU policy.

5.5.6 Dynamic Segment Size

In Figure 5.12, we show the predicted best non-referenced segment size for each of the benchmark. Insensitive benchmarks use a non-referenced segment size of eight most of the time. LRU-friendly

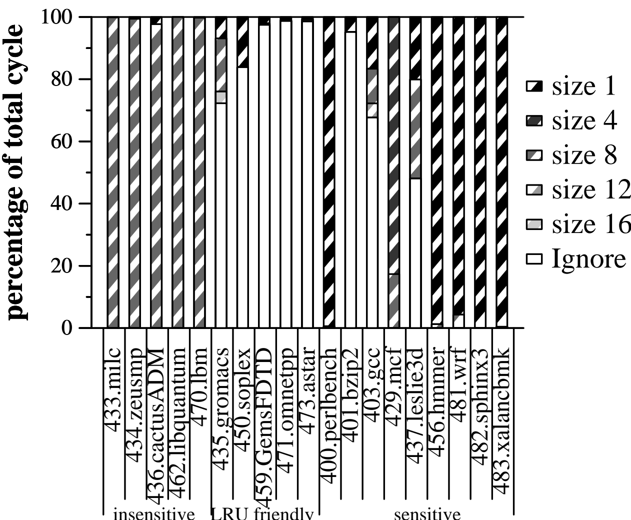


Figure 5.12: Runtime predicted best non-referenced segment size

benchmarks choose to ignore the segmentation. However, segmentation-sensitive benchmarks choose different segmentations in various phases. Benchmarks like *400.perlbench*, *482.sphinx3* and *483.xalancbmk* are thrashing workloads and always choose segment size one. *403.gcc*, *429.mcf* and *437.leslie3d* clearly go through different phases and choose different segmentations at runtime.

5.5.7 Cache Sensitivity

Figure 5.13 shows DCS performance with several LLC sizes: 1MB, 2MB, 4MB and 8MB. All of the cache configurations are 16-way associative. Our technique with the NRU policy outperforms LRU replacement on average by 5.2-8.7 for various cache sizes.

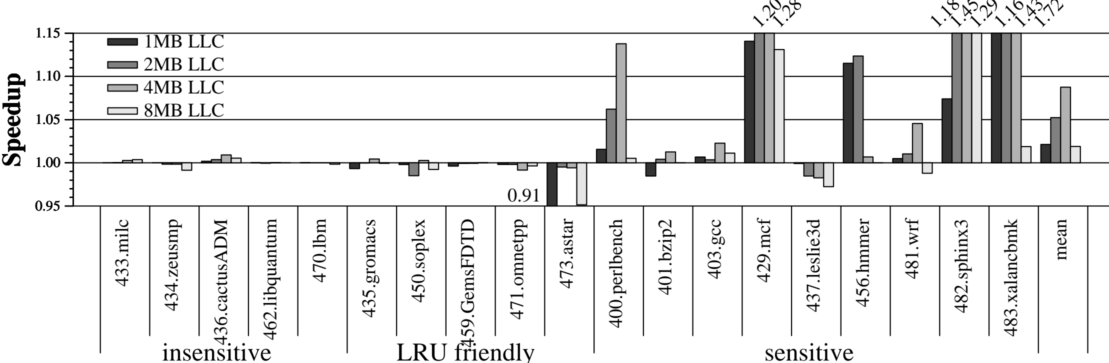


Figure 5.13: Cache size sensitivity of Dynamic Segmentation

5.5.8 DCS with Shared Cache Partitioning

In this section we show that our technique complements current way-partitioning techniques. Figure 5.14 compares cache segmentation with utility-based cache partitioning (UCP) [40] on multi-core workloads with a 8MB LLC. DCS is not aware of thread-specific characteristics. It adapts to the mixed access pattern from all threads and predicts the best segmentation size for that mix-

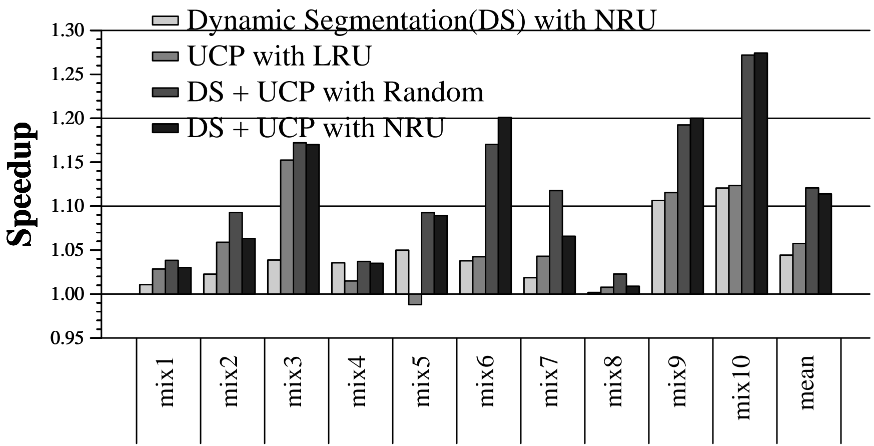


Figure 5.14: Complementing cache partitioning technique

ture. Figure 5.14 shows that it achieves 4.4% normalized weighted speedup compared to LRU replacement. We implement a combined version of DCS and UCP in which UCP predicts the best partitioning for each thread and DCS predicts the best segmentation for each thread. UCP achieves 5.4% weighted speedup over LRU. Complementing UCP with DCS achieves 11.4% speedup using an underlying NRU policy. It also achieves 12% speedup with an underlying random replacement. The space overhead for UCP is 68.75KB. However, our technique with random as the base replacement policy achieves 4.5% performance improvement over UCP with less than half of the space requirement. The space overhead of our technique is 45.25KB with NRU and 28.75KB with random replacement, which is respectively 0.55% and 0.35% of the 8MB LLC capacity. Our technique with NRU and random replacement performs similarly. Segmentation decreases the number of candidates in each set. Victims can only be selected from the specific ways belonging to the specific list of that thread. This essentially makes NRU and random perform very similarly. We have used 32 dedicated sets for UCP and 16 dedicated sets for DCS. We have also compared our technique with Thread Aware RRIP (TA-RRIP) which is not a cache partitioning technique [14]. TA-RRIP outperforms LRU on average by 10.2%. We outperform TA-RRIP even with an underlying random-policy.

5.6 Reducing Dead Blocks with Insertion Policy

Many cache management techniques rely on LRU policy [38, 40, 45]. In next chapter we introduce a dead time reduction cache management that reduces dead-on-arrival blocks in LRU policy just changing the insertion policy.

CHAPTER 6: DEAD TIME REDUCTION THROUGH INSERTION POLICY

In this section we introduce a cache management technique that reduces the dead time of dead-on-arrival blocks. We propose an adaptive insertion policy that learns the best insertion position using decision tree analysis. This way all incoming blocks are placed in an optimal insertion position where dead upon arrival blocks get evicted earlier.

6.1 Motivation

The motivation behind this work is the large number of dead-on-arrival blocks in the last level cache. The memory references are filtered through upper level caches and the access stream in the LLC does not have much temporal locality. This is why 79.1% percentage of blocks brought to the LLC are never used again.

This gives an opportunity to reduce dead time of these zero reuse blocks only by changing the insertion policy. We show that there is an optimal position in the LRU stack where inserting the blocks, dead upon arrival blocks will be evicted earlier, but live blocks will remain in the cache long enough to be re-referenced. We propose to use decision tree analysis to determine this optimal insertion position.

6.2 Description

6.2.1 Decision Tree Analysis

Our scheme considers five different insertion positions in the LRU stack. It divides the LRU stack into four equal segments. The default placement is MRU. DIP [38] considers LRU as an insertion position. We consider the middle position of the LRU stack and other two equidistant positions from the middle position. These two positions are named *near LRU position* and *near MRU position*. Figure 6.1 shows these five positions in the LRU stack. It also shows how the

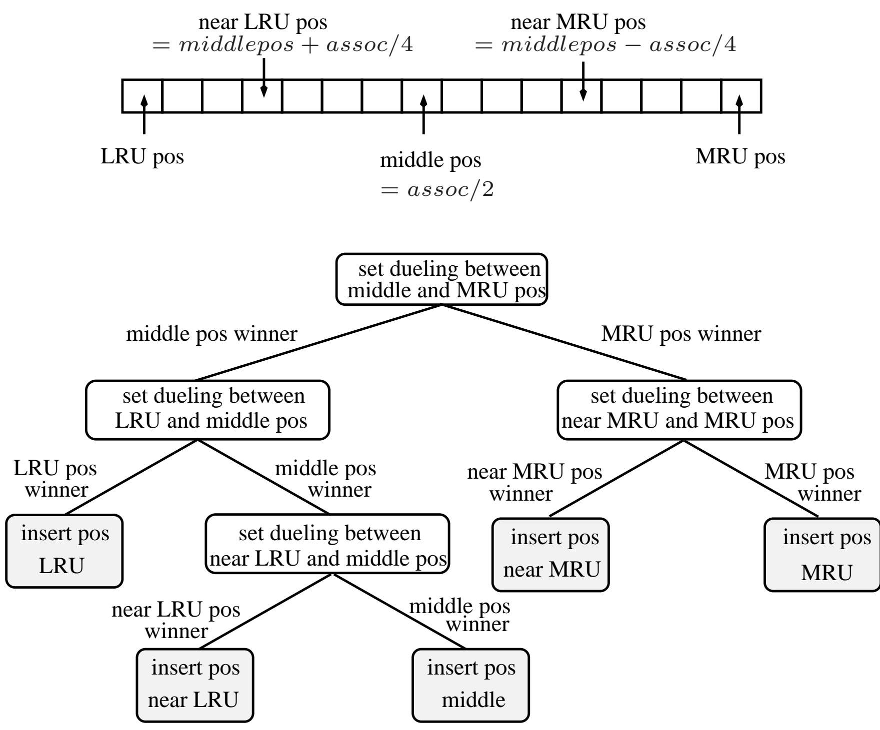


Figure 6.1: Decision Tree Analysis

appropriate insertion position is selected using the decision tree. The insertion position is chosen after a few rounds of competition as illustrated in Figure 6.1.

6.2.2 Adaptive policy in Leader Sets for Multi Set Dueling

Multi-set-dueling was proposed for multi-threaded workloads [13]. Each application has its own counter and it decides to insert in either LRU position or MRU position depending on that counter value. Multi-core multi-policy set-dueling was subsequently proposed [33]. In each core there are leader sets for each of the competing policies grouped into two. In the first round two policies in one group duel with each other. The winner policy of the first round are deployed in the partial follower sets (ϕ sets). The second level winner is then determined from the duel of these ϕ sets. Thus, the policy selection becomes a tournament where at each round half of the policies are eliminated. In the final round there are only two policies left and the winner policy is followed by all the other follower sets.

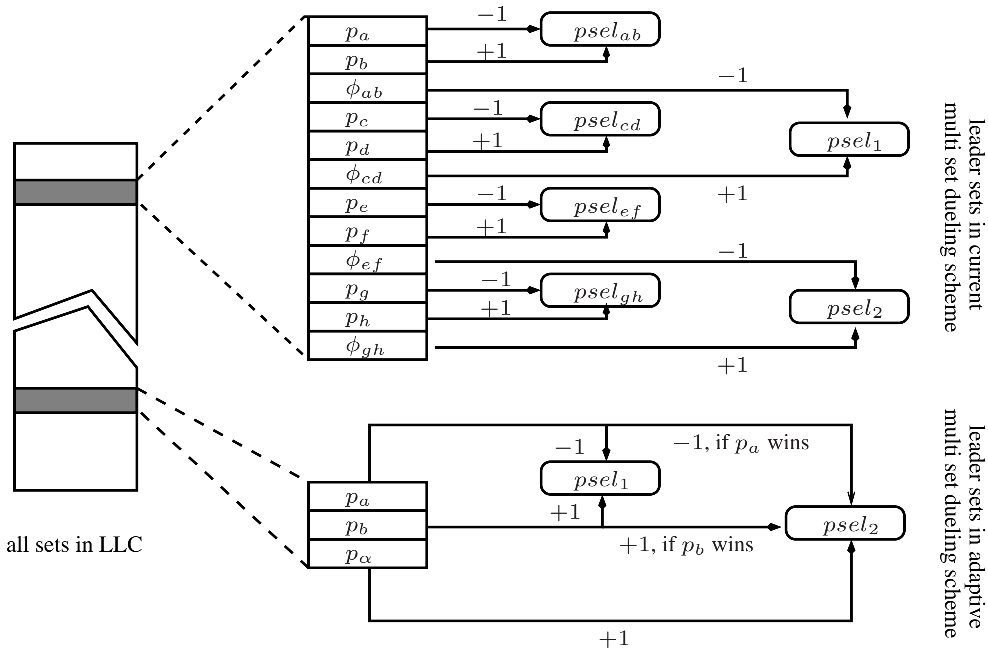


Figure 6.2: Reduction in Leader sets with adaptive policy

The problem with this approach is number of leader sets goes up with the number of policies being considered for multi set dueling. When many policies are dueling in a tournament manner, even if we can choose the best performing policy for the rest of the follower sets, all but one leader set continue using the wrong policy, potentially hurting performance significantly when the number of leader set increases. Another problem is the presence of partial follower sets. These sets are redundant as there are leader sets already present in the cache using that specific winner policy.

We have used the idea of multi set dueling in a single-core context. However, the problems of this scheme is solved by using leader set that can dynamically select specific insertion policy. We also remove the partial follower sets. Figure 6.2 shows the difference in two schemes. The first group of leader set is defined according to previous work [33]. First round is between policy p_a, p_b and p_c, p_d and p_e, p_f and p_h, p_g. The winner is deployed in partial follower sets ϕ_{ab}, ϕ_{cd}, ϕ_{ef} and ϕ_{gh}. These sets duel in pairs and the tournament goes to semi-final and final round (not shown in the figure).

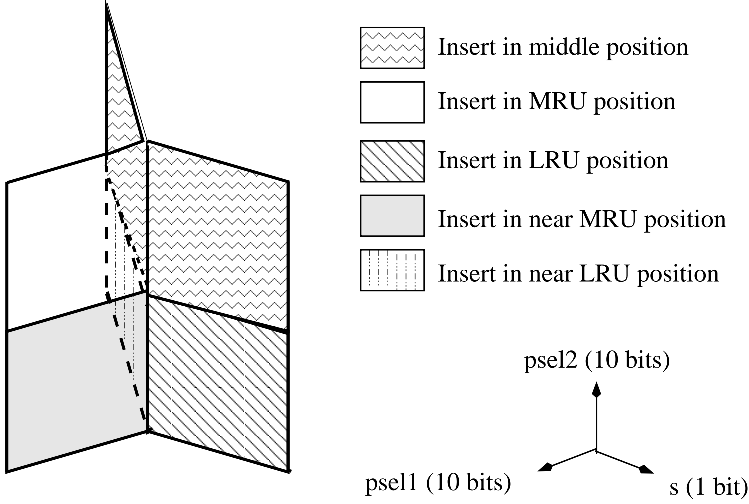


Figure 6.3: Selecting insertion policy

Table 6.1: Storage for 1MB 16 way cache

Parameter	Storage
set type	2 bits $*$ 1024 sets
two counters (psel)	20 bits
one counter (s)	1 bit
Total	2069 bits

We show our leader set with adaptive policy in the second group of the leader sets. Here we have only three kinds of leader sets. The first two leader sets implement policy p_a and p_b. The last set implements p_α. Depending on which set is winning, we can dynamically choose among the policies p_c, p_d, p_e, p_f, p_h and p_g. In the next section we describe how we use this idea in our insertion position selection.

6.2.3 Insertion Policy Selection

According to previous work [33] we should have five leader sets for five insertion positions and two partial follower sets for 1st round winner. Instead we use only three leader sets. The first round duel is between the MRU position and middle position. Counter $psel1$ determines the winner in this round. If MRU position is the winner, the last leader set inserts in the near MRU position.

Table 6.2: Configuration

Parameter	Configuration
Issue width	4
Reorder Buffer	128 entry
Branch Prediction	perfect
L1 I-Cache	32KB, 4 way LRU,
	64B blocks, 1 cycle hit latency
L1 D-Cache	32KB, 8 way LRU,
	64B blocks, 1 cycle hit latency
L2 Cache	256 MB, 8 way LRU,
	64B blocks, 10 cycle hit latency
L3 Cache	1 MB, 16 way,
	64B blocks, 30 cycle hit latency
Main Memory	200 cycle

The counter $psel2$ is responsible for the second level winner. However, if middle position was the winner in the first round, last leader set inserts in the LRU position. So the second level duel takes place between middle position and LRU position. If middle position is still the winner, the last leader set starts inserting in near LRU position. We use a one bit counter s to keep track of the policy used in this set so that follower sets know which policy to use. Figure 6.3 shows how follower sets decide which policy is winning.

6.2.4 Storage Requirement

We have four kind of sets in our scheme; leader set inserting at MRU position and middle position, adaptive leader set and follower set. This requires extra 2 bits per set. Then we need two counters ($psel1$ and $psel2$) and one extra bit for s to keep track of policies in adaptive leader set. Table 6.1 shows the space requirement for a $1MB$ 16-way last level cache.

6.3 Simulation Methodology

We have simulated our scheme with CMP$im [12]. It simulates a simple multiple issue out-of-order core. Table 6.2 shows the configuration of the simulated machine. We have used SPEC CPU

Table 6.3: Insertion Position Selected

Benchmark	Position	Benchmark	Position
400.perlbench	middle	450.soplex	middle
401.bzip2	middle	456.hmmer	nearLRU
429.mcf	nearLRU	464.h264ref	MRU
434.zeusmp	MRU	471.omnetpp	middle
435.gromacs	middle	473.astar	MRU
436.cactusADM	middle	482.sphinx3	LRU
445.gobmk	nearLRU	483.xalancbmk	LRU

2006 benchmarks. We have chosen 14 memory intensive benchmarks that achieve more than 1% instructions-per-cycle (IPC) speedup when the LLC is increased from 1MB to 2MB. We have fast forwarded the benchmarks for 40 billion instructions and then run the simulation for 100 million instructions.

6.4 Results

In this section we show the results of our scheme. Table 6.3 shows the insertion position selected by our decision tree analysis for each of the benchmark. Fig 6.4 shows the percentage of benchmarks

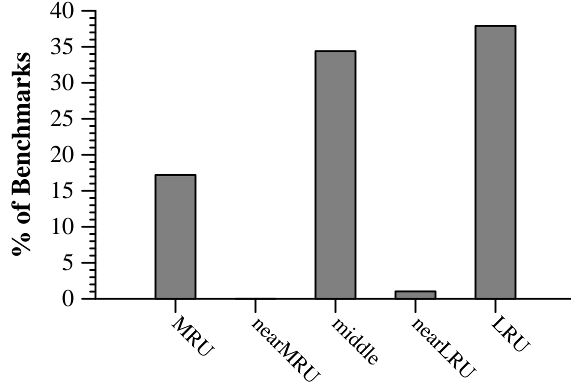


Figure 6.4: Percentage of benchmarks at each insertion position

choosing each insertion position. Among 29 SPEC CPU2006 benchmarks, DTA chooses MRU insertion position in 5 benchmarks, middle position in 10 benchmarks, near LRU position in 3

benchmarks and LRU position in 11 benchmarks.

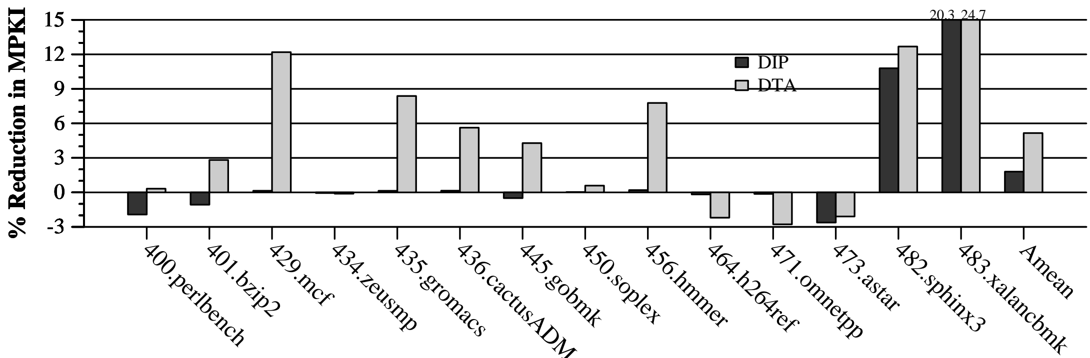


Figure 6.5: MPKI reduction compared to LRU

Fig 6.5 shows the percentage reduction in misses-per-1000-instructions (MPKI) of our scheme and DIP over LRU replacement policy. On average our policy reduces MPKI by 5.16% over LRU. On average DIP reduces MPKI by 1.8%. Harmonic mean IPC for DTA is 0.850399 which is 7.19% IPC improvement over LRU and 4.12% IPC improvement over DIP. Fig 6.6 shows the speedup of

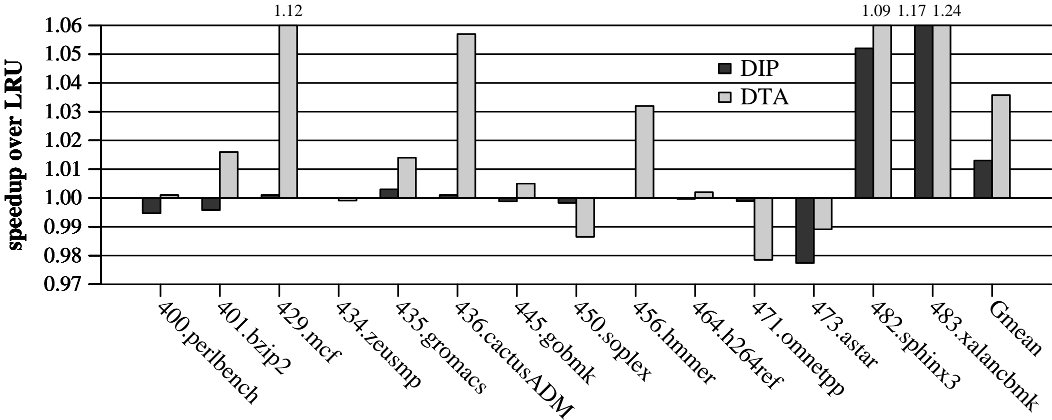


Figure 6.6: Speedup over LRU

the memory intensive benchmarks. The geometric mean speedup of our scheme over LRU is 3.57% where DIP achieves only 1.3% speedup over LRU. DTA performs poorly in 450.soplex, 471.omentpp and 473.astar. The reason is the workloads do not show uniform access across the sets. 16 dedicated sets are not enough for these benchmarks. For example if we increase the

number of dedicated sets from 16 to 32, DTA chooses the MRU position for 450.soplex and its performance becomes similar to baseline.

It should be noted that our scheme can detect phase changes in the workloads. It has two static leader sets that always use the same specific insertion positions. Only the adaptive leader set chooses the insertion position dynamically. The static sets detect the phase change and adaptive set chooses the insertion position accordingly. Our benchmarks ran only 100 million instructions and do not experience any phase change.

Fig 6.7 shows speedup of our scheme over LRU for all SPEC CPU 2006 benchmarks. It achieves 1.7% IPC improvement over the baseline. We can see that DTA dose not significantly slow down any of the non memory intensive benchmarks.

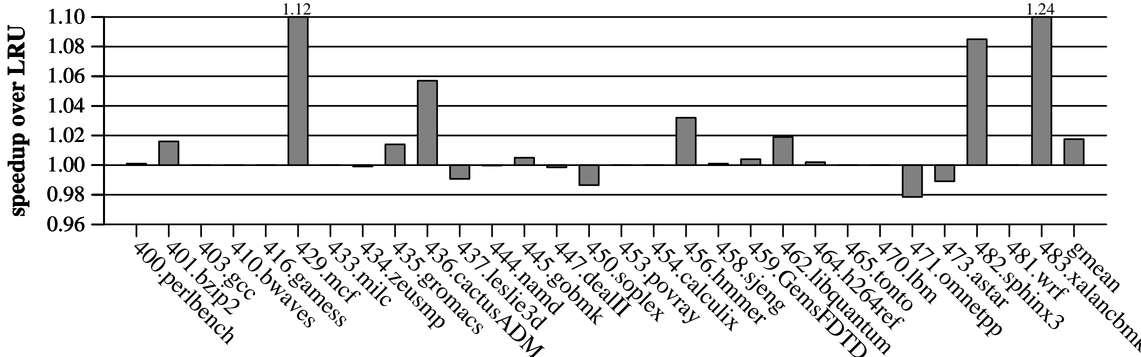


Figure 6.7: Speedup over LRU replacement policy

6.4.1 Comparison With a Dead Block Predictor Replacement

In this section we compare our result with Counting Based Dead Block Replacement (CDBR) [25]. A dead block predictor can accurately identify zero reuse lines and replace them instead of the LRU block. However, such a predictor requires a significant hardware budget. The counting based predictor needs to keep track of program counter (PC), access count, past access count and the confidence of the prediction for each cache line [25]. It also uses a 256×256 entry predictor table where each entry stores the number of access and the confidence. For a 1MB cache it uses 74KB extra storage where our scheme needs only $2,069$ bits. That is, our technique requires far less than

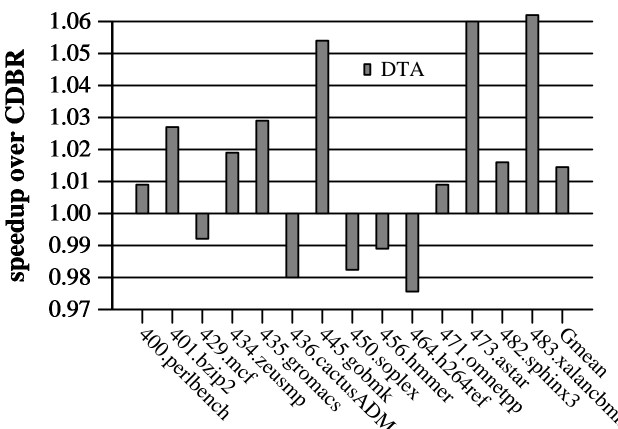


Figure 6.8: Speedup of over Counting Based Dead Replacement

1% of the storage of a dead block predictor.

Figure 6.8 shows the speedup of DTA over counting based dead block replacement. On average we achieve 1.45% speedup over CDBR. The reason our scheme performs better on average is, although the dead block predictor learns zero reuse lines, these lines do not come back to the cache frequently. So even if the dead block predictor cannot identify those lines, by contrast, our scheme inserts them in near LRU position and gets rid of them quickly.

CHAPTER 7: CONCLUSION

7.1 Summary

This dissertation proposes simple intelligent mechanisms to reduce memory system waste. In this section we summarize our contribution.

We propose a dead block prediction technique based on *sampling*. The predictor learns from memory references accessing only a few sets. This work shows that we can decouple dead block prediction from replacement policy and use cache optimizations with inexpensive replacement policies [23].

We propose the Virtual Victim Cache technique that stores victim blocks within the dead blocks in the cache. Our technique reduces dead time as well as take care of the conflict misses in the hot sets [22].

We have introduced a dynamic segmentation technique that reduces the dead time of non-referenced blocks in the last-level cache. This segmentation prefers referenced blocks over non-referenced blocks by dynamically adjusting the cache segmentation in runtime. We show that this segmentation is orthogonal to shared cache partitioning techniques [24].

We introduce an insertion policy that dynamically chooses the best the insertion position using decision tree analysis. This way dead-on-arrival blocks get evicted earlier and provides space for other useful blocks [21].

7.2 Implication

In this dissertation we explore the inefficiencies in the memory system and propose simple cache management techniques that reduce memory system waste and improve performance. As we approach the *Dark Silicon Era* [8] where chip multiprocessors will be packed with more and more transistors, but we would not be able to turn on those transistors because of the limited power

budget, this work becomes more important. When we would not be able to gain performance just by adding more cores in the chip, complex micro-architectural designs will be the only way to improve performance. This will enable us to use more transistors on-chip to implement complex techniques to improve performance.

BIBLIOGRAPHY

[1] Jaume Abella, Antonio González, Xavier Vera, and Michael F. P. O'Boyle. IATAC: a smart predictor to turn-off l2 cache lines. *ACM Trans. Archit. Code Optim.*, 2(1):55–77, 2005.

[2] Alaa R. Alameldeen, Aamer Jaleel, Moinuddin Qureshi, and Joel Emer. 1st JILP workshop on computer architecture competitions (JWAC-1) cache replacement championship. http://www.jilp.org/jwac-1/.

[3] Sorav Bansal and Dharmendra S. Modha. Car: Clock with adaptive replacement. In *in Proceedings of the USENIX Conference on File and Storage Technologies (FAST*, pages 187–200, 2004.

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

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

[6] Doug Burger and Todd M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[7] Mainak Chaudhuri. Pseudo-lifo: the foundation of a new family of replacement policies for last-level caches. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 401–412, New York, NY, USA, 2009.

[8] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, June 2011.

[9] Matthew Farrens, Gary Tyson, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *In Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, 1995.

[10] Antonio González, Carlos Aliagas, Mateo Valero, and Campus Nord. A data cache with multiple caching strategies tuned to different types of locality. pages 338–347, 1995.

[11] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. *SIGARCH Comput. Archit. News*, 30(2):209–220, 2002.

[12] Aamer Jaleel, Robert S. Cohn, Chi-Keung Luk, and Bruce 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.

[13] Aamer Jaleel, William Hasenplaugh, Moinuddin K. Qureshi, Julien Sebot, Simon Stelly Jr., and Joel 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] Aamer Jaleel, Kevin Theobald, Simon Steely Jr., and Joel 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.

[15] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen mei W. Hwu. Run-time cache bypassing, 2000.

[16] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[17] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Comput. Archit. News*, 18(3a):364–373, 1990.

[18] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27:38–46, March 1994.

[19] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *ICCD*, pages 245–250, 2007.

[20] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *In Proceedings of the 25th International Conference on Computer Design (ICCD-2007)*, pages 245–250, 2007.

[21] Samira Khan and Daniel A. Jiménez. Insertion policy selection using decision tree analysis. In *Proceedings of the International Conference of Computer Design*, Amsterdam, Netherlands, 2010.

[22] Samira Khan, Daniel A. Jiménez, Doug Burger, and Babak Falsafi. Using dead blocks as a virtual victim cache. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Vienna, Austria, 2010.

[23] Samira Khan, Yingying Tian, and Daniel A. Jiménez. Sampling dead block prediction for last-level caches. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, USA, 2010. IEEE Computer Society.

[24] Samira Khan, Zhe Wang, and Daniel A. Jiménez. Decoupled dynamic segmentation. In *Proceedings of the International Symposium on High Performance Computer Architecture,*, New Orleans, Louisiana, 2012.

[25] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4):433–447, 2008.

[26] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ACM SIGPLAN NOTICES*, pages 211–222. ACM, 2002.

[27] Joydip Kundu and Janice E. Cuny. A scalable, visual interface for debugging with event-based behavioral abstraction. In *Frontiers of Massively Parallel Computing '95*, pages 472–479, 1995.

[28] An-Chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *International Symposium on Computer Architecture*, pages 139 – 148, 2000.

[29] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. *SIGARCH Comput. Archit. News*, 29(2):144–154, 2001.

[30] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 23(2):48–59, 1995.

[31] Donghee Lee, Jongmoo Choi, Jong hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. In *In Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 134–143, 2001.

[32] Haiming Liu, Michael Ferdman, Jaehyuk Huh, and Doug Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 222–233, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[33] Gabriel H. Loh. Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 201–212, New York, NY, USA, 2009. ACM.

[34] Nimrod Megiddo and Dharmendra Modha. Arc: A self-tuning, low overhead replacement cache. In *In Proceedings of the Conference on File and Storage Technologies (FAST)*, 2003.

[35] David M. Nicol, Albert G. Greenberg, and Boris D. Lubachevsky. Massively parallel algorithms for trace-driven cache simulations. In *IEEE Transactions on Parallel and Distributed Systems*, volume vol. 5, pages 849–859, August 1994.

[36] Elizabeth J. O'neil, Patrick E. O'Neil, Gerhard Weikum, and Eth Zurich. The lru-k page replacement algorithm for database disk buffering. pages 297–306, 1993.

[37] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003.

[38] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel 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.

[39] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 167–178, Washington, DC, USA, 2006. IEEE Computer Society.

[40] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE*, pages 423–432. IEEE Computer Society, 2006.

[41] Moinuddin K. Qureshi, David Thompson, and Yale N. Patt. The v-way cache: demand-based associativity via global replacement. In *In Proceedings of the 32nd International Symposium on Computer Architecture*, pages 544–555, 2005.

[42] Steven K. Reinhardt and Erik G. Hallnor. A fully associative software-managed cache design. *Computer Architecture, International Symposium on*, 0:107, 2000.

[43] Jude A. Rivers, Edward S. Tam, Gary S. Tyson, Edward S. Davidson, and Matt Farrens. Utilizing reuse information in data cache management. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 449–456, New York, NY, USA, 1998. ACM.

[44] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA*, pages 187–198. IEEE, 2010.

[45] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 57–68, New York, NY, USA, 2011. ACM.

[46] Jennifer B. Sartor, Subramaniam Venkiteswaran, Kathryn S. McKinley, and Zhenlin Wang. Cooperative caching with keep-me and evict-me. *Annual Workshop on Interaction between Compilers and Computer Architecture*, 0:46–57, 2005.

[47] André Seznec. A case for two-way skewed-associative caches. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 169–178, New York, NY, USA, 1993.

[48] Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Memory coherence activity prediction in commercial workloads. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*, pages 37–45, New York, NY, USA, 2004. ACM.

[49] Shekhar Srikantaiah, Mahmut T. Kandemir, and Mary Jane Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *ASPLOS*, pages 135–144, 2008.

[50] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA, pages 117–, Washington, DC, USA, 2002.

[51] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, and Norman P. Jouppi. Cacti 5.1. Technical report, HP Tech Report HPL-2008-20, 2008.

[52] Gary S. Tyson, Matthew K. Farrens, John Matthews, and Andrew R. Pleszkun. A modified approach to data cache management. In *MICRO*, pages 93–103. ACM/IEEE, 1995.

[53] Zhenlin Wang, Kathryn S. McKinley, Arnold L. Rosenberg, and Charles C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, page 199, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[54] Wayne A. Wong and Jean-Loup Baer. Modified lru policies for improving second-level cache behavior. *High-Performance Computer Architecture, International Symposium on*, 0:49, 2000.

[55] Yuejian Xie and Gabriel H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *International Symposium on Computer Architecture (ISCA)*, June 2009.

VITA

Samira Khan earned her PhD degree in Computer Science from the University of Texas at San Antonio. Her primary research interest lies in the general area of computer architecture. Originally she is from Bangladesh and has earned her BS in Computer Science and Engineering from the Bangladesh University of Engineering and Technology (BUET).