REDUCING WASTE IN MEMORY HIERARCHIES

A Dissertation

by

YINGYING TIAN

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

DOCTOR OF PHILOSOPHY

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

May 2015

Major Subject: Computer Science

ABSTRACT

Memory hierarchies play an important role in microarchitectural design to bridge the performance gap between modern microprocessors and main memory. However, memory hierarchies are inefficient due to storing waste. This dissertation quantifies two types of waste, dead blocks and data redundancy. This dissertation studies waste in diverse memory hierarchies and proposes techniques to reduce waste to improve performance with limited overhead.

This dissertation observes that waste of dead blocks in an inclusive last level cache consists of two kinds of blocks: blocks that are highly accessed in core caches and blocks that have low temporal locality in both core caches and the last-level cache. Blindly replacing all dead blocks in an inclusive last level cache may degrade performance. This dissertation proposes temporal-based multilevel correlating cache replacement to improve performance of inclusive cache hierarchies.

This dissertation observes that waste exists in private caches of graphics processing units (GPUs) as *zero-reuse blocks*. This dissertation defines zero-reuse blocks as blocks that are dead after being inserted into caches. This dissertation proposes adaptive GPU cache bypassing techinque to improve performance as well as reducing power consumption by dynamically bypassing zero-reuse blocks.

This dissertation exploits waste of data redundancy at the block-level granularity and finds that conventional cache design wastes capacity because it stores duplicate data. This dissertation quantifies the percentage of data duplication and analyze causes. This dissertation proposes a practical cache deduplication technique to increase the effectiveness of the cache with limited area and power consumption.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Memory is essential to a computer system to store code and data. Modern computer systems use a hierarchical memory design to bridge the performance gap between microprocessors and main memory with reasonable cost. Memory hierarchies work by exploiting *locality of reference* [11, 28], *i.e.* the observation that memory references tend to be localized in terms of time and space, referred to as *temporal locality* and *spatial locality*, respectively.

Caches [106] store instructions and data that exhibit locality with low access latencies. Caches are usually small and fast compared to the main memory. References to memory locations that are stored in caches can be satisfied in just a few clock cycles, while a miss in the last-level cache will go all the way down to significantly slower DRAM main memory, incurring hundreds of cycles of delay. The cache plays an important role in modern processors as a performance-critical structure to reduce the average memory access latency and provide high bandwidth. Compared to main memory, cache technology typically costs more per-bit, but an efficient cache can be large enough to hold only the working set of an application, and thus have most of the accesses hit in the cache, leading to far faster accesses and often less energy consumption than main memory. However, in practices caches are often inefficient because they store useless or redundant data, leading to a significant waste of storage.

1.1 The Problem: Cache Waste

This dissertation quantifies two types of waste:*dead blocks* [64] and *data redundancy* [73].

1.1.1 Waste of Dead Blocks

Caches organize data and instructions into fixed-sized blocks of *e.g.* 64 bytes. A cache block is *dead* from the last reference to that block until the block is evicted from the cache [64]. Dead blocks lead to cache inefficiency [14] and should be replaced by useful blocks as early as possible to improve cache efficiency. Previous work [64, 58, 56] introduced several dead block prediction techniques to reduce dead blocks in last-level caches (LLCs) for chip-multiprocessors (CMPs). However, these techniques cannot be directly applied to other cache types due to different cache characteristics.

1.1.1.1 Dead Blocks in Inclusive Caches

Inclusive caches have been widely used in chip-multiprocessors to simplify cache coherence. They suffer from poor performance compared to non-inclusive or exclusive caches because of the limited capacity of the inclusive cache hierarchy, and ignorance of temporal locality in the last-level cache. Blocks that are highly referenced (referred to as *hot blocks*) are always found in higher level caches (a.k.a. core caches) and are rarely referenced in the LLC. Thus, they tend to become dead blocks in the LLC despite the fact that they have high locality. Due to the inclusion property, blocks replaced from the LLC must be invalidated from core caches. Evicting these dead blocks from the entire cache hierarchy harms performance by introducing costly off-chip misses for hot blocks that makes the inclusive cache perform even more poorly.

1.1.1.2 Dead Blocks in GPU caches

Modern graphics processing units (GPUs) include hardware-controlled caches to reduce bandwidth requirements and energy usage [7, 41]. Current GPU cache hierarchies are inefficient for general purpose GPU computing (GPGPU). GPGPU workloads tend to include data structures that would not fit in any reasonably sized caches, leading to low cache hit rates. This problem is exacerbated by the design of GPUs, which share small caches between many threads. Caching these data structures wastes cache capacity and power while evicting useful data that may otherwise fit into the cache. These blocks are *dead-on-arrival* [56] and should not be inserted into GPU caches. Previous CPU LLC-based dead block prediction techniques cannot be directly applied on GPU caches due to their sophisticated mechanisms as well as significant power and storage overhead.

1.1.2 Waste of Data Redundancy

Data redundancy is another source of cache waste. In a conventional cache, each block is associated with a requested memory block address and a copy of the data. Cache blocks with different addresses can contain copies of identical data. These *duplicated* blocks waste cache capacity and power because of the storage of redundancy. Previous work exploited specific data redundancy such as zero with compression techniques [31, 2]. This dissertation shows that many blocks in the working set of typical benchmark programs have the same values, far beyond the zero-content blocks one would expect in any program. As shown in Figure 1.1, eliminating zero-content (null) blocks can save 13% of the cache capacity while eliminating all possible duplication leads to 47.5% of cache blocks removed or invalidated in a 2MB LLC.

1.2 The Solutions

In an inclusive LLC, there are two kinds of dead blocks: blocks that are highly accessed in core caches and blocks that have low temporal locality in both core caches and the LLC. Replacing hot core-cache blocks will hurt performance. The optimized replacement candidates should only be blocks that have low temporal locality in the whole cache hierarchy, whose replacement and back-invalidation will not cause extra

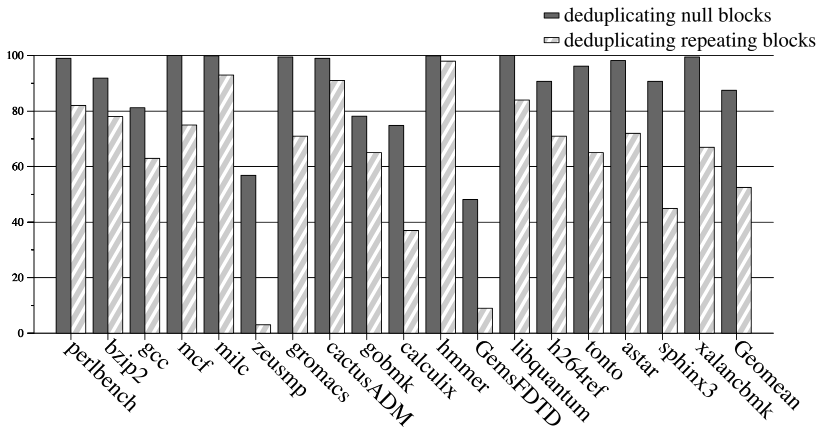

Figure 1.1: Percentage of distinct blocks for null block deduplication and all repeating block deduplication

cache misses; instead, replacing them with other useful blocks as early as possible will increase the cache efficiency and performance. This dissertation proposes Temporal-based Multi-level Correlating (TMC) cache replacement to choose LLC blocks that have low temporal locality in the whole inclusive cache hierarchy as LLC replacement candidates with high accuracy and minimal overhead.

To reduce waste in GPU caches, this dissertation develops a simple dynamic bypass predictor designed for small GPU caches with only hundreds of of bytes of storage overhead. The bypass predictor dynamically predicts if cache blocks are likely to be dead after their first references and bypasses these dead blocks to avoid polluting caches. That is, some blocks are not placed in the GPU cache, but rather bypass the cache and go directly to the consuming functional unit. Instead of increasing power overhead, the proposed GPU cache bypass technique reduces the power consumption of the baseline. Beside reducing the energy cost, the proposed bypass predictor improves GPU performance.

Data deduplication is a specific compression technique that has been widely used in disk-based storage systems [27, 111]. With data deduplication, only a single instance of identical data is physically stored. The redundant data is stored as references to the corresponding data in a deduplicated data storage to improve storage utilization. Although commonly used in disk storage and proposed for main memory compression, data deduplication is a challenge in on-chip caches with limited overhead due to several design concerns. This dissertation proposes a practical cache deduplication technique to exploit block-level data redundancy dynamically to increase the effectiveness of the cache with limited area and power consumption.

1.3 Thesis Statement

The performance and efficiency of modern processors can be improved by reducing waste in memory hierarchies.

1.4 Contributions

This dissertation makes the following original contributions:

- This dissertation introduces Temporal-based Multi-level Correlating (TMC) cache management for inclusive caches to reduce waste of dead blocks. TMC chooses blocks that will not be re-referenced in all cache levels as LLC replacement candidates. TMC samples LLC cache access patterns and correlates them with temporal locality knowledge passively acquired from higher level caches to choose temporal-aware LLC replacement candidates, providing performance improvement while consuming minimal overhead.

- This dissertation proposes a simple and effective GPU cache bypass predictor prevents streaming one-time-use values from being needlessly inserted into the cache. The predictor has high accuracy and minimal area overhead. It demonstrates performance gains and energy savings when using the proposed bypass techinque for a GPU L1 data cache. This dissertation studies limitations of current GPU cache design and the effects of a bypass predictor as they relate to using scratchpad memories.

- This dissertation quantifies the waste causes by data duplication and finds that widespread duplication exists in caches. It proposes a unified cache deduplication technique to increase effective cache capacity with limited area and power consumption. The deduplicated last-level cache improves performance without increasing physical area consumption.

2. BACKGROUND AND RELATED WORK

This dissertation proposes techniques to reduce waste of dead blocks and data redundancy in diverse types of memory hierarchies to improve efficiency and performance. To provide context for our research, we first give background and brief description of related work.

2.1 Dead Block Prediction

Cache blocks are *dead* from the last reference until they are evicted [64]. Storing dead block in caches waste capacity without improving performance. Replacing dead blocks as soon as possible with live blocks improves cache efficiency. Dead Block Prediction [64] is a technique that predicts whether cache blocks are likely to be dead after certain references and drives optimization techniques to improve performance.

Previous work introduces several dead block prediction techniques [64, 74, 55, 44, 1]. Lai *et al.* proposed a trace-based dead block predictor to drive prefetching. This predictor collects sequences (*i.e.* traces) of memory instructions that access the same block. The intuition is that if a trace leads to the last access for one block, then the same trace will lead to the last access for other blocks. Trace-based dead block predictor is also used to drive a cache coherence protocol optimization [25, 101] and dynamic self-invalidation [65]. Khan *et al.* proposed a skewed trace-based dead block predictor and utilized dead blocks as a "virtual victim cache" to store LRU victims from hot sets for future reuse [55].

Counter-based dead block predictor uses both memory addresses and memory instructions to record counters of cache access events and make dead block predictions [58]. The Live-time Predictor (LvP) tracks the number of accesses to each cache block. The Access Interval Predictor (AIP) tracks the access interval of each cache block. The counter-based dead block predictor uses dead blocks as replacement candidates, to replace dead blocks as early as possible for useful blocks.

A time-based predictor [44] was proposed to record the number of cycles a block is alive and predict a block to be dead if it is not accessed for twice of the number of cycles. This predictor is used for L1 cache prefetching and filtering a victim cache.

The cache burst predictor [74] proposes to make dead block prediction and update the prediction table on cache bursts rather than on all cache accesses. A cache burst consists of all the contiguous accesses to a block in the MRU position. The work yields little advantage for lower level caches since most bursts are filtered out by the core cache.

2.2 Improving Inclusive Cache Hierarchies

Inclusive caches have been widely used in chip multiprocessors (CMPs) to simplify cache coherence. However, they have poor performance compared with non-inclusive caches and exclusive caches. Previous work introduced several techniques to improve the performance of inclusive cache hierarchies.

Global Replacement Policy [113] was designed to use one unified replacement policy to control the replacement of cache blocks in all caches of an inclusive cache hierarchy. This proposal was only evaluated with single-threaded workloads and the results showed the global replacement policy sometimes performed worse than the corresponding local replacement policy. Grade *et al.* [35] analyzed the performance of Global Replacement Policy by deconstructing the policy with reuse-distance analysis and evaluated it in a multi-core inclusive cache hierarchy to show that the performance with global replacement policy was actually limited. Zahran *et al.* [114] proposed to make global cache placement decision based on access patterns of dierent blocks. This technique is not designed for inclusive caches because it violates the inclusion property by placing some blocks only into higher level caches but not the LLC. It can be treated as a non-inclusive cache managed by a global placement policy to achieve the capacity of an exclusive cache.

Temporal Locality Aware (TLA) [47] inclusive cache management policy suite was designed to improve the performance of inclusive caches by reducing the freqency of invalidation of inclusion victims that have high temporal locality in core caches. Successful TLA policies can identify cache blocks that have high temporal locality in core caches and avoid evicting these blocks from the LLC. It consists of three policies: Temporal Locality Hints (TLH), Early Core Invalidation (ECI), and Query Based Selection (QBS). TLA policies can only identify a limited number of highly reference blocks in core caches. Moreover, even the replace candidates in the LLC do not have high temporal locality in core caches, they may still be live blocks in the LLC whose invalidation will also hurt the cache performance by incurring hundreds of cycles of memory access penalty.

Gaur *et al.* [36] proposed a bypass and insertion algorithms for exclusive last-level caches in the LLC to improve the cache performance, but it was only designed for exclusive caches.

2.3 Improving GPU Private Caches

Graphics Processing Units (GPUs) provide tremendous throughput and high performance computing. Recently, GPUs have been used for general purpose computing. To support this effort, programming models such as CUDA [84] and OpenCL [39] have been developed for easier programming; hardware support such as memory hierarchies has been implemented in GPU cores. Recent work introduced several techniques to improve GPU memory hierarchy.

Compiler-controlled scratchpad memories [61, 54, 9] were proposed to improve the efficiency of scratchpad memories. Knight *et al.* proposed an optimizing compiler for architectures with software-managed memory hierarchies [61] to explicitly manage scratchpad memories. Kandemir *et al.* proposed a compiler-controlled dynamic on-chip scratchpad memory management technique for real-time embedded systems.

Jia *et al.* proposed a Memory Request Prioritization Buffer (MRPB) to improve GPU performance [50]. MRPB also employs cache bypassing to mitigate intra-warp contention. Instead of distinguishing reused blocks from significant amount of zero-reuse blocks, MRPB blindly and aggressively bypasses memory requests when there are resource limits, which can cause performance degradation, as stated in [50]. To evaluate MRPB in terms of programmability, Jia *et al.* created an ”unshared” version of some Rodinia benchmarks that used scratchpad memory by simply using global memory instead. Simply replacing *_local_* functions with *_global_* ones will cause significant degradation of performance and lead to biased comparison.

Rogers *et al.* proposed Cache-Conscious Wavefront Scheduling (CCWS) to improve GPU cache efficiency by avoiding data thrashing that causes cache pollution [92]. CCWS restricts the number of wavefronts that are able to access the caches by changing the scheduler to schedule a limited number of wavefronts, which adversely affects the ability of hiding high memory access latency of GPUs.

Lee and Kim proposed a thread-level-parallelism-aware cache management policy to improve performance of the shared last level cache (LLC) in heterogeneous multi-core architecture [67]. They focus on shared LLCs that are dynamically partitioned between CPUs and GPUs. Mekkat *et al.* proposed a similar idea for heterogeneous LLC management [75], to better partition LLC for GPUs and CPUs in a heterogeneous system.

2.4 Eliminating Data Redundancy

Data redundancy widely exists in storage structures. Data deduplication is used in disk-based storage systems to reduce storage consumption [27, 111, 43].

Address Correlation [96] analyzes the phenomenon of data duplication in the L1 cache without giving a feasible implementation. Non Redundant Data Cache [77] proposes a sub-block level cache deduplication technique, which requires value-based data storage overhead and an extra value search on the critical path. Content-Based Block Caching [78] was an inline deduplication technique designed to improve disk-based storage systems. The binary tree structure and the significant storage overhead of the block cache (the storage of all possible LUN/offset pairs per data entry) make it impractical in a cache level. The mergeable cache architecture [13] was proposed to use deduplication in caches by merging cache blocks with similar data. Dusser *et al.* [30] investigated zero-content data contained in cache blocks and proposed to use an augmented cache to store null blocks to increase effective cache capacity. Since the percentage of zero-content blocks is small on average, the overall performance improvement is small. The HICAMP architecture [21] utilizes memory deduplication to reduce the overhead of copying shared data. Main memory in this architecture is designed as an associative hash table, suffering from underutilization. Moreover, the lookup in overflow area designed for hash collisions is expensive. CATCH [60] was proposed to use cache-content-deduplication in instruction caches. It only works for instruction caches since it does not support modifications in cached data.

Data compression is another technology to eliminate redundant data [118, 103, 15, 107, 59, 95, 102, 93]. We focus on cache compression techniques in particular. Yang *et al.* [110] proposed Frequent Value Compression in first-level caches. By encoding frequent values during the memory accesses into a small number of bits, the capacity of a cache block is potentially increased. Zhang *et al.* [115] proposed the frequent value cache (FVC) based on the observation of frequent value locality. FVC employs a value-centric approach to hold only frequently accessed values in a compressed form. Instead of using value-based encoding scheme, Alameldeen *et al.* [3] proposed Frequent Pattern Compression (FPC), a pattern-based compression scheme for L2 caches. By storing common word patterns in a compressed form with certain prefixes, FPC provides comparable compression ratio to more complex schemes. To reduce useless decompression overhead, Alameldeen *et al.* [2] proposed an adaptive policy to dynamically trade off between the benefit of compression with the cost overhead. Hallnor *et al.* [40] proposed to use a unified compression scheme to compress and decompress data in the LLC, main memory and memory channels. Although the unified compression scheme eliminates the additional compression and decompression expense required in data transferring between the LLC and the main memory, it cannot avoid compression/decompression overhead incurred with data transferring between different cache levels. Base-Delta-Immediate Compression [87] is another data compression algorithms representing data using a base value and an array of differences. For value or pattern based compression, besides the complex compression and decompression logic and unavoidable decompress latency, another drawback is that most of cache management policies cannot be used efficiently in a compressed cache because of the variation of block sizes.

The V-Way cache [89] proposes to vary the associativity of a cache on a per-set basis to increase the effective cache capacity. Zcache [94] proposes to provide higher associativity than the number of physical ways by increasing the number of replacement candidates.

3. PRELIMINARY WORK*

A last-level cache (LLC) occupies large chip area with significant power requirements. However, LLCs are inefficient because they store dead blocks. On average, 86% of blocks in a 2MB LLC are dead [56], causing low cache efficiency.

Previous work proposes different dead block prediction techniques [26, 74, 55, 44, 1]. These techniques consume significant storage and power overhead due to a large amount of metadata. Moreover, previous dead block prediction techniques cannot be applied to last-level caches effectively due to the fact that the access pattern associated with traces of memory instruction are filtered out by the L1 caches, leaving precious little contextual information.

As preliminary work to the research in this dissertation, we investigated *sampling dead block prediction* technique that uses sampled program counters (PCs) to predict if a LLC block is likely to be dead.*A sampling dead block predictor keeps track of only metadata of a small number of cache sets and updates the prediction table only on sampler accesses rather than every single cache access.

3.1 Sampling Dead Block Predictor

A sampling dead block predictor keeps a small partial tag array, referred to as a *sampler*. The sampler samples a fews sets from the whole LLC, i.e., a sampler of 32 sampled sets from a LLC of 2,048 sets. Each sampler entry contains only a 15-bit partial tag to conserve area and energy with high enough accuracy. Each access to the LLC incurs an inquiry to the predictor for prediction; while the predictor is only

*Part of this chapter is reprinted with permission from "Sampling Dead Block Prediction for Last-Level Caches" by Samira M. Khan, Yingying Tian, and Daniel A. Jiménez, 2010. Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society Washington, DC, USA. Copyright [2010] by IEEE Computer Society.

updated on accesses to the sampler sets. The intuition behind a sampler is that the learning acquired through sampling generalizes to the entire cache. Furthermore, with the help of sampler, the replacement policy used in the LLC can be different from the one used in the sampler, i.e., less expensive replacement policies such as random and not-recently-used (NRU).

A sampling dead block predictor uses only the PC to index its prediction table, instead of a trace of PCs (a.k.a. refTrace). The PC-based dead block predictor works better than the refTrace predictor for LLCs due to the fact that temporal locality has been filtered by core caches. Thus a reference trace brings more noise rather than useful information compared to simply using the PC of the last memory instruction that accesses to the corresponding block.

Beside the structure of a sampler and PC-based prediction, the third feature of our sampling dead block predictor is the skewed organization [98] of the prediction table to reduce hash collision. The predictor keeps three hash tables, each indexed by a different hash of a 15-bit signature. Each access to the predictor yields three counter values whose sum is used as a confidence compared with a threshold; if the threshold is met, then the corresponding block is predicted dead. With the help of the skewed organization, the effect of destructive conflicts is reduced.

3.2 Evaluation

Based on our experiments, a sampling predictor can reduce the number of LLC misses over LRU by 11.7% for memory-intensive single-thread benchmarks and 23% for multi-core workloads. The reduction in misses yields a geometric mean speedup of 5.9% for single-thread benchmarks and a geometric mean normalized weighted speedup of 12.5% for multi-core workloads. Due to the reduced state and number of accesses, the sampling predictor consumes only 3.1% of the of the dynamic power and 1.2% of the leakage power of a baseline 2MB LLC, comparing favorably with more costly techniques.

3.3 Summary

Sampling dead block prediction can improve performance for last-level caches while reducing the power and storage requirements over previous techniques. However, all the proposed dead block prediction techniques, including sampling dead block prediction, are designed for monolithic caches and cannot be applied to specific cache hierarchies, i.e. inclusive cache hierarchies. This dissertation explores characteristics of dead blocks in inclusive cache hierarchies and proposes temporal-based multilevel correlating cache management technique for inclusive LLC in the next chapter.

4. REDUCING WASTE OF DEAD BLOCKS IN INCLUSIVE CACHE HIERARCHIES*

Inclusive cache hierarchies have been widely used in Chip Multiprocessors (CMPs) because of the simplicity in maintaining cache coherence [10, 20]. However, compared to exclusive [53] or non-inclusive [112, 117] cache hierarchies, inclusive cache hierarchies have limited performance due to the inclusion property that all the cache blocks in higher level caches (a.k.a. core caches) must be a subset of the shared last-level cache (LLC). When the sum of the sizes of all core caches is comparable to the size of the LLC, overall capacity of the inclusive cache hierarchy becomes limited compared to exclusive and non-inclusive caches and the performance becomes poor. Moreover, when cache blocks in the inclusive LLC are replaced, they must also be invalidated from all higher level caches to maintain inclusion. Due to the fact that temporal locality is hidden by higher level caches, hot blocks that are highly referenced in higher level caches are rarely accessed in the LLC and therefore become LLC replacement victims and are invalidated from the entire cache hierarchy, eventually causing cache misses and incurring hundreds of cycles of memory access penalties.*

4.1 Motivation

To bridge the performance gap between non-inclusive and inclusive caches, one naive solution would be increase the size of the inclusive LLC. However, the chip area occupied by caches is already more than half of the overall chip area [105, 63, 94], which contributes to significant power consumption. Simply increasing cache sizes

*Part of this chapter is reprinted with permission from "Temporal-based Multilevel Correlating Inclusive Cache Replacement" by Yingying Tian, Samira M. Khan, and Daniel A. Jiménez, 2013. ACM Transactions on Architecture and Code Optimization (TACO), ACM, New York, NY, USA. Copyright [2013] by ACM.

will not help performance improvement.

Another way to improve inclusive caches is to intelligently choose LLC replacement candidates that have low temporal locality in higher level caches. Back-invalidation of these blocks will not cause performance loss. Previous work [47] identified blocks that have high temporal locality in higher level caches and reduced the frequency of back-invalidating them, making performance of inclusive cache hierarchies similar to that of non-inclusive caches. However, blocks that have poor temporal locality in higher level caches may still have temporal locality in the LLC and the replacement of these blocks will still hurt the overall performance. If a block will not be referenced in either higher level caches or the LLC[1], replacement and consequent back-invalidation of this block will not hurt performance. In fact, cache performance can be improved by replacing these known replaceable blocks with others deemed more useful. Thus, inclusive caches are capable to outperform non-inclusive caches.

This dissertation categorizes LLC blocks into three exclusive groups based on their temporal characteristics in both higher level caches and the LLC:

- *HAH blocks*: blocks that are highly referenced in higher level caches;
- *HAL blocks*: blocks that are highly referenced in the LLC;
- *LAL blocks*: blocks that have low temporal locality in both higher level caches and the LLC, which should be the group of LLC replacement candidates.

The LLC replacement candidates chosen from LAL blocks will not hurt inclusive cache performance. By contrast, replacing these blocks with useful ones as early as possible helps improve cache efficiency.

[1] At the time an invalidated block is requested again, if it is still kept in a corresponding cache of a non-inclusive cache hierarchy with the same replacement policy, it is treated as being re-referenced before its eviction; otherwise, it will not be referenced until its eviction.

Figure 4.1 shows the average percentages of different categories of blocks that are back-invalidated due to LRU replacement in the LLC. On average, 72.51% of the back-invalidated blocks are referenced again before their eviction in corresponding sets of a non-inclusive L1 cache. These blocks are HAH blocks and should not be invalidated from the cache hierarchy. There are 15.82% of blocks not hit in the L1 cache but re-referenced in the LLC before being replaced from the LLC. These blocks are HAL blocks that have temporal locality in the LLC and should not be invalidated from whole cache hierarchy, either. The remaining 11.67% of back-invalidated blocks are not re-referenced until they are evicted from the LLC in a corresponding non-inclusive cache hierarchy. These blocks are LAL blocks and replacement of these blocks is harmless to the performance of inclusive caches, so they should be chosen as LLC replacement victims whenever possible.

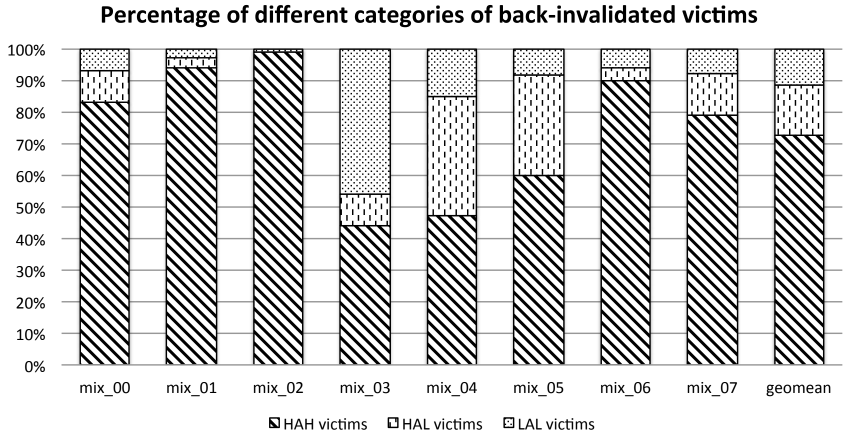

Figure 4.1: Percentage of different categories of back-invalidated blocks due to LLC LRU replacement

4.2 Temporal-based Multi-level Correlating Cache Replacement

This dissertation proposes Temporal-based Multi-level Correlating (TMC) cache replacement to choose LLC blocks that have low temporal locality in all caches as inclusive LLC replacement candidates. The technique intelligently categorizes LLC blocks into three exclusive groups using two-level categorization. It first categorizes LLC blocks based on their local temporal locality in the LLC into two groups: HAL and P-LAL (potential LAL blocks). Then it identifies HAH blocks from P-LAL blocks, and replaces the other LAL blocks when needed. It uses a *correlating temporal locality detector (CTL detector)* to detect LAL blocks with high accuracy. To first categorize HAL and P-LAL blocks, a CTL detector uses sampled program counters (PCs) to determine when a LLC block is likely to be a P-LAL block. The key intuition behind TMC is that if a memory access instruction PC leads to a P-LAL block, then there is a high probability that the same PC will lead to another P-LAL block. Previous work has found correlations between observed patterns of memory access instructions and cache accesses [25, 24, 101, 66, 55, 56, 108]. As stated in [25], it is because "*program behavior is repetitive, e.g., a critical section used a fixed set of instructions to read and modify data*". If an instruction is "*repeatable and always leads to (and can be associated with) the same event, a predictor can dynamically learn the behavior and accurately predict the event*". "*Much as path-based predictors [80] predict conditional branches dynamically based on correlating a sequence of basic-block addresses*", a PC-based predictor predicts an event dynamically based on correlating PCs of memory access intructions. In the LLC, temporal locality is filtered by higher level caches and memory access patterns are roughly consistent across groups of sets. Thus, the learning acquired through sampling a few sets generalizes to the entire LLC [90, 56]. Using sampled PC information to detect P-LAL blocks are accurate and consumes little. After the first level categorization, the CTL detector detects HAH blocks from the P-LAL group, using temporal information passively acquired from higher level caches. To get temporal information filtered by higher level caches, the naive way is to send this information to the LLC actively on every cache hit in higher level caches. However, the number of cache hit in higher level caches is extremely large and sending that number of requests to the LLC will consume a lot of bandwidth and energy. Therefore, in TMC the temporal locality of higher level caches is passively acquired by the LLC on each LLC miss, which is far less than the number of cache hit in higher level caches.

Compared to previous work, this technique has the following advantages: 1) detecting temporal-aware LLC replacement candidates with high accuracy and minimal storage overhead; 2) correlating multi-level temporal information with minimal communication overhead; 3) self-training at runtime for accurate detection. The design of correlating temporal locality detector and how it works in detail will be discussed in the following sections.

4.2.1 Correlating Temporal Locality Detector

A CTL detector consists of a detection table, a decoupled structure storing sampled LLC sets, a detection regulator and a modified invalidation message format. Figure 4.2 gives a block diagram of a CTL detector, showing the structure and related communication.

4.2.1.1 Detection Table

The detection table is a hash table of saturating counters, indexing by a hashed PC. It is accessed on sampled LLC cache accesses, which is descirbed in detail in the following subsection. Each access to the detection table yields a confidence compared with a threshold; as long as the threshold is not reached, blocks accessed by that PC

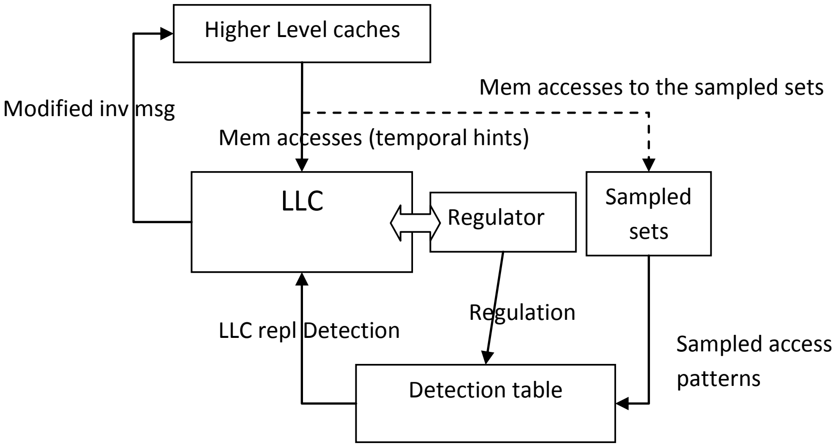

Figure 4.2: Block diagram of CTL detector

are likely to be HAL blocks, otherwise, blocks are grouped as P-LAL blocks. When an LLC block is referenced, the corresponding PC that accesses this block is hashed to index a detection table to determine its group.

To reduce the impact of conflicts in the table, The detection table uses the skewed organization [99, 76] of three tables. Each access to the detection table yields three values of counters that are summed up to compare with an threshold. In our experiments, the detection table has three 4,096-entry tables of 2-bit saturating counters, each indexed by a different hash function of a 15-bit partial PC. The skewed prediction table consumes a total of 3KB in storage.

4.2.1.2 Sampled Access Patterns

To reduce the storage and power overhead, the detection table is only updated on a small fraction of cache accesses referred to as sampled access patterns. The intuition is that memory access patterns are roughly consistent across sets. Thus, the CTL detector keeps track of program behavior by sampling a small number of LLC sets using a decoupled structure containing only partial tags and kept outside the LLC [91]. This structure can be configured differently than the configuration of the LLC to provide improved detection accuracy [56]. For instance, This dissertation finds that for a 16-way set-associative LLC, a reduced associativity of 12 ways provides accurate detection with less state. The LLC is decoupled from the sampled sets and does not require keeping extra PC information to update the detection table. Thus, each cache block in the LLC only holds two extra bits of metadata to store the categorization information, which consumes less than 0.5% of a 2MB LLC, further reducing the storage overhead. The sampled sets are accessed in parallel with the LLC. When an LLC access occurs in a sampled set, the CTL detector hashes the PC that accessed this block to index the detection table and update the corresponding saturating counter. Accesses to blocks whose sets are not in the sampled sets will not update the detection table. In on our experiments, the sampled sets contain only 64 sets of tags, randomly selected from the LLC. Each sampled set has 12 entries consisting of 15-bit partial tags, 15-bit partial PCs, and other metadata used for CTL detection, consuming 3.375KB of total storage overhead. Compared to accessing the detection table on each LLC access, the number of accesses to the detection table is reduced by more than 95%. Note that more sampled sets slightly slightly improves the detection accuracy while too many sets can increase destructive interference in the detection tables. Since power overhead is a serious issue in cache design, this dissertation chooses to slightly sacrifice the performance improvement for far less power consumption.

4.2.1.3 Modified Invalidation Message Format

The P-LAL group consists of HAH blocks and LAL blocks. In the second level categorization, a CTL detector randomly back-invalidates P-LAL blocks before replacing them from the LLC. The intuition is this: if the block is a HAH block, it will be requested soon by higher level caches; otherwise, it is a LAL block and can be replaced. To invalidate blocks from higher level caches, this dissertation simply modifies the format of invalidation message instead of changing default inclusion protocol.

In a conventional inclusive cache hierarchy, on each LLC miss, the LLC sends an invalidation message to all higher level caches with the physical address of the replacement victim. If the block is present in any caches, it is invalidated from those caches. Instead of generating extra messages, this work modifies the format of the invalidation message with extra physical address fields. On each LLC miss, the CTL detector sends one invalidation message encapsulating the physical address of a replacement victim together with N physical addresses of P-LAL blocks. There is no extra control message involved. The value of N is related to traffic overhead. Although the inclusion protocol is unchanged, the throughput of on-chip network is increased by N. A large N will also invalidate more higher level cache blocks and may cause unnecessary cache misses. Based on our experiments, $N = 1$ is sufficient for accurate detection as well as minimal communication overhead. Higher level caches de-encapsulate the invalidation message and invalidate blocks with addresses stored in the message. Higher level caches do not send any acknowledgment or temporal information to the LLC. Temporal locality information is passively acquired by the LLC with subsequent LLC accesses.

4.2.1.4 Detection Regulator

A detection regulator is used to regulate previous P-LAL detection. If the block is a HAH block, it will be requested soon by higher level caches and a LLC hit will occur. The detection regulator therefore gets the hint that the block should be kept in the higher level cache(s). Thus, the previous P-LAL is remarked as a HAH block, and its replacement state is updated. If the tag of block is located in the sampled sets, the corresponding counters in the detection table are also updated. If the block is not requested after certain cycles, it is treated as harmless for replacement. The detection regulator then marks the block as a LAL block and reinforces previous detection if the block is in the sampled set. The number of cycles the regulator waits before tagging LAL blocks is related to the accuracy of detection. If it waits longer, there is a higher probability of making a more accurate detection. However, it also delays the procedure of grouping blocks. Based on our experiments, waiting until another LLC miss occurs is sufficient to make accurate and timely decisions.

4.2.2 How Does TMC Work?

This section describes the TMC algorithm in detail.

On each LLC cache access, the technique first checks whether the set of the requested block is in the sampled sets. If so, the sampled tag array is accessed in parallel with the LLC; otherwise, only the LLC is accessed. On an access to a sampled set, if the tag is in the set, it is a sampled hit, the partial PC stored in the corresponding tag entry is used to index to the detection table, and the counter of the detection table entry is decremented by one, indicating the stored PC is likely to lead to a HAL block. The stored partial PC is updated to the PC that currently requests the block. The corresponding replacement status is updated, e.g., the accessed block is move to MRU according to LRU replacement policy. The categorization of the current block is decided by the detection table with the stored PC and comparing the corresponding counter with the threshold; if the threshold is not reached, the current block is marked as a HAL block; or it is marked as a P-LAL block. If the accessed block is not in the sampled set, it is a sampled miss, a replacement candidate is needed. If there is a LAL block marked in the sampled set, it is the replacement victim; if there is no LAL block in current set, a P-LAL block is chosen; if there is neither LAL nor P-LAL block, a normal LRU block is replaced. The partial PC stored in the replacement victim entry is indexed into the detection table, the corresponding counter in the detection table entry is incremented by one, indicating that the stored PC is likely to lead to a P-LAL block. Then the partial tag, partial PC, and replacement status are updated. Finally, the group of the block is updated by hashing the partial PC into the detection table and comparing the corresponding counter with the threshold.

The LLC is accessed in parallel with the sampled sets. On LLC hit, the PC of the memory instruction that accesses this block is indexed into the detection table to determine the categorization of the accessed block. Replacement status and other metadata are also updated. On LLC miss, a LAL block is chosen for replacement. If there is no LAL block, a P-LAL is chosen; if there is no P-LAL block, the LRU block is replaced. The corresponding metadata of the incoming block is updated and the categorization of the coming block is made by indexing the PC that caused the LLC miss into the detection table and comparing the counter with the threshold. Note that with TMC, the LLC may use a less costly replacement policies (e.g., not-recently-used replacement, random replacement, etc.) to further reduced storage overhead because the sampled sets are decoupled from the LLC and the detection table is updated only with sampled information. To fairly evaluate our technique, this work conservatively uses the LRU replacement policy in our experiments to maintain consistency with other techniques.

To maintain inclusion, the address of the replacement victim is sent back to all higher level caches for invalidation. Besides the address of the replaced block, the address of a P-LAL block (if there are P-LAL blocks marked, as shown in Figure 4.3(a)) is encapsulated into the back-invalidation message packet too for temporal hints from higher level caches. Both the replaced block and the selected P-LAL block is invalidated from all higher level caches if presented (Figure 4.3(b)). It is to detect HAH blocks from consequent behaviors of higher level caches according to the intuition: if the block is a HAH block, it will be requested soon by higher level caches (Figure 4.3(c)); otherwise, it will not be requested until being evicted. If the block is re-referenced before next LLC miss occurs, it is a LLC hit to a P-LAL block. Replacement status of this block is updated, indicating it keeps temporal locality, and this block is marked as HAH instead of P-LAL. If the set where this block locates is sampled, the corresponding counter in the detection table is also updated. If the back-invalidated P-LAL block is not referenced until another LLC miss occurs, it is marked as a LAL block and can be replaced.

Figure 4.4 shows the two level detection of the categories of LLC blocks. HAL blocks should be kept in the LLC and HAH blocks should be kept in the corresponding higher level caches; LAL blocks can be replaced from the LLC and invalidated from the whole inclusive cache hierarchy without causing performance loss.

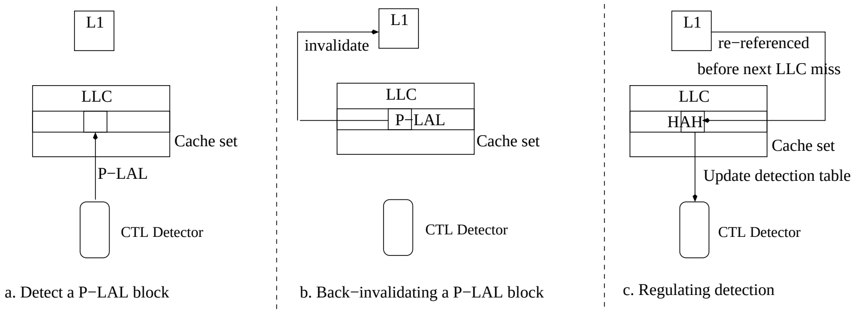

Figure 4.3: Detect HAH blocks from P-LAL blocks

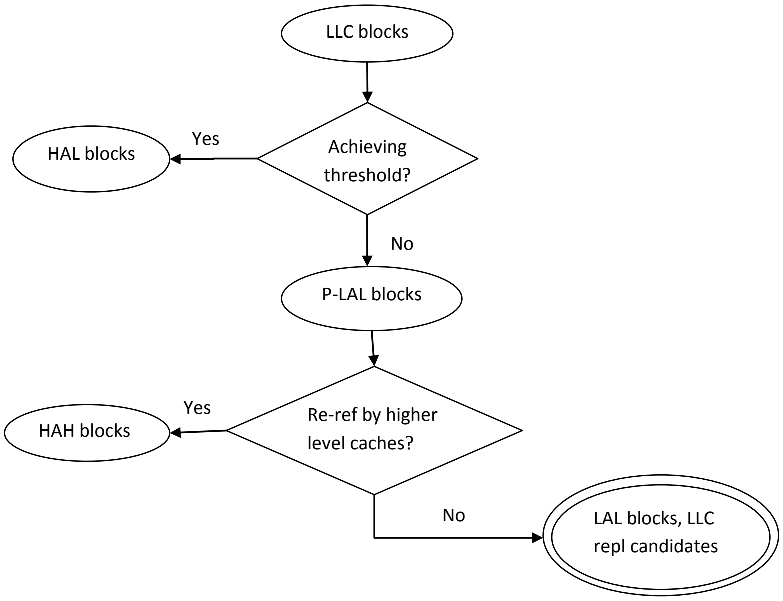

Figure 4.4: Two-level detection of LLC block categorization

4.2.3 Comparison with Previous Work

This section compares TMC with previous inclusive cache management techniques.

4.2.3.1 Temporal Locality Aware (TLA) Policy Suite

The Temporal Locality Aware (TLA) policy suite [47] was proposed to improve inclusive cache performance. It consists of three policies: Temporal Locality Hints (TLH), Early Core Invalidation (ECI) and Query Based Selection (QBS). As claimed in [47], TLH is only a limit study; ECI is a lower traffic solution with limited performance; QBS performs best among three TLA policies, achieving similar performance to a non-inclusive cache. The goal of TLA is to identify hot blocks in higher level caches (a.k.a. HAH blocks according to our definition) and avoid replacing these blocks from the LLC. Although the replacement victims chosen by TLA are not highly accessed in higher level caches, there is a chance that they will be re-referenced in the LLC (a.k.a. HAL blocks in our definition). Compared to TLA, TMC identifies hot blocks in higher level caches and also hot blocks in the LLC, and avoids replacing these blocks from the LLC. Cache efficiency is further improved by bringing useful blocks into the cache as early as possible. Therefore, instead of achieving similar performance, TMC actually outperforms non-inclusive caches significantly with low overhead.

Compared to QBS, the best management policy of TLA suite, TMC has not only better performance improvement, but also lower communication overhead. On each LLC replacement, QBS chooses a block and queries to see if it is present in any core caches. If the block is located in some core caches, QBS has to find another block in the LLC and repeats the query procedure again until it finds a block absent in all core caches to replace. If the number of queries is unlimited, up to 1.5KB of data is transferred on-chip on each LLC miss in a dual-core CMP, compared to 32 bytes per LLC miss with TMC. The query number of QBS is limited, as stated in [47], as at least two queries per LLC miss is required to achieve acceptable performance, the on-chip communication overhead is still six times more than that of TMC.

Compared to TLA suite, TMC requires extra PC information sent to the LLC. Sending this extra information to the LLC has been proposed by much previous work [25, 24, 101, 66, 55, 56, 108]. TMC has higher storage overhead, but it is as low as less than 1% of the capacity of the LLC in a dual-core CMP. TMC has lower communication overhead compared to QBS. TMC outperforms ECI and QBS by 10.7% and 8.6% respectively.

4.2.3.2 Sampling Dead Block Prediction

We also compare this work with our preliminary work: Sampling Dead Block Prediction (SDBP) [56]. SDBP is designed to identify dead blocks in the LLC and replace them with live blocks as early as possible to improve cache efficiency. Compared to other dead block prediction techniques, SDBP uses far less overhead to make predictions with much higher accuracy. However, since SDBP has no awareness of temporal locality in core caches, predictions are made based on local information of LLC accesses. Therefore the predicted dead block in the LLC can be highly referenced blocks in core caches and the replacement of these blocks will cause costly off-chip cache misses that hurt the inclusive cache performance.

We compare our work with SDBP from performance to overhead. Based on our experiments, TMC achieves an average performance improvement of 5.2% over SDBP in an inclusive cache hierarchy. Moreover, TMC performs comparable to an enhanced non-inclusive cache with SDBP, which utilizes SDBP in a non-inclusive cache. The storage overhead of TMC is 4KB compared to SDBP and the on-chip communication overhead is 32 bytes on each LLC miss.

The performance comparison will be shown in detail in Section 4.4.

4.3 Experimental Methodology

This section outlines the experimental methodology used in this study.

4.3.1 Simulation Environment

We use the MARSSx86 cycle-accurate simulator, a full system simulation of the x86-64 architecture. We use the multi-core implementations [86] with extensive enhancements for improved simulation accuracy and performance. It detailed models an out-of-order 4-wide 5-stage pipeline with a 128-entry reorder buffer, coherent caches with MESI protocol as well as on-chip interconnections. We modified the simulator to collect instructions-per-cycle (IPC) figures as well as cache misses.

The micro-architectural parameters closely model Intel Core i7 [16] with the following parameters (the same as in [47]): Three level cache hierarchy, L1, L2 and a shared LLC. The L1 and L2 caches are private in each core. The L1 I-cache and D-cache are 4-way 32KB each and the L2 cache is unified 8-way 256KB. As in the Intel Core i7, inclusion is not enforced between private L1 and L2 caches. The shared LLC is a unified 2MB cache for dual-core CMP and 2MB per core for 4-core CMP and 8-core CMP. We simulate a dual-core CMP with 2MB LLC to compare with previous work [47] that ran experiments under this configuration. However, the configuration of 2MB per core LLC is more realistic in current industrial design. The block size of all caches in the hierarchy is 64 bytes. The access latencies for the L1, L2, LLC, and main memory are 1, 10, 24, and 250 cycles, respectively. The default replacement policy of each cache is the LRU replacement policy.

In TMC, there are 64 sampled sets of tags, evenly chosen from among the sets of the LLC. Note that, for 2MB, 4MB, 8MB and 16MB caches, the number of sampled sets is constant, i.e., the storage overhead of sampled sets does *not* increase with core count. Each sampled set contains 12 entries consisting of a 15-bit partial tag, a 15-bit partial PC, and 2 bits of categorization. The detection table consists of three 4,096-entry tables of 2-bit saturating counters, also regardless of core count. For each LLC block, we store an additional 2 bits of categorization for TMC. Note that TMC does not make any prediction for prefetched blocks. Prefetched blocks are inserted and replaced using default LRU replacement policy. For QBS, it requires a control mechanism to maintain information about the presence of a queried block in all higher level caches.

4.3.2 Benchmarks

We use the SPEC CPU 2006 benchmark suite [42]. Each benchmark is compiled for the x86-64 instruction set. Note that not all the workloads will hurt inclusive cache performance. Based on our experiments, some workloads running with an inclusive LLC perform similar to a non-inclusive LLC. We classify workloads into two categories: **inclusion-sensitive** and **inclusion-insensitive** workloads. To evaluate whether a certain technique can help improve the performance of both categories of workloads, we run sixteen dual-core workloads, eight of them are inclusion-sensitive (The selection criteria is that the performance gap between the inclusive cache and the non-inclusive cache is larger than 3%.) and the other eight workloads are inclusion-insensitive (The performance gap is smaller than 3%.). We also randomly selected five 4-core workloads whose average performance gap is 1.7%, and five 8-core workloads with an average performance gap of 1.2%, to evaluate the scalability of all techniques. In each workload, benchmarks run simultaneously, restarting after one billion instructions until another two billion instructions (four billion instructions for 8-core workloads) are totally executed. Table 4.1, 4.2, 4.3 and 4.4 show the workload mixes we use in the experiments respectively. We also simulated ECI, QBS and SDBP in both the inclusive hierarchy and the non-inclusive hierarchy for comparison.

4.4 Evaluation

This section discusses the results of our experiments. In the graphs below, several techniques are referred as abbreviation. Table 4.5 gives a legend for them. The *Inclusive Cache* stands for the baseline.

Name	Benchmarks
mix-00	perlbench, mcf
mix-01	mcf, calculix
mix-02	hmmer, mcf
mix-03	gromacs, mcf
mix-04	gobmk, mcf
mix-05	gobmk, GemsFDTD
mix-06	gamess, sphinx3
mix-07	namd, xalancbmk

Table 4.1: Inclusion-sensitive dual-core workloads

Name	Benchmarks
mix-00	calculix, GemsTDTD
mix-01	astar, tonto
mix-02	gcc, mcf
mix-03	gobmk, soplex
mix-04	sphinx3, milc
mix-05	perlbench, libquantum
mix-06	bzip2, hmmer
mix-07	gromacs, h264ref

Table 4.2: Inclusion-insensitive dual-core workloads

Name	Benchmarks
mix-00	GemsFDTD, h264ref, tonto, lbm
mix-01	gobmk, sphinx3, xalancbmk, mcf
mix-02	namd, bzip2, gcc, mcf
mix-03	perlbench, gcc, namd, zeusmp
mix-04	sphinx3, gamess, zeusmp, perlbench

Table 4.3: 4-core workloads

Name	Benchmarks
mix-00	xalancbmk, tonto, mcf, sphinx3, libquantum, namd, gobmk, soplex
mix-01	perlbench, h264ref, gcc, hmmer, libquantum, soplex, calculix, GemsFDTD
mix-02	zeusmp, calculix, namd, gromacs, xalancbmk, bwaves, gamess, sphinx3
mix-03	omnetpp, h264ref, libquantum, gcc, hmmer, GemsFDTD, calculix, soplex
mix-04	astar, soplex, xalancbmk, GemsFDTD, h264ref, calculix, libquantum, hmmer

Table 4.4: 8-core workloads

Name	Technique
Inclusive Cache	Inclusive Baseline with default LRU policy in each cache
ECI	Early Core Invalidation cache management policy in an inclusive LLC
QBS	Query Based Selection cache management policy in an inclusive LLC
Inclusive SDBP	Dead block replacement with SDBP in an inclusive LLC
TMC	Temporal-based Multi-level Correlating cache replacement in an inclusive LLC
Non-inclusive LRU	Non-inclusive cache with default LRU policy in each cache
Non-inclusive SDBP	Dead block replacement with SDBP in a non-inclusive LLC

Table 4.5: Legend for the baseline and various cache optimization techniques

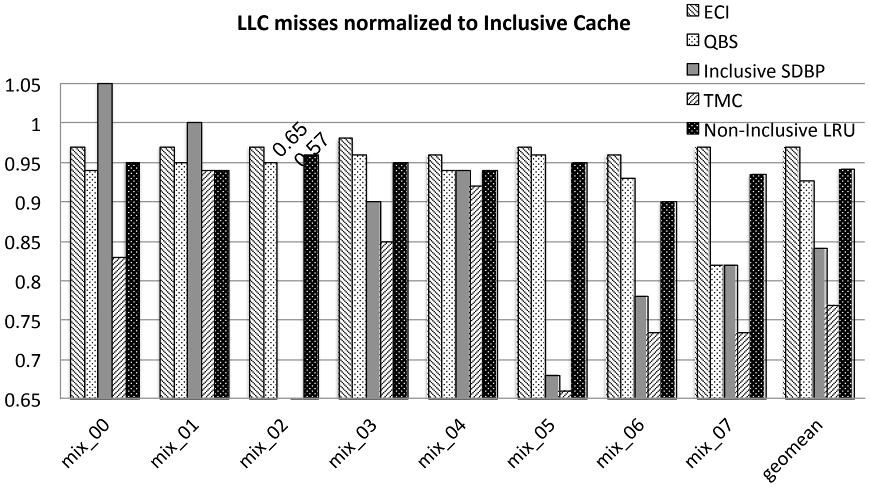

Figure 4.5: LLC misses for inclusion-sensitive workloads

4.4.1 Performance Improvement with Inclusion-Sensitive Workloads

Figure 4.5 shows the number of LLC misses normalized to Inclusive Cache for all inclusion-sensitive workloads. On average, ECI reduces LLC misses by 3% and QBS reduces it by 7.4%. Inclusive SDBP reduces LLC misses by 15.9%. For workload *mix_00*, instead of reducing the LLC misses, Inclusive SDBP increases the LLC misses by 5% because Inclusive SDBP is unaware of temporal information of other cache levels and a predicted dead block in the LLC may still be alive in higher level caches.

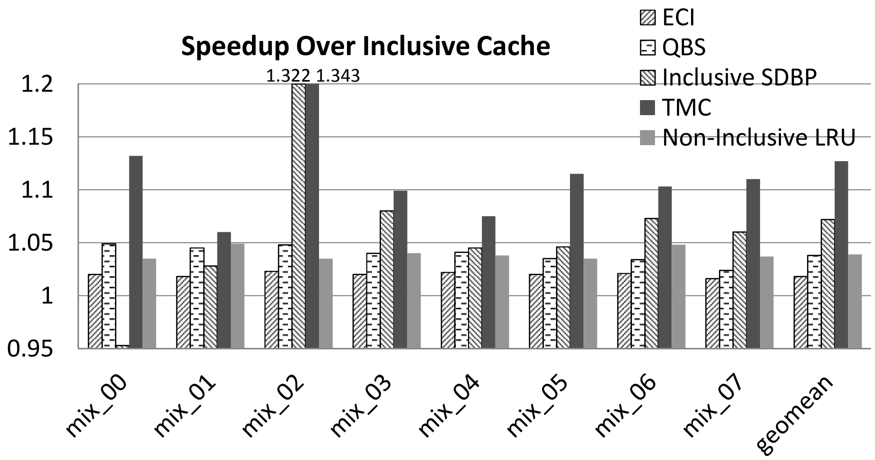

Figure 4.6: Performance improvement of inclusion-sensitive workloads

TMC reduces the LLC misses for all workloads to generate an average reduction of LLC misses by 23.2%. Non-inclusive LRU reduces the LLC misses of inclusive baseline by 5.9%.

Reducing cache misses leads to improved cache performance. Performance improvement normalized to Inclusive Cache (the IPC of enhanced inclusive cache with certain technique divided by the IPC of Inclusive Cache) is shown in Figure 4.6. The eight dual-core workloads are inclusion-sensitive workloads with an average performance gap (the IPC of Non-inclusive LRU divided by the IPC of Inclusive Cache) of 3.9%.

ECI improves the performance for all workloads and yields an average speedup of 1.8% while the QBS policy improves performance by 3.8%, which is similar to the performance of non-inclusive LRU. Inclusive SDBP, using SDBP technique in inclusive caches, gives a geometric mean speedup by 7.2% over Inclusive Cache. However, for workload *mix_00*, instead of improving the performance, Inclusive SDBP only achieves 95.3% of inclusive baseline. TMC improves the performance for all work-

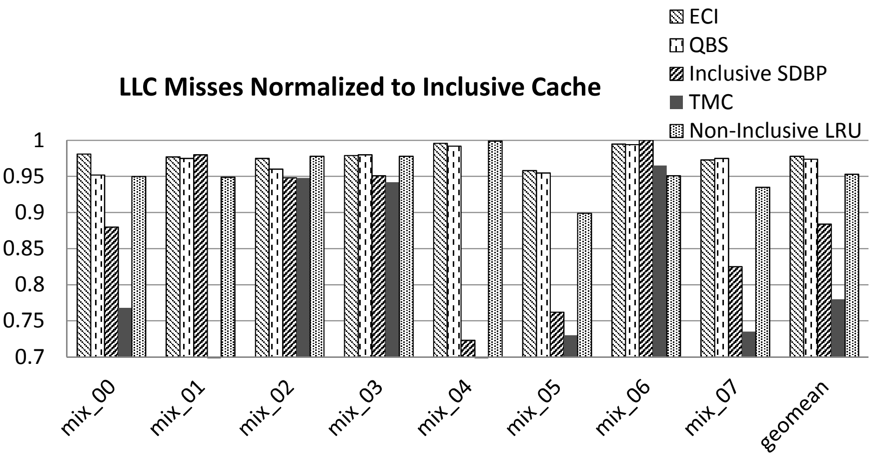

Figure 4.7: LLC misses for inclusion-insensitive workloads

loads to produce a geometric mean speedup of approximately 12.7%. With a 95% level of confidence, the margin of error is ± 5.7%.

4.4.2 Performance Improvement with Inclusion-Insensitive Workloads

As stated above, not all workloads are sensitive to the inclusive property. Before using a technique in inclusive caches, we must guarantee that the technique will not hurt the performance for insensitive workloads. We select eight insensitive workloads. None of the performance gap of these eight workloads is greater than 3% and the average performance gap is 1.4%. Figure 4.7 and Figure 4.8 show the number of LLC misses and the performance improvement normalized to that of Inclusive Cache for the inclusion-insensitive workloads.

On average, ECI reduces LLC misses by less than 3%. Simiar to ECI, QBS reduces it by 2.6%. Inclusive SDBP produces a reduction of LLC misses by less than 12%. For workload *mix_06*, instead of reducing the LLC misses, Inclusive SDBP increases the LLC misses by 2.43%. TMC generates an average reduction of LLC

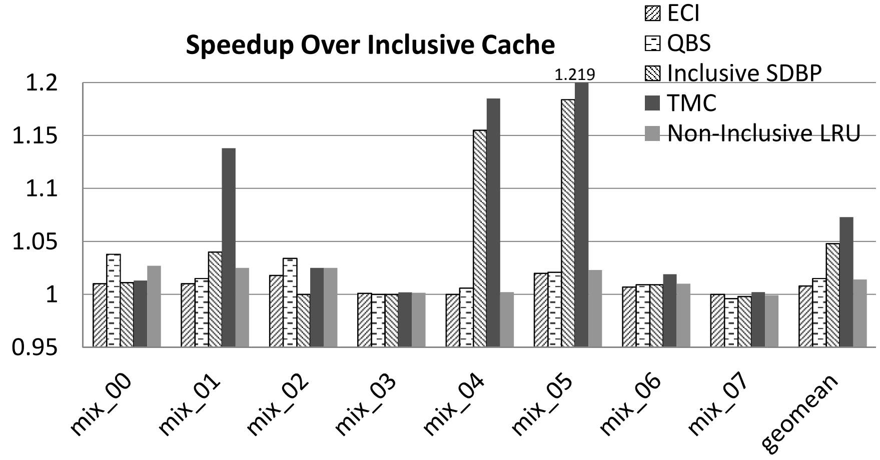

Figure 4.8: Performance improvement of inclusion-insensitive workloads

misses of 22%. Non-inclusive LRU reduces the LLC misses of inclusive baseline by 4.7%.

As shown in Figure 4.8, ECI yields a slight speedup of 0.8% over Inclusive Cache without hurting the performance of any workload. QBS gives a geometric mean speedup of 1.5% over inclusive baseline, performing comparable to non-inclusive LRU. However, it reduces the performance of workload *mix_07* by less than 1%. Inclusive SDBP achieves performance improvement of 4.8%. However, it also reduces the performance of *mix_07* by less than 1%. TMC improves the performance of all workloads and yields an average speedup of 7.3% over the inclusive baseline. With a 95% level of confidence, the margin of error is ± 5.8%.

Based on the results, TMC performs well for both inclusion-sensitive and inclusion-insensitive workloads while QBS and Inclusive SDBP hurt the performance compared to the baseline for some inclusion-insensitive workloads. ECI does not reduce the performance of any inclusion-insensitive workloads but the performance improvement it achieves is limited.

4.4.3 Compared to an Enhanced Non-inclusive Cache

By accurately replacing LAL blocks from inclusive caches as early as possible, the performance of inclusive caches not only achieves, but outperforms non-inclusive caches using default LRU replacement policy. We observed that both Inclusive SDBP and TMC far outperformed non-inclusive LRU. To quantify how much performance improvement that Inclusive SDBP and TMC can achieve over non-inclusive LRU, we compare Inclusive SDBP and TMC to Non-inclusive SDBP that uses SDBP in a non-inclusive LLC and can be treated as an upper-bound of an enhanced non-inclusive cache.

Figure 4.9 shows the improved inclusive cache performance normalized to the enhanced non-inclusive cache in a dual-core CMP with inclusion-sensitive workloads.

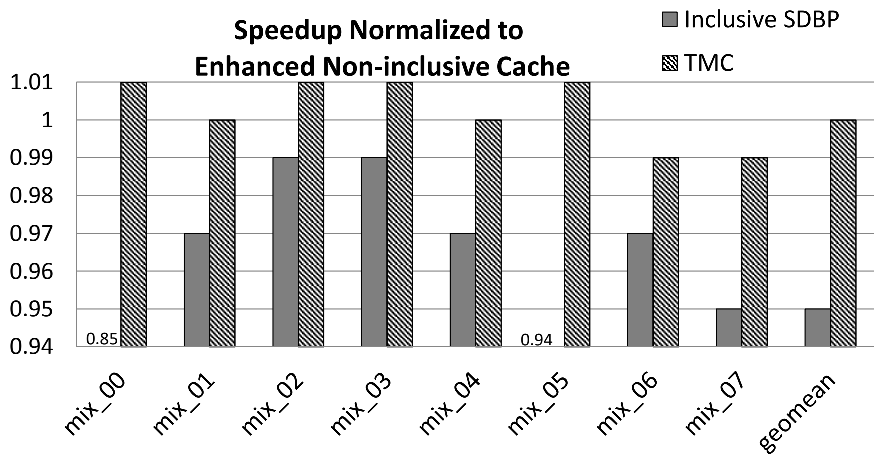

Figure 4.9: Speedup normalized to enhanced non-inclusive cache

Compared to Non-inclusive SDBP, Inclusive SDBP reduces the inclusive cache performance for all eight workloads by 5% on average. TMC reduces the performance for two workloads and improves four workloads to perform comparable to

Non-inclusive SDBP.

In conclusion, inclusive caches with TMC outperform non-inclusive caches and even comparable to optimized non-inclusive caches, while maintaining the simplicity of cache coherence.

4.4.4 Scalability Analysis

To evaluate the scalability to different number of cores, we randomly select five groups of inclusion-insensitive 4-core workloads with an average performance gap of 1.7% and five groups of inclusion-insensitive 8-core workloads with an average performance gap of 1.2%. The capacity of the LLC in the experiments is 2MB per core, i.e. 8MB for the 4-core configuration and 16MB for the 8-core configuration. Figure 4.10 and Figure 4.11 show the normalized LLC misses of each technique on 4-core and 8-core workloads, respectively. Figure 4.12 and Figure 4.13 show the performance improvement of each technique on both 4-core and 8-core workloads.

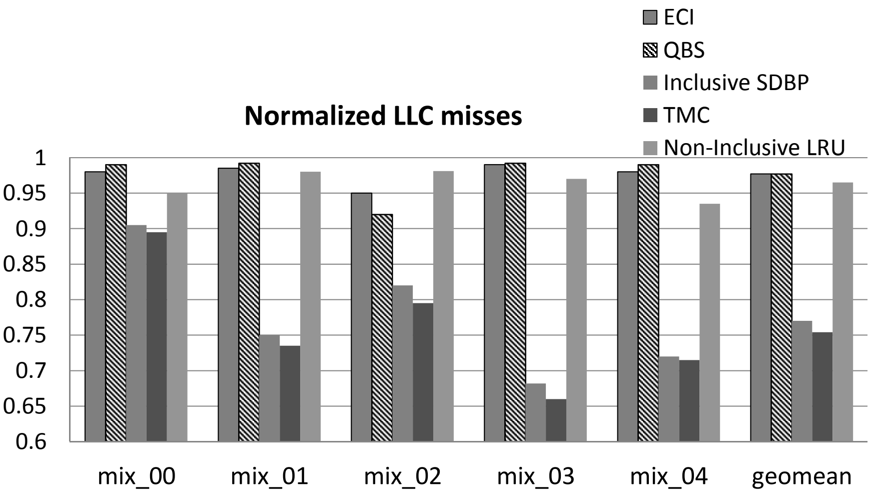

Figure 4.10: Normalized LLC misses for 4-core workloads

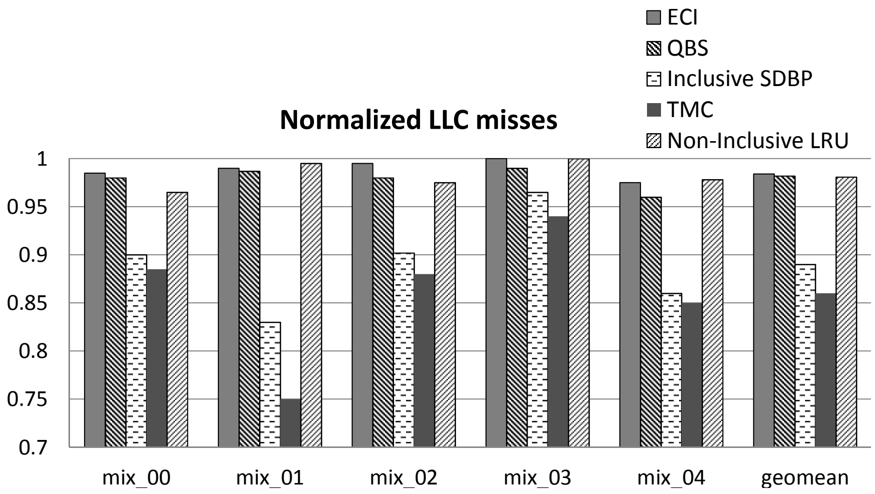

Figure 4.11: Normalized LLC misses for 8-Core workloads

In a 4-core CMP, as shown in Figure 4.10, ECI reduces the average LLC misses of five workloads by 2.3%. QBS makes slight reduction of LLC misses for four workloads and generates and average reduction of 2.3%. Inclusive SDBP reduces the LLC misses by 23% on average while TMC yields a reduction of 24.6%. Compared to Inclusive Cache, Non-inclusive LRU reduces the LLC misses by 3.5%.

The normalized LLC misses for each technique compared to an inclusive cache for 8-core workloads are shown in Figure 4.11. On average, ECI reduces LLC misses by 1.6% while QBS produces a reduction of 1.8% to the inclusive baseline. Inclusive SDBP reduces the LLC misses by 11% and TMC generates an average reduction of 14%. Non-inclusive LRU reduces the LLC misses by 1.9% compared to the inclusive cache.

As shown in Figure 4.12, in a 4-core CMP, ECI yields an average speedup of 1.2% over Inclusive Cache while QBS gives a speedup of 1.5%. Inclusive SDBP improves the inclusive cache performance by 8.7% and TMC improves it by 9.8%. With a 95% level of confidence, the margin of error is ±4.4%. None of the techniques hurts cache

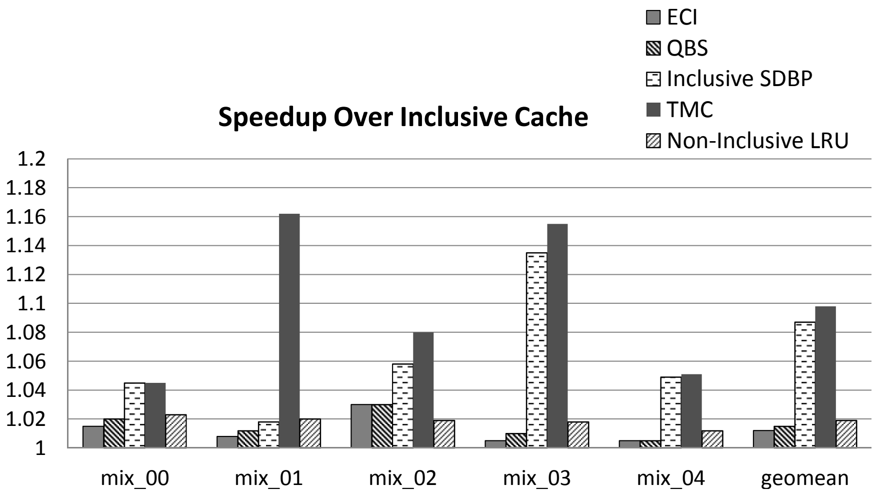

Figure 4.12: Performance improvement for 4-core workloads

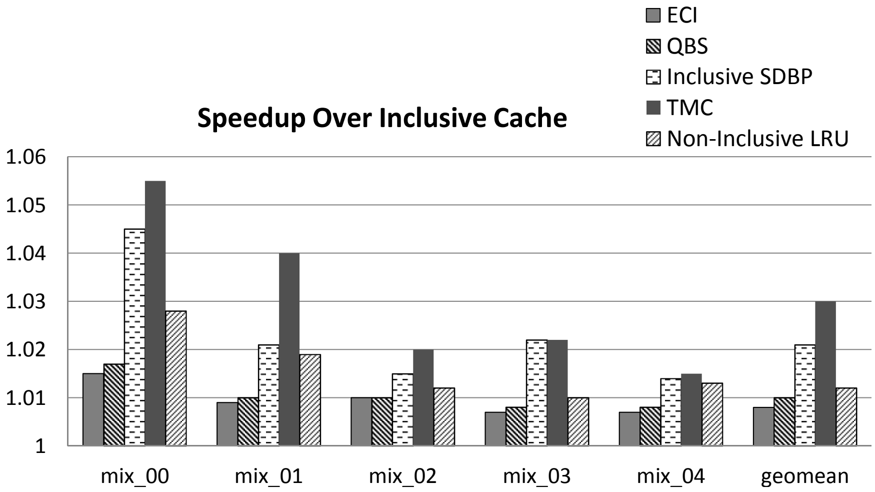

Figure 4.13: Performance improvement for 8-core workloads

performance for any 4-core workloads.

The performance improvement of each technique for 8-core workloads is shown in Figure 4.13. On average, ECI produces a slight speedup of 0.8% over the baseline while QBS produces an average speedup of 1%. Inclusive SDBP improves the performance of inclusive caches by 2.1%. TMC yields an average speedup of 3% over the inclusive cache. With a 95% confidence level, the margin error is ±1.3%. This performance is better than non-inclusive LRU that performs better than the baseline 1.2%.

Note that for 8-core workloads, the performance improvement is lower compared to dual-core or 4-core workloads with any of the techniques. This is because our methodology scales the size of the LLC with the core count. The number of back-invalidated HAH blocks significantly increases when the size of the LLC is not significantly larger than the sum of all higher level caches [47, 112] which hurts the performance of inclusive caches. In practical design, both Intel and AMD 8-core processors have more modest sized LLCs [16, 6]. Previous TLA work also ran experiments on 8-core CMP with 8MB L3 cache. Thus, we ran more experiments to evaluate the scalability of TMC on 8-core CMP with a more practical 8MB LLC. Figure 4.14 shows the performance improvement of TMC and Non-inclusive LRU normalized to 8MB Inclusive Cache. On average, TMC achieves 12.2% of speedup over an 8MB inclusive cache, scales well with the increased number of cores.

In general, all the techniques are scalable to different numbers of cores. ECI and QBS help inclusive caches perform similarly to non-inclusive caches while Inclusive SDBP and TMC perform better than non-inclusive caches.

Based on previous evaluation, inclusive caches with ECI cannot perform as well as non-inclusive caches. With QBS, inclusive caches perform similarly to non-inclusive ones by paying significantly high communication overhead. Both Inclusive SDBP and TMC help inclusive caches achieve better performance than non-inclusive caches with reasonable overhead.

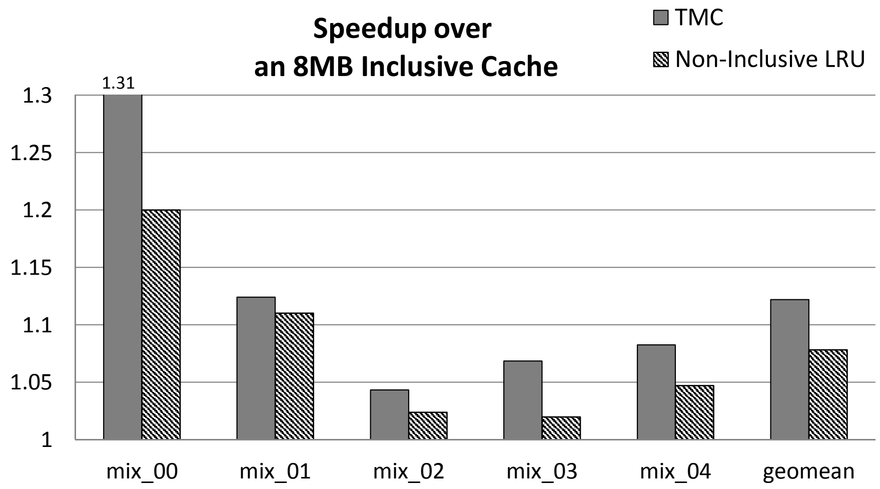

Figure 4.14: Performance improvement for 8-core workloads on 8MB LLC

4.4.5 Detection Accuracy and Coverage

The detection of LAL blocks is not 100% accurate. A misprediction may cause extra delay. Mispredictions come from false positives and false negatives. False positives are more harmful because they detect useful blocks as LAL blocks to cause costly off-chip cache misses. The coverage of a detector is the ratio of LAL detection to all detection. Higher coverage means the detector can help find more opportunity for the optimization. Figure 4.15 shows the coverage and false positive rates of the CTL detector in TMC. On average, it detects a block as a LAL block for 45% of LLC cache accesses and has a much low false positive rate as of 1.6%, explaining why it achieves high average speedup.

4.4.6 Overhead Analysis

This section evaluate the storage and power overhead of TMC.

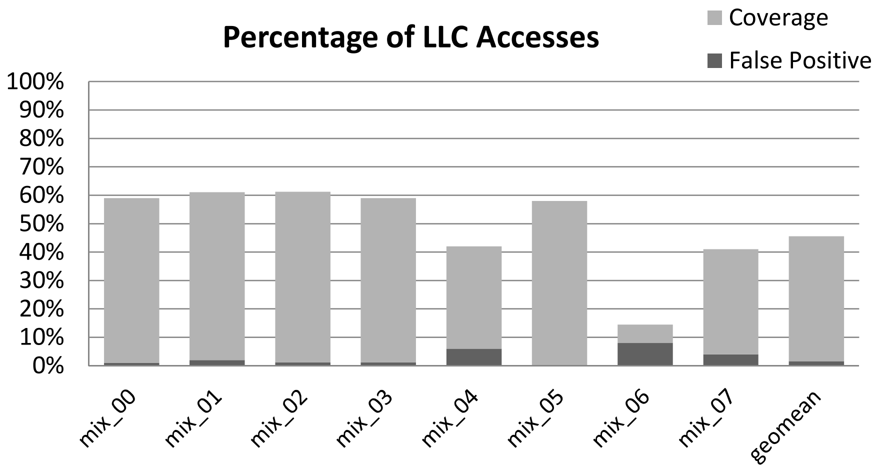

Figure 4.15: Coverage and false positive rate of CTL detector

4.4.6.1 Storage Overhead

The detection table consists of three 4,096-entry tables of 2-bit saturating counters, consuming a total of 3KB in storage. The sampled sets contain 64 sets. Each set has 12 entries consisting of 15-bit partial tags, 15-bit partial PCs, and 2-bit categorization indicator, consuming 3.375KB of total storage overhead. To indicate the groups of LLC blocks, each LLC block also keeps a 2-bit indicator, consuming 8KB in total for a 2MB LLC. Thus, the CTL detector consumes a total of 14.375KB, which is less than 1% of the capacity of a 2MB LLC in a dual-core CMP.

4.4.6.2 Power Overhead

The storage overhead costs power overhead. Table 6.4 shows the results of CACTI 6.5 simulations [79] to determine the leakage and dynamic power of the TMC technique. The sampled sets were modeled as the tag array of a cache with as many sets as in the sampler. The detection tables was modeled as a tagless RAM with three banks accessed simultaneously, To attribute extra power to cache metadata (2 bits per cache block), 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. As shown in Table 6.4, the dynamic power of the CTL detector is 0.078W and that of the extra metadata is 0.008W. So the total dynamic power of TMC is 0.086W. The leakage power of the CTL detector is 0.005W and that of the extra metadata is 0.002W. Therefore the total leakage power of TMC is 0.007W. The baseline LLC has a dynamic power of 2.75W and a leakage power of 0.512W. Thus, the TMC technique consumes 3.1% of the dynamic power consumption and 1.4% of the leakage power budget.

	Dynamic Power	**Leakage Power**
Detector structure	0.078	0.005
Extra metadata	0.008	0.002
Total	0.086	0.007

Table 4.6: Dynamic and leakage power of TMC (Watts)

4.4.6.3 Communication Overhead

TMC tends to replace and back-invalidate LAL blocks. When the CTL detector is warming up lacks knowledge, there might be back-invalidation of HAH blocks that increases the number of L1 misses and on-chip traffic between the L1 cache and the LLC due to the re-fetch of these HAH blocks. After the CTL detector has gone through sufficient training, the P-LAL blocks that the detector invalidates are likely to be LAL blocks, whose invalidation will not cause any future re-fetch and thus will not increase the number of L1 misses. By contrast, with the improved efficiency of the whole inclusive cache hierarchy, cache misses in *each level* will be reduced. Figure 4.16 shows the normalized number of L1 misses in each core of TMC to that of the inclusive cache in a 4-core CMP. Most of the L1 misses are TMC is similar to that in inclusive caches. In workload *mix-03*, there is an increased on-chip communication overhead of 12.9% between core 0 and the LLC but reduced overhead of 23.2% between core 2 and the LLC compared to the overhead in Inclusive Cache.

There is no re-fetch overhead to off-chip memory compared to the inclusive cache with default LRU replacement policy. The overall communication overhead to off-chip memory is reduced with TMC due to the increased efficiency of the LLC. On average the off-chip memory accesses are reduced by 24.6%, as shown in Figure 4.16.

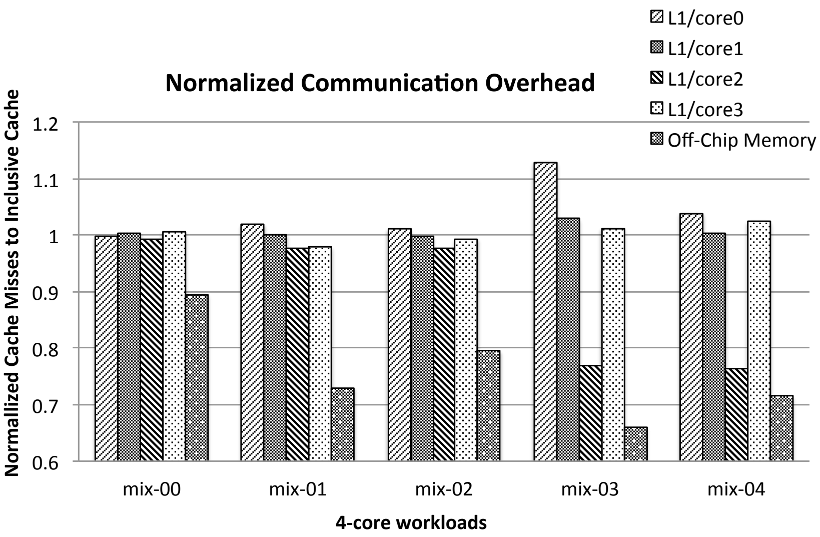

Figure 4.16: Normalized communication overhead for 4-core workloads

4.5 Summary

In this chapter we have quantified dead blocks in inclusive cache hierarchies as LAL blocks and evaluated the performance improvement and overhead of the proposed temporal-based multilevel correlating cache management.

Generic dead block prediction techniques designed for monolithic caches cannot be applied directly to cache hierarchies with specific features. To continue exploring dead blocks in diverse memory hierarchies, we explore dead blocks in GPU cache hierarchies in the next chapter.

5. REDUCING WASTE OF DEAD BLOCKS IN GPU PRIVATE CACHES*

A Graphics Processing Unit (GPU) is a highly parallel processor consisting of hundreds to thousands of concurrently operating arithmetic logic units. Though they were originally hard-coded circuits meant only to accelerate 3D graphics computations, modern GPUs are now fully programmable general-purpose processors. General purpose GPU computing uses GPUs to accelerate applications in domains such as science, engineering, physics, media, and statistics [104].*

GPUs hide long memory access latencies through a high degree of thread-level parallelism. If one group of threads is stalled on a long latency memory request, many others can take that opportunity to execute. This is acceptable for most graphics workloads, but some GPGPU workloads can cause the whole pipeline to stall by causing all available thread groups to wait on memory. In addition, both graphics and general-purpose applications can heavily tax the memory bandwidth of a GPU. As such, GPUs traditionally used small read-only texture caches and scratchpad memories in order to increase available bandwidth to their computational pipelines. However, these resources are difficult to use for GPGPU workloads because they require either the programmer or compiler to decide whether particular memory accesses should go through these subsystems.

Modern GPU architectures have adopted hardware-controlled cache hierarchies between globally accessible DRAM and the compute units to aid programs that are unable to use the GPU's shared memory [81]. For example, AMD's Graphics Core Next (GCN) architecture [7] has a 16KB private L1 cache for each compute unit and

*Part of this chapter is reprinted with permission from "Adaptive GPU Cache Bypassing" by Yingying Tian, Sooraj Puthoor, Joseph L. Greathouse, Bradford M. Beckmann, and Daniel A. Jiménez, 2015. Proceedings of the 8th Workshop on General Purpose Processing using GPUs, ACM, New York, NY, USA. Copyright [2015] by ACM.

64-128KB of shared L2 cache per memory channel. Nvidia's Fermi architecture [81] has a 16KB/48KB configurable private L1 cache for each streaming multiprocessor and 768KB of shared L2 cache.

5.1 Motivation

Hardware-managed GPU caches are used for two main purposes: 1) to cache data with immediate spatial and temporal locality, and 2) as write-combining buffers to reduce the memory bandwidth and energy requirements of the system. Although caches are effective write-combining buffers for GPGPU workloads, they are less useful at exploiting locality [52]. The underlying reason for this is the streaming nature of GPGPU memory accesses resulting in good spatial locality but very low temporal locality.

5.1.1 Memory Characteristics of GPGPU Programs

Traditional graphics workloads traverse large scenes of 3D vertices while calculating shading values, performing mathematical transformations, and laying textures on surfaces. These algorithms stream large amounts of data from memory, consuming hundreds of megabytes to render a single frame. Because such large working sets are completely impractical to hold in on-chip caches, GPUs have traditionally had copious memory bandwidth and enough parallelism to keep these long latency accesses from stalling.

This bandwidth and latency hiding has subsequently affected the kinds of general-purpose applications that are commonly ported to run on GPUs. GPGPU applications often look like graphics workloads: highly parallel, regular, and with large storage and bandwidth needs. Although these workloads may exhibit good data reuse, the distance between repeated accesses to the same value is such that most of the reusable data is evicted from the cache before it can be touched again.

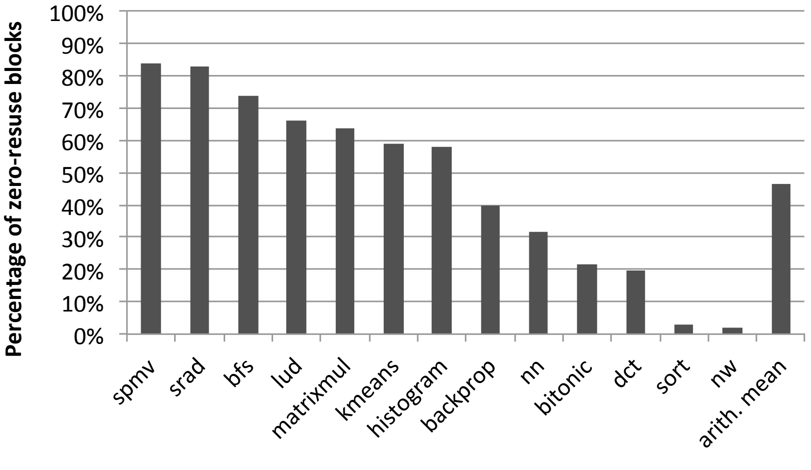

Figure 5.1: Zero-reuse blocks in the L1 data cache

Figure 5.1 demonstrates this idea across a series of benchmarks from the Rodinia suite [17] and a selection of AMD APP SDK [8] programs. The *zero-reuse* bars represent the percent of cache blocks that are evicted from a 16KB L1 cache before they are touched again. This data shows that an average of 46% (and a maximum of 84%) of cache blocks are evicted by the pseudo-LRU replacement algorithm without being touched again. Inserting this data into the cache costs energy, but only results in pollution and the potential eviction of other useful blocks.

Streaming data accesses in these programs, coupled with large data sets, are the primary reasons for these long reuse distances. For graphics applications, GPUs traditionally used different memory subsystems for data that would cache well (such as textures), allowing other data to bypass these specialized caches. Similarly, scratchpad memories (called Local Data Stores on AMD GPUs [4, 7] and Shared Memory on Nvidia GPUs [81, 82]) can be used to manually store reusable data while skipping streaming values. Some GPUs now include compiler hints to say that particular static loads are streaming and so should not be cached [61, 54, 9].

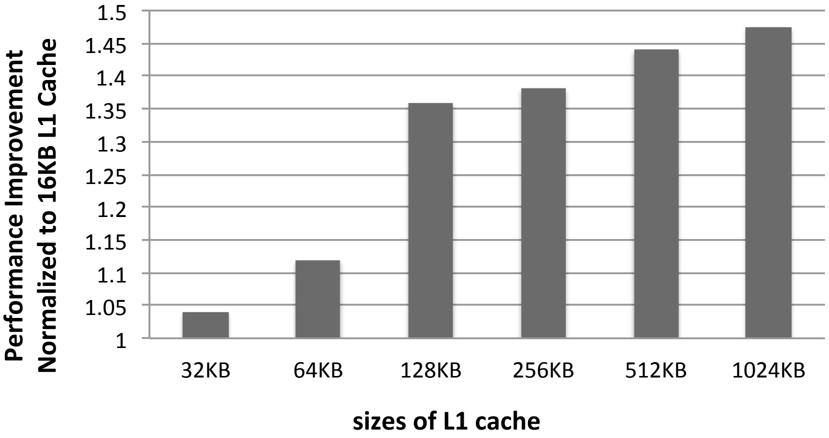

Figure 5.2: Performance improvement normalized to a 16KB L1 cache with different cache sizes

As GPGPUs extend further into non-traditional domains, more programmers whose expertise lies outside GPU architectures are using these devices. Such explicitly managed memory systems are known to be more difficult to use than hardware-controlled caches [71], requiring such structures limits the market for GPUs to only expert programmers. Moreover, scratchpad memories are not portable across devices or generations of designs. Scratchpad sizes and layouts change over time, further increasing the programmer's burden. With these issues in mind, we focus on hardware mechanisms that can improve existing GPU caches and be transparent to software and programmers.

5.1.2 Improving GPU Caches

Two major problems have been identified with GPU caches: 1) They are not effective at exploiting temporal locality due to noise from streaming data; and 2) insertions and evictions of useless data consumes energy without performance gain.

Figure 5.2 shows the average performance improvement of different L1 data cache sizes normalized to a 16KB baseline over a series of GPGPU benchmarks described in Section 5.3.2. This demonstrates that more powerful caching systems have the capability to increase the GPU's performance. However, L1 caches larger than 16-64KB are impractical for current GPU designs.

Current AMD GPUs have 16KB of L1 data cache per compute unit. The previous generation of Nvidia chips had a dynamically configurable 16KB or 48KB L1D. The current generation of Nvidia GPUs, Kepler, can configure its L1 data cache to be 16, 32, or 48KB [82]. However, this L1 cache is only used to store local data, such as register spills, and is always bypassed when accessing global data, i.e. there is essentially no hardware-controlled R/W L1 data cache [83].

These cache sizes are unlikely to increase significantly as the general performance benefit from adding extra cache space does not outweight the extra area taken up by these caches. That area could instead be dedicated to more computational resources, which would directly increase performance in traditional graphics and many GPGPU applications. Unfortunately, at these sizes, the large GPGPU data structures and streaming data cause unnecessary cache evictions, reducing reuse and wasting energy. They are not cacheable because of the thrashing or streaming access patterns [48].

If these zero-reuse blocks were not inserted into the cache when accessed, only useful data would be installed. This data would also be more likely to remain in the cache and be reused before being evicted. Therefore, a bypass decision mechanism could increase the efficiency of the cache without requiring either effort on the programmer's part or a large amount of area.

5.2 Adaptive GPU Cache Bypassing

This dissertation proposes a dynamic GPU cache bypassing technique that prevents zero-reuse blocks from being placed in the L1 data cache of the GPU compute units that access them. If a block is unlikely to be accessed again before it is evicted from the cache, the mechanism instead sends the data directly to the compute unit, bypassing the cache. This technique saves energy by avoiding needless insertions followed by later evictions and improves performance by reducing cache pollution.

The most important question for such a technique is: how can the hardware decide whether a block is zero-reuse when it fetches data during a cache miss? Previous CPU cache bypassing techniques proposed to make decisions using mechanisms such as frequency of accesses [51, 58], temporal locality information [37], or reuse distance [49]. Using information related to memory addresses is impractical in GPU caches due to the large number of data accesses. Single Instruction Multiple Data (SIMD) units used in GPUs simultaneously perform the same task on different items of data, resulting in a high degree of data parallelism and large numbers of memory addresses. Using memory address-related information to make bypass decisions would require a large amount of storage, which is not amenable to GPUs. Figure 5.3 shows a study of the number of 64B memory blocks accessed in our set of benchmarks. Hundreds of thousands of memory blocks are accessed during the execution of these small kernels.

Compared to the large amount of data accessed in GPGPU workloads, the number of memory instructions is much smaller because program behavior is dominated by a few small kernels and a high degree of thread-level parallelism.

Figure 5.4 shows that there are far fewer distinct load instructions executed in each benchmark. Rather than hundreds of thousands of data addresses, there are instead only tens to hundreds of distinct program counters (PCs) of memory instructions. Thus, a predictor indexed using PCs of memory instructions is more practical than one indexed with accessed addresses. There are fewer distinct entries, requiring far less on-chip storage, and there are fewer distinct values concurrently generated,

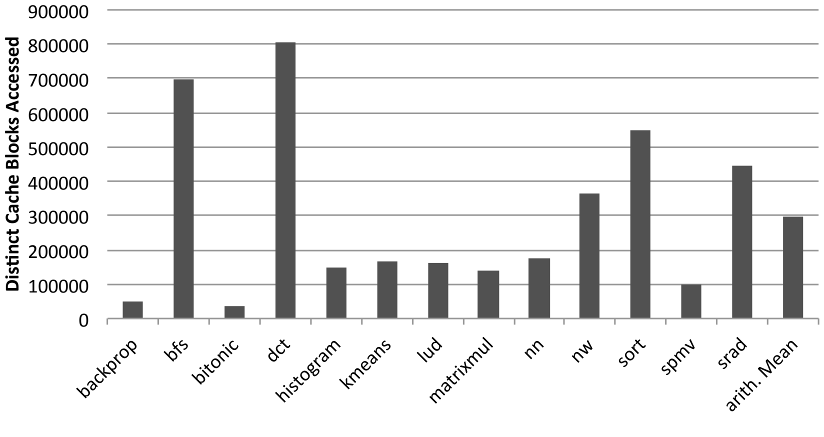

Figure 5.3: Number of distinct blocks accessed in execution of each benchmark

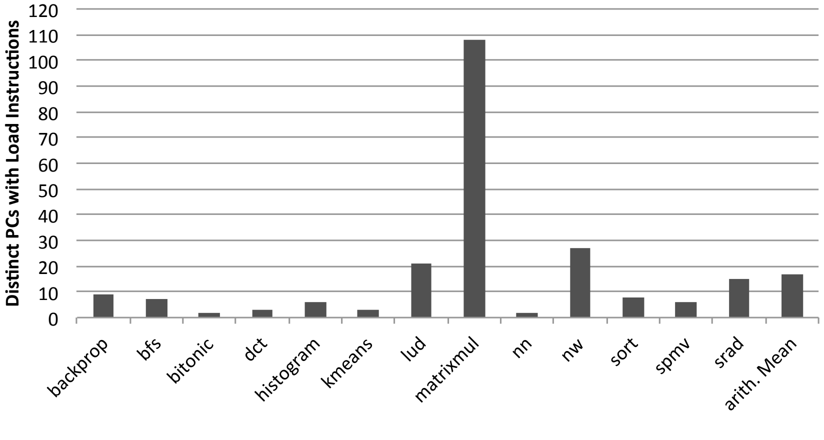

Figure 5.4: Number of distinct load instruction PCs executed in each benchmark

reducing the port count of the predictor. Beyond the capacity concern, a PC-based predictor can be more accurate because it learns to generalize the behavior of a single instruction to multiple data blocks.

Previous CPU dead block prediction techniques leverage the fact that sequences of memory instruction PCs tend to lead to the same behavior for different memory blocks [64, 74]. The preliminary work showed that in last level caches (LLCs), the PC of the last memory instruction to touch a particular block is highly correlated with whether or not the block will be used again, leading to a compact and highly accurate predictor [56]. Wu *et al.* used this observation to classify LLC blocks in terms of their likely reuse distances [109].

This intuition is extended to predict zero-reuse blocks in GPGPU workloads. Although both this technique and the sampling dead block prediction (SDBP) [56] use PCs to make a prediction, the intuition behind them is different. SDBP is designed for LLCs, where much of the temporal locality has been filtered by higher level caches. Thus, using the PC of the last memory instruction rather than a trace of PCs as in previous work [64, 74] achieves higher accuracy in LLCs. By contrast, our technique is designed for GPU L1 caches, where temporal information is complete. However, this dissertation proposes to use the PC of the last memory instruction, rather than sequences of memory instructions, because of the observation of characteristics of GPGPU memory accesses as shown in Figure 5.4. Since GPU kernels are small and frequently launched, the interleaving changes frequently. This interleaving has a negative impact on warm-up time for the predictor when using PC traces rather than the last PC.

5.2.1 Structure of PC-based Bypass Predictor

This section describes the design of a PC-based bypass predictor.

Figure 5.5 shows the structure of the PC-based bypass predictor in a GPU L1 cache. The predictor keeps a 128-entry prediction table aside the L1 cache, where each entry contains a 4-bit saturating counter. This table is indexed by a hashed PC and consumes 64 bytes of storage of each L1 cache. The number of entries of the prediction table is very small taking advantage of the characteristics of GPU

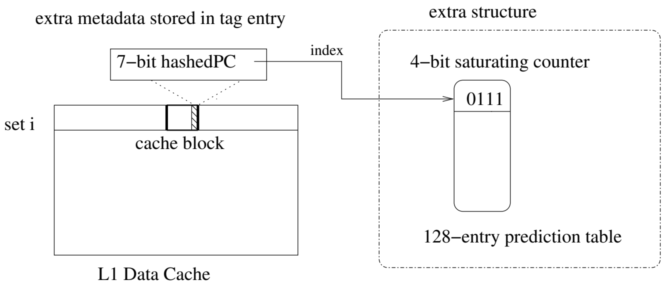

Figure 5.5: Structure of PC-based bypass predictor in GPU L1 cache

programs that there are only few distinct PCs. Each access to the prediction table yields a confidence compared with a threshold; if the threshold is met, then the corresponding block accessed by that PC is predicted as zero-reuse. Beyond the prediction table, each tag entry stores one more item of metadata: a hashed PC value (7 bits) that records the last memory instruction that referenced the current block.

No matter how high the prediction accuracy is, a bypass misprediction in this design is irreversible. That is, when a bypass decision related to a PC is made, no blocks accessed by that PC will be placed into the L1 cache. If the prediction is wrong, all subsequent blocks accessed by this PC will miss in the L1 cache, causing additional penalties for accessing lower cache levels. To correct potential mispredictions, each L2 cache block keeps an extra bit, called the *bypassBit*, to help verify predictions. When a block is selected to be bypassed on a L1 cache miss, the prediction is sent to the L2 cache with the memory request. The L2 cache stores this information in the corresponding L2 entry (set $bypassBit = 1$). If the block is referenced again before being evicted from the L2 cache, this information is sent back to the L1 cache with the requested data, indicating that the previous bypass prediction might be incorrect. The requested block will not be bypassed this time. Instead, it is placed into the L1 cache for potential verification.

5.2.2 Prediction Algorithm Details

This section describes the prediction algorithm in detail.

```
On each L1 access (address, PC):
If (the access is a hit) {
     /* corresponding prediction entry is updated to indicate a
        reused block */
     predictionTable[block[address].hashedPC)]--;
     /* PC information is stored in the cache entry for future
        verification */
     block[address].hashedPC = hash(PC);
     /* update LRU replacement status */
     block[address].LRU_stack = 0;
}
else {
    /* get bypass prediction */
    bool isBypassed = predictionTable[hash(PC)] >= threshold ? true
       : false;
     /* send memory request to L2, along with the prediction */
    SendMemReq (address, isBypassed);

    if (!isBypassed) {
       /* if the prediction is to not bypass
        * a victim block(VictimAddr) has to be replaced
        * corresponding prediction entry is updated to indicate a
           zero-reuse block
        */
        predictionTable[block[VictimAddr].hashedPC]++;
        /* bypassBit stored in L2 cache is sent back with requested
            data */
        bypassBit = L2Block[address].bypassBit;
        L2Block[address].bypassBit = false;
        Data = RecvMemPkt(address,  L2Block[address].data,
           bypassBit);
        /* cache installation */
        block[address].data = data;
        block[address].hashedPC = hash(PC);
        block[address].LRU_stack = 0;
    }
    else {
        /* if the prediction is to bypass, use the bypassBit to
           confirm */
        bypassBit = L2Block[address].bypassBit;
        L2Block[address].bypassBit = false;
        Data = RecvMemPkt(address,  L2Block[address].data,
           bypassBit);
        if(bypassBit) {
             /* if the bypssBit indicates a previous misprediction,
                 do not bypass */
             isBypassed = false;
             block[address].data = data;
             block[address].hashedPC = hash(PC);
             block[address].LRU_stack = 0;
        }
        else {
              /* bypass L1 cache */
        } } }
```

Listing 5.1: Pseudocode of PC-based bypassing prediction

Listing 5.1 gives the pseudocode of our PC-based bypass predictor. The least-recently-used (LRU) replacement policy is used in this example. On each L1 access, the L1 cache is searched for the tag of the requested block. If there is a tag match, then the last PC that accessed this block led to a reused block. A prediction table entry indexed by the hashed PC stored in the cache entry is decremented to indicate a potentially reused block. The current PC is hashed and stored in the cache entry, with the corresponding replacement status updated.

If it is a cache miss, the bypass prediction of the requested block is made and sent to lower level caches with the memory request. If the predictor decides not to bypass this block, the LRU block is replaced with the incoming block. The prediction entry indexed by the hashed PC stored in the LRU block entry is updated, indicating this PC likely leads to zero-reuse blocks. On receiving the requested block, the corresponding metadata is updated.

If the prediction is to bypass, the requested block will not be placed into the cache. However, there is a chance that the prediction is incorrect. If the bypassBit sent from the L2 cache is set, it is possible that this block would be reused (since it is hit in the L2 cache). In this case, instead of being bypassed again, this block is placed into the L1 cache for potential re-references and misprediction correction. The misprediction correction does not distinguish if the bypassBit set by a previous bypass prediction is from a different compute unit. The intuition is that different compute units behave similarly in GPUs. Thus, using prediction information from other compute units will not interfere with one another; by contrast, it helps correct potential mispredictions with limited information.

Note that previous warp scheduling proposals such as Cache-Conscious Wavefront Scheduling (CCWS) [92] were also designed for increasing GPU cache efficiency. Our work is orthogonal to warp scheduling techniques and can be used along with them for better performance. To fairly evaluate our technique as a GPU cache management technique, this dissertation conservatively uses "Oldest-First" scheduling technique which minimizes cache thrashing caused by warp interference.

5.2.3 Comparison with Counter-based Bypass Prediction

Counter-based bypass prediction [58] is a CPU last-level cache bypassing technique. It proposes to use an event counter in each cache block to record an event of interest such as cache accesses. When the counter reaches a threshold, the block observes no more reuse. This information is stored in a prediction table indexed by hashed block addresses and PCs. To bypass zero-reuse blocks, the block addresses and PCs of bypass victims are indexed to the prediction table for prediction before inserting into the cache. Compared to PC-based bypass prediction which tracks repetitive program patterns, counter-based prediction tracks block access patterns. GPU program features a small number of distinct PCs addressing a large amount of distinct data. To record block-level reuse patterns, counter-based prediction keeps extra information per block and a large prediction table. Due to the limited capacity of the GPU L1 caches, counter-based prediction consumes too much on-chip area to be practical in GPU cache designs.

Counter-based bypass prediction achieves worse performance on average and much higher storage overhead compared to PC-based bypass technique. Based on our experiments, on average, in each 16KB L1 cache, counter-based prediction takes more than 10.5KB of storage overhead, while PC-based prediction takes less than 256 bytes of overhead in each L1 cache, and a total 0.5KB of storage overhead in a shared 256KB L2 cache. In addition, PC-based bypass prediction outperforms counter-based prediction by 2.3%. A detailed evaluation is given in Section 5.4.

5.3 Experimental Methodology

This section outlines the experimental methodology used in this study.

5.3.1 Simulation Environment

We use an in-house APU simulator that extends gem5 [12]. The simulator runs with a microarchitectural timing model of a GPU that directly executes the HSA Intermediate Language (HSAIL) [33] and produces detailed statistics including execution cycles, cache miss rate and traffic. Table 5.1 shows the configuration of the GPU side of the evaluated system, which is similar to the AMD Graphics Core Next architecture [7]. The warp scheduling policy is oldest-first, which attempts to minimize cache thrashing caused by wavefront interference. All caches use a default Pseudo-LRU replacement policy. Compared to the baseline system, each L1 bypass predictor requires a 128-entry prediction table of 4 bit counters and additional metadata of 7-bit in each tag entry, costing 224 bytes in total of storage overhead in each L1 cache. To help verify prediction accuracy, each L2 tag entry contains one extra bit of bypassBit, taking 0.5KB in total. We also evaluate counter-based bypass prediction. For a 16KB L1 cache, counter-based bypass predictor contains a prediction table of 128*128 two dimensional matrix structure, containing 5-bit of prediction information. Each tag entry contains 20-bit extra information for hashed PC, counters, and the prediction. The storage overhead of counter-based bypass predictor is 10.626KB.

5.3.2 Benchmarks

We evaluate 13 benchmarks from Rodinia [17], AMD SDK [8], Opendwarfs [32] and one custom microbenchmark implementing a 4-byte radix sort with high data reuse. These workloads represents all OpenCL benchmarks we have that can be

GPU Clock	1GHz
Compute Units	8
Compute Unit SIMD Width	64 scalar units by 4 SIMDs
GPU L1-I/D Cache	8-way 16KB, 64B, 1 cycle of tag access, 4 cycles of data access
GPU Shared L2 Cache	16-way 256KB, 64B, 4 cycles of tag access, 16 cycles of data access
L3 Memory-side Cache	16-way 4MB, 15 cycles of tag access, 30 cycles of data access

Table 5.1: System configuration

Program	**Input**	**MI**	**Description**
matrixmul	512×512	395.6	matrix multiplication
spmv	256×256	215.8	sparse matrix-vector multiplication
bfs	1M	202.7	breath-first search
nn	342080	130.4	k-nearest neighbor
kmeans	16384	121.8	kmeans clustering
bitonic	131072	114.3	bitonic sort
srad	512×512	102.2	speckle reducing anisotropic diffusion
backprop	8192×16	89.7	back propagation
dct	2048×2048	76.2	discrete cosine transform
sort	65536	76.2	radix sort
histogram	1024	43.1	histogram
nw	512×512	30.4	needleman-wunsch
lud	1024×1024	14.2	LU decomposition

Table 5.2: Workloads and inputs

compiled and run in our simulator. Table 5.2 lists the characteristics of the evaluated benchmarks. The benchmarks are sorted by *memory intensity* (MI, calculated as the global memory accesses per 1000 instructions) [116]. Among all the benchmarks, benchmark *matrixmul*, *spmv*, *bfs* are memory-intensive workloads and benchmark *dct*, *sort*, *histogram*, *nw* and *lud* are compute-intensive workloads. We use medium to large inputs for each benchmark.

5.4 Evaluation

This section gives detailed analysis of the bypass predictor, regarding energy, performance, and prediction accuracy.

5.4.1 Energy Saving

In this section we evaluate the energy savings of the bypass predictor. Insertion of zero-reuse blocks wastes energy without performance improvement and may even cause cache pollution. Cache bypassing significantly reduces the energy consumption by preventing unnecessary filling of data into caches. A large amount of streaming data is bypassed from caches, reducing the energy cost and potential cache pollution.

In a conventional L1 cache, on each L1 cache access, both the tag and data arrays are accessed in parallel for fast response. On a cache miss, both the tag and data arrays will be accessed again to fill the selected cache block with data from lower level of the memory hierarchy. With cache bypassing, on each L1 cache access, the tag and data arrays are accessed in parallel together with a direct access to a very small prediction table. On a cache miss predicted to bypass, the data is sent directly to the compute unit without accessing the cache structure again. As shown in Figure 5.6, on average 58% of cache fills are prevented with cache bypassing.

Energy (nJ)	**16KB baseline**	**bypassing**
per tag access	0.00134096	0.0017867
per data access	0.106434	0.106434
per prediction table access	N/A	0.000126232
Dynamic Power (mW)	44.2935	36.1491
Static Power (mW)	7.538627	7.72904

Table 5.3: Power cost

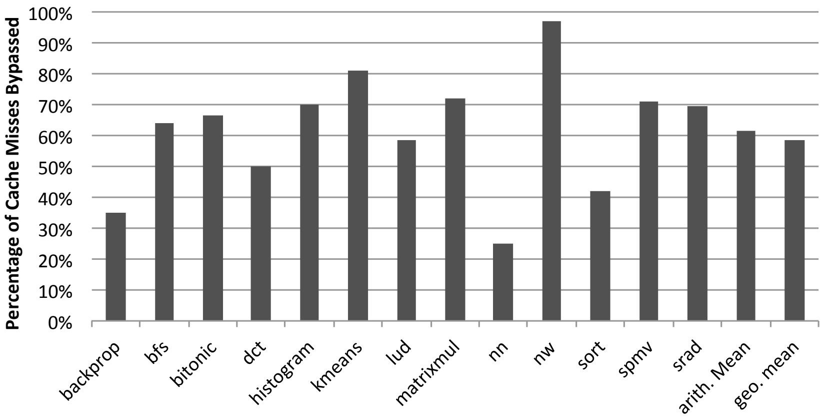

Figure 5.6: Ratio of bypasses to cache misses

The reduction of unnecessary cache fills significantly reduces the energy consumption compared to the baseline. Table 5.3 shows the results of CACTI 6.5 simulations [79] to determine the energy reduction by adding a PC-based bypass predictor compared to the 16KB baseline. The extra structure of the prediction table is modeled as a tag array (with 4-bit tags) of a direct-mapped cache with 128 sets. Each tag entry in the L1 cache with bypassing has 8 more bits[1] and the data array remains unchanged. Figure 5.7 gives the reduction in energy with PC-based bypassing compared to the 16KB baseline. The energy cost of the 16KB baseline is reduced by up to 49%, and on average by 25% with bypassing. Table 5.3 also shows the quantified power cost. On average, PC-based bypassing reduces dynamic power by 18% over the 16KB baseline and increases the leakage power by only 2.5%.

5.4.2 Performance

Bypassing improves the cache efficiency by preventing unnecessary filling of data into caches to cause cache pollution. Therefore data stored in caches are likely to be

[1] We add 7 bits in each tag entry for prediction. To use CACTI correctly, we evaluated it as 8 bits.

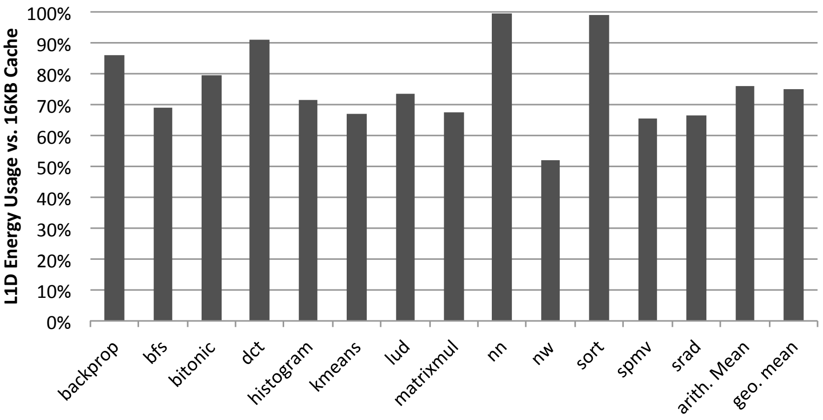

Figure 5.7: Energy usage of 16KB cache with bypassing (relative to baseline)

useful. In another word, bypassing improves cache efficiency and overall performance.

In this section we evaluate cache miss reduction and performance improvement over a 16KB L1 cache baseline for PC-based bypass prediction, counter-based bypass prediction, and compare them to a large 32KB L1 cache baseline. For brevity, we use Baseline, PC-based predictor, counter-based predictor and 32KB Cache as abbreviations, respectively.

Figure 5.8 shows L1 misses normalized to the baseline system for each benchmark with different techniques and Figure 5.9 shows the speedup, i.e. the execution time of benchmarks on the baseline system divided by the execution time on the evaluated system. To help analyze the results, Figure 5.10 shows the hit rate in the L1 cache of each benchmark in the baseline system.

PC-based bypass prediction offers a significant performance improvement in benchmarks *matrixmul*, *bfs*, and *spmv*. These benchmarks observe intermediate or low L1 hit rate in the baseline (as shown in Figure 5.10) because most of the data that should be reused are replaced due to cache pollution. As shown in Figure 5.1, these benchmarks have a high percentage of zero-reuse blocks while very low or none ratio of blocks that are only accessed once during execution. With PC-based bypass prediction, streaming data is bypassed and previously doomed useful blocks are kept in the L1 cache. Cache efficiency is significantly improved for these benchmarks. Among these three benchmarks, *bfs* produces a speedup of 13% over the baseline, *spmv* yields a speedup of 9% and *matrixmul* generates a speedup of 6%. Compared to PC-based bypassing, the counter-based bypass predictor provides much less speedup for benchmarks *bfs* and *spmv* but yields a better performance for benchmark *matrixmul*. In comparison, the 32KB Cache provides less performance improvement for all three benchmarks.

Benchmarks *backprop* and *srad* have intermediate to low L1 hit rate as well as a low reuse rate 5.10. For these two benchmarks, most zero-reuse blocks are accessed only once during execution. The performance of benchmark *backprop* with a PC-based predictor is improved by 4.3% and *srad* reaches a speedup of 4% over the baseline.

Benchmarks *sort*, *dct*, and *lud* are compute-bound benchmarks [18]. Increasing cache size does not significantly improve performance for these benchmarks. Their overall performance mainly depends on the compute ability of SIMD processors. All three evaluated techniques yield an average speedup of about 3%.

Some benchmarks observe little performance improvement with all evaluated techniques. Benchmarks *kmeans* and *histogram* invoke many kernel launches and frequently shared data between the CPU and the GPU. The performance is thus dominated by pulling data from CPU side, resulting in no significant performance improvement with any of the techniques. Benchmark *bitonic* contains frequent barrier synchronizations [34], causing the program to execute in lock-step with no observed performance improvement with any techniques while larger cache sizes degrade the performance due to the cache walk required when kernels complete. Benchmark *nw* puts all reused data into the scratchpad memory for computation and write through data to global memory when the computation is finished. As shown in Figure 5.6, with PC-based bypassing, benchmark *nw* has more than 95% of cache insertions prevented. Therefore, for benchmark *nw*, there is little performance improvement while around 50% of energy reduction with PC-based cache bypassing.

Storage is a key issue in GPU cache design. On average, the PC-based bypassing prediction in a 16KB cache outperforms both the counter-based prediction and the 32KB cache system while using far less overhead, which means almost half of the chip area dedicated for private caches is saved without performance degradation. The tension between number of compute units and the size of caches makes it infeasible to increase the cache size naively. For example, to double the cache size of 16KB L1 caches in a 'Tahiti' graphics card with 32 parallel compute units [5] without increasing the chip area, we estimate that up to 4 CUs would need to be removed, leading to a theoretical maximum throughput degradation of 12.5% [22, 70, 23] [2].

5.4.3 Prediction Accuracy and Coverage

In this section we evaluate prediction accuracy and coverage of PC-based bypassing.

There are two groups of mispredictions: false positives and false negatives. False positives are more harmful because they wrongly bypass reused blocks. Further rereferences cause extra misses. The coverage of the bypass predictor is the ratio of bypass prediction to all prediction made on cache misses. Higher coverage means

[2] Based on estimates derived from die images and expert teardowns [22, 23], the total chip area is $352mm^2$ and 32 CUs take up approximately $176mm^2$. The computational logic in each CU is estimated to be approximate $3.7mm^2$ and a 16KB cache structure takes $1.8mm^2$. Doubling the cache size to 32KB leads to an increase of $0.8mm^2$ in area. A chip of roughly the same area of $176mm^2$ would therefore require removing 4 CUs to fit the extra cache storage.

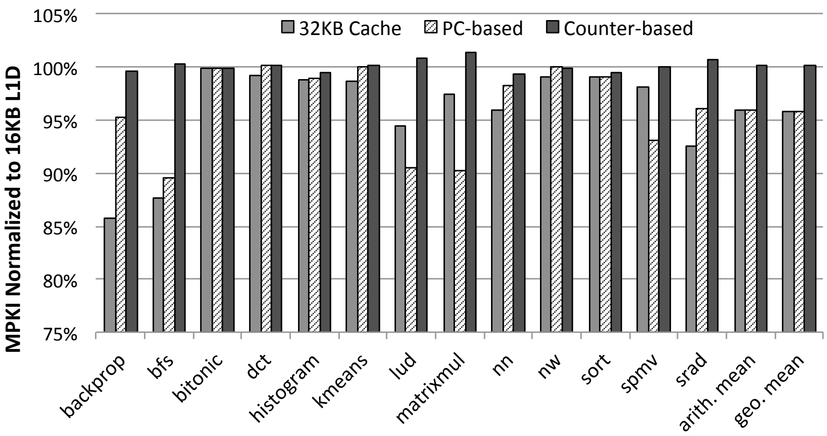

Figure 5.8: Reduction in L1 misses for different techniques

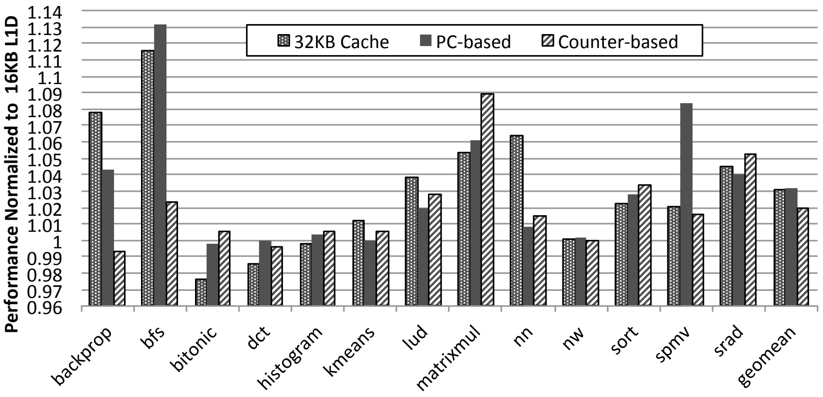

Figure 5.9: Speedup over the baseline for different techniques

more opportunity for the optimization. Figure 5.11 shows the coverage and false positive rates of the PC-based bypass predictor. On average, the coverage rate is 58.6%, and the false positive is 12%.

Note that the reason why the false positive rate is higher than previous work [56] is because we include incorrectly bypassed or replaced blocks as false positives. Sampling-based dead block prediction [56] calculated false positive as (number of ac-

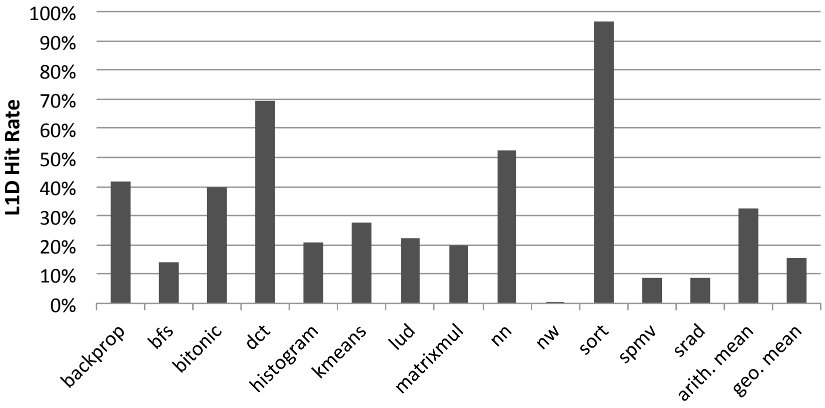

Figure 5.10: L1 cache hit rate of each benchmark in the baseline

cesses to predicted dead blocks / number of dead predictions), so only re-referenced blocks predicted dead are categorized as false positives. Using the same computation as sampling-based dead block prediction gives a false positive rate of 1% for the GPU cache bypassing.

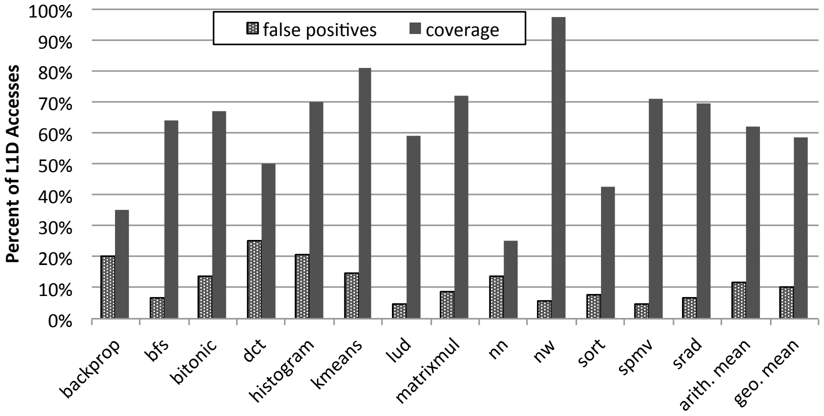

Figure 5.11: False positive and coverage of bypassing predictor

5.4.4 A Case Study of Benchmark Needleman-Wunsch

GPU L1 caches can be treated as hardware-controlled scratchpad memories. Both of them store reused data shared within a compute unit. Programmers use scratchpad memories to bypass streaming-like data by explicitly storing only reused data into the scratchpad memories. A GPU L1 cache with bypassing stores reused data by adaptively bypassing streaming-like data without programmer intervention. We quantify the extent to which dynamic L1 cache bypassing can make up for the potential performance lost in production environments where the effort to program scratchpad memories is impractical.

To explore the effectiveness and limitation of adaptive L1 cache bypassing, we take a Rodinia benchmark *Needleman-Wunsch* for a case study. *Needleman-Wunsch (nw)* uses a global optimization algorithm for DNA sequence alignment in bioinformatics [17]. It dynamically loads the northern and western edges of a 2-D matrix into the scratchpad memory and processes the data in the scratchpad memory. After computation, results are written through to the main memory. Most of the kernel is spent doing partial computation in the scratchpad memory. There is very little reuse observed in L1 caches because the scratchpad filters reused data. We re-wrote the source code of *nw* to remove the use of the scratchpad memory (benchmark *nw-noSPM*). Note that we did not simply replace the *_local_* functions into *_global_* functions (which will cause significant degradation of performance); rather, we re-wrote the source code by understanding the original algorithm resulting in a best-effort program without the use of scratchpad memories.

Figure 5.12 shows the execution time of *nw* and *nw-noSPM* with different configurations. As shown in the left of Figure 5.12, performance is slightly changed with different cache configurations due to the highly reuse in the scratchpad memory.

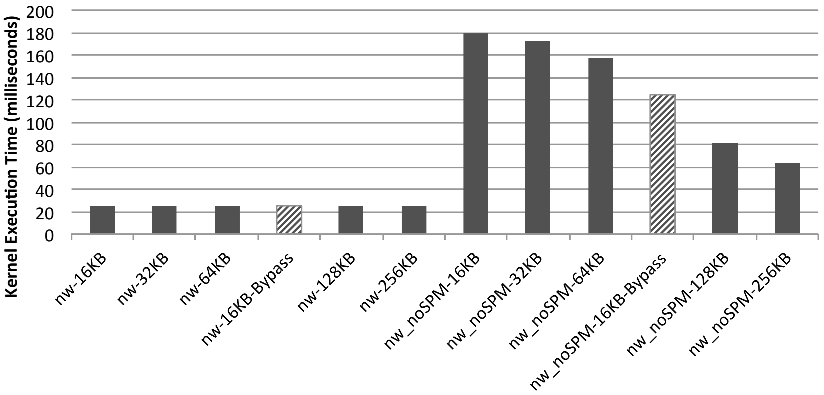

Figure 5.12: Execution time of nw with different configurations

Without using scratchpad memories, *nw-noSPM* takes 7 times longer than the original program. With the help of cache bypassing, the gap is reduced by 30%, which outperforms a 64KB L1 cache. Note that cache bypassing is running with 16KB L1 caches.

This limited study shows that, while the technique currently cannot replace scratchpad memories programmed by expert programmers, it can improve performance in production environments where such programming effort is impractical, as well as programmability. We believe improvements such as our predictor bring GPU programming closer to general purpose programming in terms of programmability while retaining the performance advantage of highly parallel GPUs.

5.5 Summary

In this chapter we have quantified dead blocks in GPU private caches as zero-reuse blocks and proposed a simple but effective GPU cache bypassing technique to reduce unnecessary cache insertion. Adaptive GPU cache bypassing dynamically bypasses zero-reuse blocks to improve performance as well as reducing energy consumption.

Besides dead blocks, memory hierarchies store another kind of waste: data redundancy. To continue exploring waste of memory hierarchies, this dissertation quantifies data redundancy and proposes a practical data redundancy elimination techinique in the next chapter.

6. REDUCING DATA REDUNDANCY IN THE LAST LEVEL CACHES*

Conventional cache design wastes capacity because it stores redundant data. When a memory request is issued, the data fetched from the main memory also is brought into caches for future requests. This data is associated with a tag derived from its physical memory address. Cache blocks with different block addresses may contain identical data. The same chunk of data is duplicated in the cache because the addresses differ. As an example, Figure 6.1 shows the average percentage of duplicated blocks stored in a 2MB last-level cache (LLC) in 18 randomly selected SPEC CPU2006 benchmarks [42]. The ratio of duplication varies with the workload, but there always are duplicated blocks stored in the cache for all the benchmarks. 13 of 18 benchmarks have more than 20% duplicated cache blocks. Among the benchmarks, *hmmer* has the smallest percentage of duplicated blocks (2.7% on average) and *zeusmp* has the largest percentage of duplicated blocks (97.8%). On average, 35.1% of cache blocks are duplicated for all the benchmarks.*

6.1 Reasons of Cache Deduplication

This phenomenon happens mainly because of program behavior and input characteristics. Examples of redundancy-causing program behavior are copying and assignment generating duplicate data stored at different memory locations. Listing 6.1 shows a code snippet of assignment in the SPEC CPU2006 benchmark *xalancbmk*. Elements in the vector *objToStore* are stored in the buffer *serEng*. After running this code, there are two copies of the same data stored in the cache. A similar

*Part of this chapter is reprinted with permission from "Last-level Cache Deduplication" by Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh, 2014. Proceedings of the 28th ACM international conference on Supercomputing, ACM, New York, NY, USA. Copyright [2014] by ACM.

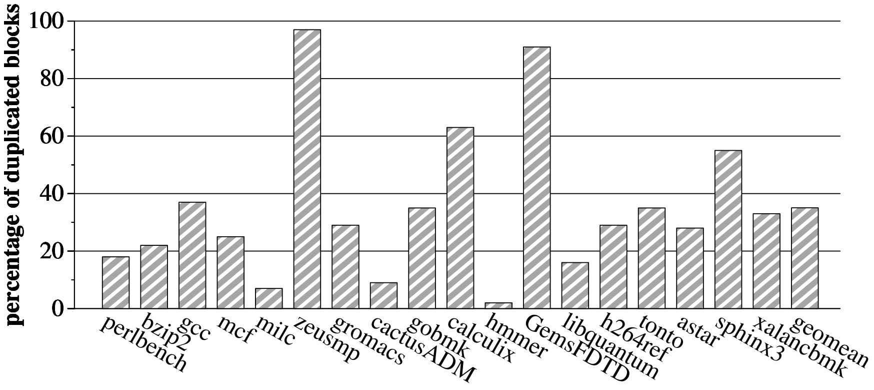

Figure 6.1: Average percentage of duplicated blocks in LLC

phenomenon happens with memory operations like *memcpy()* [1].

```
if (serEng.needToStoreObject(objToStore)) {
    int vectorLength = objToStore->size();
    serEng<<vectorLength;
    for ( int i = 0; i < vectorLength; i++) {
        XercesStep* data = objToStore->elementAt(i);
        serEng<<data;
     }
}
```

Listing 6.1: storeObject() in XTemplateSerializer.cpp

Another source of duplication is program input. For example, the input of the SPEC CPU2006 benchmark *zeusmp* is “a spherical blastwave with radius r=0.2 and located at the origin” [42], which contains perfect symmetry, leading to a significant amount of data similarity (97.8% of cache blocks are duplicated in our experiment). Similar input characteristics exist in benchmark *GemsFDTD*, in which more than

[1] Some compilers and ISAs generate specialized code so that certain copies bypass the cache. For instance, Intel’s C compiler and libraries will use a non-temporal store for *memcpy()* if the size of the data moved is larger than 256KB [38]. However, shorter instances of copying continue to lead to significant cache data duplication.

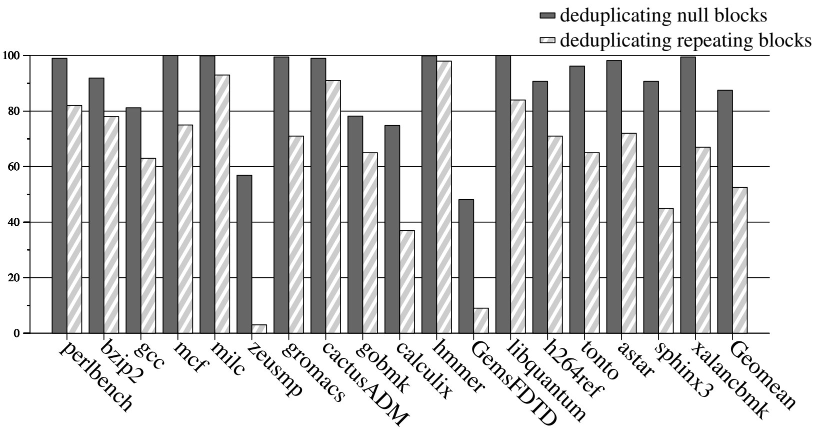

Figure 6.2: Percentage of distinct blocks for null-block deduplication and full-block deduplication

90% of cache blocks are duplicated. Symmetric data is common especially in scientific workloads, causing copious duplication of non-zero values.

Previous cache-compression techniques proposed to compress specific values that cause data duplication [2, 30, 87] such as zero. Based on our experiments, eliminating zero-content (null) blocks can save only 13% of the cache capacity, while eliminating all possible duplication leads to 47.5% of cache blocks removed/invalidated, as shown in Figure 6.2. In other words, almost half of the cache capacity can be saved with data deduplication.

The majority of duplication contains non-zero data values resulting from input and/or computation with a random distribution of the number of copies depending on program behavior. As an example, we take a random execution point of *xalancbmk* to show the nature of duplication degree and duplicated data. At a random execution point, in a 2MB cache, there are 14,931 distinct blocks out of 29,278 of cache blocks (i.e., 51% of blocks are distinct). There are 2,414 chunks of data associated with two tags each, so 16% of blocks are duplicated once. There are 1,157 zero-content blocks.

If only zero-content blocks are compressed, only 4% of total capacity is saved. If all the duplication can be eliminated from the cache, more than 38% of the capacity of the 2MB cache can be saved, which is about three times larger than a modern processor's typical 256KB L2 cache.

6.2 Challenges of Reducing Data Redundancy

Cache compression has been proposed to improve effective cache capacity [2, 3, 19, 68, 69, 115, 110, 40, 87] by compressing redundant value. Storing compressed cache blocks potentially reduces cache misses by increasing effective capacity. However, the processes of compression and decompression significantly increase cache access latency, thus degrading performance. The zero-content augmented cache [30] was proposed to reduce the storage of cache blocks that contain null data. Storing only physical addresses and valid bits of null blocks in an augmented cache saves cache area and improves overall performance. However, the percentage of zero-content blocks is relatively small on average, the performance improvement is also small.

Data deduplication is a specific compression technique to eliminate duplicated copies of repeating data. It has been used widely in disk-based storage systems [27, 111, 43]. With data deduplication, only a single instance of identical data is stored physically. The redundant data is stored as references to the corresponding data in a deduplicated data storage to improve storage utilization. Although commonly used in disk storage and main-memory compression, data deduplication is a challenge in caches with limited overhead due to the following concerns:

How to detect duplication: The first challenge is the way to compare data to detect possible duplication. Duplication can be detected by comparing the analyzed data either with all the stored data or to a specific part of a tree-based data array. Because caches contain a large number of blocks, direct comparison with all blocks is prohibitively expensive. A tree-based structure requires more metadata to maintain the tree while the time complexity is still too high for a large number of nodes. Indexing using a hash function is a fast solution to find the data with which to compare. However, simply using a hash function to index the data array is inefficient because of underutilization of the data array. A practical duplication-detection technique must be fast as well as storage-efficient.

When to detect duplication: The second challenge is the point at which to process duplication detection. Caches play an important role in bridging the performance gap between processors and the main memory, in which access latency is critical to the overall system performance. The process of duplication detection should not affect the cache latency.

Deduplication granularity: Previous work [115, 110, 3, 2, 30, 87] used sub-block level granularity to compress all possible compressible data. Granularity at the sub-block-level may lead to a higher rate of deduplication, but it also causes increased access latency, additional power overhead, and more complex hardware design. Although the effective capacity can be increased more with sub-block-level deduplication, the system performance may be degraded because of the increased access latency. The trade-offs among compression degree and increased cache latency and overhead makes compression granularity another challenge for cache deduplication.

Write hit and replacement of duplicated blocks: The last challenge in cache deduplication design is dealing with write hits and replacement of duplicated blocks. When a store instruction writes duplicate data, the updated block must be allocated a new entry to differentiate from the previous value. When duplicate data is invalidated or evicted from a deduplicated cache, all tags that are associated with this data also should be invalidated. Previous work proposed storing all possible tags in each data entry [78], which is impractical in a cache design due to the limited capacity. An intelligent and low-overhead data management is required in a practical cache deduplication design.

6.3 Deduplicated Last-Level Cache

This dissertation proposes a practical LLC design eliminating duplicated cache blocks, called a deduplicated LLC. To address the challenges cited in the previous section, deduplicated LLC uses augmented hashing to detect duplication, which is fast and makes the most of the utilization of the cache capacity. It uses post-process detection [62] to hide possibly increased cache latency. It uses block-level-deduplication granularity to compare the analyzed block with the data already stored in the cache, regardless of its content, to exploit data duplication fully with limited overhead. For the replacement policy of the duplicated blocks, we propose the distinct-first random replacement (DFRR) policy for efficiency.

6.3.1 Structure

Figure 6.3 shows the structure of a deduplicated LLC. It consists of three decoupled structures: a tag array, a data array, and a hash table. With cache deduplication, the mapping from the data store to the tag store is no longer one-to-one. The structure of the data store is decoupled from that of the tag store. The data array is used only to place distinct data, while the tag array keeps the semantics of cache blocks by storing blocks with tags, pointers to the data array, and other metadata. More than one tag can share a data block. Cache-management techniques (e.g., intelligent replacement policy, increased number of blocks, and so on) are related only to the tag array. With the decoupled structures, changes in the tag array need not affect the design of the data array.

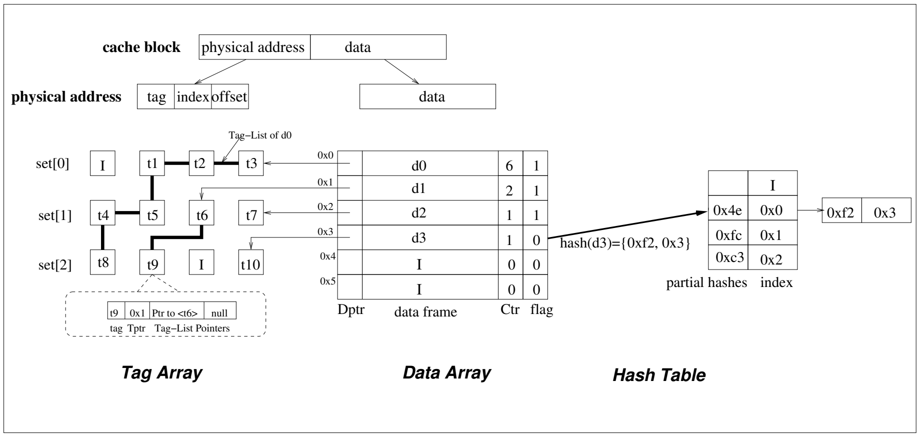

Figure 6.3: Structure of a deduplicated LLC. Blocks t1, t2, t3, t4, t5 and t8 are duplicated blocks, sharing identical data d0; t6 and t9 share data d1; t7 is a distinct block with data d2; and, t10 is inserted as a distinct block and has not been analyzed for deduplication yet.

6.3.1.1 Tag Array

The tag array is a set-associative structure that keeps the semantics of cache blocks. Each entry in the tag array contains the following fields: required metadata of a cache block as in a conventional cache (e.g., tag bits, LRU bits, valid bit, and dirty bit), a reference that indexes the data array, and two references that point to other tag entries that maintain a doubly-linked list of tags all pointing to the same data block. The reference to a data entry, referred to as a *tag-to-data pointer (Tptr)*, identifies a distinct entry in the data array. When there is a tag match, *Tptr* directly indexes the data associated with this cache block. When a tag is inserted in the tag array, it also is inserted into the doubly-linked list of tags of duplicated blocks (if there are any) associated with the corresponding data.

When a tag is replaced from the tag array, it also is deleted from the linked list. With these pointers, all tags stored in the tag array that share identical data are linked. The linked list of tags of duplicated blocks is referred to as the tag-list and the two pointers in each tag entry are referred to as tag-list pointers. When there is a replacement in the data array, all associated tags can be tracked along with the tag-list of the data block and invalidated. The replacement of the data array will be discussed in Section 6.3.2.3; in practice, this process has very low latency. The tag array can be treated as a conventional cache storing only metadata. It uses requested memory addresses to search specific sets for matching tags. When cache misses occur, the tag array uses the regular cache replacement policy (i.e., least-recently used (LRU) to choose replacement candidates rather than replacement in the data array, which uses the DFRR policy).

In our experiments, we use the traditional least-recently-used (LRU) replacement policy in the tag array for fair evaluation. The left-most structure shown in Figure 6.3 gives an example of the tag array in a deduplicated LLC. This tag array is a 4-way set-associative structure, with three sets. As shown at the bottom of the structure, the second (from left to right) tag entry in *set[2]* contains the tag *t9*, the *Tptr* that indexes the corresponding data *d1 - 0x1*. One tag-list pointer to the previous block in the tag-list - *t6* and the other tag-list pointer is set as NULL because there is no next block of *t9*. As drawn in bold in Figure 6.3, Blocks t3, t2, t1, t5, t4, and t8 are in the tag-list of duplicated data *d0*, and t6 and t9 are in their own list. Blocks t7 and t10 are distinct blocks, because there is only one tag in the tag-list of each data block.

6.3.1.2 Data Array

Each entry in the data array contains a data frame, a counter, a pointer, and a one-bit deduplication flag. The counter (referred to as *Ctr*) indicates the number of tags stored in the tag array that share this data. When a tag is inserted into the tag array, the corresponding *Ctr* in the data array is incremented by 1. When a tag is replaced or invalidated from the tag array, the corresponding *Ctr* is decremented by 1. When a *Ctr* becomes zero, the data block can be reused. The pointer (referred to as a data-to-tag pointer (Dptr)) identifies the head of the tag-list. *Dptr*s of invalid entries are used to keep a free list of available data entries. The one-bit deduplication flag indicates whether the current data block has been analyzed for deduplication (discussed in Section 6.3.3). The data array can be treated as a direct-mapped cache, accessed only by *Tptr*s from the corresponding tag entries. The structure shown in the middle of Figure 6.3 gives an example of a data array. There are six entries in the data array; four of them are valid. Data *d0*, located in *0x0*, is shared by six blocks (*Ctr* equals 6), heading with tag *t3* in the tag-list. Data blocks *d2* and *d3* are distinct blocks, linking to only one tag each, *t7* and *t10*, respectively. However, *d3* has not been analyzed for duplication detection yet (i.e., the flag is unset).

6.3.1.3 Hash Table

The third structure in deduplicated LLC is an augmented hash table. This work uses an augmented hash table to implement a two-level look-up to make the most of the cache capacity. The first level of look-up occurs in the hash table indexed by the hashed data, and the second level occurs in the data array redirected by the indices stored in the hash node. To reduce the number of hash collisions, the hash table is implemented as a sequence of small associative arrays representing buckets. Each node in a bucket contains a 16-bit pointer indexing the data array, a 1-bit valid bit, and a 15-bit partial-hash value.

On each duplication detection, the new data are hashed to a hash table entry containing a bucket of nodes as shown in the right-most structure in Figure 6.3. To reduce access to the data array, each node stores a partial hash value as well as the index into the data array. The new data is compared with indexed data only if the partial hash values match. For the hash function, we use five-level exclusive-OR gates using the same technology used for hashing long branch history for high-performance branch predictors [100]. Each level of the exclusive-OR gate halves the number of bits by taking the exclusive-OR of the upper half of the input bits with the lower half of the input bits. Hashing is completed within one cycle assuming a clock period of at least 10 FO4 delays.

Based on our experiments, a small hash table is sufficient to keep the percentage of hash collisions extremely low (less than 1%). However, hash collisions are practically unavoidable when hashing a large set of possible keys (cache data). Hash collision resolution will be discussed in Section 6.3.4.

6.3.2 Operations

A deduplicated cache has different operations on cache hits and cache misses. On a cache access, the tag of the requested block is compared in parallel with all tags in a specific set of the tag array. If the look-up fails, a cache miss has occurred; otherwise, a cache hit has occurred.

6.3.2.1 Cache Miss

On a cache miss, the requested block is brought from the main memory as in a conventional cache. The placement of the cache block then is separated into two parts: placement in the tag array and placement in the data array. The data of the block is placed in an invalid data entry randomly chosen from the free list maintained using the *Dptr*s. The tag of the block is placed in the corresponding set of the tag array indexed using the memory address. The *Tptr* in the tag entry and the *Dptr* in the data entry then are updated to point to each other, and *Ctr* is increased by 1. If there is no invalid entry in the set of the tag array, the regular replacement policy (LRU in our experiments) is used to choose a replacement victim. If there is no invalid entry in the data array, we use DFRR to choose a data replacement victim (details in Section 6.3.2.3).

At this time, the requested cache block is not analyzed for duplication (with the deduplication flag unset). Instead, it is placed in the cache directly with an unset deduplication flag, indicating it has not been processed for deduplication, and without incurring any deduplication latency. The duplication detection to this block will not be launched until next cache miss occurs, as described in Section 6.3.3. The corresponding hash node of the data replacement victim then is invalidated.

6.3.2.2 Cache Hit

A cache hit can be either a read hit or a write hit. In a deduplicated cache, write hits modify the data of blocks, incurring re-hash of the updated data for another duplication detection, while read hits are unrelated to deduplication. Thus, the operations on read hits and write hits are different:

- When there is a read hit in the tag array, the *Tptr* in the matching entry directly indexes the data array to retrieve the requested data. Replacement information then is updated in the tag array. The data array is unchanged.

- When there is a write hit in the tag array, the requested data is indexed by the *Tptr*. If it is a distinct block (*Ctr* equals 1), the data can be modified immediately and the deduplication flag is unset to indicate an unanalyzed block. If it is a duplicated block, instead of modifying the data array directly, an invalid data entry is allocated to place the updated data. In this case, the write hit to a duplicated data is processed similar to a cache miss. Then the dirty bit in the tag entry is updated as well as the replacement information.

6.3.2.3 Distinct-first Random Replacement

This dissertation uses a DFRR policy in data array replacement. To find a replacement candidate, the DFRR policy goes to a random position of the data array and checks if the data is distinct. If it is distinct, the entry is chosen as replacement victim; if not, another random entry is checked. To limit the amount of checking, up to four locations can be checked on each replacement. If there is no distinct block among the checked blocks, the block with the fewest duplicates out of the four entries is replaced. Corresponding tag entries are back-invalidated in the tag array to maintain integrity.

Based on our experiments, on each data replacement, on average 1.004 blocks are checked randomly to find the replacement victim. The intuition behind DFRR is that no invalid data entry means there are too many distinct blocks, so one or two random checks will be enough to find a distinct block to replace. The latency of finding a new data entry can be hidden completely.

6.3.3 An Example of Hash-based Post-Process Deduplication

This dissertation proposes to use hash-based post-process duplication detection to process deduplication fast with limited overhead. Hash-based post-process duplication detection is launched on LLC misses to avoid possible increased latency. The cache block that is under deduplication detection is blocked. Delaying the detection process until the cache is less busy and the processed block has less chance to be accessed (due to locality) helps avoid dynamically increased cache latency. Figure 6.4 gives an example of how it works. In this example, the tag array is a 4-way associative structure with two sets, the data array has three entries, and the hash table has four buckets. Each bucket contains a chain of two nodes. Each valid tag entry contains a *Tptr* pointing to the corresponding data entry. For simple illustration, we do not show the replacement states in the tag array, nor do we show *Dptr*s, *Ctr*s, and deduplication flags in the data array.

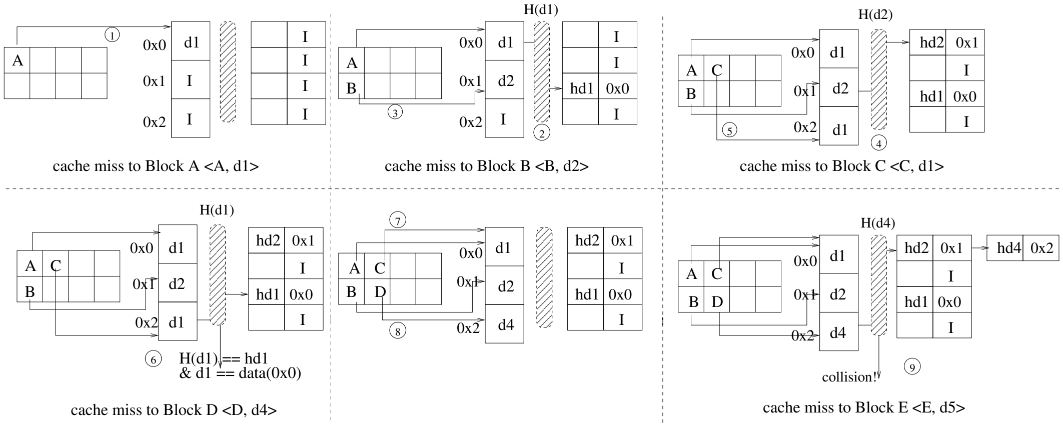
cache miss to Block A <A, d1>
cache miss to Block B <B, d2>
cache miss to Block C <C, d1>
cache miss to Block D <D, d4>
cache miss to Block E <E, d5>
H(d1)
H(d2)
H(d4)
0x0
0x1
0x2
d1
d2
d4
I
hd1
hd2
hd4
H(d1) == hd1
& d1 == data(0x0)
collision!
1
2
3
4
5
6
7
8
9

Figure 6.4: An example of hash-based post-process last-level cache deduplication

On a cache miss to Block A, the requested block is fetched from the main memory. The tag is inserted in the tag array and the data *d1* is inserted in an invalid data entry, as in Step 1. On the next cache miss to Block B, during the memory access time, the previously placed data *d1* of Block A is detected for duplication. The hash value of *d1* indexes a bucket in the hash table (Step 2). Because the bucket is empty, the location of *d1* and its hash value *hd1* are placed in this bucket. After Block B is fetched from the memory, it is filled in the cache (Step 3).

On a cache miss to Block C, the previously placed data *d2* of Block B needs duplication detection. The bucket of *d2* is also empty, so the position of *d2*, *0x1*, and its hash value *hd2* are inserted in the bucket (Step 4). Block C later is filled in the cache by placing the tag in the tag array and inserting the data in an empty data entry at *0x2* (Step 5).

On a cache miss to Block D, the data of Block C (located at *0x2*) hashes to a bucket containing a hash value *hd1* and index *0x0*. Because the hash of the data of Block C equals *hd1*, the data is compared with the data located at *0x0*, resulting in a match (Step 6). Thus, the *Tptr* of Block C is updated to *0x0*, and the data entry in *0x2* is invalidated (Step 7). The *Dptr* of *d1* is updated to point to Block C. After requested Block D is fetched, it is filled in the cache by placing its data in the empty entry at *0x2* (Step 8).

On a cache miss to Block E, the previously placed data *d4* of Block D is analyzed for deduplication. The hash value of *d4* does not equal the one stored in the hash node, so there is no further data comparison. A hash collision incurs. The location of *d4* and its hash value are inserted in the chain of the hashed bucket (Step 9).

6.3.4 Hash Collision Resolution

Hash collisions are unavoidable with a practical hash function. In a deduplicated cache, a hash collision occurs when the hash bucket is full. Thus, a strategy is required for hash collision resolution:

- If there is a distinct block indexed in the current bucket, this block is back-invalidated from the data array and the tag array, respectively. The bucket node then is updated to the location of the colliding data. This procedure can be treated as a replacement in a hash bucket.

- Because of the extremely low probability (lower than 0.1% in our experiments), if data indexed in the current bucket are all duplicated, no replacement occurs in this bucket. The current deduplication procedure just exits and a new detection is launched if there is any unanalyzed data. In this case, we may lose a chance to eliminate a possibly duplicated block. However, it will not cause any extra cache misses to degrade the cache performance because the mapping from the tag to the data is kept one to one.

Based on our experiments, a hash bucket with 16 nodes is sufficient to keep the rate of hash collision as low as 1%. Detailed analysis concerning hashing is described in Section 6.5.5.

6.4 Experimental Methodology

This section outlines the experimental methodology used in this work.

6.4.1 Simulation Environment

We use the MARSSx86 cycle-accurate simulator [85], a full-system simulation of the x86-64 architecture that runs both single-core and multi-core workloads to evaluate the proposed deduplicated LLC. It models an out-of-order 4-wide x86 processor with a 128-entry re-order buffer and coherent caches with MESI protocol as well as on-chip interconnections.

The micro-architectural parameters are consistent with Intel Core i7 processors [72], including a three-level cache hierarchy: L1 I-caches and L1 D-caches, L2 caches, and a shared LLC. The L1 and L2 caches are private to each core. The L1 I-cache and D-cache are 4-way 32KB each and the L2 cache is unified 8-way 256KB. The shared LLC is a unified 16-way 2MB-per-core cache. The default replacement policy for each cache is LRU. Access latencies to the L1 cache, L2 cache, LLC, and main memory are 4, 10, 40, and 250 cycles respectively, in keeping with the methodology of recent cache research work [45, 57, 46, 29]; we show in Section 6.5.6 that our results are not changed significantly with alternate latencies. For the deduplicated LLC, both the number of sets and the associativity of the tag array can be increased to accommodate more blocks. We evaluate both ideas by doubling the number of sets and associativity of the tag array, respectively. The reason to double the size of the tag array is to compare the duplicated LLC with a double-sized conventional

LLC. The actual size of the tag array can be increased arbitrarily to achieve better performance with commensurate power and area consumption. Based on the experiments, the evaluated deduplicated LLC with a double-sized tag array fits in the area of the LLC of the baseline. We show a detailed cost analysis in Section 6.5.3.

The replacement policy in the tag array is LRU, while the replacement policy in the data array is the proposed DFRR.

We also compare our work with two cache-compression techniques: adaptive cache compression [2] and ZCA cache [30]. With adaptive cache compression, the L1 and L2 caches have the same configuration as in a conventional cache hierarchy. Data stored in L1 and L2 caches are uncompressed and only the LLC supports compression. The compressed LLC is a unified 16-way (up to 32-way dynamically) 2MB-per-core set-associative cache with decoupled tag and data stores. Instead of storing a 64-byte data block, the data store is broken into 8-byte segments. An uncompressed 64-byte block is stored as eight 8-byte segments, while a compressed block is compressed into one to seven segments. Data segments are stored continuously in each set with tag order. We conservatively ignore the very high cost of replacement in the contiguous storage variant of the compressed cache.

In our experiments, the access latency of a compressed LLC is constant at 24 cycles. We ignore the decompression latency of 5 cycles to evaluate the cache-deduplication technique better. We also assume that the compression process, occurring on each LLC replacement, can be hidden by the memory-access latency. Thus, the extra compression latency is ignored in our experiments.

With the ZCA cache technique, the L1 and L2 caches have the same configuration as the baseline. The L3 cache is a 2MB-per-core set-associative main cache along with an 8,192-entry, 8-way ZCA cache consuming 156KB of storage overhead. Because accesses to the ZCA cache are in parallel with accesses to the main cache, the access latency is unchanged.

6.4.2 Benchmarks

The benchmarks used in the experiments are selected randomly from the SPEC CPU2006 benchmark suite. We use SimPoint [88] to identify a single one-billion-instruction characteristic interval (i.e., SimPoint) of each benchmark. Each benchmark is compiled for the x86-64 instruction set and run with the first *ref* input provided by the *runspec* command. Benchmarks are categorized into three groups based on the average percentage of duplicated blocks:

- *Deduplication-sensitive benchmarks*: average percentage of duplicated blocks is greater than 50%;
- *Deduplication-friendly benchmarks*: average percentage of duplicated blocks is between 20% and 50%; and,
- *Deduplication-insensitive benchmarks*: average percentage of duplicated blocks is lower than 20%.

Table 6.1 shows the group and the percentage of duplicated blocks of each benchmark as well as the LLC misses per 1,000 instructions (MPKI), instructions per cycle (IPC), and the number of instructions fast-forwarded (FFWD) to reach the interval given by SimPoint in a baseline system. Memory-intensive benchmarks are shown in boldface.

For multi-core workloads, we randomly generate 12 mixes of quad-core workloads from the 18 benchmarks, listed in Table 6.2 with their characteristics of duplication. Each benchmark in a workload runs simultaneously with the others, restarting after one billion instructions, until all of the benchmarks have executed at least two billion instructions.

Group	**Benchmark**	**% Duplicated Blocks**	**MPKI (LRU)**	**IPC (LRU)**	**FFWD**
Dedup-sensitive (S)	**zeusmp**	97.1%	9.05	0.580	405B
	GemsFDTD	90.6%	16.46	0.466	1060B
	calculix	63%	0.04	1.130	4433B
	sphinx3	54.6%	9.00	0.530	3195B
Dedup-friendly (F)	gcc	37.3%	1.38	1.292	64B
	gobmk	34.9%	0.35	1.072	133B
	tonto	34.9%	0.04	1.259	44B
	xalancbmk	33.4%	35.95	0.144	178B
	h264ref	30%	0.09	1.700	8B
	gromacs	28.8%	0.59	1.244	1B
	astar	27.9%	9.7	0.366	185B
	mcf	24.7%	83.54	0.126	370B
	bzip2	22.1%	0.886	1.127	368B
Dedup-insensitive (I)	**perlbench**	18.2%	1.67	0.882	541B
	libquantum	16.1%	24.82	0.162	2666B
	cactusADM	9%	24.7	0.22	81B
	milc	7%	1.01	1.299	272B
	hmmer	2.7%	2.75	0.844	942B

Table 6.1: The 18 SPEC CPU2006 benchmarks with LLC cache misses per 1,000 instructions for LRU, instructions per cycle for LRU in a 2MB cache, and number of instructions fast-forwarded to reach the simpoint (B = billions). Memory-intensive benchmarks in boldface.

6.5 Evaluation

In this section we give performance evaluation and detailed analysis of cache deduplication with respect to capacity, storage, and power overhead, hashing effectiveness, and the cache sensitivity to different sizes of hash table. .

6.5.1 Performance Improvement

In a deduplicated cache, both the number of sets and the associativity of the tag array can be increased to place more cache blocks. In a compressed cache, the number of sets cannot be increased and the associativity is increased dynamically up to twice as large as an uncompressed cache. In a ZCA cache, up to 64MB null blocks can be mapped.

Mixes	Benchmarks
mix1 (FFSF)	gcc, gobmk, zeusmp, xalancbmk
mix2 (ISSF)	milc, sphinx3, zeusmp, gobmk
mix3 (SSSF)	GemsFDTD, zeusmp, calculix, xalancbmk
mix4 (FFSS)	astar, gobmk, calculix, GemsFDTD
mix5 (FISF)	sphinx3, milc, zeusmp, xalancbmk
mix6 (IFSS)	hmmer, gcc, sphinx3, calculix
mix7 (IFFF)	hmmer, gcc, xalancbmk, gromacs
mix8 (FSSF)	gcc, calculix, GemsFDTD, h264ref
mix9 (FFII)	gobmk, gromacs, hmmer, perlbench
mix10 (FIIF)	h264ref, hmmer, libquantum, xalancbmk
mix11 (IISF)	libquantum, hmmer, GemsFDTD, tonto
mix12 (ISFF)	perlbench, zeusmp, mcf, gcc

Table 6.2: 12 mixes of quad-core workload ('F' stands for deduplication-friendly, 'S' for deduplication-sensitive and 'I' for deduplication-insensitive)

We compare the performance of each technique with a double-sized conventional cache as an upper bound (doubled-sets). In our experiments, we show the performance improvement (normalized to an 8MB conventional LLC) of an 8MB compressed LLC, an 8MB deduplicated LLC with doubled number of sets (16,384 sets, 16-way), an 8MB deduplicated LLC with doubled associativity (8,192 sets, 32-way), an 8MB conventional LLC with a 8,192-entry ZCA cache, and a 16MB conventional LLC (16,384 sets, 16-way).

Figure 6.5 shows the LLC cache misses normalized to an 8MB conventional LLC of each technique for quad-core workloads. On average, ZCA cache reduces the LLC misses by 5.5%. Cache compression reduces the LLC misses by 12%. Cache deduplication in a doubled-set LLC reduces average misses by 18.5%. Cache deduplication in a doubled-associativity LLC reduces average misses by 19%. The doubled-size conventional LLC reduces the cache misses by 18.4%.

Reducing cache misses translates into improved performance. Figure 6.6 shows the performance improvement of each technique normalized to an 8MB conventional

LLC. The ZCA cache improves performance by 6.9%. The compressed cache yields an average speed-up of 10.8% compared to the baseline. Cache deduplication in a doubled-set LLC gives an improvement of 15%, and cache deduplication in a doubled-associativity LLC yields a speed-up of 15.2%. The upper-bound 16MB conventional cache delivers an average speed-up of 15.1% compared to the 8MB baseline. A 12MB conventional LLC delivers an 8.7% speed-up, and a 14MB LLC delivers an 8.9% speed-up.

Overall, the deduplicated LLC performs comparably to a double-sized conventional LLC.

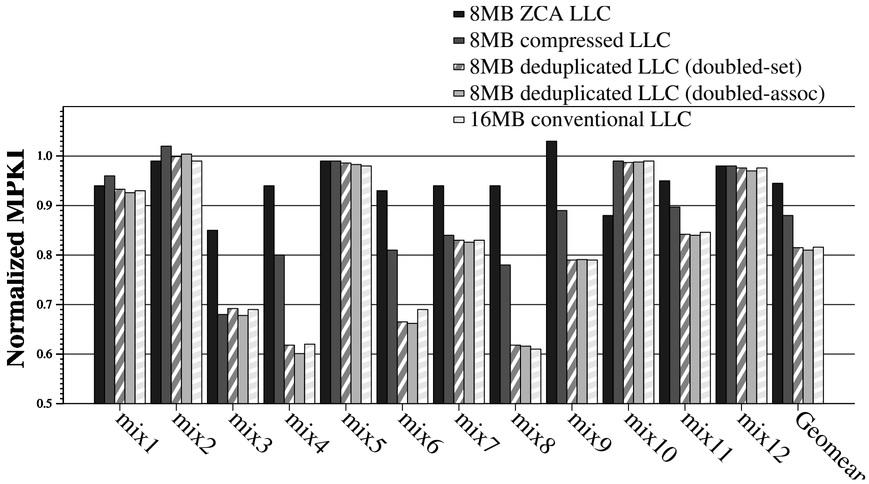

Figure 6.5: Reduction in LLC misses normalized to 8MB conventional LLC

6.5.2 Effective Cache Capacity

Figure 6.7 shows the average amount of duplication in each quad-core workload. On average, each block of data stored in the data array is shared by 2.23 tags. In other words, effective cache capacity is increased by 112% with cache dedupli-

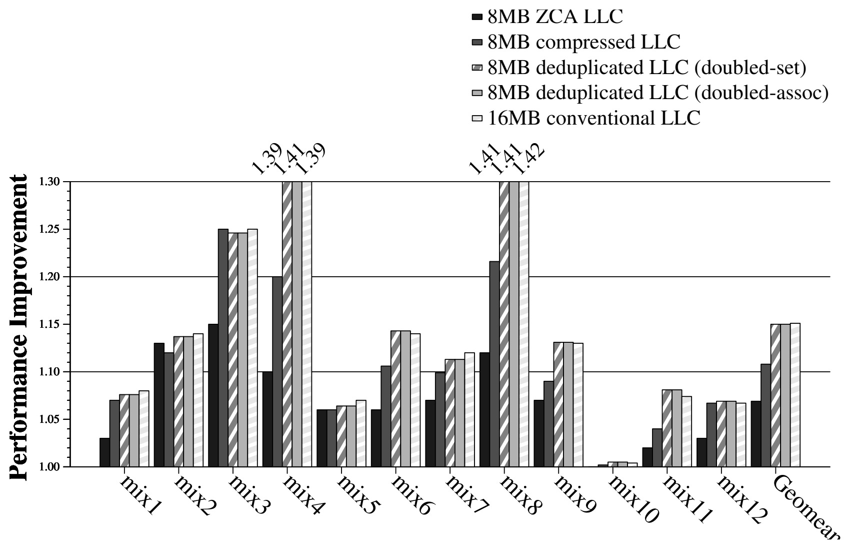

Figure 6.6: Performance Improvement normalized to 8MB conventional LLC

cation. For workloads mix6, mix7, mix9, mix10, and mix11, which all contain the most deduplication-insensitive benchmark *hmmer*, cache deduplication still works by eliminating duplication by about 38%.

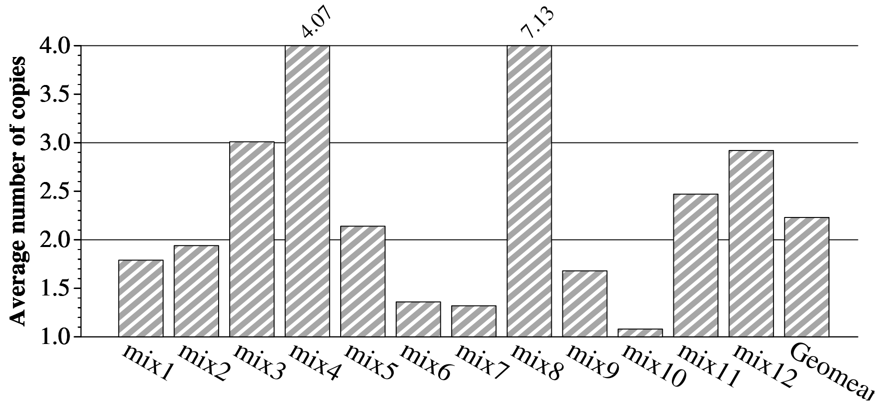

Figure 6.7: Average amount of duplication

6.5.3 Storage Analysis

Although the effective capacity is increased, the physical area is reduced. Table 6.3 shows the detailed storage requirements of both the baseline and the deduplicated LLC in a quad-core CMP. The 8MB deduplicated LLC occupies only 87.8% of the physical area of a conventional 8MB LLC (i.e., it reduces physical area by 12.2% compared to the conventional LLC). The area savings lead to reduced leakage power cost, as shown in Section 6.5.4.

	Conventional LLC	**Deduplicated LLC**
Each tag store entry contains:		
Tag	29 bits	28 bits
Status (valid+dirty+LRU)	6 bits	6 bits
Tptr	-	17 bits
*Rptr*s	-	36 bits
Number of tag entries	131,072	262,144
Total size of tag store	560KB	2784KB
Each data store entry contains:		
Data	512 bits	512 bits
Dptr	-	18 bits
Ctr	-	18 bits
Dedup flag	-	1 bit
Number of data entries	131,072	65,536
Total size of data store	8192KB	4392KB
Additional structure(s):		
Size of hash table	-	8,192
Length of chain	-	16
Size of node	-	32 bits
Total size of hash table	-	512KB
TOTAL SIZE	8,752KB	7,688KB

Table 6.3: Storage cost analysis

6.5.4 Power and Energy

Table 6.4 shows the results of CACTI 6.5 simulations [79] to determine the leakage and dynamic power of the deduplicated LLC compared to the conventional LLC. The tag array is modeled as the tag store of a conventional 16MB set-associative cache. The data array is modeled as a 4MB direct-mapped cache with 37 bits of tags. The hash table is modeled as the data store of a 512KB direct-mapped cache with block size of 4 bytes.

Due to the nature of deduplicated caches, accesses to the LLC are increased while accesses to the main memory are decreased. Based on the experiments, compared to an 8MB conventional cache, the number of accesses to the tag array of the 8MB deduplicated cache is increased by 38% and the number of accesses to the data array is increased by 33%. The number of accesses to the off-chip main memory is decreased by 26% with the deduplicated LLC.

Compared to the energy cost of accessing caches, the energy cost of accessing the off-chip memory is significantly higher. According to the results of previous work [97], the energy consumed to activate and precharge a page and to read a block is $5nJ$ with a row buffer size of 8KB. Thus, as shown in Table 6.5, the average dynamic energy consumption of the deduplicated LLC accesses is 3.3% higher than that of the conventional LLC, while the dynamic energy cost of the memory accesses is reduced by 34.5% with the deduplicated LLC.

6.5.5 Hashing Analysis

In this section we give detailed analysis in regard to the hash function and the hash table used in the experiments.

	Structures	Dynamic Energy per Read Port (nJ)	Dynamic Power per Read Port at max freq (W)	Leakage Power per Bank (W)
Conventional	Tag store	0.0389	0.0605	0.5205
	Data store	1.3148	2.0482	3.0297
	Total	1.3537	2.1087	3.5502
Deduplicated	Tag array	0.1225	0.2564	0.9207
	Data array	0.8793	2.3149	1.8441
	Hash table	0.0234	0.0746	0.0445
	Total	1.0543	2.6534	2.9278

Table 6.4: Dynamic and leakage power of each LLC design

	Structures	Dynamic Energy (J)
Conventional	Tag store	0.0005
	Data store	0.0175
	Memory	0.0222
Deduplicated	Tag array	0.0021
	Data array	0.0156
	Hash table	0.0009
	Memory	0.0165

Table 6.5: Dynamic energy cost of each LLC and main memory

6.5.5.1 Number of Look-ups

Figure 6.8 shows the average number of look-ups in each deduplication process. On each duplication detection, the analyzed data is compared with all the data indexed in the hash bucket until a match occurs or it mismatches with all the data. On average, there are 4.9 look-ups in each duplication detection. The number of look-ups is related to the deduplication latency, described in Section 6.5.6. For workloads such as mix6, mix10, and mix11, the number of look-ups is higher because of the nature of deduplication-insensitive benchmarks: most analyzed data is distinct, causing more look-ups in each duplication detection.

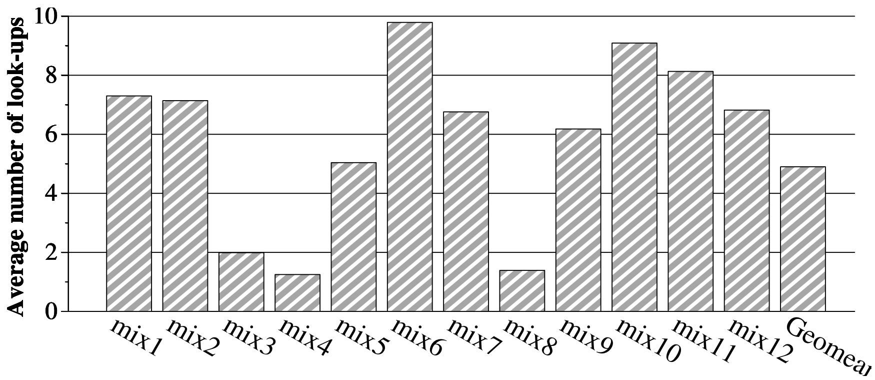

Figure 6.8: Average number of look-ups for data comparison

6.5.5.2 Hash Collisions

With a practical hash algorithm, hash collisions are unavoidable. Figure 6.9 shows the average percentage of hash collisions for each quad-core workload. On average, the percentage of hash collisions is as low as 1%.

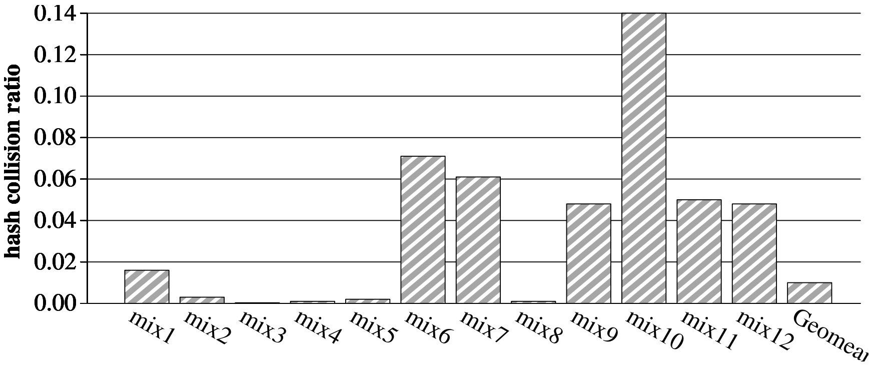

Figure 6.9: Hash collision

6.5.5.3 Hash Table Sensitivity

The size of the hash table in our experiments is 8,192 buckets with 16 nodes per bucket, leading to a 512KB storage overhead. Reducing the size of the hash table to 4,096 buckets leads to an increased number of look-ups of 5.7 on average, and the percentage of hash collision is increased to 1.3%. The performance improvement is barely changed; the difference is 0.1%. We performed experiments to measure the behavior of our technique in the presence of context switching. Our results indicate that this technique yields at least the same improvement compared to the baseline configuration in the presence of OS context-switching among multiple applications.

6.5.6 Process Latency

The deduplication latency is hidden by the memory access. On each LLC miss, the duplication detection is launched to analyze a previously stored cache block. The analyzed data is hashed to a bucket and compared with all the data indexed in that bucket until a match occurs or mismatches with all the indexed data. Data comparison is completed well within one cycle using a simple circuit, assuming 12 FO4 delays [3]. Thus, the duplication detection takes *(number of look-ups* × *(1 + data comparison))* cycles on average, which is less than 10 cycles and thus totally hidden by the memory-access latency of 250 cycles.

In adaptive cache compression, as claimed in [2, 3], compression latency is 3 cycles and decompression latency is 5 cycles. The extra access latency is on the critical path to degrade performance. Even if the compression latency of 3 cycles can be hidden by the memory-access latency, the decompression latency is unavoidable.

7. CONCLUSION

Let us recall the thesis statement from the introduction:

The performance and efficiency of modern processors can be improved by reducing waste in memory hierarchies.

This dissertation exploits two types of waste in three different types of memory hierarchies: dead blocks in inclusive cache hierarchies, dead blocks in GPU private caches, and data redundancy in last level caches. This dissertation proposes several techniques to eliminate waste from diverse memory hierarchies effectively with limited overhead. In this section, we review the contribution of these techniques.

7.1 Reducing Waste Caused By Dead Blocks in Inclusive Cache Hierarchy

In this work, we propose temporal-based multi-level correlating (TMC) cache replacement for inclusive cache hierarchies. It chooses blocks that will not be re-referenced in all cache levels as LLC replacement candidates. Replacing these blocks with useful ones as early as possible significantly helps improve cache efficiency and overall performance.

This dissertation proposes to sample LLC cache access patterns and correlate them with temporal locality knowledge passively acquired from higher level caches to choose temporal-aware LLC replacement candidates, which provides high performance improvement while consuming minimal overhead.

This dissertation shows that inclusive caches with TMC is more efficient than not only the inclusive baseline but also the “upper-bound”– non-inclusive caches. Inclusive caches with TMC perform as well as enhanced non-inclusive caches while keeping the advantage of simplifying cache coherence of CMPs.

7.2 Reducing Waste Caused By Dead Blocks in GPU Private Caches

In this work, we propose a simple but effective GPU cache management technique. It prevents streaming one-time-use values from being needlessly inserted into the cache with high accuracy and minimal area overhead.

This dissertation demonstrates performance gains and energy savings when using our bypass predictor for a GPU L1 data cache.

This dissertation studies limitations of current GPU cache design and the effects of a bypass predictor as they relate to using scratchpad memories. In particular, this dissertation compares an application that uses scratchpad memories to a rewritten version of the same application that does not require the complexity of manual memory layout in the context of our optimization.

7.3 Reducing Wasted Caused By Data Redundancy in Last-Level Caches

In this work, we find that widespread duplication exists in caches and quantify the cache duplication effect in 18 SPEC CPU2006 benchmarks.

This dissertation proposes a unified cache-deduplication technique to improve cache performance with increased effective cache capacity. By exploiting block-level value redundancy, cache deduplication significantly increases cache effectiveness with limited area and power consumption.

This dissertation proposes a novel LLC design with cache deduplication. Compared to a conventional LLC, the deduplicated LLC uses similar chip area and power consumption while performing comparably to a double-sized conventional LLC.

REFERENCES

[1] Jaume Abella, Antonio González, Xavier Vera, and Michael FP O'Boyle. Iatac: a smart predictor to turn-off l2 cache lines. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):55–77, 2005.

[2] A.R. Alameldeen and D.A. Wood. Adaptive cache compression for high-performance processors. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 212–223. IEEE, 2004.

[3] A.R. Alameldeen and D.A. Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. *Dept. of Computer Sciences, University of Wisconsin-Madison, Tech. Rep*, 2004.

[4] AMD. AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience. 2010.

[5] AMD. AMD Radeon HD 7970 Graphics. 2011.

[6] AMD. Amd fx processors, 2012.

[7] AMD. AMD Graphics Cores Next (GCN) Architecture. 2012.

[8] AMD. Accelerated Parallel Processing (APP) SDK. 2013.

[9] Federico Angiolini, Francesco Menichelli, Alberto Ferrero, Luca Benini, and Mauro Olivieri. A post-compiler approach to scratchpad mapping of code. In *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267. ACM, 2004.

[10] J.L. Baer and W.H. Wang. *On the inclusion properties for multi-level cache hierarchies*, volume 16. IEEE Computer Society Press, 1988.

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

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[13] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F.T. Chong. Multi-execution: multicore caching for data-similar executions. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 164–173. ACM, 2009.

[14] Douglas C Burger, James R Goodman, and Alain Kagi. *The declining effectiveness of dynamic caching for general-purpose microprocessors*. University of Wisconsin-Madison, Computer Sciences Department, 1995.

[15] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

[16] J. Casazza. Intel core i7-800 processor series and the intel core i5-700 processor series based on intel microarchitecture (nehalem). *White paper, Intel Corp*, 2009.

[17] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[18] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of parallel and distributed computing*, 68(10):1370–1380, 2008.

[19] D. Chen, E. Peserico, and L. Rudolph. A dynamically partitionable compressed cache. 2003.

[20] X. Chen, Y. Yang, G. Gopalakrishnan, and C.T. Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, pages 81–88. IEEE, 2006.

[21] D. Cheriton, A. Firoozshahian, A. Solomatnikov, J.P. Stevenson, and O. Azizi. Hicamp: architectural support for efficient concurrency-safe shared structured data access. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 287–300. ACM, 2012.

[22] Chipworks. Inside the ASUS AMD 7970 graphics card - TSMC 28nm! 2012.

[23] Chipworks. A Look at Sony's Playstation 4 Core Processor. 2013.

[24] An chow Lai. Dead-block prediction and dead-block correlating prefetchers. In *In Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, 2001.

[25] An chow Lai and Babak Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 139–148, 2000.

[26] An chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *In Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, 2001.

[27] T.E. Denehy and W.W. Hsu. Duplicate management for reference data. *Research Report RJ10305, IBM*, 2003.

[28] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970.

[29] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):35, 2012.

[30] J. Dusser, T. Piquet, and A. Seznec. Zero-content augmented caches. In *Proceedings of the 23rd international conference on Supercomputing*, pages 46–55. ACM, 2009.

[31] J. Dusser and A. Seznec. Decoupled zero-compressed memory. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 77–86. ACM, 2011.

[32] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. OpenCL and the 13 dwarfs: a work in progress. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 291–294, New York, NY, USA, 2012. ACM.

[33] HSA Foundation. Deeper Look Into HSAIL And It's Runtime. 2012.

[34] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of*

the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pages 407–420. IEEE Computer Society, 2007.

[35] Rahul V Garde, Samantika Subramaniam, and Gabriel H Loh. Deconstructing the inefficacy of global cache replacement policies. 2008.

[36] J. Gaur, M. Chaudhuri, and S. Subramoney. Bypass and insertion algorithms for exclusive last-level caches. In *Proceeding of the 38th annual international symposium on Computer architecture*, pages 81–92. ACM, 2011.

[37] Antonio González, Carlos Aliagas, and Mateo Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 9th international conference on Supercomputing*, pages 338–347. ACM, 1995.

[38] Ronald W Green. Memory movement and initialization: Optimization and control. April 4th, 2013.

[39] OpenCL Working Group. The OpenCL specification, version 1.2, revision 16, 2011.

[40] E.G. Hallnor and S.K. Reinhardt. A unified compressed memory hierarchy. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 201–212. IEEE, 2005.

[41] Blake A Hechtman, Shuai Che, Derek R Hower, Yingying Tian, Bradford M Beckmann, Mark D Hill, Steven K Reinhardt, and David A Wood. Quickrelease: a throughput oriented approach to release consistency on gpus. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[42] J.L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[43] B. Hong, D. Plantenberg, D.D.E. Long, and M. Sivan-Zimet. Duplicate data elimination in a san file system. In *Proceedings of the 21st Symposium on Mass Storage Systems (MSS04), Goddard, MD*, 2004.

[44] Zhigang Hu, Stefanos Kaxiras, and Margaret Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 209–220, Washington, DC, USA, 2002. IEEE Computer Society.

[45] A. Jaleel, E. Borch, M. Bhandaru, SC Steely, and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 151–162. IEEE, 2010.

[46] A. Jaleel, H.H. Najaf-Abadi, S. Subramaniam, S.C. Steely, and J. Emer. Cruise: cache replacement and utility-aware scheduling. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 249–260. ACM, 2012.

[47] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 151–162, Washington, DC, USA, 2010. IEEE Computer Society.

[48] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[49] Jonas Jalminger and P Stenstrom. A novel approach to cache block reuse predictions. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 294–302. IEEE, 2003.

[50] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *20th International Symposium on High Performance Computer Architecture (HPCA-20)*, 2014.

[51] Teresa L Johnson, Daniel A Connors, Matthew C Merten, and W-MW Hwu. Run-time cache bypassing. *Computers, IEEE Transactions on*, 48(12):1338–1354, 1999.

[52] Hadi Jooybar, Wilson WL Fung, Mike O'Connor, Joseph Devietti, and Tor M Aamodt. GPUDet: a deterministic GPU architecture. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 1–12. ACM, 2013.

[53] N.P. Jouppi and S.J.E. Wilton. Tradeoffs in two-level on-chip caching. *ACM SIGARCH Computer Architecture News*, 22(2):34–45, 1994.

[54] Mahmut Kandemir, J Ramanujam, Mary Jane Irwin, Narayanan Vijaykrishnan, Ismail Kadayif, and Amisha Parikh. A compiler-based approach for dynamically managing scratch-pad memories in embedded systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(2):243–260, 2004.

[55] Samira M. Khan, Daniel A. Jiménez, Doug Burger, and Babak Falsafi. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 489–500, New York, NY, USA, 2010. ACM.

[56] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 175–186, Washington, DC, USA, 2010. IEEE Computer Society.

[57] S.M. Khan, Y. Tian, and D.A. Jimenez. Sampling dead block prediction for last-level caches. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 175–186. IEEE, 2010.

[58] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement and bypassing algorithms. *IEEE Trans. Comput.*, 57:433–447, April 2008.

[59] J. Kieffer. *Data Compression.* Wiley Online Library, 1971.

[60] M. Kleanthous and Y. Sazeides. Catch: A mechanism for dynamically detecting cache-content-duplication and its application to instruction caches. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1426–1431. ACM, 2008.

[61] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 226–236, New York, NY, USA, 2007. ACM.

[62] P. Koutoupis. Data deduplication with linux. *Linux Journal*, 2011(207):7, 2011.

[63] N.A. Kurd, S. Bhamidipati, C. Mozak, J.L. Miller, T.M. Wilson, M. Nemani, and M. Chowdhury. Westmere: A family of 32nm ia processors. In *Solid-*

State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International, pages 96–97. IEEE, 2010.

[64] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, pages 144–154, New York, NY, USA, 2001. ACM.

[65] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.

[66] Alvin R. Lebeck and David A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.

[67] Jaekyu Lee and Hyesoon Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[68] J.S. Lee, W.K. Hong, and S.D. Kim. Design and evaluation of a selective compressed memory system. In *Computer Design, 1999.(ICCD'99) International Conference on*, pages 184–191. IEEE, 1999.

[69] J.S. Lee, W.K. Hong, and S.D. Kim. Adaptive methods to minimize decompression overhead for compressed on-chip caches. *International journal of computers & applications*, 25(2):98–105, 2003.

[70] Leonidas. AMD R1000/Tahiti Die-Shot. 2012.

[71] Jacob Leverich, Hideho Arakida, Alex Solomatnikov, Amin Firoozshahian, Mark Horowitz, and Christos Kozyrakis. Comparing memory systems for chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 358–368. ACM, 2007.

[72] D. Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 2009.

[73] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. *SIGPLAN Not.*, 31(9):138–147, September 1996.

[74] 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 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 222–233, Washington, DC, USA, 2008. IEEE Computer Society.

[75] Vineeth Mekkat, Anup Holey, Pen-Chung Yew, and Antonia Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 225–234. IEEE Press, 2013.

[76] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. *SIGARCH Comput. Archit. News*, 25(2):292–303, May 1997.

[77] Carlos Molina, Carles Aliagas, Montse García, Antonio Gonzàlez, and Jordi Tubella. Non redundant data cache. In *Proceedings of the 2003 international*

symposium on Low power electronics and design, ISLPED '03, pages 274–277, New York, NY, USA, 2003. ACM.

[78] C.B. Morrey III and D. Grunwald. Content-based block caching. In *Proceedings of 23rd IEEE Conference on Mass Storage Systems and Technologies*, 2006.

[79] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. Cacti 6.0: A tool to model large caches. *Research report hpl-2009-85, HP Laboratories*, 2009.

[80] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, pages 15–23. IEEE Computer Society Press, 1995.

[81] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. 2009.

[82] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. 2012.

[83] Nvidia. Tuning CUDA Applications for Kepler. 2013.

[84] CUDA Nvidia. Programming guide, 2008.

[85] A. Patel, F. Afram, S. Chen, and K. Ghose. Marssx86: A full system simulator for x86 cpus. In *Proc. DAC*, 2011.

[86] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[87] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Todd C Mowry, Phillip B Gibbons, and Michael A Kozuch. Base-delta-immediate compression: A practical data compression mechanism for on-chip caches. In *Proceedings of the*

21st ACM International Conference on Parallel Architectures and Compilation Techniques (PACT), 2012.

[88] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '03, pages 318–319, New York, NY, USA, 2003. ACM.

[89] M.K. Qureshi, D. Thompson, and Y.N. Patt. The v-way cache: Demand-based associativity via global replacement. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 544–555. IEEE, 2005.

[90] Moinuddin K Qureshi, Aamer Jaleel, Yale N Patt, Simon C Steely, and Joel Emer. Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 381–391. ACM, 2007.

[91] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[92] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 72–83. IEEE Computer Society, 2012.

[93] D. Salomon. *Data compression: the complete reference.* Springer-Verlag New York Inc, 2004.

[94] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 187–198. IEEE, 2010.

[95] K. Sayood. *Introduction to data compression.* Morgan Kaufmann, 2000.

[96] Resit Sendag and Peng fei Chuang. Address correlation: Exceeding the limits of locality. *IEEE Comput. Archit. Lett.*, 1(1):13–16, January 2002.

[97] O. Seongil, S. Choo, and J.H. Ahn. Exploring energy-efficient dram array organizations. In *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pages 1–4. IEEE, 2011.

[98] A. Seznec. A case for two-way skewed-associative caches. In *ACM SIGARCH Computer Architecture News*, volume 21, pages 169–178. ACM, 1993.

[99] André Seznec. A case for two-way skewed-associative caches, 1993.

[100] Andre Seznec. Analysis of the o-geometric history length branch predictor. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, pages 394–405. IEEE, 2005.

[101] Stephen Somogyi, Thomas F. Wenisch, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Memory coherence activity prediction in commercial workloads. In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, WMPI '04, pages 37–45, New York, NY, USA, 2004. ACM.

[102] J. Storer. Data compression methods and theory. 1988.

[103] T. Welch. Technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[104] Wen-mei W. Hwu. *GPU Computing Gems Emerald Edition*. Access Online via Elsevier, 2011.

[105] D.F. Wendel, R. Kalla, J. Warnock, R. Cargnoni, S.G. Chu, J.G. Clabes, D. Dreps, D. Hrusecky, J. Friedrich, S. Islam, et al. Power7, a highly parallel, scalable multi-core high end server processor. *Solid-State Circuits, IEEE Journal of*, 46(1):145–161, 2011.

[106] Maurice V Wilkes. Slave memories and dynamic storage allocation. *Electronic Computers, IEEE Transactions on*, (2):270–271, 1965.

[107] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[108] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Jr. Simon C. Steely, and Joel Emer. Ship: signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 430–441, New York, NY, USA, 2011. ACM.

[109] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C Steely Jr, and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 430–441. ACM, 2011.

[110] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 258–265. ACM, 2000.

[111] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan. Debar: A scalable high-performance de-duplication storage system for backup and archiving. In

Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pages 1–12. IEEE, 2010.

[112] M. Zahran, K. Albayraktaroglu, and M. Franklin. Non-inclusion property in multi-level caches revisited. *INTERNATIONAL JOURNAL OF COMPUTERS AND THEIR APPLICATIONS*, 14(2):99, 2007.

[113] Mohamed Zahran. Cache replacement policy revisited. In *Proceedings of the 6th Workshop on Duplicating, Deconstructing, and Debunking*. Citeseer, 2007.

[114] Mohamed Zahran and Sally A. McKee. Global management of cache hierarchies. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 131–140, New York, NY, USA, 2010. ACM.

[115] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *ACM SIGOPS Operating Systems Review*, volume 34, pages 150–159. ACM, 2000.

[116] Jishen Zhao, Guangyu Sun, Gabriel H. Loh, and Yuan Xie. Energy-efficient GPU design with reconfigurable in-package graphics memory. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, ISLPED '12, pages 403–408, New York, NY, USA, 2012. ACM.

[117] Ying Zheng, B. T. Davis, and M. Jordan. Performance evaluation of exclusive cache hierarchies. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–96, Washington, DC, USA, 2004. IEEE Computer Society.

[118] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, 1977.