NTNU
Norwegian University of
Science and Technology

Evaluating Shared Last Level Cache Partitioning Algorithms

Thomas Alexander aan de
Wiel

Embedded Computing Systems
Submission date: June 2017
Supervisor: Magnus Jahre, IDI

Norwegian University of Science and Technology
Department of Computer Science

Problem Description

Chip Multiprocessors (CMPs) or multi-core architectures are becoming increasingly popular, both in industry and academia. CMPs often share on-chip cache space between cores. When the CMP is used to run multiprogrammed workloads, different processes compete for cache space. Severe competition can lead to considerable performance degradation, and researchers have proposed many techniques that aim to alleviate this problem. The performance of a management technique depends heavily on other architectural components and the benchmarks used in the evaluation. Thus, it is difficult to meaningfully compare techniques without implementing them in the same simulation framework.

The first task is to implement Vantage in the simulation framework currently in use at NTNU. This framework already supports TADIP, DRRIP, UCP, PIPP and PriSM in addition to conventional LRU-based replacement. Second, the student should carry out extensive simulations to establish the performance of the different techniques across a range of architectural configurations. If time permits, the student should identify the root cause of the observed performance differences.

Abstract

Over the past few decades, the development of Dynamic Random-Access Memory (DRAM) has mainly focused on increasing capacity and lowering costs. However, microprocessor development has experienced enormous improvements in latency. This has led to an increasing memory latency-gap, that unaddressed can lead to significant underutilization of available microprocessor resources. To bridge this gap, memory hierarchies including several levels of cache memories have been introduced. Chip Multiprocessors (CMPs) or multi-core architectures commonly share the Last Level Cache (LLC). Sharing allows for destructive interference, as several cores can start to compete for cache space. With CMPs becoming commonplace and as their core count increases, scalable algorithms that partition the LLC among the cores of a CMP are becoming increasingly important.

This thesis describes the implementation of the zcache and the cache partitioning algorithm Vantage in a simulation framework based on Sniper, a parallel multi-core simulator. We utilize this simulation framework to establish the performance improvements of Vantage and the cache partitioning algorithms Thread-Aware Dynamic Insertion Policy (TADIP), Dynamic Re-Reference Interval Prediction (DRRIP), Promotion/Insertion Pseudo Partitioning (PIPP), Utility-Based Cache Partitioning (UCP) and Probabilistic Shared Cache Management (PriSM) over a range of architectural configurations, with the conventional Least Recently Used (LRU) algorithm as baseline. Moreover, we identify several root causes that lead to the observed performance differences. We find that Vantage, by using the highly associative zcache, attains the highest performance improvements and is the most scalable cache partitioning algorithm in our evaluation. The scalability and performance of cache partitioning algorithms utilizing the conventional set-associative cache are mainly limited by the restricted associativity that the set-associative cache provides. We further find that although individual improvements in the System Throughput (STP) can reach up to approximately 20% in our evaluation, the overall impact of cache partitioning is minor, improving the STP and the Harmonic Mean of Speedups (HMS) by a maximum of 3% with respect to LRU.

Acknowledgment

The author would like to thank Associate Professor Magnus Jahre for his guidance, feedback and all the interesting discussions during the course of this thesis.

Contents

List of Figures

List of Tables

Acronyms

ATD Auxiliary Tag Directory. 23–25, 29, 31, 34, 59, 73, 82

BBV Basic Bit Vector. 55

BFS Breadth-First Search. 16–18, 44, 49, 50, 80

BIP Bimodal Insertion Policy. 23–25, 27, 65, 74, 75

BRRIP Bimodal Re-Reference Interval Prediction. 27, 65, 74, 75

CDF Cumulative Distribution Function. 20, 21

CMP Chip Multiprocessor. ii, 1–3, 10, 22, 24, 29, 47, 52–54, 56, 59, 62–65, 73, 82, 84, 85, 87

CPI Cyles per Instruction. 53, 54

DIP Dynamic Insertion Policy. iv, 22–24, 27

DRAM Dynamic Random-Access Memory. ii, 2, 18, 53, 80, 84, 85

DRRIP Dynamic Re-Reference Interval Prediction. ii, iv, v, viii, 4, 26, 27, 59, 63–65, 72, 74, 75, 80, 82, 83, 86, 87

DSS Dynamic Set-Sampling. 24, 29, 59, 60

FS Futility Scaling. 4

HMS Harmonic Mean of Speedups. ii, 54, 59, 62–64, 67–72, 74, 80, 82–86

ILP Instruction Level Parallelism. 1

IPC Instructions Per Cycle. 54, 56, 59, 75

LIP LRU Insertion Policy. 23–25, 33, 74

LLC Last Level Cache. ii, viii, 3, 18, 22, 30, 31, 44–50, 52, 53, 55, 56, 59, 60, 63, 73, 75, 77, 78, 80, 83–85

LRU Least Recently Used. ii, iv, v, 3, 5, 10–14, 21–23, 26, 29, 31–34, 39–41, 53, 58, 59, 62–72, 74, 75, 77, 80–84, 86, 87

LUT Lookup Table. 40–42

MESI Modified-Exclusive-Shared-Invalid. 53

MPKI Misses-Per-Kilo-Instructions. 56, 65, 74

MRU Most Recently Used. 11, 13, 23, 32, 33

MSB Most-significant Bit. 23

MSHR Miss Status Holding Register. 53

MTD Main Tag Directory. 7–10, 23, 24, 34, 59

PIPP Promotion/Insertion Pseudo Partitioning. ii, 4, 5, 28, 32, 33, 59, 60, 63–65, 67–71, 82, 83, 86

PriSM Probabilistic Shared Cache Management. ii, v, viii, 4, 5, 28, 33, 34, 58–60, 63, 64, 66, 67, 69, 71–73, 76–78, 80, 82, 83, 86, 87

QoS Quality of Service. 22

RRIP Re-Reference Interval Prediction. 26, 64, 75

RRPV Re-Reference Prediction Value. 26, 27, 59

SRRIP Static Re-Reference Interval Prediction. 26, 27, 74, 75

STP System Throughput. ii, 54, 59, 62–72, 74, 77, 80, 82–86

TADIP Thread-Aware Dynamic Insertion Policy. ii, iv, v, viii, 4, 24, 25, 27, 59, 63–65, 72, 74, 75, 80, 82, 83, 86, 87

UCP Utility-Based Cache Partitioning. ii, v, 4, 5, 28, 29, 31, 32, 58–60, 63–65, 67–73, 76–78, 80, 82, 83, 86

UMON Utility Monitor. 29, 42, 59, 60, 80

1 | Introduction

This chapter starts by providing the necessary context for this thesis. This then leads to a motivation for the performed research. We proceed with a list of requirements that need to be fulfilled. This is followed by the contributions of this thesis. An outline of this thesis concludes this chapter.

1.1 Context

To overcome the challenges that the computer industry has faced and is facing in its continuous search for improved performance, several important developments in both microprocessor and memory design have taken place. In this section we will give an overview of these developments, including how they have affected each other.

1.1.1 Microprocessor Development

Since the early 1970's and three decades onward, microprocessors have experienced a three orders of magnitude increase in performance [5]. This rapid growth in performance has been enabled by technology scaling of the transistors used in microprocessors. In 1965, Gordon Moore made the observation that the transistor density of chips doubles approximately every two years. Shortly after, in the early 1970's, Robert N. Dennard laid out several rules for scaling down transistors, known as Dennard Scaling [5]. Dennard Scaling implies that as transistor dimensions are scaled by a factor $1/\kappa < 1$, the power density stays constant [4, 12] and the frequency of operation of the transistors increases by a factor of κ. In accordance with Moore's law, transistors were scaled down with a scaling factor $\kappa \approx 1/\sqrt{2}$, doubling the transistor density. Consequently, this meant that around every two years, the speed of microprocessors could be increased by a factor of approximately $\sqrt{2} \approx 1.4$ without an increase in power consumption [5].

Furthermore, with increasing transistor densities, more and more transistors could be fit onto a single chip. This left space for several core microarchitecture techniques exploiting the parallelism present in the software running on the processors (Instruction Level Parallelism (ILP)) to increase performance even more [25].

However, since the early 2000's, single-core processor performance has hit the *power wall*: performance has stagnated due to limited power budgets. One of the main rea-

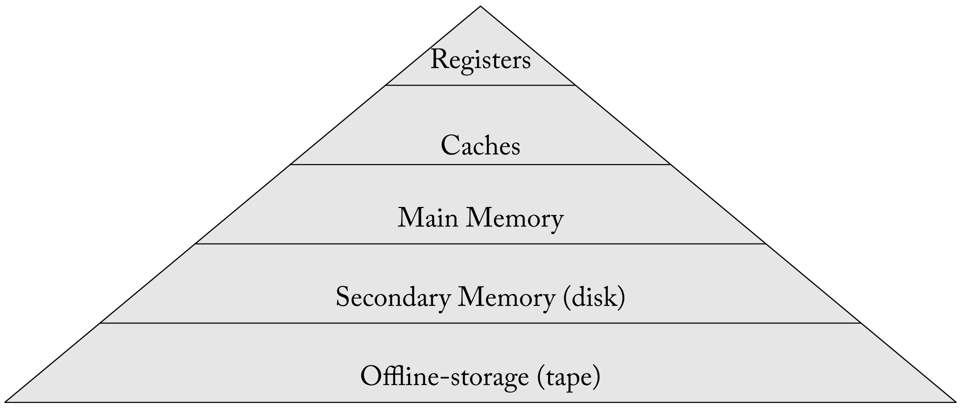


Figure 1.1: Example of a Memory Hierarchy

sons is that Dennard Scaling has failed. This has led to an increasing power usage of microprocessors as their clock-frequency was increased to improve performance. Furthermore, advanced core microarchitecture techniques used to improve single-core performance are power-inefficient [23]: they lead to minor performance improvements at the cost of significant increases in power usage.

To keep benefiting from increasing transistor densities, a shift from single- to multi-core processors or Chip Multiprocessors (CMPs) has taken place. Where single-core processors take advantage of parallelism by exploiting ILP using advanced core microarchitecture techniques, CMPs exploit Thread-Level Parallelism (TLP) to take advantage of parallelism. This allows the cores to be simpler, as no additional hardware for extracting ILP is required [23]. Furthermore, by using multiple cores the clock frequency of each core can be decreased with respect to a single-core processor, while still obtaining better performance [30]. Reduction of the clock frequency leads to, an often linear, decrease in supply voltage [25]. Since the power usage is proportional to the square of the supply voltage, this can result in a significant power reduction. CMPs can therefore achieve better performance than single-core processors while consuming less power.

1.1.2 Bridging the DRAM and Microprocessor Performance Gap: Cache Memories

The development of memory, especially that of Dynamic Random-Access Memory (DRAM) that usually serves as main memory, has taken a different approach compared to that of microprocessors. DRAM development has continually focused on increasing capacity and lowering costs, but improvements in latency have not been as big as those for microprocessors. This has caused DRAM to form a bottleneck, as it is no longer able to supply data to the processor at the rate it is being requested. To address this bottleneck, cache memories have been incorporated in memory hierarchies. An example of such a memory hierarchy is depicted in Figure 1.1. Memory hierarchies aim to emulate the view of a big fast memory. The closer to the processor, the smaller the memory is.

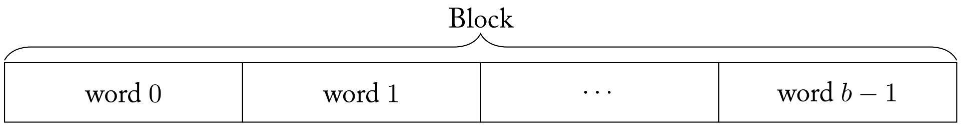


Figure 1.2: A Cache Block Containing b Words

Smaller memories have a higher cost-per-bit, but are able to operate at higher speeds than their larger counterparts. The latency of the larger memories below can be (partially) hidden by exploiting the principle of locality of memory references. This locality can be *temporal*, meaning that the accessed data is likely to be accessed in the near future. An example of temporal locality are *loops* over the same data. By buffering this data from main memory into the small cache memory, the next time this data is referenced by the processor, it can be accessed quickly. Moreover, memory references can exhibit *spatial locality*: when the processor accesses a word of memory, it is likely that other words in the vicinity of the requested word will be referenced. Practical examples in software include arrays and structures. By caching not only the requested word, but a *block* of memory, consisting out of several words (see also Figure 1.2), serialization of the latency of memory references is prevented.

Effective caching is characterized by a high *hit-ratio*, the ratio of finding the requested blocks in the cache (*cache hits*), divided by the total amount of requests. An important factor affecting the hit-ratio of a cache is the cache-size [1]. With an increasing cache-size, the hit-ratio increases, as more data can fit in the cache. However, enlarging a cache results in larger access times, limiting its usefulness. With the introduction of multi-core architectures, needs on bandwidth have increased, thereby making the use of large caches appealing as they reduce the need on bandwidth to main memory [1]. To emulate a large and fast cache, it is useful to make use of a two level cache hierarchy [39, 1]. The higher level cache is relatively small, but provides low access times. The lower level cache is larger and provides good hit ratios. This idea is not limited to a two level hierarchy and contemporary consumer CMPs typically employ a three-level cache hierarchy. In the following, we will refer to (the smallest) cache at the top of the cache hierarchy as the highest-level or L1-cache. (Larger) caches further down the hierarchy will be referred to as lower level caches (L2, L3 and so on). The cache at the bottom of the hierarchy is the largest and is commonly referred to as the Last Level Cache (LLC).

Another factor affecting the hit-ratio is the *associativity* of the cache. Hill and Smith [16] define associativity as the number of places where a block can be resident in the cache. When associativity is limited, blocks can start to compete for the same place in the cache. This can lead to an increase in *cache misses*, requests to blocks that are not present in the cache. Cache misses caused by limited associativity are commonly referred to as *conflict misses*. Fully associative caches, where a block can be placed anywhere in the cache, completely eliminate conflict misses. However, they are impractical due to their large associated hardware overhead. On the other hand, direct mapped caches, where a block can be placed at only one location, have a relatively low hardware-cost, but increase conflict misses considerably. The set-associative cache forms a middle-ground

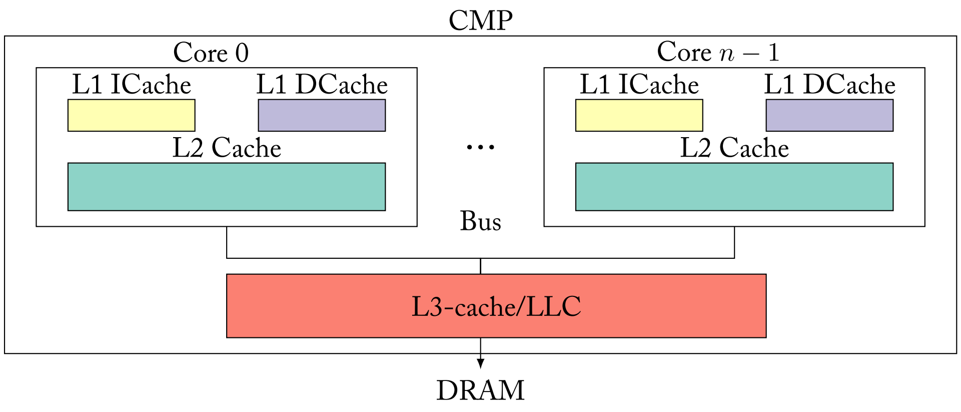


Figure 1.3: n-core CMP with a Shared LLC

between these two cache designs, allowing a block to be mapped to several, but not all, places in the cache. Set-associative caches have become the mainstream cache-design in contemporary CMPs.

1.1.3 Last Level Cache Partitioning

To increase the utilization of resources, the LLC is commonly shared on a CMP. An example of such a n-core CMP with 3-levels of caching and a shared LLC is shown in Figure 1.3. Sharing of the LLC allows for interference when different applications are running concurrently on the cores of the CMP. This interference can be destructive, leading to a decrease in performance.

Commonplace in CMPs are set-associative LLC's managed by the Least Recently Used (LRU) algorithm. However, the LRU algorithm implicitly allocates cache space to the cores based on their access frequencies, which can degrade overall performance severely. It is therefore desirable that certain quota's are set and enforced such as to optimize performance. To this end, several cache partitioning algorithms based on the use of a set-associative cache have been introduced.

Recently, the cache partitioning algorithms *Vantage* and *Futility Scaling (FS)* have been introduced [33, 41] to address the issues of cache partitioning algorithms that stem from the use of set-associative caches. Both Vantage and FS are cache partitioning techniques based on statistical properties provided by highly-associative, but unconventional, caches such as the zcache.

1.2 Motivation

A sheer volume of cache partitioning algorithms is available, yet individual algorithms are commonly only compared to a few competing algorithms. Moreover, these comparisons often use different simulators and different setups, making it difficult or even impossible to (fairly) compare the performance of available cache partitioning algorithms. To provide a more thorough comparison, a simulation framework has been developed at NTNU [24] including the cache partitioning algorithms Utility-Based Cache Partitioning (UCP), Thread-Aware Dynamic Insertion Policy (TADIP), Dynamic Re-Reference Interval Prediction (DRRIP), Promotion/Insertion Pseudo Partitioning (PIPP) and Probabilistic Shared Cache Management (PriSM). However, all these algorithms are based on the conventional set-associative cache. With the introduction of the cache partitioning algorithms Vantage and FS it is desirable to implement one of these algorithms, such as to evaluate the possible advantage that comes with the use of the highly-associative caches that Vantage and FS are based on.

1.3 Requirements

Based on the problem statement and problem description for this thesis, we extract the following mandatory requirements:

(R_1) Give an overview of the cache partitioning algorithms already present in the simulation framework at NTNU (UCP, TADIP, DRRIP, PIPP and PriSM), as well as Vantage.

(R_2) Implement Vantage in the simulation framework.

(R_3) Compare the performance of the cache partitioning algorithms from Requirement (R_1) by evaluating them over a range of architectural configurations.

Furthermore, we extract one optional requirement:

(O_1) Analyze the performance of the evaluated algorithms and identify the root causes of the observed performance differences.

1.4 Contributions

The main contributions of this thesis are:

- An overview of conventional caches and the zcache, supplemented by an overview of the cache partitioning algorithms present in the simulation framework at NTNU and of Vantage (Requirement (R_1)).

- A parallel software implementation of both the zcache and Vantage in the simulation framework at NTNU (Requirement (R_2)).

- An extensive performance evaluation of the cache partitioning algorithms considered (Requirement (R_3)).

- An implementation of PriSM-UCP in the simulation framework and the identification of the root causes of the observed performance differences through several case studies(Requirement (O_1)).

1.5 Overview

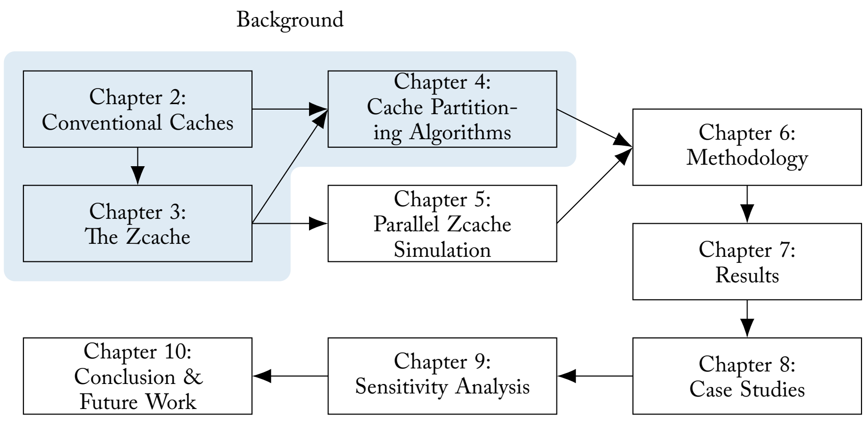

Background
Chapter 2: Conventional Caches
Chapter 3: The Zcache
Chapter 4: Cache Partition-ing Algorithms
Chapter 5: Parallel Zcache Simulation
Chapter 6: Methodology
Chapter 7: Results
Chapter 8: Case Studies
Chapter 9: Sensitivity Analysis
Chapter 10: Conclusion & Future Work

Figure 1.4: Thesis Outline

The necessary background of this study is too extensive and varied to fit in one chapter. It has therefore been divided into several chapters. To give the reader an overview, of the interdependencies and the general flow between the chapters that follow, we have depicted those in Figure 1.4.

Chapters 2 to 4 together form the required background for this work. Chapter 2 will start with an overview of the operation of conventional caches. This serves as the required background for Chapter 3, that continues to explore caches, but focuses on the unconventional zcache. Together these chapters form the basis to discuss all cache partitioning algorithms present in our simulation framework and Vantage in Chapter 4. Furthermore, Chapter 3 lays out the foundation for Chapter 5, in which a parallel software implementation of the zcache is described. Chapter 4 together with Chapter 5 support Chapter 6, in which the performance evaluation of the researched cache partitioning algorithms is described. This chapter is followed by Chapter 7, that outlines the results of the main experiment that has been performed to evaluate the researched cache partitioning algorithms. Chapter 8 furthers with several case-studies explaining the root causes of observed performance differences. Chapter 9 follows with experiments regarding the sensitivity-analysis of our framework. At last, this document is concluded in Chapter 10.

2 | Conventional Caches

Caches are relatively small and fast memories used to exploit temporal and spatial locality to hide the latency of bigger, but slower memories. This chapter will introduce the background of how conventional caches are operated.

2.1 Data Mapping

The natural unit of memory from the point of view of a processor core is a *word*. Usually, a word corresponds to the size of an integer or the size of a general purpose register of the corresponding processor [40]. As already indicated in Section 1.1.2, caches operate on *blocks* of words to exploit spatial locality. When the processor requests a word at the word address A_w, the cache needs to be searched for the block with block address A_b containing this word. For a block-size of b words, the corresponding block address A_b is given by:

$$A_b = \left\lfloor \frac{A_w}{b} \right\rfloor \tag{2.1}$$

and the block-offset o_b, at which the word resides in a block, is given by:

$$o_b = \mod(A_w, b). \tag{2.2}$$

Where the block with block address A_b resides in the cache, is determined by the *mapping* from main to cache memory. Since caches can only contain a subset of the blocks in main memory, this mapping is an *one-to-many* mapping. This means that several blocks from main memory may map to the same location, also called a *cache line*, in the cache. Therefore, a mapping does not suffice to determine whether a certain block of data is present in the cache. A part of the block address, the *tag*, is required to distinguish between the blocks mapping to the same cache line. A cache can therefore be regarded as a collection of blocks storing the actual data complemented by a directory of tags to associate an address with each block. In what follows, we will refer to the directory of tags as the Main Tag Directory (MTD). Furthermore, in literature on cache memories, it is common to find the terms cache line and cache block used interchangeably and in the following we continue to do so.

An example illustrating the introduced terminology can be found in Figure 2.1, which depicts a cache containing n blocks that in turn each contain b words.

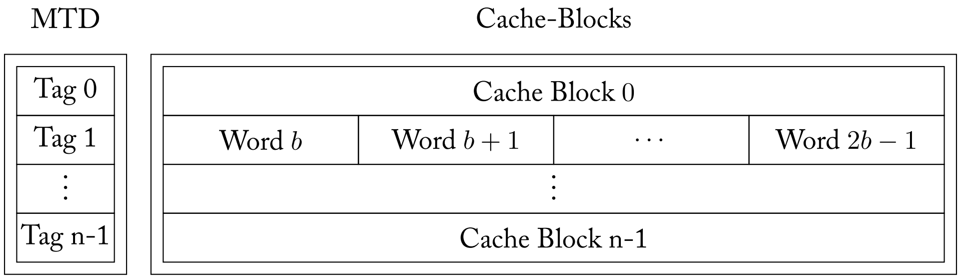


Figure 2.1: A General Cache Architecture

2.1.1 Direct-Mapped Cache

In a direct-mapped cache, a block maps to only *one* location in the cache. To explain the mapping in a direct-mapped cache, we start with the word address A_w, as issued by a processor core to the cache. In Section 2.1 we already subdivided the word address into two sub-addresses: the block-offset o_b and the block address A_b. To explain the operation of a direct-mapped cache, we now subdivide A_b into two sub-addresses: the cache line index i and the tag t. The cache line index i is obtained through a mapping function based on the modulo operator [40] and is given by:

$$i = \mod(A_b, n), \tag{2.3}$$

with n the number of cache lines in the cache. To distinguish between several blocks mapping to the same cache line, we define the tag t as:

$$t = \left\lfloor \frac{A_b}{n} \right\rfloor. \tag{2.4}$$

When the index i of a cache line is known, the tag t as given by Equation (2.4) completely specifies the block address:

$$A_b = i + n \cdot t.$$

The tag t is therefore sufficient to distinguish between several blocks that map to the same cache line. Furthermore, since o_b gives the offset of the requested word in the corresponding block (see Equation (2.1)), the tuple (t, i, o_b) completely specifies the corresponding word address:

$$A_w = (i + n \cdot t) \cdot b + o_b.$$

So far, we have been as general as possible and n and b need not be a power of two. However, when n and b are indeed powers of two, the complexity of the hardware that calculates the mapping decreases considerably, as the modulo operators involved can be re-written in terms of the binary and operator:

$$\mod(a, 2^n) = \text{and}(a, 2^n - 1). \tag{2.5}$$

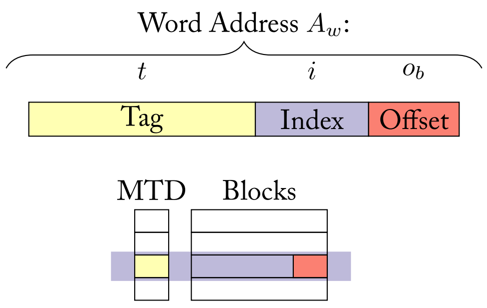


Figure 2.2: A Direct-Mapped Cache

Instead of using a full divisor, it therefore suffices to use a basic and-gate. Furthermore the tag-calculation (Eq. (2.4)) can be performed using a bit-shift instead a full integer division. Cases where n or b are not powers of two, are rare in commercial caches. In the following we will assume the number of cache lines and the number of words in a block to be a power of two, unless stated otherwise. An example illustrating the introduced terminology is shown in Figure 2.2.

2.1.2 Set-Associative & Fully-Associative Caches

Set-associative and fully associative caches differ from the direct-mapped cache in that they allow a block to be inserted at multiple cache lines in the cache instead of one. In a set-associative cache, a block can map to a *limited* amount of cache lines in the cache. Those cache lines together form a *set*. The memory of a set-associative cache is divided into several of those sets. When a set contains n cache lines, we refer to such a cache as a n-way set-associative cache. Since in a fully associative cache a block can map to any cache line in the cache, it can be seen as a set-associative cache with *one* set and the amount of ways equal to the total amount of cache lines in the cache. Therefore, it suffices to explain the operation of a set-associative cache.

When a set-associative cache is used, the word address A_w as issued to the cache by a processor core, can be divided into three sub-addresses: $A_w = (t, s, o_b)$. As in the case with the direct-mapped cache (see Section 2.1.1), t represents the tag and o_b represents the offset of the word in the corresponding block. However, unlike the direct-mapped cache, s does not give the *cache line*, but the *set* of the block that the requested word is contained in:

$$s = \quad \mod (A_b, n), \tag{2.6}$$

with n the number of *sets*.

An example of a 4-way set-associative cache is shown in Figure 2.3. In this example, a set consists of 4 cache blocks spread out over 4 *ways*. Sets can therefore be regarded as "rows" in the set-associative cache, whereas the ways represent the "columns". In this example, each way contains 4 blocks.

Although fully-associative caches can completely eliminate conflict misses, set-associative caches are preferred in hardware implementations. In a fully associative cache the tags

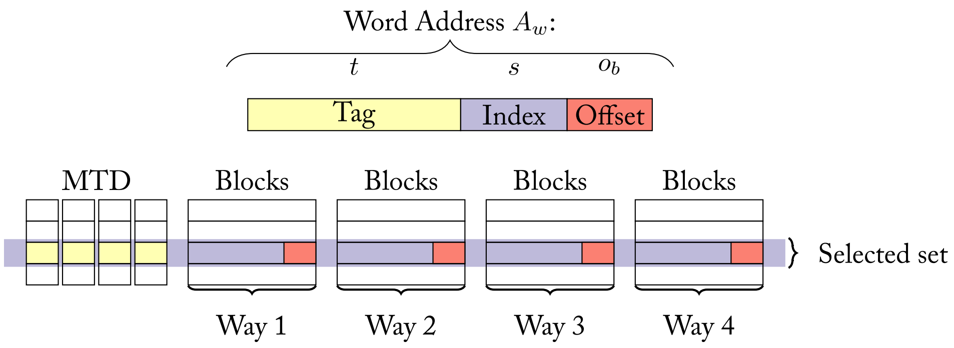


Figure 2.3: A 4-way Set-Associative Cache

of *all* cache lines need to be compared upon a search for a block in the cache, whereas this is limited to n tags in a n-way set associative cache. The hardware overhead associated with the tag comparators have made the set-associative cache a popular compromise between the direct mapped cache and the fully associative cache and is the most common cache found in CMP's.

2.2 Conventional Replacement Algorithms

When a cache miss occurs, the requested block is not present in the cache. To load it into the cache, it is necessary that there there is place available to do so. In a direct-mapped cache this means that if the cache line the block maps to, is already occupied by another block, then this block needs be *evicted*, removing it from the cache. However, in a set-associative cache and fully associative cache the missing block can be inserted at several different cache lines. Therefore, a *replacement algorithm* is required to determine which block will be evicted.

Belady's MIN algorithm is an optimal replacement algorithm to do so [3], yet impractical to implement. Belady's MIN selects the block that will be used the furthest in future for eviction. Knowledge of future cache accesses is therefore required when a cache miss occurs. Usually, it is not known what applications will utilize the cache and such knowledge is not available.

The LRU replacement algorithm, or variants thereof, forms a commonly used alternative [18] and is the conventional replacement algorithm. In this section we will describe the LRU replacement algorithm in more detail. This is continued by a classification of memory access patterns and their influence on LRU.

2.2.1 LRU

Least Recently Used (LRU) selects the least recently used block for eviction. To do so, it ranks cache lines based on their recency, which is represented by the *LRU-chain*. The block at the head of this chain has been accessed most recently, whereas the block at

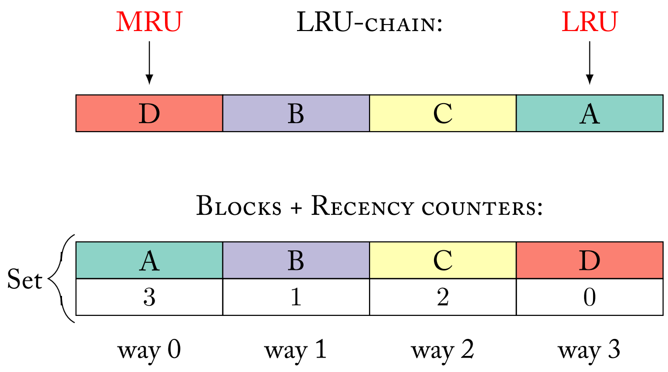


Figure 2.4: A LRU Chain & LRU Bits

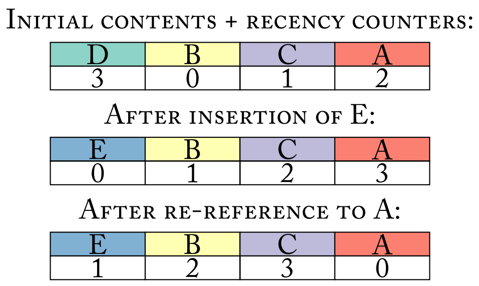


Figure 2.5: LRU Managed Cache Set with 4 Ways

the tail has been accessed least recently.

As a block maps to only one set in the cache, only cache lines within that set are are considered for eviction. Therefore, LRU can rank all cache lines on a per set-basis. The recency ranking is kept by storing a *recency counter* per cache line. This terminology is depicted in Figure 2.4 for a 4-way set associative cache with the Most Recently Used (MRU) and LRU positions marked. The LRU position is associated with the highest recency counter, whereas the MRU position is associated with the lowest recency counter.

After a cache line has been evicted, the missing block is inserted and its recency counter is set to 0, indicating that this is the most-recently used cache line. Consequently, all other recency counters are incremented to enforce a strict ordering. Upon a cache hit, the recency counters of all cache lines that have a lower value than that of the recency counter of the referenced cache line are incremented and the referenced cache line is moved to the MRU position. The LRU replacement algorithm, as described, is illustrated in Figure 2.5.

A set-associative cache managed by the LRU replacement algorithm possesses the *stack property* [28]. Let A and B denote two set-associative caches with the same number of sets, n, but varying amount of ways, a and b respectively with $a \leq b$. The stack property states that when a cache hit to a block occurs in A, a cache hit to the same block is guaranteed in B.

In this particular case, all memory accesses that are performed on a certain set in cache A are exactly the same as in the corresponding set in cache B (see Equation (2.6)). If a hit occurs to a block h in cache A, but misses in cache B, this would require h in B to have been evicted. This implies that at least b consecutive references to blocks other than h must have occurred to this set in B. Since A and B experience the same memory accesses per set, this means that also in cache A at least b consecutive references have been made to a block other than h in the corresponding set. Consequently h must have been evicted from A, as $a \leq b$. This contradicts the possibility of cache hit to h in A, but a miss in B. The stack property does not hold for set-associative caches with varying associativity *and* varying ways: in this situation the memory references are spread out differently over the sets.

2.2.2 Memory Access Patterns and Their Effects on LRU

Having described the LRU replacement algorithm, we now turn our attention to a classification of memory access patterns. For non-optimal replacement algorithms, certain memory access patterns can result in sub-optimal performance. In this section we will describe the effect of four types of memory access patterns on LRU. We follow the four-case distinction as in [18] and classify an memory access pattern as:

1. **Recency-Friendly** when the access pattern exhibits significant amounts of temporal locality. An example of such an access pattern is [18]:

$$(a_1, a_2, \ldots, a_{k-1}, a_k, a_k, a_{k-1}, \ldots, a_2, a_1)^N . \tag{2.7}$$

   Since Belady's MIN prescribes that one should evict the entry that is to be used furthest-in-future, LRU is optimal on this access pattern for any k.

2. **Cache-Trashing** when the access pattern accesses an amount of memory, called the *working set*, that is bigger than the cache in such a fashion that no re-use of data in the cache is possible. An example of such an access pattern is

$$(a_1, a_2, \ldots, a_k)^N \tag{2.8}$$

   with k bigger than the amount of lines the cache. In this case LRU will have evicted $a_i, i \in \{1, 2, \ldots, k\}$ before the following re-reference[1]. This effect is called *trashing*.

3. **Streaming** when the access patterns exhibits no or marginal reuse of data. An example of such an access pattern is:

$$(a_1, a_2, \ldots, a_k) \qquad k = \infty. \tag{2.9}$$

   When no re-use of data is present, caching does not help to improve performance, regardless of the replacement algorithm.

[1]For simplicity reasons we assume LRU to rank the cache lines globally, as is the case in a fully-associative cache. In a set-associative cache a_i is likely, but does not necessarily need to have been evicted. This depends on the specific addresses in the access pattern.

4. **Mixed** when the access patterns has characteristics of two or more of the aforementioned access patterns. An example of such a pattern is a recency-friendly pattern containing *scans*: bursts of data that is referenced only once or in the distant future [2, 18]. An example of a pattern with scans is given by:

$$(a_1, a_2, \ldots, a_k)\,(b_1, b_2, \ldots, b_m)\,(a_1, a_2, \ldots, a_k)^N\,(b_1, b_2, \ldots, b_m)\,, \tag{2.10}$$

with $N \gg 1$. When $m + k$ is bigger than the amount of cache lines in the cache, the scan $(b_1, b_2, \ldots, b_m)$ starts to remove the recency-friendly data $(a_1, a_2, \ldots, a_k)$ from the cache. Furthermore, LRU places all data from the scan at the MRU position: b_1 would evict a_1 from the cache, b_2 would evict a_2 from the cache and so on. Belady's MIN would evict a_k to make place for b_1, but would evict b_1 to make place for b_2 and so on. Therefore, Belady's MIN preserves more of the recency-friendly data in the cache than LRU.

3 | The Zcache

Recently the zcache has been introduced by Sanchez and Kozyrakis [32]. Zcaches are based on the skew-associative cache. Both the skew-associative cache and the zcache are highly associative, but unconventional cache designs [33]. They take a different approach to increasing associativity than the conventional set-associative cache.

This chapter starts with a description of the skew-associative cache, which forms a stepping stone to describing the operation of the more complex zcache. We conclude this chapter with a description of a LRU-approximation for skew-associative caches and zcaches.

3.1 The Skew-Associative Cache

The skew-associative cache has been introduced by Seznec and Bodin [36]. The difference between a n-way skew-associative cache and a n-way set-associative cache is found in the mapping of cache lines: in a skew-associative cache, a different mapping function can be used per way, whereas the set-associative cache uses *one* mapping function for all ways.

To explain the mapping in a skew-associative cache, we start with the with word address A_w corresponding to the word requested by a core. A_w can be subdivided into two sub-addresses:

$$A_w = (A_b, A_0).$$

A_b denotes the block address of the block that contains the requested word. A_0 gives the word offset in the corresponding the cache block. A skew-associative cache applies a hash function $\mathcal{H}_i$ to A_b to obtain the cache line A maps to for every way $i \in \{0, \ldots, n-1\}$. Since the whole block address is used by the hashing functions, in general A_b also functions as the tag. The mapping in a 3-way skew-associative cache is illustrated in Figure 3.1.

Seznec and Bodin [36][35] have shown that 2-way and 4-way skew-associative caches can outperform 4 and 16-way set associative caches, respectively. Skew-associative caches therefore behave as set-associative cache designs with higher way-counts.

Since a skew-associative cache uses a mapping function per way, it can be seen as a generalization of the set-associative cache: if the hash-functions are chosen to be the same for all ways, set-like behavior is obtained. However, in general, when two

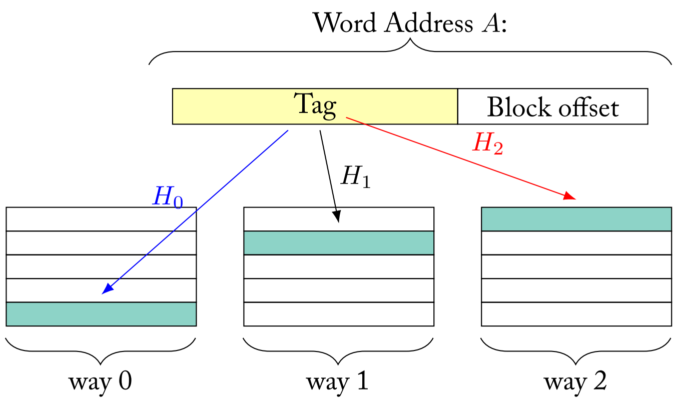


Figure 3.1: Example of a 3-way Skew-Associative Cache

addresses A and B map to the same cache line in way i, they do not necessarily map to the same cache line in way $j \neq i$. This makes it impossible to group cache blocks in sets: the concept of a set is unknown to skew-associative caches [32].

3.2 Multi-Level Skewing: The Zcache

The mapping in a zcache functions similarly to a skew-associative cache: the zcache applies a hash function $\mathcal{H}_i$ to the block address A_b to find the cache line in way i that the block maps to [32]. When a n-way skew-associative cache needs to evict a block, only n blocks, as given by the mapping, are considered for eviction. In a n-way zcache the amount of replacement candidates can *exceed* n.

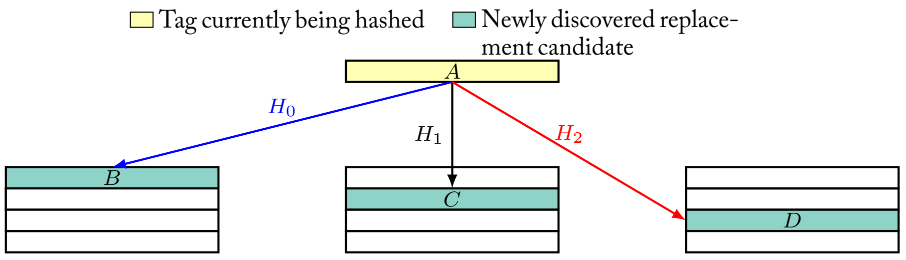
When a zcache offers $m > n$ replacement candidates, not all replacement candidates can reside at the cache lines as given by the mapping. Therefore, inserting the missing block at a cache line that has been made available by eviction does not ensure it will be found when the cache is searched. To resolve this issue, the zcache performs a *relocation* process after a cache line has been evicted. This eventually results in an available cache line at one of the locations given by the mapping functions.

3.2.1 The Replacement Process

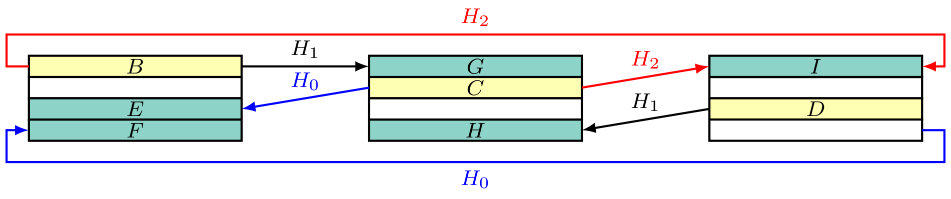
We will now consider the replacement process of the zcache in more detail. This process can be divided in the process of finding the replacement candidates and the process of performing relocation.

Finding Replacement Candidates

As in a skew-associative cache, the block address A_b of a requested block functions as the tag. When a cache miss occurs and a block with tag A is to be inserted, a n-way zcache hashes A with the hash function of every way, obtaining n replacement


Tag currently being hashed
Newly discovered replacement candidate
A
H_0
H_1
H_2
B
C
D

(a) First-Level Replacement Candidates Found by a Zcache



(b) Second-Level Replacement Candidates Found by a Zcache

Figure 3.2: Finding the Replacement Candidates in a Zcache - An example depicting a 3-way zcache.

candidates. These cache lines form the *first-level* replacement candidates. This stage is depicted in Figure 3.2a, where B, C, D form the first-level replacement candidates.

Whereas a skew-associative cache only performs one level of hashing, the zcache performs $L \geq 1$ levels of hashing. In level $n > 1$, instead of A, the tags of the replacement candidates found in level $n-1$ are hashed. Each replacement candidate from level $n-1$ is hashed with the hash functions $j \neq i$, with i the way the replacement candidate resides in. This results in $n-1$ replacement candidates per hashed replacement candidate. A second level of hashing is illustrated in Figure 3.2b. We observe that the tag B and C hash to the same cache line in the third way, leading to a *repeat*. Therefore, only 5 out of the 6 replacement candidates found in the second level are new. The occurrence of such repeats is very rare in caches with hundreds of blocks per way, and can therefore be ignored for LLCs [32].

The hashing process as described, hashes the replacement candidates in a Breadth-First Search (BFS)-like fashion, but this is not necessarily a restriction [32]. However, the BFS can be cheaply implemented in hardware and is therefore the used by Sanchez and Kozyrakis [32]. In the following we will restrict ourselves to the BFS and refer to the entire process of finding replacement candidates as the *BFS-walk*.

In the case of a BFS-walk, the amount of replacement candidates R, ignoring possible repeats, equals:

$$R = n \times \sum_{i=0}^{L-1} (n-1)^i \tag{3.1}$$

with L the amount of levels of the BFS and n the amount of ways of the cache. For

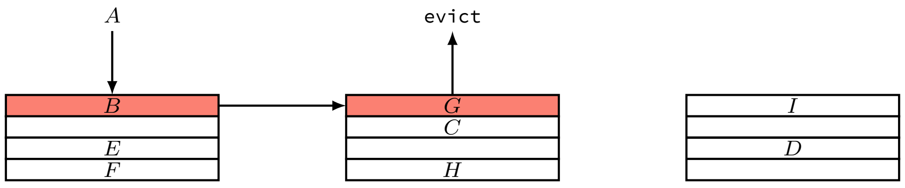


Figure 3.3: Path $\mathcal{P}$ to a Selected Replacement Candidate as Found by a Zcache

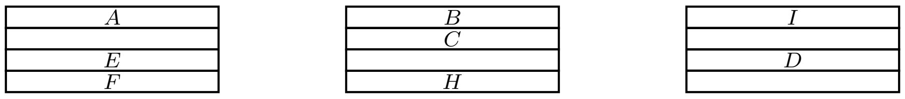


Figure 3.4: Contents of a Zcache after Relocation & Insertion

$L = 3$ and $n = 4$, this results in 52 replacement candidates. This is a substantial increase over the 4 replacement candidates a normal set-associative or skew-associative cache with 4 ways would provide.

In the following the notation Zx/y will refer to a zcache having x ways and offering y replacement candidates. Note that this notation implicitly defines L. Z4/52 for example implies that $L = 3$.

Relocating Replacement Candidates

Upon a cache miss, the zcache selects a replacement candidate r for eviction. If r is a first-level replacement candidate, no additional work is required. As the first-level replacement candidates correspond to the cache lines that the missing block maps to, the missing block can be directly inserted. However, if r is *not* a first-level replacement candidate, it is necessary that a cache line corresponding to one of the first-level candidates becomes available. To make such a cache line available, the zcache backtracks the path $\mathcal{P}$ that was taken from the root of the BFS-walk to the selected replacement candidate.

We continue our example from Figure 3.2 and depict $\mathcal{P}$ in Figure 3.3 where the replacement candidate G is considered for eviction and $\mathcal{P} = B \rightarrow G$ (highlighted in the figure). The relocation process starts by evicting the selected replacement candidate. Subsequently, the ancestor of the evicted candidate in $\mathcal{P}$ is moved to the location of the evicted candidate, as indicated by the arrow from B to G in Figure 3.3. Since the evicted block was found by hashing the tag of the ancestor (see Figure 3.2b), the ancestor itself maps to the same cache line as the evicted block. This ensures that the ancestor will still be found after being relocated, when being searched for in the cache. This relocation of the ancestors is repeated up to the root of $\mathcal{P}$, eventually resulting in an available cache line that corresponds to one of the cache lines as given by the mapping of the missing block. In Fig. 3.3 the cache line made available is the cache line that B resided at before it was relocated. The final result after relocation and insertion of the missing block A is depicted in Figure 3.4.

During the relocation process, it is important that all other outstanding operations, involving cache blocks that are relocated, are blocked to ensure correct operation of the cache. To avoid increasing cache latency, one can allow the BFS-walk to run concurrently with other cache operations. By doing this, benign race conditions can occur: a block that has been identified as the best replacement candidate by the BFS-walk might have been accessed directly afterwards, making another replacement candidate preferred [32]. In small caches such as first level caches, these benign race conditions might be common, but in larger caches these race conditions are rare and one solution is to ignore them [32]. In our evaluation, only the LLC can be a zcache. Since LLCs are relatively large, we choose to ignore those race conditions.

This elaborate replacement process is performed only upon a miss and happens concurrently with the read of the missing block from DRAM. Therefore, it does not have to increase the latency of the cache: Sanchez and Kozyrakis [32] show that the time taken by the this replacement process is significantly less than the latency of DRAM in the case of 21 replacement candidates.

3.2.2 Associativity of the Zcache

In Section 1.1.2 we have defined associativity as the amount of positions a block can reside at in a cache. However, given a n-way set-associative and a n-way zcache, this definition results in the same associativity for both caches. Still, Sanchez and Kozyrakis [32] have shown that zcaches can obtain higher hit-ratios than set-associative caches of the same size, but with more ways, implying higher associativity. To quantify the associativity more generally, Sanchez and Kozyrakis [32] propose the use of an *associativity distribution*. To calculate the associativity distribution, the cache its replacement algorithm is assumed to globally rank the cache lines. Cache lines that are preferred for eviction get a higher rank r. If the cache contains N cache lines, then $0 \leq r \leq N-1$. The *eviction-priority* of a cache line e is given by:

$$e = \frac{r}{N-1}. \tag{3.2}$$

The associativity distribution $F_a(e)$ of a cache is defined as the Cumulative Distribution Function (CDF) of the eviction priorities of the evicted blocks. In the optimal case, a fully associative cache, the cache line with $e = 1$ is always evicted, resulting in the following associativity distribution:

$$F_a(e) = \begin{cases} 1 & \text{for } e = 1 \\ 0 & \text{elsewhere} \end{cases} \tag{3.3}$$

However, most caches have a limited amount of replacement candidates. The associativity distribution will therefore be a smeared out towards lower eviction priorities, as it is not always possible to select the cache line with $e = 1$. Therefore, the more the associativity distribution is skewed towards Equation (3.3), the higher the associativity.

The associativity distribution depends on the used replacement algorithm and the workload utilizing the cache. Therefore, determining the associativity distribution in general can be a tedious task. However, under certain circumstances, it can be characterized solely by the *number of replacement candidates* that a cache provides. Those

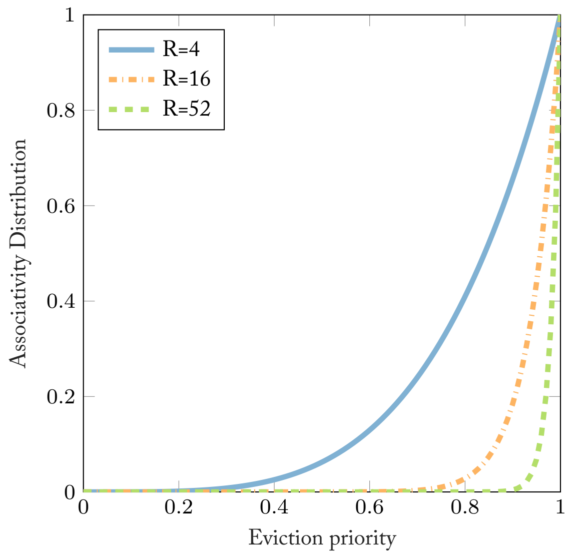


Figure 3.5: Associativity Distribution of a Cache under the Uniformity Assumption

circumstances are referred to as the *uniformity assumption* [32]. Under the uniformity assumption, the cache always returns R replacement candidates and their eviction properties $E_1, E_2, \ldots E_R$ can be modeled as i.i.d. random variables. Furthermore, the distribution of the eviction priorities is uniform, that is $E_i \sim U(0,1)$.

Consequently, the CDF of the eviction priority E_i under the uniformity assumption is given by:

$$F_{E_i}(x) = x.$$

Since the replacement policy evicts the candidate with the highest eviction priority, we can define another random variable $A = \max(E_i), i \in 1, 2, \ldots R$ that takes on the maximum priority of all candidates considered. The CDF of A, which is the associativity distribution, is then given by [32]:

$$\begin{aligned} F_A &= Pr(A \leq x) = Pr(E_1 \leq x \wedge \ldots \wedge E_n \leq x) \\ &= Pr(E_i \leq x)^n = x^R \end{aligned} \tag{3.4}$$

The higher R becomes, the more Equation (3.4) resembles Equation (3.3) and the higher the associativity. This is effect is visible in Fig. 3.5, which shows the associativity distribution as given by Equation (3.4) for a cache providing 4,16 and 52 replacement candidates.

Although a zcache and skew-associative cache do not truly randomly select their replacement candidates[1], Sanchez and Kozyrakis [32] show that both zcaches and skew-

[1]Also a zcache will not *always* provide R replacement candidates as repeats might occur (see also Section 3.2.1).

associative caches practically meet the uniformity assumption. Therefore, the associativity of a zcache and skew-associative cache is characterized directly by the amount of replacement candidates [32].

3.2.3 Hashing in the Zcache

The family of hashing functions used for the zcache by Sanchez and Kozyrakis [32] is the H3-family. We define the H3-family of hashing functions following Carter and Wegman [8].

Let x be an i-bits number that is to be hashed to a j-bits number. Furthermore let A denote the set of all i-bits numbers and B the set of all j-bit numbers. Let M be the set of all arrays of length i whose elements are from B. Each element $m \in M$ can therefore be thought of as an $i \times j$ Boolean matrix [8]. For each Boolean matrix $\mathbf{M}' \in M$, a hash-function $f_{\mathbf{M}'} : A \to \mathbb{B}^j$, is defined as:

$$f_{\mathbf{M}'}(x) = x_1 \cdot \mathbf{M}'^T_{1,*} \oplus x_2 \cdot \mathbf{M}'^T_{2,*} \oplus \ldots \oplus x_i \mathbf{M}'^T_{i,*}, \tag{3.5}$$

with x_k the kth bit of x and $\oplus$ the exclusive-or (XOR) of two Boolean vectors. The set of all hash-functions defined by Equation (3.5) denotes the H3-family of hashing functions.

For a zcache, each way is associated with a hash function from this family and is selected at random [32].

Let H denote the set of all H3-hashing functions from A to B. The H3-family is *universal* [31], which means that for any pair of distinct input values (a, b) in A, no more than a fraction $\frac{1}{|B|}$ of all hashing functions in H result in the same hash value (collide)[8]. This, in turn, leads to an approximately uniform distribution of hash values. This is a desirable property as it ensures that the amount of blocks that map to any certain cache line is approximately equal [31].

Furthermore, the hash values of two H3-hash functions $h_1 : A \to B$ and $h_2 : A \to B$, $h_1 \neq h_2$ are *pairwise independent* [31]. That is, for any distinct pair of input values (a, b), $a \in A, b \in A$ the following holds

$$P\big(h_1(a) = x \wedge h_2(b) = y\big) = \frac{1}{|B|^2} \tag{3.6}$$

meaning that the output values of both h_1 and h_2 behave as if they were randomly chosen. This ensures that the quality (the randomness as assumed by the uniformity assumption) of the replacement candidates does not degrade with the number of levels [31].

3.3 LRU at Cache Line Granularity: Bucketed-LRU

Since the cache blocks in a zcache and skew-associative can not be grouped into sets, implementing LRU would require that the cache blocks are globally ranked. This implicates large hardware-overheads, as the recency counters need to be able to store M values, with M the size of the cache in cache lines.

To reduce the hardware overhead, Sanchez and Kozyrakis [32] propose an algorithm that approximates LRU, which is referred to as *bucketed-LRU*. In this algorithm a 8-bit timestamp is added to each cache line. Based on a global timestamp counter (also 8-bits), the replacement candidate that has the oldest timestamp is chosen for eviction. Since the timestamps are 8-bits long, it is proposed that the global counter is incremented only every k accesses with k set to 5% of the cache-size. This makes it is rare for a block to stay in the cache after a wraparound of the global counter, without being evicted or referenced [32].

4 | Cache Partitioning Algorithms

The LLC is one of the most important shared resources on a CMP [19]. Therefore, it is important that it is utilized in an efficient manner. Although LRU-managed set-associative caches are widespread in CMPs, LRU implicitly manages the cache space among the cores based on their access frequency. This can result in serious performance degradation. To this end, several cache partitioning algorithms have been proposed that address the shortcomings of LRU.

Cache partitioning algorithms can be defined as algorithms that manage the usage of the cache among several independent concurrent processes. Before continuing to describe any cache partitioning algorithm, we divide a cache partitioning algorithm in an *allocation policy* that prescribes how big the sizes of all partitions should be and an *partitioning scheme* that enforces the partition sizes as given by the allocation policy [33]. Allocation policies can target Quality of Service (QoS), fairness and hit-maximization [22]. In the following we will restrict ourselves to the discussion of hit-maximizing cache partitioning algorithms, as not every cache partitioning algorithm in our framework is designed to support QoS or fairness.

The first part of this chapter will be devoted to algorithms that do not have an explicit allocation policy. We will refer to these algorithms as *implicit cache partitioning algorithms*. These algorithm focus on regulating memory access patterns and thereby implicitly partition the cache. (Bucketed-)LRU as previously described fall under this category. The second part of this chapter will give an overview of several algorithms that explicitly partition the cache among the processes running on the CMP. We will refer to those algorithms as *explicit cache partitioning algorithms*.

As in previous works [22, 28, 42, 33], we assume that each core of the CMP is set up to run one application. Therefore, while discussing any cache partitioning algorithm, we will use the terms application and core interchangeably.

4.1 Implicit Cache Partitioning Algorithms

In this section we will start with an overview of the implicit cache partitioning algorithm Dynamic Insertion Policy (DIP), as the implicit cache partitioning algorithms TADIP and DRRIP in our simulation framework are based on several concepts of DIP. After DIP we give an overview of the operation of TADIP and we conclude this section with a description of DRRIP.

4.1.1 DIP

Dynamic Insertion Policy (DIP) is an implicit cache partitioning algorithm that targets trashing access patterns (see also Section 2.2.2 for a description on memory access patterns). It modifies the LRU replacement algorithm by changing the policy that determines how a block is inserted. This policy is also referred to as the *insertion policy*. Instead of only using the LRU Insertion Policy (LIP), DIP supplements LIP with the Bimodal Insertion Policy (BIP). BIP inserts an incoming cache line predominantly at the LRU position and at the MRU position with a low probability. This is regulated by the bi-modal throttle parameter ϵ that determines the percentage of cache lines placed at the MRU position. LRU can therefore also be viewed as BIP with $\epsilon = 1$.

When the working set of an application is bigger than the cache, always inserting at the MRU position can lead to a trashing (see Section 2.2.2). By using BIP instead, only few blocks are inserted at the MRU position and can stay in the cache as most blocks are inserted at the LRU position and therefore do not replace the blocks at higher recency positions. In this way, a part of the working set can be kept in the cache, improving performance. DIP dynamically chooses which policy is best to incur the fewest misses. Two methods, DIP-Global and DIP-SD (Set-Dueling) have been developed by Qureshi et al. [29].

DIP-Global

In DIP-Global, besides the MTD of the DIP managed cache, two Auxiliary Tag Directorys (ATDs) are present. These ATDs can can be seen as copies of the MTD. The two ATDs have the same amount of ways as the MTD and all access made to the MTD are made visible to the ATDs. One ATD is set up to use LIP whereas the other is setup to use BIP. Every time a miss occurs in the LIP-ATD, a saturating counter called the *Policy Selector* (PSEL) is incremented. A miss in the BIP-ATD decrements the PSEL. Therefore the higher the value of the PSEL, the more misses have been incurred by LIP and vice versa: the lower the value of the PSEL, the more misses have been incurred by

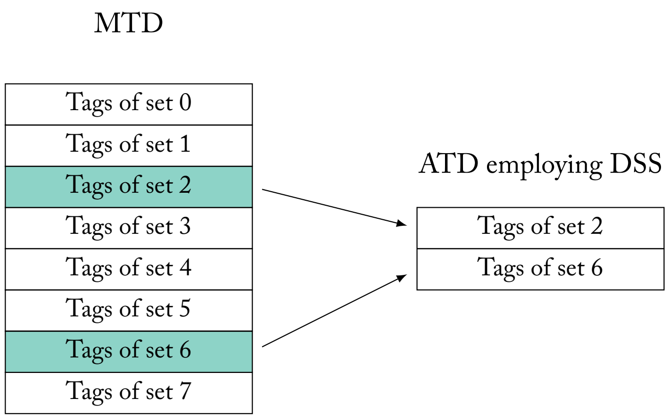


Figure 4.1: ATD Employing DSS

BIP. Whenever the Most-significant Bit (MSB) of the PSEL is set, the counter has a value in the upper halve of its range, indicating that BIP is incurring fewer misses than LIP. Therefore, when the MSB is set, BIP is selected as insertion policy for the MTD. LIP is selected as insertion policy for the MTD when the MSB is cleared.

DIP-SD

The use of two ATDs is not practical in a hardware implementation, as the overhead caused by duplicating the MTD is considerable. To reduce the overhead, Dynamic Set-Sampling (DSS) [28] may be used. Instead of duplicating the complete MTD, the idea is that by sampling the tags of only a few sets of the MTD, the global behavior of the cache can still be accurately approximated [29]. An ATD utilizing DSS is illustrated in Figure 4.1.

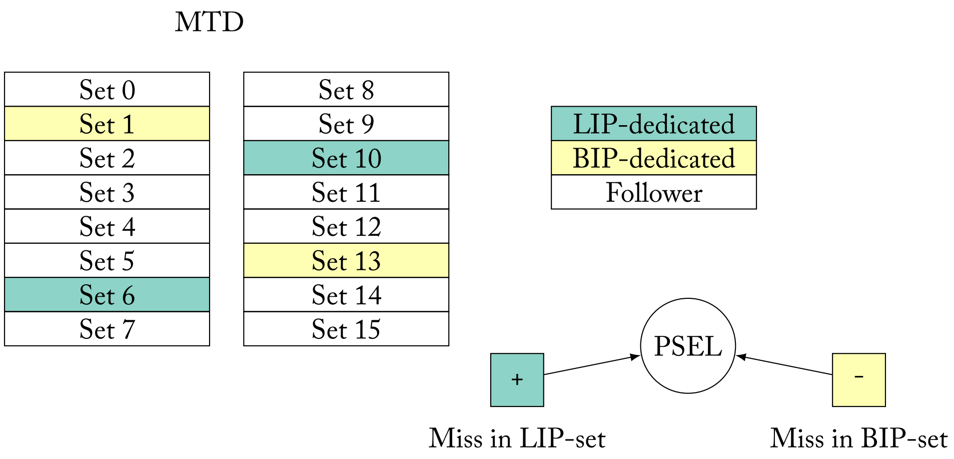


Figure 4.2: DIP-SD, after [29]

Qureshi et al. [29] use the concept of DSS for the definition of DIP-SD, that completely eliminates the need for additional ATDs. This method, *Set-Dueling*, dedicates few of the cache sets to use either LIP or BIP, the so-called *dedicated sets*. The policy that incurs the fewest misses on the dedicated sets is used as the policy for all other sets, the *follower sets*. DIP-SD is illustrated in Figure 4.2.

4.1.2 TADIP

Thread-Aware Dynamic Insertion Policy (TADIP), introduced by Jaleel et al. [17], is a thread-aware version of DIP. DIP does not differentiate between the behavior of the applications running on a CMP and makes a global decision as to which insertion policy is used for the whole cache.

TADIP makes this decision for *each* application running on the CMP. When N applications are running on a CMP, there are 2^N possible decisions as to which combination of policies to use. The selection of policies, P, can be represented as a tuple

$P = (p_0, p_1, \ldots, p_{N-1})$, where p_i indicates the selected policy, BIP or LIP, for application i. The policy-decisions can be made statically (through the use of profiling) or dynamically [17]. Using profiling to decide the best policies is impractical in general situations: the best policy depends on the workload, but also on the cache-size [17]. As Jaleel et al. [17] we will not pursue static decision making any further.

Since there are 2^N possible decisions, dynamically determining the optimal combination of policies can be costly. For example, when ATDs are used to determine the optimal combination, 2^N ATDs would be required to get statistics for each combination. Jaleel et al. [17] propose the use of set-dueling to reduce the overhead. Two versions based on set-dueling are proposed: TADIP-I(solated) and TADIP-F(eedback).

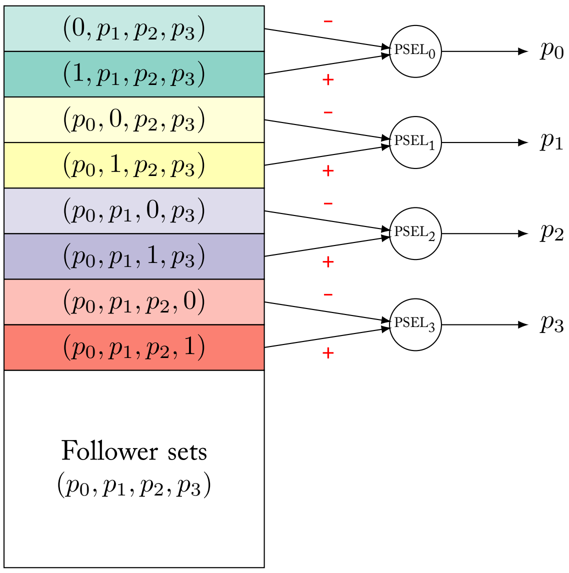


Figure 4.3: Dueling and Follower Sets for TADIP-F - BIP=0, LIP=1, after [17]

TADIP-I uses $N + 1$ dedicated sets. The first dedicated set uses LIP for all applications. The remaining N dedicated sets use LIP for all applications except one. That is, the first of these N dedicated set uses BIP only for application 1, the second only for application 2 and so on. Furthermore TADIP-I uses one saturating counter $PSEL_i$ for every application i. A miss in the first dedicated set increments the saturating counter of all applications. Misses in the dedicated sets that use BIP for application i only decrement $PSEL_i$. In this way, TADIP-I can learn what the best insertion policy is for each application in isolation: if the whole cache would use LIP, would the use of BIP for this application improve performance?

TADIP-F uses at least one pair of dedicated sets per application: one set is dedicated to LIP and the other to BIP for the application. To take the optimal decision of all other cores into account, the dedicated pairs of application i apply the optimal policy p_j as given by the dedicated sets of application $j \neq i$ when a cache line by a core j is inserted in the dedicated set of core i. This scheme is illustrated in Figure 4.3.

Of those two versions, TADIP-F has been shown to perform better than TADIP-I [17]: determining the best-policy for an application by taking the currently selected policy of all other applications into account is beneficial.

4.1.3 DRRIP

In Section 2.2.2 we have shown that mixed-memory access patterns containing scans can degrade the performance of LRU. Jaleel et al. [18] introduce Re-Reference Interval Prediction (RRIP) to address this issue.

The idea is to store a M-bits value for each cache line in a set. This value, the Re-Reference Prediction Value (RRPV), gives the re-reference prediction for an individual cache line. The RRPV is an indication of the predicted time before a next reference to a cache line. Unlike the recency-counters as used by LRU, the RRPV's in a set do not strictly order all cache lines in that set. Therefore, it is possible that several cache lines within a set have the same RRPV. A RRPV of 0 indicates that a cache block is predicted to be re-referenced in the near-immediate future. A RRPV of 2^{M-1} on the other hand, is a prediction for a re-reference in the distant future. Therefore, the lower the RRPV, the sooner block is predicted to be re-referenced. Blocks with a higher RRPV, and therefore more distant predicted re-reference interval, are prioritized for eviction. Upon a cache miss, the corresponding set is searched for blocks with a distant re-reference interval (2^{M-1}). However, a set might contain several of those blocks. In such cases, a tie-breaker is used to select a block for eviction. If no such block is present, all RRPV's in the set are incremented and the search for a block for eviction is repeated. Updating the RRPV's by incrementing them allows RRIP to remove stale blocks from the cache [18].

RRIP mainly aims at preventing blocks with a distant re-reference interval from polluting the cache [18]. However, in general, there is no external information about the re-reference intervals available. This gives rise to Static Re-Reference Interval Prediction (SRRIP) algorithm, which statically predicts the re-reference intervals of blocks upon cache hits and misses.

Upon a cache miss, always predicting a near-immediate re-reference interval for the missing block is not robust: blocks from a scan will unnecessarily occupy space in the cache. Always predicting a distant re-reference interval is not robust either, as blocks with near-immediate intervals are more likely to be evicted. This would reduce the performance for memory access patterns that predominantly access blocks with near-immediate re-reference intervals [18]. SRRIP uses a middle way: it inserts a missing block with a re-reference interval that is *long*. A long re-reference interval is represented by a RRPV of $2^M - 2$. Long distance intervals prevent blocks with distant reference intervals from polluting the cache. At the same time they leave some time for hits to occur to blocks that *do* have near-immediate re-reference intervals.

When a cache hit occurs, SRRIP needs to update the RRPV of the corresponding block. The policy that determines how the RRPV is updated is called the *promotion policy*. Jaleel et al. [18] propose two promotion policies: Hit-Priority(HP) and Frequency Priority(FP). The HP-policy predicts that the block corresponding to the hit will be re-referenced in the near-immediate feature, therefore setting its RRPV to 0. In case that a block is only used once after its insertion, the HP-policy can degrade the per-

Initial state:

A	B	C	Empty
1	0	2	3

Insertion of D

A	B	C	D
1	0	2	2

Insertion of E
(tie-brake between C and D in favor of C)

A	B	E	D
2	1	2	3

Hit to D

A	B	E	D
2	1	2	2

Figure 4.4: SRRIP Managed Cache Set with 4 Ways, 2bits RRPV's, and the FP-Policy

formance as the block stays longer in the set than necessary. The FP-policy addresses this issue by decrementing the RRPV value on a hit (unless it is already 0). Therefore, several hits might be required before the RRPV of the corresponding block equals 0. SRRIP-FP is illustrated with 2-bits RRPV's in Figure 4.4.

When the re-reference interval of all the blocks is larger than the available cache, SRRIP is trashing the cache [18]. Similarly to DIP, which uses BIP to counter trashing access patterns, SRRIP can be combined with Bimodal Re-Reference Interval Prediction (BRRIP), which inserts the majority of the cache blocks with a *distant* re-reference interval. BRRIP, infrequently (with probability ϵ), inserts a block as having a *long* re-reference interval (RRPV of $2^M - 2$). In this manner, parts of the working set of the trashing application can be kept in the cache, improving performance.

Similar to TADIP-F, set-dueling is used to dynamically select whether to use SRRIP or BRRIP. This algorithm, combining SRRIP and BRRIP, is referred to as Dynamic Re-Reference Interval Prediction (DRRIP).

4.2 Explicit Cache Partitioning Algorithms

Several explicit cache partitioning algorithms have been proposed, but they do not always introduce both a partitioning scheme and allocation policy [33, 42, 41]. The explicit cache partitioning algorithms considered in this thesis are UCP, PIPP, PriSM and Vantage. Their details regarding the inclusion of an allocation policy and enforcement scheme are given in Table 4.1.

Reference	Algorithm	PS	AP
Qureshi and Patt [28]	Utility-Based Cache Partitioning (UCP)	✓	✓
Xie and Loh [42]	Promotion/Insertion Pseudo Partitioning (PIPP)	✓	UCP
Manikantan et al. [22]	Probabilistic Shared Cache Management (PriSM)	✓	✓
Sanchez and Kozyrakis [33]	Vantage	✓	interpolated UCP

Table 4.1: Overview of Evaluated Explicit Cache Partitioning Algorithms - PS=Partitioning Scheme, AP=Allocation Policy

4.2.1 UCP

Utility-Based Cache Partitioning (UCP), proposed by Qureshi and Patt [28], is an explicit partitioning algorithm using set-associative caches, including both an allocation policy and partitioning scheme. The allocation policy of UCP is based on *utility*, an indication of the usefulness of cache resources to a certain application. Qureshi and Patt [28] use UCP to perform partitioning by way (way partitioning). In the following we will implicitly assume that UCP is used to partition the cache at way-granularity.

Allocation policy

The utility $U_{i,a}^{b}$, corresponding to an increase in the allocation of application i from a to $b > a$ ways, is given by:

$$U_{i,a}^{b} = miss_a - miss_b, \tag{4.1}$$

where $miss_a$ and $miss_b$ give the amount of misses for application i when it receives a respectively b ways [28]. That is, the utility $U_{i,a}^{b}$ equals the amount of misses *saved* by allocating $b - a$ more ways to application i. Let $\vec{a} = (a_1, a_2, \ldots a_M)^T$ denote the vector containing the allocations of M applications sharing the cache. Let N be the amount of ways of the set-associative cache used. As no more than a total of N ways can be allocated, the sum of all allocations equals N:

$$\sum_{k=1}^{M} a_k = N.$$

UCP's allocation policy tries to maximize the combined utility of all applications running on the CMP. We write this as the following optimization problem:

$$\begin{aligned} \underset{\vec{a}}{\text{maximize}} \quad & \sum_{i=1}^{M} U_{i,0}^{a_i} \\ \text{subject to} \quad & \sum_{i=1}^{M} a_i = N. \end{aligned} \tag{4.2}$$

Maximizing the combined utility corresponds to minimizing the total amount of misses of all applications combined, as an increase in utility is directly proportional to a decrease in misses (see Equation (4.1)) [28].

To obtain the data needed to calculate the utility for every application, Qureshi and Patt [28] propose the use of an Utility Monitor (UMON). An UMON is based on the stack property of LRU (see also Section 2.2.1): if an application hits in a cache with $a \leq b$ ways, then the application will also hit in a cache with b ways.

The term $miss_a - miss_b$ in Equation (4.1) is equal to the amount of additional hits a b-way set-associative cache receives over an a-way set-associative cache. By tracking the amount of hits h_i for every recency position $i \in \{0, 1, \ldots, N-1\}$ in the LRU-chain, $miss_a - miss_b$ can be calculated as

$$miss_a - miss_b = \sum_{k=a}^{b-1} h_k. \tag{4.3}$$

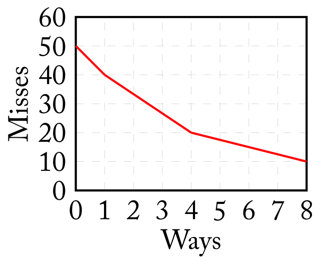
For an UMON it is therefore sufficient to track the hits of one N-way set associative cache instead of N caches with $i \in 1, 2, \ldots, N$ ways.

To keep track of the hits, an ATD is used per application. By adding a hit-counter for every recency position in the ATD, the required information is gathered. This can be done a per set basis, in which every set has N hit counters for every recency position in the set(UMON-local). UMON-local allows for partitioning per set, but incurs too much hardware overhead to be practical. To save hardware, one hit counter can be used for the same recency position among all sets (UMON-global). UMON-global allows for way-partitioning. To save even more hardware on UMON-global, DSS (see Section 4.1.1) can be applied to the ATD.

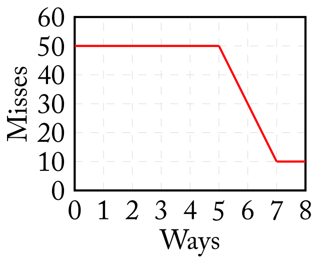
Finding an optimal solution to the optimization problem in Equation (4.2) has been shown to be NP-hard [28]. To keep the complexity of finding an allocation bounded, Qureshi and Patt [28] propose the use of a greedy algorithm at the cost of a possibly non-optimal solution. This algorithm, the *look-ahead algorithm*, makes use of the *marginal utility* $MU_{i,a}^{b}$ to calculate an allocation. When the allocation of an application i is increased from a to b ways, the marginal utility is given by:

$$MU_{i,a}^{b} = \frac{U_{i,a}^{b}}{b-a}. \tag{4.4}$$

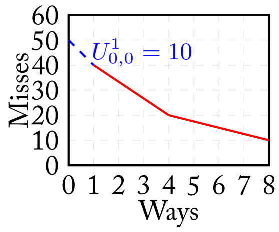
The look-ahead algorithm starts by allocating zero ways to every application, that is



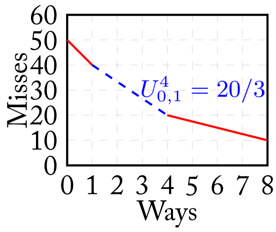
(a) Miss-curve application 1



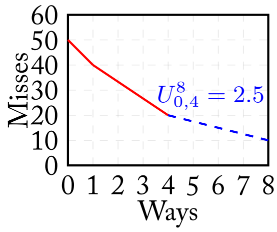
(b) Miss-curve application 2



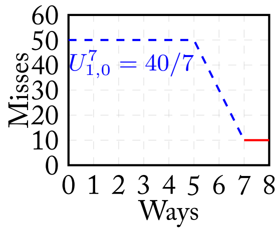
(c) Step 1: application 1



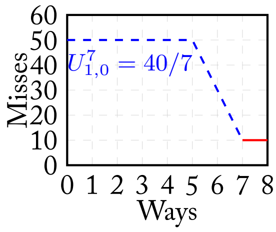
(d) Step 2: application 1



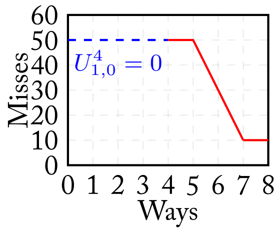
(e) Step 3: application 1



(f) Step 1: application 2



(g) Step 2: application 2



(h) Step 3: application 2

Figure 4.5: Example of the Look-Ahead Algorithm

$\vec{a} = \vec{0}$. It then starts searching for the highest marginal utility

$$MU^b_{i,a_i}, \ \ b = a_i + 1, \ldots, a_i + N - \sum_{k=1}^{M} a_k$$

among all applications. If the highest marginal utility has been found for increasing the allocation of application x to b ways, then a_x is set to b and the search repeats until $\sum_{k=1}^{M} a_k = N$, indicating that all ways have been allocated.

We will now illustrate the look-ahead algorithm with an example for two applications sharing the LLC, as depicted in Figure 4.5. In this example $N = 8$, $M = 2$ and the look-ahead algorithm takes the following steps:

1. Calculate the maximum possible marginal utility for application 1 and 2, starting with $\vec{a} = (0, 0)^T$. $N - \sum_{k=1}^{M} a_k = 8$ ways can be allocated. The highest marginal

utility for application 1 is 10 and corresponds to an increase of its allocation by 1 way. The highest marginal utility for application 2 is $40/7$ and corresponds to an increase of its allocation by 7 ways. Application 1 has the highest marginal utility and wins the search. Its allocation is therefore set to 1 way: $a_1 = 1$.

2. Calculate the maximum possible marginal utility for application 1 and 2, starting with $\vec{a} = (1, 0)^T$. $N - \sum_{k=1}^{M} a_k = 7$ ways can be allocated. The highest marginal utility for application 1 is $20/3$ and corresponds to an increase of its allocation by 3 ways. The highest marginal utility for application 2 is $40/7$ and corresponds to an increase of its allocation by 7 ways. Application 1 has the highest marginal utility and wins the search. Its allocation is therefore increased by 3 ways: $a_1 = 4$.

3. Calculate the maximum possible marginal utility for application 1 and 2, starting with $\vec{a} = (4, 0)^T$. $N - \sum_{k=1}^{M} a_k = 4$ ways can be allocated. The highest marginal utility for application 1 is 2.5 and corresponds to an increase of its allocation by 4 ways. The highest marginal utility for application 2 is 0. Application 1 has the highest marginal utility and wins the search. Its allocation is therefore increased by 4 ways: $a_1 = 8$.

This results in the allocation $\vec{a} = (8, 0)^T$ and 40 misses are saved. The allocation $\vec{a} = (1, 7)^T$ would save 50 misses. This illustrates the non-optimality of the look-ahead algorithm.

In this example, the look-ahead algorithm allows that an application is allocated zero ways in the cache. When the cache-hierarchy is *inclusive*, all data present in a higher level cache *must* be present in the lower level caches. When a shared LLC is partitioned, allocating no cache space to a core makes it impossible to include the data of this core its higher-level caches. Qureshi and Patt [28] modify the look-ahead algorithm to allocate at least one way to application, to ensure that inclusiveness can be enforced.

To keep the overhead of re-partitioning low, Qureshi and Patt [28] suggest that the look-ahead algorithm is run with re-partitioning periods of 5 million cycles. After every period, the hit counters in the ATD are halved: in this way, the history of an application is taken into account.

Partitioning scheme

UCP's partitioning scheme, as defined by Qureshi and Patt [28], is based on LRU with the following addition and modification:

1. Each cache line's tag is extended to include the core that installed the block currently residing in this cache line.

2. Upon a cache miss, the amount of blocks in the corresponding set belonging to the core causing the miss are counted: when this amount equals or exceeds the allocation, the LRU-block among all blocks of this core in this set is evicted. Otherwise, the LRU entry of all blocks belonging to the other cores is evicted.

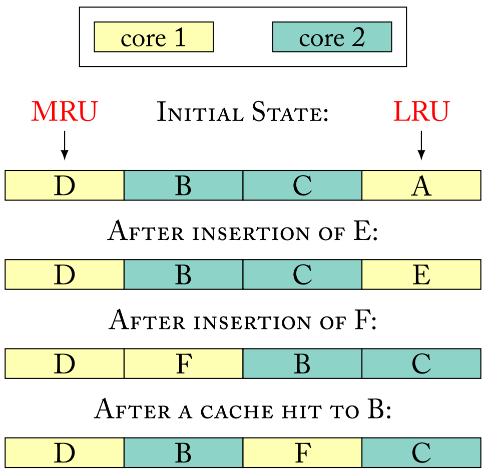


Figure 4.6: A PIPP Managed Cache Set

This partitioning scheme will therefore never allow a partition to grow when it has reached its target size. However, due to a change in allocation by the look-ahead algorithm, it might be that a partition temporarily exceeds its target size (in case it was downsized). This is transient behavior and is the consequence of regulating partition sizes through eviction. Neglecting this transient behavior, UCP's partitioning scheme is strict.

4.2.2 PIPP

Promotion/Insertion Pseudo Partitioning (PIPP), introduced by Xie and Loh [42], is a partitioning scheme that does not enforce *strict* partition sizes. It is based on a set-associative cache using the conventional LRU-replacement algorithm, but modifies how blocks are inserted and updated upon a cache hit. Xie and Loh [42] suggest the use of UCP as allocation policy.

Let N denote the amount of cores of the CMP and w the amount of ways of the set-associative cache used. Furthermore let $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$ be a partitioning of the cache such that $\sum_{i=1}^{N} \pi_i = w$. PIPP modifies the insertion policy of LRU to insert a block belonging to core i at position π_i. Position 1 corresponds to the LRU position and w to the MRU position. This is illustrated in Figure 4.6 for two cores with $\Pi = \{1, 3\}$.

Upon a cache hit, the referenced block (if possible) is moved one position closer to MRU in the LRU-chain with a probability of p_{prom}. On a miss, PIPP behaves the same as conventional LRU, evicting the LRU cache line, regardless of to which core it belongs to. This is illustrated in Figure 4.6. The lower the insertion position is, the more likely it is that the data item gets evicted in the near future and vice versa: the higher the position, the more likely it is to stay in the cache in the near future. In this way PIPP approximately tracks the target partition sizes.

Applications that are characterized by streaming access patterns are unlikely to re-use a lot of cache blocks that they insert. The misses incurred by such applications can remove blocks from the cache-sets that are useful to other applications. Therefore, Xie and Loh [42] suggest the use of a streaming detector that detects whether an application is streaming. This detector compares the amount of misses of a certain application incurred in a certain partitioning-period with a threshold θ_m. When θ_m is exceeded, an application is considered streaming. Furthermore, the miss rate of an application is compared to a threshold θ_{mr}. When exceeded, the application is considered streaming as well. When an application is considered streaming, the insertion policy used for this application is adjusted: regardless of the allocation, the insertion position of a streaming application equals the total amount of streaming cores. In this way, each streaming application is effectively allocated on way [42].

When a hit occurs to a block belonging to a core that is considered streaming, the block is moved (if possible) to one position closer to MRU with a probability $p_{stream} \ll p_{prom}$. The reasoning behind using a smaller probability, is that only the blocks of a streaming application that do exhibit a lot of re-use get promoted to positions closer to MRU[1]. This prevents the other blocks from the streaming access pattern from polluting the cache. If all cores are streaming, the insertion policy is set to LIP, as there is no non-streaming application running that could be negatively affected by the insertion of blocks from streaming cores.

Chaudhuri [9] shows that PIPP suffers from a decrease in performance when the congestion at the tail of the LRU chain is high. This corresponds to situations in which many cores insert their entries close to the LRU position. This is for example the case when the amount of cores equals the amount of ways and each core is allocated one way in the cache. In such situations competition for the same cache blocks increases, thereby increasing the amount of conflict misses, which hurts the performance. Manikantan et al. [21] suggest the use of a minimum insertion position of 8 ways above LRU in a 16-way set-associative cache, to avoid that inserted blocks are evicted before they are able to be promoted. Both Manikantan et al. [21] and Olsen [24] confirm that PIPP performs considerably worse without the use of a minimum position. In the following we will refer to PIPP with a minimum insertion position simply as PIPP.

4.2.3 PriSM

Probabilistic Shared Cache Management (PriSM) is an explicit probabilistic cache partitioning algorithm, including both an allocation policy and a partitioning scheme. PriSM partitions a set-associative cache at a block-granularity. The partitioning scheme is based on an eviction probability distribution $\mathcal{E}$, that prescribes what the probabilities are that a block belonging to core i is evicted. Adjusting the eviction-rate of core i allows for adjusting its partition size.

[1]Strictly speaking such a pattern would be a mixed memory access pattern, as both streaming and recency-friendly characteristics are present.

Partitioning Scheme

Upon a cache miss, PriSM selects a *victim-core* of which a cache line is to be evicted, based on the discrete probability distribution $\mathcal{E}$. $\mathcal{E}$ gives the probability E_i of eviction of a block from core i. $\mathcal{E}$ is determined based on the concept of *occupancy*. The occupancy of a core i is defined as the fraction of the cache it owns. $\mathcal{E}$ should be chosen such that the eviction rate of core i regulates its current occupancy C_i to a value close or equal to its target occupancy T_i, as set by the allocation policy.

Let M_i denote the fraction of the total amount of misses caused by core i in an interval of W misses. Together with the target occupancy T_i and current occupancy C_i, the eviction probability E_i can be determined to regulate C_i to T_i. The reasoning is that during an interval of W misses, the amount of cache lines owned by core i grows by an amount of $M_i \times W$ due to the misses incurred by core i. This means that at the end of the interval, core i would have an occupancy of $C_i + M_i \times \frac{W}{N}$, with N the amount of cache lines of the cache. However, given that $E_i > 0$, on average $E_i \times W$ cache blocks of core i get evicted. Therefore, at the end of the interval, the expected occupancy τ_i of core i equals:

$$\tau_i = C_i + (M_i - E_i) \times \frac{W}{N} \tag{4.5}$$

By setting τ_i in Equation (4.5) equal to T_i, one can solve for corresponding eviction probability E_i. However, in certain cases E_i might be larger than 1 or lower than 0. Since E_i represents a probability, it is clamped to the interval $[0, 1]$:

$$E_i = \begin{cases} 0, & \text{if } ((C_i - T_i)) \times \frac{N}{W} + M_i) < 0, \\ 1, & \text{if } ((C_i - T_i)) \times \frac{N}{W} + M_i) > 1, \\ (C_i - T_i) \times \frac{N}{W} + M_i, & \text{otherwise} \end{cases} \tag{4.6}$$

Once a victim core has been selected based on $\mathcal{E}$, the cache set corresponding to the missing block is inspected for cache lines belonging to the victim core. If those are present, the LRU cache line belonging to the victim core is evicted. In the cases that no such cache line is present, the LRU cache line among the cache lines of all cores i with $E_i > 0$ is evicted. If no such lines are present, the LRU cache line in the set is evicted.

To calculate the eviction probabilities, it is necessary that a value for C_i is available. C_i can be obtained by adding counters for each core to the MTD to track its cache-usage. Furthermore, it is required that M_i is known. However, at the start of an interval M_i is not known yet, because knowledge of M_i would imply knowledge about the future. Instead of the expected fraction of misses for the current interval, the fraction of the previous interval is used to approximate M_i. M_i can therefore be approximated by adding a miss-counter for every core to the MTD.

Allocation Policy

Manikantan et al. [22] propose a hit-maximizing allocation policy for PriSM. This variant of PriSM is referred to as PriSM-H. PriSM-H uses an ATD per application to track

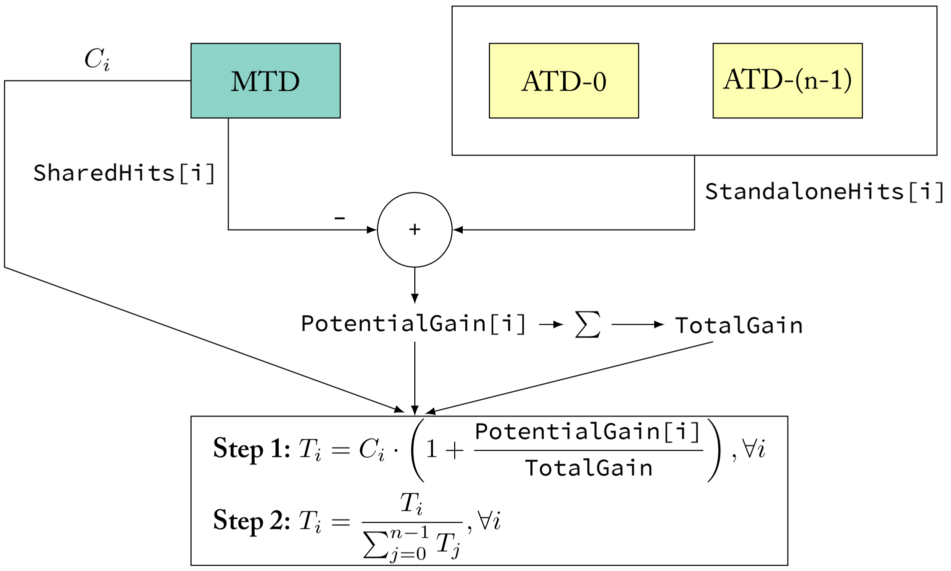

C_i
MTD
ATD-0
ATD-(n-1)
SharedHits[i]
StandaloneHits[i]
-
+
PotentialGain[i] → $\sum$ → TotalGain
Step 1: $T_i = C_i \cdot \left(1 + \frac{\texttt{PotentialGain[i]}}{\texttt{TotalGain}}\right), \forall i$
Step 2: $T_i = \frac{T_i}{\sum_{j=0}^{n-1} T_j}, \forall i$

Figure 4.7: PriSM-H's Allocation Policy

the amount of hits, `StandAloneHits[i]`, that a core i would receive if it were allocated the whole cache. By using hit counters in the MTD, the actual amount of hits per core, `SharedHits[i]` is tracked. The potential gain for core i, `PotentialGain[i]`, is calculated as the difference between `StandAloneHits[i]` and `SharedHits[i]`. The overall gain, `TotalGain`, is calculated by summing all `PotentialGain`'s. Furthermore, the MTD is setup to track the current occupancy C_i for each core i. This is depicted in Figure 4.7.

When the `TotalGain` is less than zero, the current allocation is maintained. Otherwise the target allocations are re-calculated in two steps (see also Figure 4.7):

1. For each application, set the target occupancy to the current occupancy increased by factor that is proportional to the fraction of the total gain, as contributed to by the application.

2. For each application, normalize the target occupancies T_i.

4.2.4 Vantage

Vantage is an explicit cache partitioning algorithm including only a partitioning scheme, that enforces partition sizes at a cache-line granularity. Vantage has been introduced based on prior-work on the zcache and assumes that the cache used by Vantage meets the uniformity assumption.

Vantage divides the cache into two regions:

1. A managed region;

2. An unmanaged region.

Of these two regions, only the managed region can be partitioned. Vantage is therefore not able to partition the whole cache. The unmanaged region removes interference between cache partitions as a partition will lend (evict) data from, when sized appropriately, the unmanaged region instead from partitions belonging to other applications.

Cache lines are assigned to a region by tagging them accordingly. A cache line is inserted in the managed region, from which it can be *demoted* to the unmanaged region. Once in the unmanaged region, a cache line either gets evicted or *promoted* to the managed region in case of a cache hit. By adjusting the demotion rate of every partition, the target sizes are enforced. Therefore, in general, all evictions take place in the unmanaged region, except in the rare cases that none of the replacement candidates offered by the cache come from the unmanaged region [33].

Maintaining the Size of the (Un)Managed Region

To maintain the sizes of the managed and unmanaged region, there should be a balance between replacements and promotions on one side an demotions on the other: one demotion is necessary per every promotion or replacement. Let u denote the fraction of the cache allocated to the unmanaged region. The fraction m of the cache allocated to the managed region therefore equals $m = (1 - u)$. With R replacement candidates, there are $\binom{R}{i}$ possible ways of having i candidates coming from the managed region, each having a probability[2] of $(1-u)^i u^{R-i}$. That is, the amount of replacement candidates i from the managed region follows a binomial distribution: $B(i, R) = \binom{R}{i}(1-u)^i u^{R-i}$. If we ignore promotions (which in general are rare compared to demotions [33]) and exactly one demotion occurs upon each replacement, the associativity distribution of the *managed region* can be approximated as [33]:

$$F_M(x) \approx \sum_{i=1}^{R-1} B(i, R) F_{A_i}(x) \tag{4.7}$$

$$\approx E[F_{A_i}(x)], \tag{4.8}$$

with $F_{A_i}(x) = x^i$ denoting the associativity distribution for a cache with i replacement candidates conforming to the uniformity assumption. The cases $i = 0$ and $i = R$ are not included in the sum in Eq. (4.8). In those cases, no candidates are demoted as there are either no candidates from the managed region to be demoted, or all candidates come from the managed region. In the latter case, one of the replacement candidates needs to be evicted from the managed region, rather than demoted. However, both cases have a negligible probability if the unmanaged region is sized appropriately [33].

To find the minimum size of the unmanaged region such that the worst case probability of having to perform an eviction of a block from the managed region equals P_{ev}, we repeat the associativity distribution of the whole cache (Eq. (3.4)):

$$F_A(x) = x^R.$$

[2]Since Vantage is derived based on the uniformity assumption.

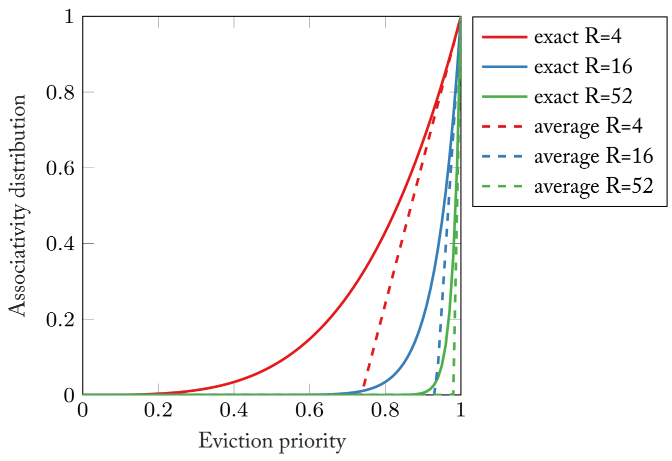


Figure 4.8: Associativity Distribution of the Managed Region of Vantage, $u = 0.3$

In the best case, the unmanaged region is filled with blocks that all have higher eviction priorities than those in the managed region. Evaluating $F_A(x)$ at $x = (1 - u)$ then gives the probability that a block is evicted from the managed region. Solving for u gives the minimum size of the unmanaged region for a given P_{ev}:

$$u \geq 1 - \sqrt[R]{P_{ev}} \tag{4.9}$$

Aperture Based Demotion

Sanchez and Kozyrakis [33] show that associativity in the managed region can be significantly improved if *on average* one demotion takes place for every promotion or eviction. This is implemented by means of an *aperture* A. All replacement candidates with an eviction priority e above $1 - A$ are evicted by the cache controller. Assuming that the cache provides R replacement candidates on a miss, then on average $R \cdot m = (1 - u) \cdot R$ replacement candidates come from the managed region. To maintain the size of the managed region the aperture should be:

$$A = \frac{1}{R \cdot m}. \tag{4.10}$$

When using an aperture for demoting, the associativity distribution of the managed region is given by:

$$F_M = \begin{cases} 0 & \text{if } x < 1 - A \\ \frac{x-(1-A)}{A} & \text{if } 1 - A \leq x \leq 1 \\ 1 & \text{if } x > 1 \end{cases} \tag{4.11}$$

Fig. 4.8 plots the associativity distributions for both exactly one demotion per replacement/promotion and one demotion on average for a cache with 4,16 and 52 replacement candidates. In all cases demoting on average results in an increase of associativity.

Enforcing the Partition Sizes

The managed region itself is partitioned into P partitions. The sizes of these partitions $T_1, T_2, \ldots T_P$ are given by a certain allocation policy. Let the rate of insertion of blocks into every partition be given by $C_1, C_2, \ldots C_P$. To maintain the actual partition sizes close to their target sizes, it is necessary that an aperture A_i is set per partition, such that the rate of demotions matches its churn. Partitions with a larger churn than average and partitions with a smaller size than average require larger apertures. On the other hand, partitions with a lower churn than average or a bigger size than average require a lower aperture.

For a partition i with current size S_i, on average, a fraction $S_i / \sum_{k=1}^{P} S_k$ of the $R \cdot m$ replacement candidates from the managed region come from partition i. That is, on average, the amount of replacement candidates of partition i from the managed region equals $\left(\sum_{k=1}^{P} S_k / S_i\right) \cdot R \cdot (1-u)$. To demote at the rate at which partition i is causing misses, its aperture needs to be multiplied by the fraction of the total churn corresponding to C_i. To maintain the partition size S_i of partition i constant, its aperture A_i should therefore be:

$$A_i = \frac{C_i}{\sum_{k=1}^{P} C_k} \frac{\sum_{k=1}^{P} S_k}{S_i} \frac{1}{R \cdot (1-u)}. \tag{4.12}$$

It might be the case that a certain partition requires an aperture $A_i > 1$. This indicates that more blocks from this partition need to be demoted than there are replacement candidates available. However, even setting $A = 1$ can be problematic, as it completely sacrifices associativity [33]: from Eq. (4.11) we find that setting the aperture equal to 1, results in all candidates encountered being demoted.

Sanchez and Kozyrakis [33] therefore propose the use of a maximum aperture A_{max} to maintain a certain amount of associativity. For applications requiring $A_i > A_{max}$ to enforce their target partition size, some leeway is incorporated when sizing the unmanaged region. If sized appropriately, this allows those partitions to outgrow their target sizes to a stable partition size at which $A_i = A_{max}$, by taking space from the unmanaged region. Sanchez and Kozyrakis [33] show that the additional fraction that should be allocated to the unmanaged region equals $\frac{1}{RA_{max}}$, and is independent of the amount of partitions used.

Feed-back based controller

Determining the aperture A_i as given by Eq. (4.12), requires that the churn C_i of partition i is known. Sanchez and Kozyrakis [33] propose the use of a feedback-based controller to avoid having to know these churns. Furthermore, by using negative-feedback Vantage is more robust: the theoretical analysis that led to Eq. (4.12) assumes that the uniformity assumption holds. Although zcaches practically meet this assumption, they

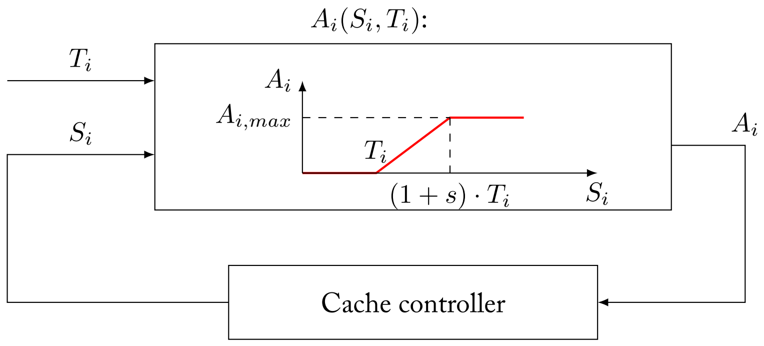


Figure 4.9: Negative Feedback to Calculate the Aperture A_i

do not do so *exactly*, which might lead to partition sizes drifting away from their targets. By allowing the partitions to outgrow their targets by some slack s, the aperture can be adjusted using negative feedback as given by Eq. (4.13). This is illustrated in Figure 4.9.

$$A_i(S_i) = \begin{cases} 0 & \text{if } S_i \leq T_i \\ \frac{A_{max}}{s}\frac{S_i - T_i}{T_i} & \text{if } T_i < S_i \leq (1+s)T_i \\ A_{max} & \text{if } S_i > (1+s)T_i \end{cases} \tag{4.13}$$

Setpoint-Based Demotion Scheme

Tracking all eviction priorities to perform the demotions, incurs too much hardware overhead to make Vantage usable. Therefore, Sanchez and Kozyrakis [33] propose a setpoint-based scheme, based on bucketed-LRU to perform demotions.

In this scheme, every partition i has a current timestamp value, `CurrentTs`, which is incremented every k_ith access. k_i is set to 1/16th of the target partition size, `TargetSize`, to make wrap-arounds of `CurrentTs` rare. Furthermore, every cache block is tagged with a timestamp value, (`Timestamp`), to keep track of the recency ranking for the bucketed LRU-scheme. To perform demotions, a *setpoint* timestamp, `SetpointTs`, is determined per partition. All replacement candidates below this setpoint timestamp (in modulo 256 arithmetic) are demoted in case the partition is exceeding its size. There is no strict ordering in modulo arithmetic, but when wrap-arounds are rare one can estimate the order of the timestamps by estimating their distance from `currentTs`. We follow Wang and Chen [41] and estimate the distance d between two timestamps a and b as

$$d = \mod (a + 256 - b, 256) \tag{4.14}$$

The larger the distance between `TimeStamp` and `CurrentTs`, the older the block. The setpoint scheme based on this definition is illustrated in Fig. 4.10. Every time c candidates from a certain partition i are seen, the amount of candidates that have been

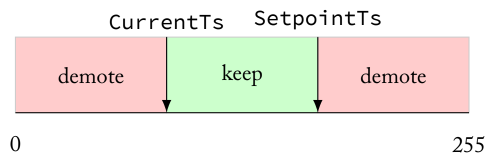


Figure 4.10: Setpoint-Based Demotion Scheme

Range (in cache-lines)	d_i
1024-1060	16
1061 - 1097	32
1098 - 1133	48
1134 - 1170	64
1171 - 1206	80
1207 - 1243	96
1244 - 1280	112
1280+	128

Table 4.2: Demotion Threshold LUT

demoted in the meanwhile, d_i, determine whether the setpoint is decremented or incremented: when $d_i > c \cdot A_i$, more candidates have been demoted than the aperture requires, and the setpoint is decremented. If $d_i < c \cdot A_i$, not enough candidates have been demoted and the setpoint is incremented. Whenever the setpoint is incremented/decremented, d_i is reset and this procedure is repeated. In their evaluation, Sanchez and Kozyrakis [33] use $c = 256$ and re-partitioning takes place every 5 million cycles. As the target partition sizes fluctuate relatively sparingly with a 5 million cycle re-partitioning period, Sanchez and Kozyrakis [33] propose that the apertures as given by Eq. (4.13) are not continuously recalculated, but instead an 8-entry Lookup Table (LUT) is used. This LUT contains how many candidates need to be demoted for 8-ranges of partition sizes. An example of such a table is shown in Table 4.2.

Overall operation

Now that we have seen all the components necessary for a practical implementation of Vantage, we describe the overall operation of a Vantage cache controller utilizing negative feedback, bucketed-LRU and the setpoint-based demotion-scheme.
To operate, Vantage requires that every cache-line has additional state variables. These are given in Table 4.3. Furthermore, Vantage keeps several variables per partition. These variables are given in Table 4.4.

Cache hit The procedure executed by the Vantage controller upon a hit to a block owned by partition i is as follows:

- Set the block's `Timestamp` value to the partitions `CurrentTs`.

Variable	Description
`Timestamp`	Timestamp for use by bucketed lru and the setpoint demotion scheme.
`Partition`	Partition identifier, used to distinguish between several partitions and the unmanaged region.

Table 4.3: Vantage Cache Line State

Variable	Description
`CurrentTs`	The current timestamp of the partition, used to implement bucketed LRU.
`AccessCounter`	Counter to count the amount of accesses made to this partition. Used by the setpoint mechanism.
`ActualSize`	The current actual size of the partition.
`TargetSize`	The target size of the partition.
`SetPointTs`	The setpoint timestamp for the partition.
`Candsseen`	The amount of candidates seen from this partition. Used by the setpoint demotion mechanism.
`CandsDemoted`	The amount of candidates demoted from this partition. Used by the setpoint demotion mechanism.
`ThrSz[8]`	8 partition ranges of the demotion threshold LUT.
`ThrDems[8]`	8 demotion values of the demotion threshold LUT.

Table 4.4: Vantage Partition State

- Increase the partitions `AccessCounter`
- If the `AccessCounter` equals `Actualsize/16`, increment `CurrentTs` and `SetPointTs`. `SetPointTs` is incremented as well to prevent changing the distance from `CurrentTs`. Furthermore, `AccessCounter` is reset.

Cache Miss Upon a cache miss the following procedure is followed:

- For all R replacement candidates offered by the cache, demote a replacement candidate from partition i when `ActualSize` of partition i exceeds its `TargetSize` and the `Timestamp` of the replacement candidate is below the `SetPointTs` (see Figure 4.10). If demoted, tag the partition of the block as unmanaged and set its `Timestamp` value to the unmanaged region's `CurrentTs`. Furthermore decrement partition i's `ActualSize` and increment its `CandsDemoted` value. Regardless of whether a replacement candidate has been demoted, `Candsseen` is incremented.

- The oldest replacement candidate from the unmanaged region is evicted. If no such candidate is present, and all candidates come from the managed region, evict one of the demoted candidates arbitrarily. If this is not the case, one replacement candidate among all candidates is arbitrarily evicted.
- Insert the missing block, tag its partition accordingly and increase the `ActualSize` of the partition that caused the miss. Set the `Timestamp` to `CurrentTs` of the corresponding partition.
- Increase the `AccessCounter` of the partition that caused the miss.
- If the `AccessCounter` equals `Actualsize`/16, increment `CurrentTs` and `SetPointTs`. Furthermore, `AccessCounter` is reset.

Set-point adjustment Whenever `Candsseen` from a partition reaches c, the setpoint of the partition is adjusted:

- The `ThrSize` and the `ThrDems` corresponding to the actual size of the partition is fetched from the demotion threshold LUT.
- If `candsDemoted` exceeds `ThrDems`, `SetPointTs` is decremented.
- If `candsDemoted` is lower than `ThrDems`, `SetPointTs` is incremented
- `Candsseen` is reset.

Allocation policy

Vantage does not include an allocation policy. Instead Sanchez and Kozyrakis [33] make use of UMON's' containing 16-ways and the look-ahead algorithm (see also Section 4.2.1). As Vantage supports partitioning at cache-line granularity instead of at way-granularity, Sanchez and Kozyrakis [33] interpolate the miss-curves from the applications to 256 points to serve as the input for the look-ahead algorithm. In this way the cache can be partitioned at a higher granularity ($1/256th$ of the cache-size instead of 1/16th).

5 | Parallel Zcache Simulation

In this chapter we will give an overview of the most important addition and changes that have been made to the simulation framework to support the zcache, as it is one of the main contributions of this work. Although the zcache is only used to implement Vantage, describing the implementation of Vantage is less interesting as its operation as outlined in Section 4.2.4 can be easily mapped to software. The zcache is introduced from a hardware point of perspective by Sanchez and Kozyrakis [32]. The mapping of parallel hardware to parallel software is not trivial and gives additional insight in how zcaches and set-associative caches interact with each other in the cache-hierarchy.

The cache partitioning simulation framework in use at NTNU is based on Sniper [7], a parallel multi-core simulator. This chapter begins with an overview of Sniper's memory architecture and a global overview of the modifications made to support the zcache architecture. This is followed by a more in-depth discussion of the default locking system present in Sniper that ensures functional correctness of parallel simulation of the caches. We then focus on the new locking system that has been added in order to support the zcache architecture. This chapter ends with a discussion of several design choices that have been made in our implementation with regards to the timing of the zcache.

5.1 Sniper's Memory Architecture

Figure 5.1 is an UML diagram of the (partial) memory architecture present in Sniper. The part of the memory architecture relevant to this thesis is highlighted. We distinguish the following classes:

1. **A Cache controller** (`CacheCntlr`). The cache controller in Sniper models the timing of cache accesses, handles the cache hierarchy and maintains cache coherency. The caches modeled in Sniper are set-associative caches and the cache hierarchy is *inclusive*. This means that all data in a higher-level cache must be contained in the lower level caches. Since every core in Sniper has its own thread for accessing memory, Sniper makes use of a master cache controller (`CacheMstrCntlr`), which is owned by a single core and serves as a proxy for all other cores that are being redirected through a `CacheCntlr` object.

2. **A Cache**(`Cache`). A Cache object class handles accesses to the cache sets in the

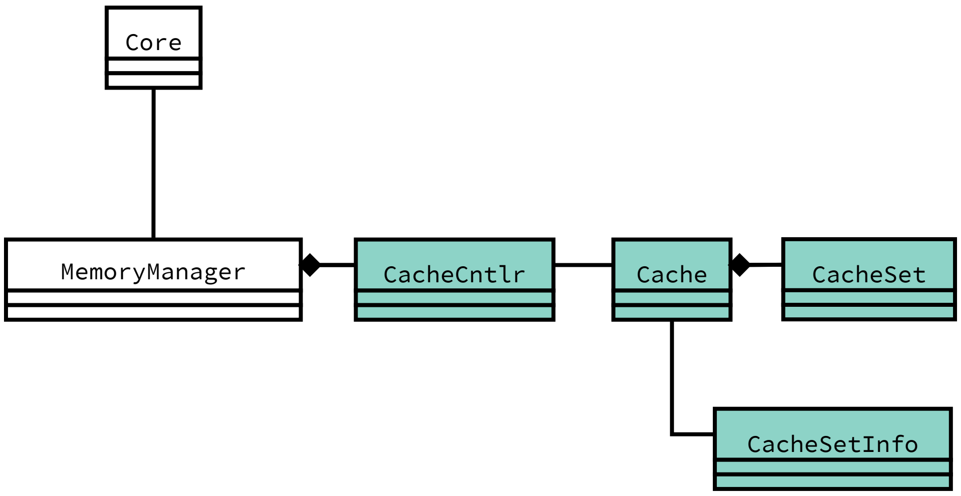


Figure 5.1: Overview of Snipers Memory Architecture

corresponding cache. Given a memory access for address A, the cache object will ensure the request is forwarded to the corresponding cache set.

3. **A Cache-Set** (`CacheSet`). A Cache-set object contains the actual data of a cache set. It handles insertion, eviction and updating of a cache line *within* a cache-set.

4. **A Cache-set-info** (`CacheSetInfo`). A cache-set info object contains data common to all cache sets (such as for example the amount of ways per set).

When implementing cache partitioning algorithms based on a set-associative cache, such as the ones that were already present in the framework, modifications are kept contained to the `CacheSet` and `CacheSetInfo` classes. Sniper's memory architecture as-is can not be used for zcaches, as modifications due insertion, replacement, demotion and promotion of cache lines are not kept contained to one set as modeled by Sniper. In our implementation we have modified *all* highlighted classes:

1. The Cache controller includes a new locking system to handle concurrent accesses to the LLC. This is to ensure functional correctness.

2. The Cache objects has been modified to support indexing on a cache line granularity, rather than on a set-granularity.

3. A CacheSet has been derived specifically for the zcache, containing code that handles the BFS-walk and the relocation process.

4. A CacheSetInfo object has been derived specifically for the zcache, containing the necessary status variables to allow the CacheSet object access to any set in the cache, as is required by the BFS-walk and relocation process.

Of all modifications, the new locking system will be looked at in more detail: this system is the crucial component in the mapping of the zcache from hardware to software and is required to achieve functional correctness.

5.2 Functional Correctness: Locking in the LLC

Since the LLC is shared by several software-threads, Sniper makes use of a locking system to ensure functional correctness. As operations on the LLC by one core can affect the contents of private caches of other cores, it is not enough to just protect the LLC. If for example, core a evicts a block x from the LLC, then due to inclusiveness of the LLC, x is invalidated in the private caches of all cores containing x. If x was concurrently operated on in a private cache by another core b, undefined behavior might occur as result of a race condition.

To ensure correctness of operation it is therefore necessary that:

1. Concurrent accesses to the same set in the LLC are prevented, such that race conditions due to sharing of data in the LLC cannot occur.

2. Memory accesses to sets in private caches that are affected by an operation on the LLC, are prevented from executing concurrently with the operation in the LLC.

Since the cache architecture in Sniper is inclusive, in the worst case, changes to the LLC can propagate up to the L1-level. One solution would therefore be to have *one* lock for all cores sharing the LLC. Upon starting a memory operation, which starts at the L1-level, a core acquires this lock in the shared mode. When an operation is about to operate on the LLC, the lock is upgraded to an exclusive lock, preventing all other cores from continuing their memory operations. However, this system is too conservative, as it does not allow other cores to access data in their private caches that is not affected by the current operation on the LLC.

To determine which sets in the caches in the levels above the LLC are possibly affected by a memory operation in the LLC, Sniper assumes that [1]:

1. All caches employs modulo-based indexing;

2. All caches at a certain level have the same amount of sets;

3. The amount of sets in a cache is a power of 2;

4. The amount of cache-sets increases monotonically as one moves down the cache-hierarchy.

Let $a = 2^x, x \in \mathbb{N}^+$ denote the amount of sets in the L1-caches. The set i that is possibly affected in the L1-cache of any core sharing the LLC by an operation in the LLC is given by:

$$i = \mod (A_b, a),$$

[1] These assumptions have been extracted from the source-code.

with A_b the address the block being operated on in the LLC. Let $b = 2^y, y \in \mathbb{N}^+$ denote the amount of sets in the L2-caches. If two blocks with addresses $A_{b,1}$ and $A_{b,2}$ map to the same set $c \in \mathbb{N}$ in the L2 cache, then their block addresses can be written as

$$\begin{aligned} A_{b,1} &= \alpha \cdot b + c & \alpha &\in \mathbb{N} \\ A_{b,2} &= \beta \cdot b + c & \beta &\in \mathbb{N}, \end{aligned}$$

with c the set they map to in the L2-cache. As $b \geq a$ and both a and b are powers of 2, b is divisible by a. This means that we can write:

$$\begin{aligned} A_{b,1} &= \delta \cdot a + c & \delta &\in \mathbb{N} \\ A_{b,2} &= \kappa \cdot a + c & \kappa &\in \mathbb{N} \\ &\implies \\ \bmod\ (A_{b,1}, a) &= \quad \bmod\ (A_{b,2}, a)\,. \end{aligned}$$

Therefore, all addresses that map to the same set in the L2, map to the same set in the L1 cache[2]. By applying this argument recursively, one can reason that this holds true for the LLC[3] as well: competing memory operations in the LLC map to the same set in the L1-cache.

Snipers locking system keeps an array of locks with the amount of locks equaling the amount of sets in the L1-cache in the master cache controller. Whenever a memory

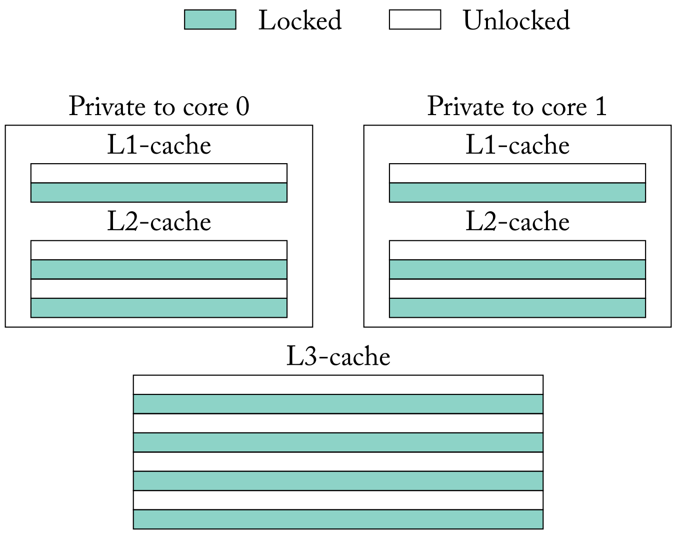


Figure 5.2: Default Locking System

[2]This is not necessarily the case when the amount of sets does not monotonically increase or is not a power of 2.

[3]in case the LLC does not reside at the L2-level.

operation is starting, the corresponding simulator thread acquires the lock (through the master controller) corresponding to the affected L1-set in the shared state. As soon as an operation moves down from the L1-level, the lock is acquired in the exclusive state. Since competing operations in the LLC affect the same set in the L1-caches, concurrent accesses to a set in the LLC cannot happen.

An example showing the sets locked due to an operation by core 0 in the LLC is illustrated in Figure 5.2 for a 2-core CMP. It might seem that this system locks more sets in the L2 and L3 cache then required: after all, there is only *one* set in the L2 and L3 cache that corresponds to the address of the memory operation issued. However, the system needs to restrict access at the *L1 level*. The sets in the L2 and L3 cache, that need not necessarily be locked, cannot be accessed by any other core anyway: as those sets maps to the same L1-set, this would violate the mutual exclusivity of the lock.

5.3 Hash-Locking System

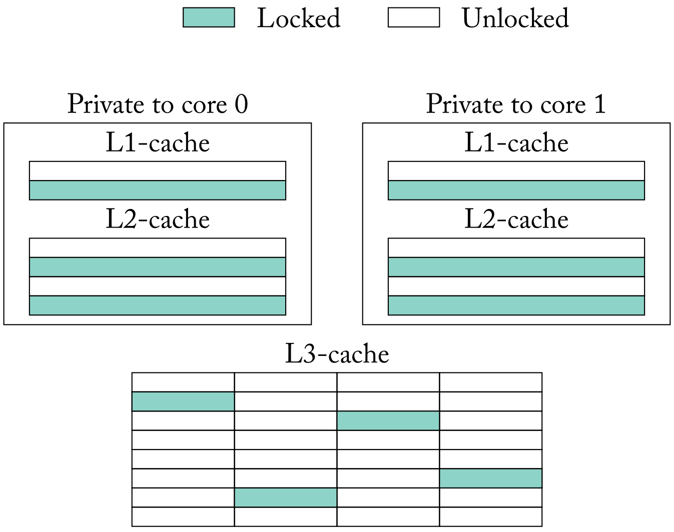


Figure 5.3: Default + Hash Locking System

When the LLC is indexed on a cache line basis rather than on a set-basis, as is the case for zcaches, the default locking system in Sniper cannot handle concurrent memory accesses in the LLC correctly. To allow for non-conflicting concurrent operations in the L1-caches, it is necessary that a locking system that can lock on cache line granularity in the LLC is present. In this work, only the LLC employs indexing at cache line granularity. This means that the default locking system will be used to protect the caches at all higher levels.

A first and intuitive step towards the new locking system would be to acquire the locks in the LLC corresponding to the hash of the block-address A_b in each way i. A cache hierarchy using this new system in combination with the default locking system is illustrated in Figure 5.3. We will refer to this new locking system as *the hash-locking system*. While it correctly prevents concurrent operations to the same cache lines in the LLC, the system does leave opportunity for race conditions to occur.

5.3.1 Race Conditions

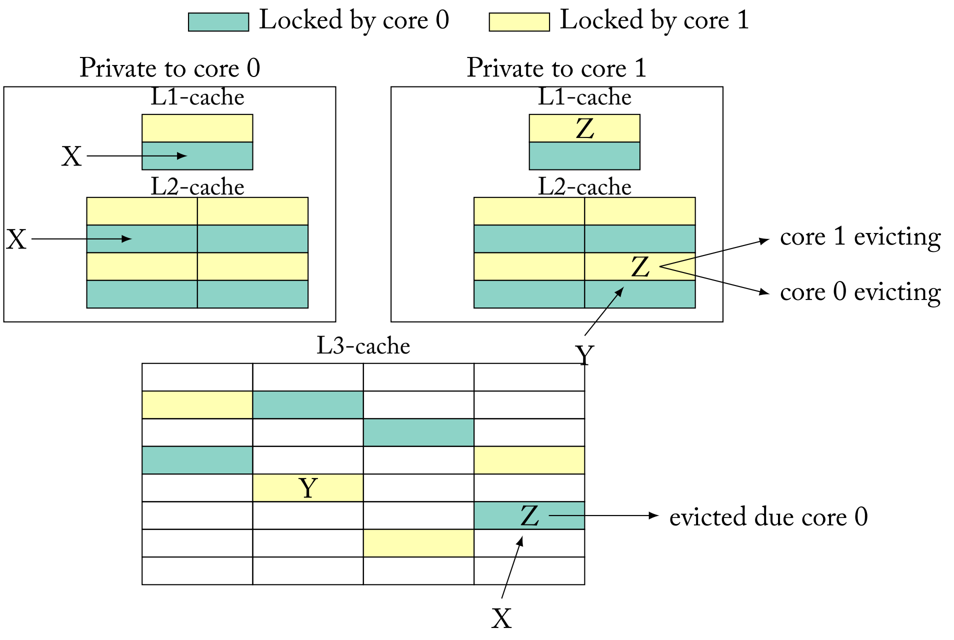


Figure 5.4: Example of a Race Condition in LLC

One problematic example of a race condition that can occur in a 3-level cache hierarchy with a zcache as LLC is depicted in Figure 5.4. In this example, a cache miss to a block y results in eviction of the cache line z from the L2-cache of core 1. Concurrently, core 0 has initiated a replacement process in the LLC to insert a missing block x and is about to evict z from the LLC. In this example the hash-locks and default locks corresponding to x and y have been acquired. However, as y and x do *not* map to the same sets in the L1, nor do they have any hash-locks in common[4], the two operations are allowed to run concurrently by the default- and the hash locking system. Now, either core 1 evicts z from its L2-cache or core 0 does so to maintain inclusiveness of the caches. However, in both cases both cores are expecting z to be present in the L2 cache of core 1.

With a set-associative cache, all (side-)effects of a memory request are confined to *one* set at every cache-level. When the LLC is a zcache, this no longer holds true: the hash- instead of modulo-indexing breaks this property and multiple sets in the set-

[4]This is possible due to the random nature of H3-hashing.

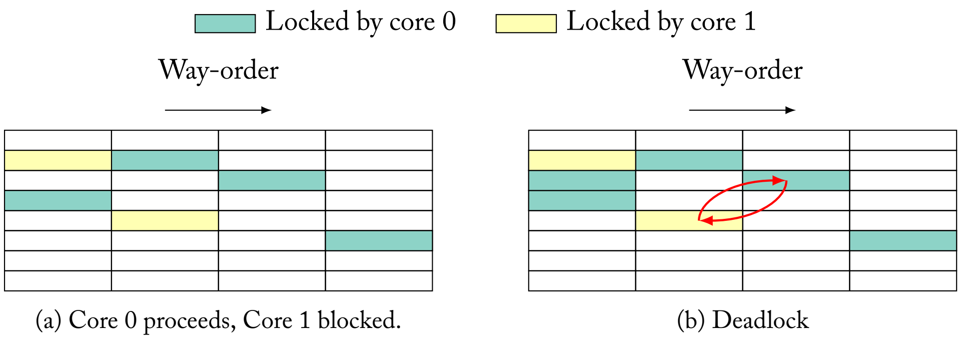


Figure 5.5: Deadlock upon Replacement in the LLC

associative caches in the levels above the LLC might be affected, as we have seen in the example in Figure 5.4. This is the origin of the race conditions that can still occur.

To prevent those types of race conditions, it is necessary that upon an *eviction* in the LLC, the path $\mathcal{P}$ to the replacement candidate is inspected. Any hash-locks associated with the cache lines on $\mathcal{P}$ that have not been acquired yet need to be acquired. Furthermore,[5] the tag of all blocks on $\mathcal{P}$ need to be inspected to find out to which set in the L1-cache they map to, such that their corresponding lock in the default locking system can be acquired. In our work we employ zcaches with 3-levels of hashing. This means that the length of $\mathcal{P}$ is *at most* 3 (but can be shorter), restricting the additional overhead of acquiring the additional locks.

5.3.2 Deadlocks

So far, we have only focused on how to prevent race conditions to ensure correctness of operation when using hash-locking. However, the order of locking is essential to prevent *deadlocks*. As depicted in Figure 5.3, upon issuing a memory operation, the hash-locks corresponding to the issued address are taken. To prevent the most basic form of a deadlock, a certain *order* needs to be present in how the hash-locks are acquired. In our implementation we use the intuitive ordering by way: that is, the hash-locks are taken in increasing order of the way of the cache they are located in.

However, the hash-locking system that locks in way-order is not completely eliminating the possibility of a deadlock situation. The reason is that upon a miss, the zcache performs a relocation process. To ensure correctness of operation, it is necessary that all replacement candidates that lie on the path $\mathcal{P}$ of the eviction candidate are locked as well (see Section 3.2.1). This means that possibly more locks (both in the hash- and default locking system) need to be acquired upon eviction of a cache line. However, before starting the corresponding memory operation at the L1-level, it is not known yet whether the access will actually result in a miss in the LLC and whether (and which) additional cache lines need to be locked. Only when the operation is started and the

[5]This is what is essential to prevent the race conditions.

LLC-controller notices the item is missing, the BFS-walk to discover replacement candidates in the zcache is performed.

Figure 5.5a depicts two cores that are about to start their memory operation at the L1-level. The way-order in this picture is from left to right. In this particular situation, core 0 and core 1 are competing for access to the same cache line in the third way, and core 0 has acquired the hash-lock corresponding to this cache line first. This means that the thread corresponding to core 0 can continue execution, leaving core 1's thread to wait. Core 0 proceeds its memory operation and finds that the requested item is missing in the LLC. However, it might be the case that one of the replacement candidates found on $\mathcal{P}$, is in fact a candidate already locked by core 1. This is depicted in Figure 5.5b, indicating a circular dependency between core 0 and core 1: core 1 is waiting for core 0 to release its lock in the third way, whereas core 0 is waiting for core 1 to release its lock in the second way, resulting in a deadlock.

When a replacement process is performed, the block to be inserted is *missing* in all caches[6]. Therefore, no core can modify the block or is expecting the block to be present in any cache. This makes it possible for a core to temporarily release *all* its acquired locks upon entering the replacement process and start executing the BFS-walk. This allows other cores to access the blocks visited by the BFS-walk in LLC concurrently. This corresponds to the benign race conditions as described in Section 3.2 and is therefore functionally correct. Once all locations in the LLC that need to be locked are discovered by the BFS-walk, first *all* hash-locks required are acquired (in way-order). Once this is done, the tags of the cache lines on $\mathcal{P}$ to the selected replacement candidate can be safely inspected. Subsequently, the additional locks of the default system can be locked as well. If the tags in the LLC are inspected before acquiring the hash-locks, it might be the case that the incorrect locks in the default locking system are acquired.

However, upon releasing all locks, it becomes possible for another core to proceed and execute its memory operation. If that operation is a miss to the same block, then once control returns to the core that initially released all its locks , this core assumes the block to be missing while it might already be present. Preventing this from happening would complicate the locking system more than necessary: our simulation framework has been developed to make use of multiprogrammed workloads where sharing of data does not occur and therefore this type of situation cannot occur. To even observe this issue, we would need to significantly change our evaluation methodology. These changes would not benefit our cache partitioning investigation in any way and has therefore not been prioritized.

5.4 Timing Aspects

During the implementation of the zcache, we have neglected the delay possibly incurred by the blocking of other memory operations during relocation. First of all, in our evaluation we employ a zcache with 3 levels: this results in only 3 additional blocks in the LLC to be locked for relocation. With a cache-size of at least 8MiB in our evaluation, this means that at most $\approx 2.29 \cdot 10^{-3}\%$ of LLC is subject to this blocking; the chance

[6]assuming an inclusive-cache as modeled by Sniper.

of outstanding memory operations on those locations is negligible. We have therefore chosen to ignore the timing aspect of this blocking.

Furthermore, Sniper allows the threads that simulate the cores to run unsynchronized during a certain period of time. This means that every core has its own local time, and the ordering in which memory accesses are simulated does not necessarily correspond to an ordering based on the times that these accesses took place. Moreover, every cache access is modeled functionally: that is, their effect takes place immediately and the local core-time is increased by the delay of the operation. It is therefore impossible to block or revert such an operation once it has started.

6 | Methodology

In this chapter we will outline our methodology as used to evaluate the performance of the researched cache partitioning algorithms.

Performance evaluation can be divided into four sub-components, of which the first component to be discussed in this chapter is the target architecture. Second are the performance metrics that allow for meaningful comparison and interpretation of the raw performance data. The third component described is the evaluation method of the target architecture. This is followed by the fourth component, the method used to create the multi-core workloads used in our experiments.

The chapter is concluded with the setup of our main experiment including several details regarding the used cache partitioning algorithms.

6.1 Target Architecture

In our evaluation we target an architecture based on the Intel Nehalem Architecture. In total we target three CMPs having 4, 8, respectively 16 cores. All CMPs utilize a 3-level inclusive cache hierarchy. Table 6.1 lists the overall configuration details of these CMPs. Since we target CMPs with different core counts, we double the LLC size as the core-count doubles, similar to other cache partitioning studies [22, 18, 17]. The amount of ways in the set-associative LLC is set to 16. This is the default in Sniper to model the LLC of the Nehalem Architecture. Furthermore, 16-ways are common in modern consumer CMPs [11] and other similar studies on cache partitioning [41, 18, 17]. When a zcache is used as LLC (as required by Vantage), we use a zcache with 52 replacement candidates as [33]. Furthermore, the latencies associated with the zcache are the same as those used for the 16-way set-associative cache, again following Sanchez and Kozyrakis [33]. This is conservative with respect to the zcache, as it is able to provide lower latencies than a 16-way set-associative cache [32].

We do not change the cache-latencies of the LLC as its size increases, because we are interested in the effects of cache partitioning. Changing the latencies as well would obscure those effects. Moreover, all introduced cache partitioning algorithms add certain delays to the cache, but those are not specified and assumed to be negligible in the corresponding literature. Accurately determining the latencies of the LLC would therefore still lead to an approximate overall latency when the cache partitioning algorithms are used. To have feasible LLC latencies, we use the default cache latencies of

Processor cores	2.66GHz, OOO, dispatch width=6, commit width=4, x86-64 ISA
L1 I-cache	32 KiB, 4-way set-associative, 64-byte cache lines, private, LRU-managed, 1 cycle tag access time, 4 cycles data access time
L1 D-cache	32 KiB, 8 way set-associative, 64 byte cache lines, private, LRU-managed, 1 cycle tag access time, 4 cycles data access time
L2 cache	256 KiB, 8-way set-associative, 64 byte cache lines, private, LRU-managed, 3 cycle tag access time, 8 cycles data access time
L3 cache	8/16/32MiB (for 4/8/16 cores), 16-way set associative or Z4/52, 64 byte cache lines, shared, varying cache partitioning algorithm, 10 cycle tag access time, 30 cycles data access time
Branch-predictor	Pentium-M based
Coherence protocol	Modified-Exclusive-Shared-Invalid (MESI)
DRAM	45ns latency, 51.2 GB/s bandwidth,
Miss Status Holding Register (MSHR)	8 entries at each cache level[1]

[1] This is a limitation of Sniper [15]

Table 6.1: CMP Configuration Details

Sniper corresponding to an 8MiB 16-way set associative cache.

To ensure that the performance of our CMP configurations is not unrealistically restricted by their bandwidth to DRAM, we choose to model the bandwidth of a modern DDR3 controller supporting 4 channels and DDR3-1600MHz, as used by 4,8 and 10-core Xeon processors [10]. This results in a DRAM-bandwidth of 51.2GB/s for all CMPs.

6.2 Performance Metrics

Evaluating the performance of single threaded benchmarks is well understood and the appropriate performance metric is time [13]. Under the 'Iron Law of Performance', the time T taken by a single-threaded program executing N instructions is given by

[13, 37]:

$$T = N \times CPI \times \frac{1}{f}, \tag{6.1}$$

where Cyles per Instruction (CPI) stands for the amount of *useful* instructions executed per cycle and f is the frequency of the microprocessor used to execute the program. CPI is a function of the microarchitecture of the processor and can therefore be used to asses the performance of different microarchitectures [13]. Instructions Per Cycle (IPC), the reciprocal of CPI, indicates how many useful instructions are executed on average per cycle. IPC and CPI are popular metrics in single-core performance evaluation, as they can be easily quantified using architectural simulation [13].

However, once the performance of applications running on a CMP needs to be assessed, IPC and CPI are not adequate anymore: Eyerman and Eeckhout [14] reason that their use skew the results in case of prolonged running time due to synchronization mechanisms. Furthermore, when multiple programs run together, they interfere with each other and overall performance is the result of a balance between single-program and overall system performance [14].

Eyerman and Eeckhout [14] reason from a system-perspective and suggest the use of System Throughput (STP) for multi-core performance evaluation. When n applications are running on a multi-core, the STP is defined as [14]:

$$STP = \sum_{i=1}^{n} \frac{IPC_i^{MP}}{IPC_i^{SP}}, \tag{6.2}$$

where IPC_i^{MP} and IPC_i^{SP} are the IPC of application i running in the presence of all other $n-1$ applications and the IPC of application i running in isolation respectively. Therefore, the STP is equal the sum of all individual speedups.

Another prevalent metric in assessing multi-core performance is the Harmonic Mean of Speedups (HMS) [13]:

$$HMS = \frac{n}{\sum_{i=1}^{n} \frac{IPC_i^{SP}}{IPC_i^{MP}}}. \tag{6.3}$$

HMS generally results in lower values than using the arithmetic mean of speedups (which equals STP/n), when one or more applications have a lower IPC speedup [14]. This is reasoned to capture the fairness of the several speedups more fairly. If for example, one application experiences a major speedup, whereas all other applications experience a small slowdown, the speedups are unbalanced. While the arithmetic mean of the speedups might indicate good performance (as one application is sped up significantly), the harmonic mean tends to result in a lower value indicating that the speedups are relatively unfair. To evaluate the performance of the workloads both from a system- and user-perspective, we use both the STP and the HMS in our evaluation.

6.3 Evaluation of the Target Architecture

Quantitatively evaluating the performance of the target architecture can be done by using a simulation model, by imitating the target architecture (emulation) or by constructing the prototype of the architecture in real hardware. Prototyping and emulation require that all details about the target architecture are known. Simulation on the other hand, abstracts most of the details away, and only certain parameters need to be set. Furthermore, simulation allows for quick adjustments of the architecture and makes it relatively easy to add new components to the modeled system. Moreover, quantitative evaluation using simulation is an accepted method and in fact has become a mainstay in computer architecture research [38].

For these reasons, we use a simulator in our evaluation framework. The simulator of choice is Sniper, a parallel multi-core simulator [7]. Due to Snipers parallel nature, it suffers from a causality problem [6]: Sniper divides its simulation into several time-quanta of duration Q. During each quantum, all simulation-threads run without synchronization, to synchronize at the end of the quantum. This means that memory accesses in one thread can functionally be simulated at the same (wall-)time as at another core, but still carry a different time-stamp, potentially causing them to be out of order. As Sanchez and Kozyrakis [34] state, at a small timescale, say $\approx$ 1000 cycles, most concurrent access to the cache happen to unrelated lines. Breaking the order of those accesses, but simulating their timing in a detailed fashion, is approximately equivalent to simulating them in order. To limit the inaccuracy of simulation we proceed using a quantum of 100ns (approximately 266 cycles, which is well below 1000), which has been shown to provide an effective trade-off between Sniper's simulation speed and accuracy [24].

To significantly reduce the simulation time while retaining representative performance estimates, we adopt the SimPoint methodology [27]. The idea is that every program, although having varying behavior over time, can be divided into several representative intervals. This representative behavior is represented by Basic Bit Vector (BBV)'s, which capture the relative frequency of the execution of code blocks during a given portion of execution. Once a program is profiled to obtain the BBV's, the BBV's are compared to each other for similarity. Intervals executing the same code blocks with the same frequency are clustered using machine-learning algorithms, resulting in a k-clustering of BBV's. One point, a *SimPoint*, is selected per cluster and weighted to serve as a representative for the program. To obtain an estimate of the performance of a program, it is now only necessary to simulate k intervals instead of the whole program.

Due to Snipers dependency on and integration with the Pin-instrumentation tool [20], Pinpoints, a SimPoint implementation of SimPoints for Pin has been used. We follow Patil et al. [26] and make use of SimPoint-intervals that are 250 million instruction in size to ease warm-up effects of the caches, as result of starting with empty caches. For each benchmark we generate one SimPoint, thereby simulating only one interval. The reason for doing so, is that different intervals of a benchmark can have different demands on the LLC. This means that when one simulates multiple intervals, discontinuity between demands on the LLC needs to be taken into account, which is non-trivial. Furthermore, we are primarily interested in the difference in performance between cache partitioning algorithms. To observe those differences, intervals exhibiting different characteristics with regards to cache usage are required, but accurate estimation of the complete individual benchmark performance is not necessary.

insensitive	medium-sensitive	higly-sensitive
`GemsFDTD`	`hmmer`	`sphinx3`
`leslie3d`	`bzip2`	`gamess`
`gromacs`	`sjeng`	`xalancbmk`
`libquantum`	`tonto`	`povray`
`zeusmp`	`wrf`	`gcc`
`calculix`	`perlbench`	`soplex`
`bwaves`	`cactusADM`	`lbm`
`namd`	`h264ref`	`omnetpp`
`gobmk`	`astar`	
`dealII`		
`milc`		

Table 6.2: Benchmark classification

6.4 Workload Generation

The benchmarks used in our simulations are those from the SPEC CPU®2006 benchmark suite, used by other similar recent cache-management studies [32, 41, 22]. Our workloads have been generated based on a cache sensitivity classification. This section will start with the method of classification, followed by how the classified benchmarks have been combined into workloads.

6.4.1 Benchmark Classification

In order to profile the benchmarks for their sensitivity to changes in allocated cache-space and associativity, we proceed as follows:

1. For each benchmark, we profile it using Sniper, while increasing the way count (1,2,4,8,16,32) of a set-associative LLC. The configuration of the architecture used is the same as that of the 4-core CMP in Table 6.1, except for the changing LLC and the use of one core. The LLC has a fixed amount of sets per way (8192) and a block-size of 64 bytes. Changing the way-count implies an increase in cache size and increase in associativity (reduction in conflict misses). While increasing the way count, the tag and data delays of the LLC are kept constant: in this way a cache of 16MiB is simulated that is being partitioned (with a way as partitioning unit).

2. For every benchmark the speedup based on the IPC is calculated.

3. All benchmarks with a speedup up to 1.3 are classified as *insensitive* (class *i*). Benchmarks having a speedup above 1.3 up until 1.5 are classified as medium sensitive (class *m*). The rest of the benchmarks is classified as highly sensitive (class *h*).

Using this method, the benchmarks have been classified as listed in Table 6.2. Although strictly speaking it is not necessary to profile the benchmarks for all listed way-counts to calculate their maximum speedup, doing so *does* give additional insight in the behavior of the used benchmarks. We have therefore included those profiles in Figure 6.1. The cache sensitivity profiles in Figure 6.1 also include the Misses-Per-Kilo-Instructions (MPKI) (in red), as this metric gives additional insight in the memory-access patterns of the benchmarks. MPKI is defined as the amount of misses per 1000 (kilo) instructions and therefore gives an indication of how many misses, on average, are incurred by a benchmark. These profiles are used later on to discuss the performance of the evaluated cache partitioning algorithms in Chapters 7 to 9.

6.4.2 From Classification to Workloads

Based on our classification, we generate our workloads for 4,8 and 16 cores, using an approach similar to that of Sanchez and Kozyrakis [33]. The 4-core workloads are generated as follows:

1. All unique combinations of the workload-types (*i*, *m*, *h*) are generated. This results in 15 classes. Each of those classes, containing four benchmarks, is represented by a tuple of 4 characters, indicating the sensitivity classification of the corresponding benchmark. `iimh` for example, refers to a class in which two benchmarks are cache-insensitive, one medium sensitive and one highly sensitive.

2. For each class, five workloads are generated by randomly selecting benchmarks corresponding to the types as given by the classes (repetition allowed). This results in 75 workloads in total. The workloads that have been generated can be found in Table A.1.

Following a similar procedure for the 8 and 16-core workloads would result in significantly more classes (45 and 153 for 8- respectively 16-core workloads). We, again, follow an approach similar to that of Sanchez and Kozyrakis [33]. For the 8- and 16-core workloads, our classes are the same as for the 4-core workloads. For example, a workload from class $iimh$ is generated by randomly selecting 2, respectively 4 benchmarks from the i,i,m and h benchmarks. This procedure is performed 5 times per class resulting in 75 workloads in total. To be able to evaluate more benchmarks and due to a limited computing budget, we have excluded `mcf`. We found that the workloads containing `mcf` could increase the average running time of a workload by more than 400%. The 8- and 16-core workloads that have been generated can be found in Tables B.1 and C.1.

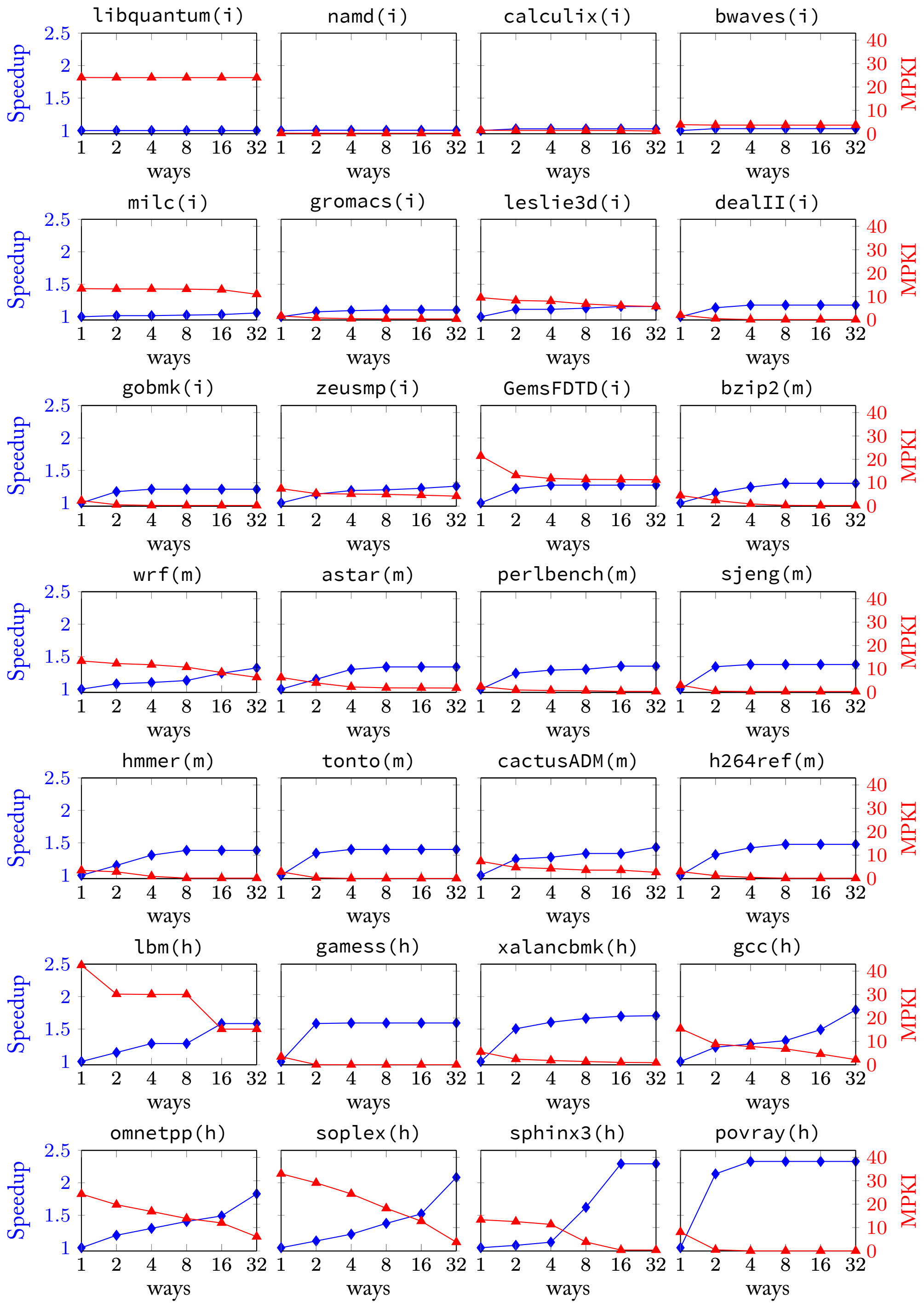


Figure 6.1: Cache Sensitivity Profiles - A way corresponds to 0.5MiB

6.5 Main Experiment

Our main experiment is set up to evaluate the performance of eight cache partitioning algorithms. We use conventional LRU on a set-associative cache as our base-line. The evaluated algorithms, together with their used parameters are shown in Table 6.3. To find out what the added effect of using Vantage over using a zcache is, we use Z-LRU (bucketed-LRU) as an additional reference of LRU for Vantage. PriSM-UCP represents the PriSM partitioning scheme combined with the allocation policy of UCP.

The parameters as shown in Table 6.3 are based on the (tuned) parameters used in the literature that the corresponding algorithms have been introduced in. All algorithms that use UMON's, are using UMON-global without DSS. When DSS is used, only few sets of the MTD are duplicated and sampled in the ATD's. As our workloads contain different benchmarks, each having different memory access patterns, the accuracy of DSS can differ among workloads. To eliminate this dependency, we have opted to not use DSS.

Algorithm	**Parameters**
Z-LRU	bucketed-LRU, 8-bit timestamps, $k = 5\%$
TADIP-F	32 dueling sets per 4MiB of LLC, $\epsilon = \frac{1}{32}$
DRRIP	3-bits RRPV's, 32 dueling sets per core per 4MiB cache. $\epsilon = \frac{1}{32}$, FP-promotion policy.
PIPP	$p_{prom} = \frac{1}{32}$, $p_{stream} = \frac{1}{128}$, $\theta_{mr} = 0.25$, UMON-global utilizing 16 way UMON's, minimum insertion position of 8 positions above LRU. 5M cycle partitioning period
PriSM-H	W = number of cache lines.
PriSM-UCP	UMON-global utilizing 16 way UMON's, W = number of cache lines
UCP	UMON-global utilizing 16 way UMON's, 5M cycle partitioning period
Vantage	$u = 0.05$, $A_{max} = 0.5$, $s = 0.1$, $c = 256$, $k = 5\%$, UMON-global utilizing 16 way UMON's, 5M cycle partitioning period

Table 6.3: Evaluated Cache Partitioning Algorithms and Their parameters

Each algorithm is evaluated over the three CMP configurations as described in Section 6.1 using the workloads listed in Tables A.1, B.1 and C.1. We set up our simulations to restart a benchmark once it completes its execution, until at least every benchmark in the workload has completed once: this is to ensure that cores keep competing for space in the LLC. This might involve situations where several benchmarks have been partially executed once the simulation ends. Furthermore, when a benchmark is repeated, several blocks of its working set might already be present in the cache, affecting its performance.

We have therefore modified Sniper to support multiple runs of a benchmark in private mode. Using this modification, we have traced the IPC of every benchmark over intervals of 1 million instructions (0.4% of the 250M cycle interval used by the SimPoints) during a run including 20 repetitions in isolation [1]. This enables us to calculate the STP and HMS more accurately: instead of calculating those metrics based on the average IPC of a benchmark of one run in isolation, we can average the IPC over the actual interval that has been executed for the benchmark and use this value in our calculations.

6.6 Several Details Regarding the Framework

This section is concerned with several remarks regarding the implementation of the cache partitioning algorithms in our framework.

PIPP PIPP has been introduced using DSS with a miss-count threshold $\theta_m = 4095$ and a miss-rate threshold $\theta_{mr} = \frac{1}{8}$. The use of miss-counts to detect streaming benchmarks has been shown to function poorly in our framework, and is therefore not present in the simulation framework [24]. Furthermore θ_{mr} has been set to $\frac{1}{4}$ in the framework [24].

PriSM As already indicated by Olsen [24], the use of Equation (4.6) to calculate the eviction probability distribution $\mathcal{E}$ can result in all eviction probabilities summing to a value higher than one. In that case, $\mathcal{E}$ no longer represents a probability distribution.

This can be explained as follows: in steady state, where the cache has been filled completely with data $\sum C_i = \sum T_i = \sum M_i = 1$ and the use of Equation (4.6), without clamping, results in $\sum E_i = 1$:

$$\begin{aligned}\sum E_i &= (\sum C_i - \sum T_i) \times \frac{N}{W} + \sum M_i \\ &= 0 \times \frac{N}{W} + 1 \\ &= 1\end{aligned} \tag{6.4}$$

However, when clamping is applied, Equation (6.4) no longer holds and $\sum E_i = 1$ does not necessarily hold. The same is true in case the cache is not completely filled yet, as $\sum C_i < 1$. During this thesis we have therefore altered the simulation framework to use $\mathcal{E}$ differently. To be able to select a victim-core, we propose that the selection algorithm is altered to normalize all E_i's when $\sum E_i > 1$, otherwise leaving all E_i's intact. The reason for doing so is that if $\sum E_i \leq 1$, normalization would result in an increase of all non-zero eviction priorities, which conflicts with Equation (4.6). When $\sum E_i > 1$, normalization is required, since PriSM prescribes that *one* victim core is selected, *before* the replacement policy selects a candidate. Normalization does not alter the proportionality between all non-zero priorities.

[1]During our experiments, the maximum amount of repetitions of any benchmark never exceeded 20.

UCP & Inclusive Caches As we use an inclusive caching architecture in our evaluation, the look-ahead algorithm as implemented in the framework allocates at least the the size of the L2-cache in the LLC to each application (see also Section 4.2.1).

Since Vantage can partition at cache-line granularity rather than per way, the miss-curves as required by the look-ahead algorithm are interpolated to contain 256 points (see Section 4.2.4). Since UMON's work with hit counters, miss curves cannot directly be obtained: this would require information about the amount of misses that would occur when no cache-space is allocated to a specific core. In the case of UCP and PIPP this is no problem, as they can directly use the hit-counters to calculate the partition targets (see Equation (4.3)).

To obtain the miss-curves, a counter per core that counts every access to the cache could be added. The amount of misses can then be calculated by subtracting the values from the hit-counters from the access counter.

We have implemented another method in the framework: as the look-ahead algorithm allocates cache on basis of *utility*, that is, how many cache misses can be saved, knowing the absolute amount of misses is not required. It is sufficient to have a miss-curve that differs from the *real* miss curve by just an offset. One can create such a curve using Algorithm 1.

Algorithm 1 getMisses

```
1: procedure GETMISSES(way)
2:     misses=0
3:     for i=N-1; i>way; i- do                ▷ Assuming zero-based indexing
4:         if way < i then
5:             misses += hitCounters[i]
6:     return misses;
```

7 | Results

This chapter describes the results of the main experiment as described in Section 6.5. We note that all performance results are *estimates* of the SPEC benchmarks based on simulation of representative regions of all used benchmarks (PinPoints).

7.1 Overall HMS and STP

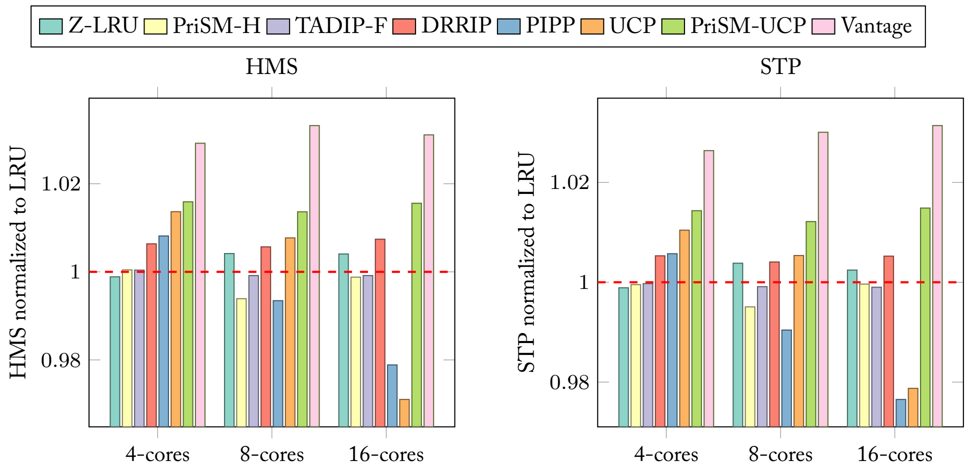


Figure 7.1: Overall HMS & STP for all Workloads and all Core-Counts

Figure 7.1 shows the average HMS and STP of all workloads normalized to LRU for the 4-,8- and 16-core CMP configurations. Our first observation is that the trends in the STP generally follow the trends in the HMS. This indicates that the speedups of the individual benchmarks are relatively in balance with the overall speedup of the workloads. As all algorithms target Hit-maximization, it is unlikely that one algorithm would attain a significantly more fair distribution of speedups than another. The HMS and STP of all algorithms both deviate at most approximately 3% from LRU.

From Figure 7.1, we further observe that Vantage is the best performing cache partitioning algorithm both in terms of the HMS and the STP. Vantage achieves an overall

HMS and STP that are approximately 3% higher than that of LRU for all core-counts. Moreover, as the core-count increases, the improvements in STP experience a minor *increase*. Furthermore, we observe that the differences in performance compared to Z-LRU are significant: for all core-counts, Vantage attains an overall STP and HMS that are more than 2% higher than those of Z-LRU. Z-LRU is an implicit cache partitioning algorithm and partitions the cache efficiently when the overall memory access pattern of a workload is recency-friendly. However, the overall memory access pattern of a workload is a composition of the memory access patterns of the individual benchmarks and is therefore not recency-friendly in general. This leads to the limited performance improvements of Z-LRU.

This observation can be made more general: Figure 7.1 shows that explicit cache partitioning algorithms, although not always scalable to higher core-counts, are able to attain more significant performance improvements than the implicit cache partitioning algorithms. This can be explained as follows: the evaluated implicit cache partitioning algorithms TADIP, Z-LRU and DRRIP address the memory access pattern of each core separately[1]. Possible destructive interference due to the interleaving of memory access patterns of several cores sharing the LLC is not addressed, thereby limiting the performance. By explicitly managing the cache space, as the explicit partitioning algorithms do, isolation can be provided to the cores. In turn, this can lead to increased performance.

PriSM-UCP is the best performing algorithm after Vantage, improving both STP and HMS with at least 1% with respect to LRU. Remarkable is the performance gap observed between PriSM-H and PriSM-UCP: PriSM-H, unlike PriSM-UCP, does not improve overall STP at any core-count, nor does it significantly improve the HMS with respect to LRU. As both algorithms use the same partitioning scheme, differences in observed performance need to be attributed to differences in the allocation policies. We will provide a more extensive analysis in a case-study (see Section 8.1).

We further observe that UCP provides improvements in HMS and STP of approximately 1% for the 4-core CMP configuration. As soon as the core count doubles, performance improvements drop significantly and for the 16-core CMP configuration, UCP is not anymore able to improve upon LRU. The reason for this observed performance drop, is that the granularity at which UCP partitions the cache is too coarse. As the core counts increase, the amount of ways of the set-associative cache stays constant in our evaluation (16 ways). UCP needs to allocate one way to each core due to inclusiveness of the cache hierarchy (see Section 4.2.1). For the 16-core configuration, this leaves no ways left to allocate and all partitions are of equal size. At 4 and 8 cores UCP is still able to allocate varying amount of ways to each core, which allows for a better distribution of the cache space among the cores, improving performance with respect to LRU.

The performance of PIPP shows a pattern similar to that of UCP. PIPP, like UCP allocates partitions on a per-way basis, which is too coarse for higher core-counts. Unlike UCP, PIPP cannot provide improvements with respect to LRU from 8-cores on-

[1]Although TADIP-F and DRRIP use a feedback mechanism to take the optimal insertion policy of other cores into account while determining the best policy for a certain application, they do not globally consider all access patterns combined to optimize performance.

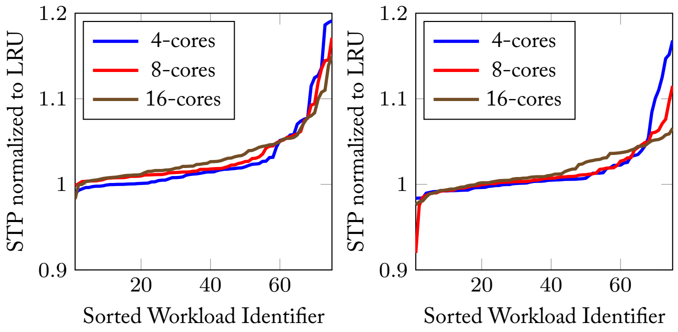
ward. As already indicated in Section 4.2.2, an increasing amount of cores leads to more conflicts at the lowest insertion position in the cache-sets. In turn, this results in an increasing amount of newly inserted cache-lines being evicted before they are able to be promoted sufficiently, thereby decreasing the effectiveness of PIPP's partitioning scheme.

DRRIP is able to consistently provide performance improvements with respect to LRU, albeit with approximately 0.5% in both HMS and STP. This indicates that RRIP-replacement is favorable over recency-replacement, confirming the results of Jaleel et al. [18]. TADIP-F on the other hand, shows no significant differences compared to LRU. We have found that the set-dueling mechanism as used in both TADIP-F and DRRIP does not always detect trashing memory access patterns in our evaluation, thereby restricting the possible performance improvements. We further investigate this matter in a case-study (see Section 8.2).

From Figure 7.1, we observe that although the use of Z-LRU does not provide any overall benefit for the 4-core configuration, it starts to provide small overall improvements in STP and HMS for the 8- and 16-core CMP configurations. As the core-count of the CMP increases, more cores can compete for the same blocks in the cache, which can result in an increase in conflict misses. In our evaluation, the zcache offers 52 replacements candidates, significantly more than the 16 replacements the set-associative cache, as used by LRU, offers. This makes it more likely that Z-LRU is able to find better replacement candidates than LRU, resulting in a reduction of conflict misses. This can explain the minor increase in overall performance.

7.2 STP per Workload

In Section 7.1 we have shown that the overall improvements in HMS and STP are relatively small (at most 3%). This is due to most workloads not benefiting significantly from cache partitioning. However, individual improvements can be significantly larger: for example individual improvements in STP can reach up to almost 20%. To give a more complete overview of the results, we therefore include the individual improvements in STP normalized to LRU for each evaluated algorithm for the 4,8 and 16-core CMP configurations. For the sake of brevity and since trends in HMS closely follow those in STP, we do not include HMS.

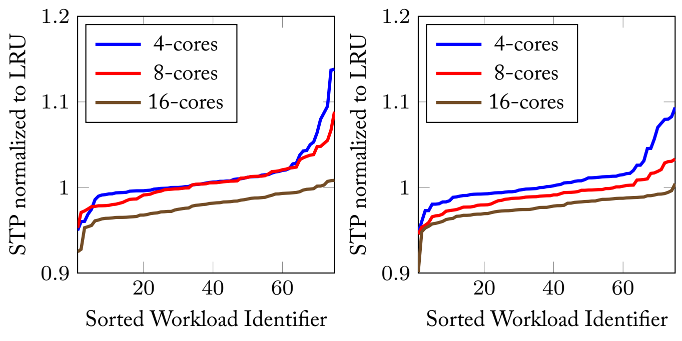


(a) STP per Workload for Vantage (b) STP per Workload for PriSM-UCP

Figure 7.2: STP per Workload for Vantage and PriSM-UCP

Figure 7.2 shows the STP per workload for Vantage and PriSM-UCP, which have been shown to be the two best overall performing algorithms in Section 7.1. The improvements shown in Figure 7.2 are sorted by increasing STP per algorithm: the curves are therefore an indication of how an algorithm performs overall, but cannot directly be compared to each other, as the workload-identifier in one figure might refer to another workload in the other. We observe that Vantage achieves the highest improvements in STP at all core counts. Although the maximum attained STP improvement with respect to LRU decreases with increasing core count, the decrease is relatively small compared to PriSM-UCP: the total decrease in the maximum improvement in STP is 11% for PriSM-UCP, whereas this is only 4% for Vantage. This indicates that PriSM-UCP does not scale as well with an increasing amount of partitions as Vantage. This is caused by the relatively quick decreasing accuracy at which PriSM is able to enforce its partition sizes: the fixed way-count of the set-associative cache, as used in the evaluation, does not provide enough replacement candidates to adequately partition the cache when the partition count increases. We further explain this in a case study (see Section 8.3).

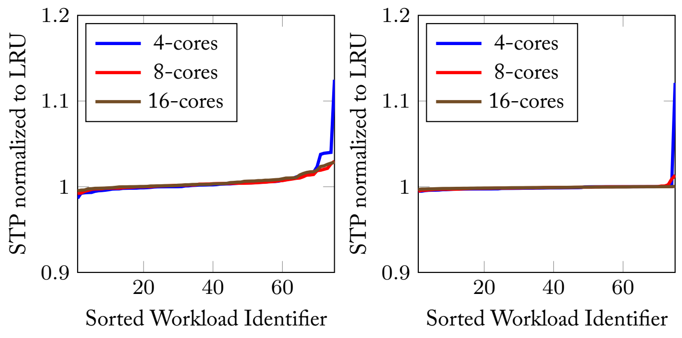
Figure 7.3 shows STP per workload for UCP and PIPP, sorted by increasing STP. In Section 7.1 we have observed that both UCP and PIPP do not scale as the amount of partitions increase. This is indeed reflected by the STP curves in Figure 7.3. We observe



(a) STP per Workload for UCP (b) STP per Workload for PIPP

Figure 7.3: STP per Workload for UCP and PIPP

that the maximum STPs of PIPP are lower than those of UCP and that the overall performance of PIPP decreases quicker. Although PIPP uses the same allocation policy as UCP, the partition sizes enforced by PIPP are only *approximates* of their targets. It is therefore possible that certain partitions outgrow their targets, lowering the maximum attainable STP and hurting overall performance. UCP's partitioning scheme does not allow partitions to outgrow their targets (see Section 4.2.1) [2], thereby providing more isolation to the cores. This can lead to increased performance with respect to PIPP.



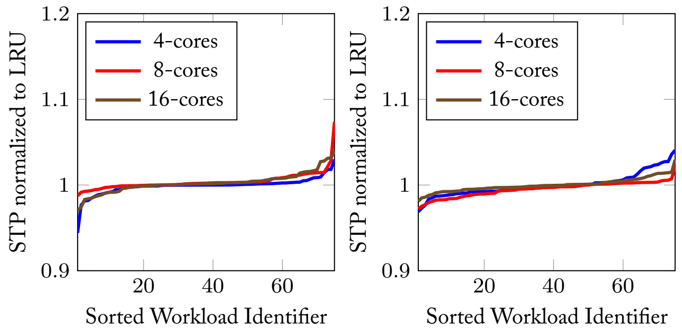
(a) STP per Workload for DRRIP (b) STP per Workload for TADIP-F

Figure 7.4: STP per Workload for DRRIP and TADIP-F

Figure 7.4 shows STP per workload for DRRIP and TADIP-F, sorted by increasing STP. For both DRRIP and TADIP-F we observe that the maximum improvement in STP is relatively small for all core-counts except at a core count of 4. For the 4-core CMP configuration, we observe that *one* workload experiences a significant improvement in STP of approximately 12%, both for DRRIP and TADIP-F. The

[2] Only in cases where a partition is downsized, its actual size can be temporarily bigger as data from the partition needs to be removed from the cache by evictions.

corresponding workload is the same for both algorithms: `ihhh-3`. This workload contains `libquantum`, which is properly detected as trashing the cache by the set-dueling mechanism (see also its sensitivity profile in Figure 6.1). Both TADIP-F and DRRIP therefore restrict its cache usage. `libquantum`, on average, has the highest MPKI of all benchmarks in this workload. This means that LRU allocates significantly more cache space to `libquantum` (due to allocating on an access-frequency basis), than TADIP-F and DRRIP. TADIP-F and DRRIP therefore leave more cache space available for the three other highly sensitive benchmarks in this workload, allowing for the observed improvement. The majority of workloads, however, attain STPs that are within 1% of those of LRU for both DRRIP and TADIP-F. As noted earlier this is due to set-dueling mechanism not always detecting trashing access patterns. A case-study further illustrating this problem is provided in Section 8.2.



(a) STP per Workload for Z-LRU

(b) STP per Workload for PriSM-H

Figure 7.5: STP per Workload for Z-LRU and PriSM-H

Figure 7.5 shows STP per workload for Z-LRU and PriSM-H sorted by increasing STP. For Z-LRU we observe that, unlike any other evaluated cache partitioning algorithm, the maximum attainable STP for 8 and 16 cores is *higher* than the maximum STP at 4-cores. As explained in Section 7.1 this could be attributed to the use of a zcache that offers more replacement candidates than the set-associative cache in our evaluation. Z-LRU experiences several performance decreases with respect to LRU for several workloads. Z-LRU (bucketed-LRU) is only an approximation of LRU. Furthermore, replacement candidates used by Z-LRU can come from every block in the zcache, whereas this is limited to sets for LRU. These differences lead to different behavior and can result in LRU outperforming Z-LRU on certain workloads.

From Figure 7.5 we observe that the STP attained by PriSM-H does not differ more than approximately 5% from that of LRU. Furthermore, the obtained speedups are approximately balanced out by the decreases in STP. This reflects our overall results which show that PriSM-H, overall seen, does not improve upon LRU.

7.3 HMS and STP per Class

To give more insight in to which workloads are susceptible to increases or decreases in the HMS respectively the STP, we group the HMS and the STP per class for all CMP configurations.

7.3.1 4-Core Results

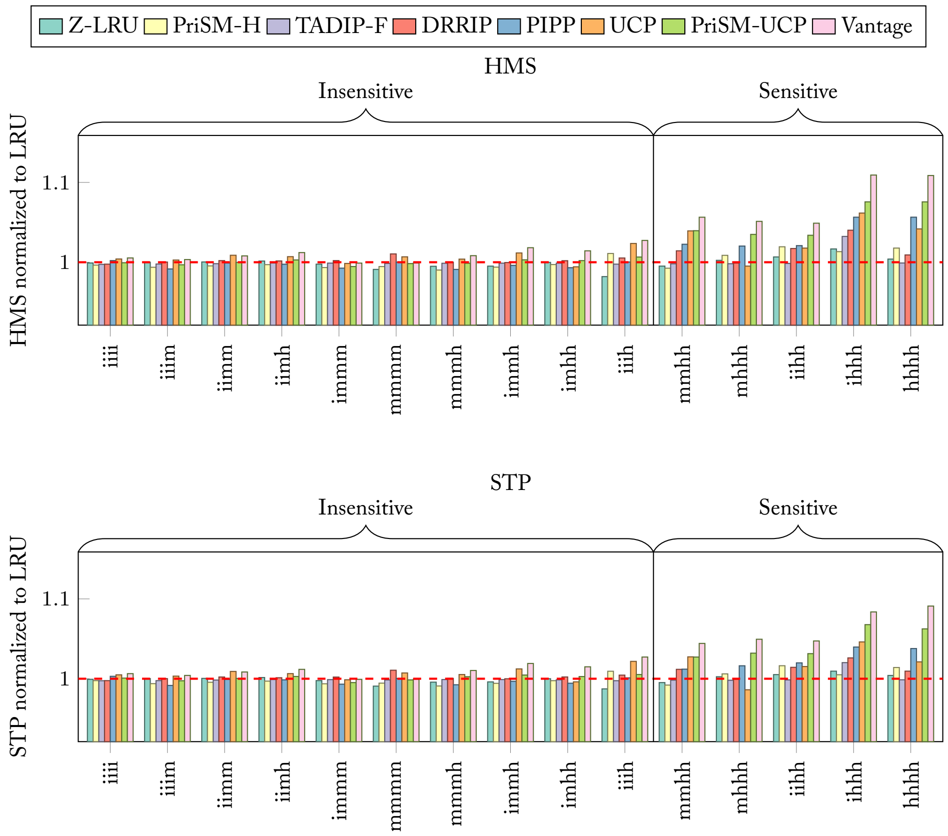


Figure 7.6: HMS & STP for all 15 workload classes, 4-cores

Figure 7.6 depicts the average HMS and STP for every class for all 4-core workloads for all evaluated algorithms normalized to LRU. We make the global observation that the trends in the HMS closely follow those in the STP. For sake of brevity we therefore restrict the discussion mainly to the STP as the observations made for the STP apply to the HMS as well. We further observe that Vantage is the top-performer for the majority of classes across all configurations.

A select group of classes, consisting of `mmhh`, `mhhh`, `iihh`, `ihhh` and `hhhh`, contains the highest improvements in STP and HMS. The common factor between all those classes is that at least half or more of the workloads are highly-sensitive benchmarks. Therefore, these workloads have the greatest potential for improvements over LRU. The rest of all classes form the insensitive group and show less than 5% difference with LRU.

We observe that the sensitive group of classes is lead by Vantage. Second is PriSM-UCP. PIPP and UCP are competing with each other in the highly sensitive group for the third place, with PIPP performing better than UCP in the classes `mhhh`, `iihh` and `hhhh`. Unlike UCP, PIPP includes a streaming detector which could give an advantage to PIPP for certain workloads. UCP, in general, directly enforces the maximum partition sizes (see Section 4.2.1), whereas PIPP only approximates them, which might be a reason for UCP outperforming PIPP in the other classes.

Per class, as was also observed for the overall results, we observe that the explicit partition algorithms provide the highest speedups.

7.3.2 8-Core Results

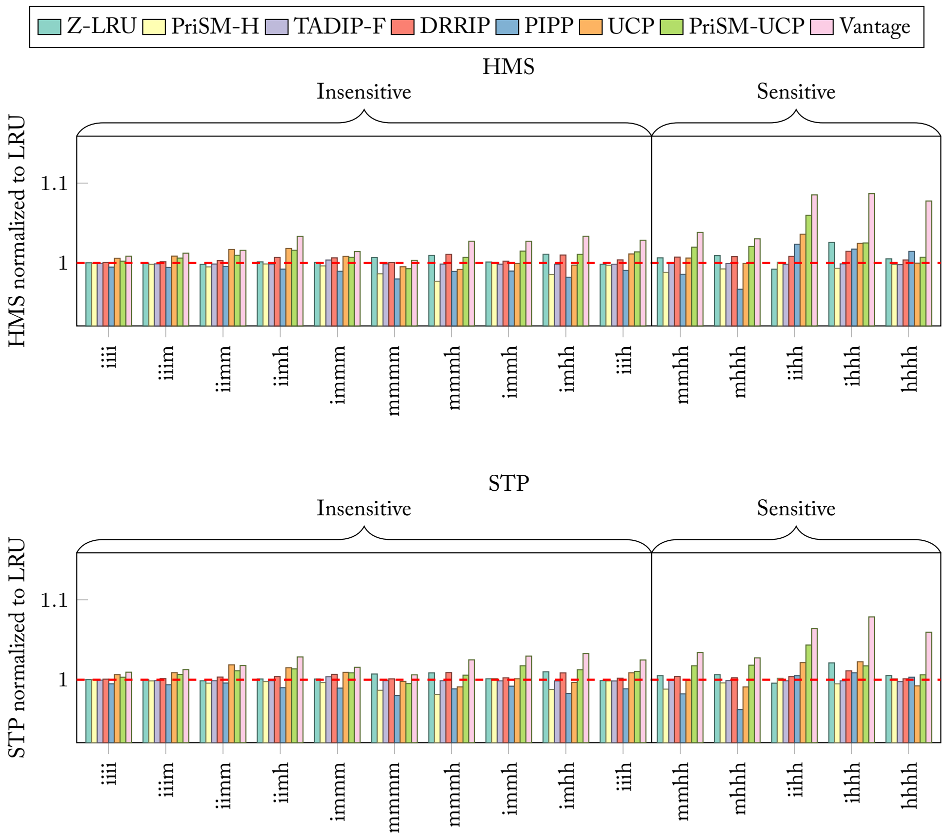


Figure 7.7: HMS & STP for all 15 workload classes, 8-cores

Figure 7.7 depicts the average HMS and STP for every class for all 8-core workloads for all evaluated algorithms normalized to LRU. Again the trends in the HMS closely follow those in the STP and we restrict our discussion mainly to the STP.

From Figure 7.7, we observe that the highest speedups are again predominantly contained in the same group of classes as for the 4-core workloads. We also observe that the highest speedups in the sensitive group are again restricted to the explicit partitioning algorithms, although it is predominantly Vantage that attains them.

Compared to the 4-core workloads, we observe that within the sensitive group the speedups achieved by Vantage in the classes `ihhh` and `hhhh` are significantly higher than those attained by the other explicit partitioning algorithms. As we have already observed, PIPP and UCP do not scale well with core count, limiting their improvements for 8-cores. Furthermore, PriSM-UCP's ability to accurately enforce partition sizes, decreases as core count increases (For further details see the case-study in Section 8.3), leading to lower maximum speedups than those of Vantage.

Another observation is that for the class `ihhh`, both the STP and HMS of Z-LRU are significantly higher than those of LRU. This is caused by Z-LRU speeding up the `sphinx3` benchmark by 10% for a workload in this class, which is big enough to significantly affect the average STP of this class. During the development of the zcache we found that `sphinx3`, in general, is highly sensitive to changes in associativity and that a zcache with only 16 (instead of 52 as used in our evaluation) replacement candidates was already significantly outperforming a LRU-managed 16-way set-associative cache.

7.3.3 16-Core Results

Figure 7.8 depicts the average HMS and STP for every class for all 16-core workloads for all evaluated algorithms normalized to LRU. Again the trends in the HMS closely follow those in the STP. PIPP and UCP now perform worse than LRU for all classes due to their unscalability. We further observe that the highest speedups are again predominantly contained in the same group off classes as for the 4-core workloads. But now, of all partitioning algorithms, only the explicit partitioning algorithms Vantage and PriSM-UCP are able to improve STP by more than 5%. These algorithms are, with the exception of PriSM-H, the only algorithms that explicitly partition on cache-line granularity, thereby giving them an advantage.

We further make the interesting observation that the HMS and STP of PIPP and UCP *do* show significantly different trends in the sensitive group: PIPP consistently attains the highest HMS in this group, but UCP, in general, attains better values for the STP. This means that the individual speedups in a workload as attained by UCP are less balanced than that of PIPP. For the `hhhh` class we have tracked this down to the highly-sensitive benchmark `povray` in the `hhhh`-2 workload. `povray` saturates to its maximum performance at 2MiB of cache space (see also its cache sensitivity profile in Figure 6.1). As already noted, for the 16-core configuration, the look-ahead algorithm allocates one way (2MiB) to every core. As UCP can strictly enforce partition sizes in this case[3], it can speedup `povray` significantly with respect to LRU. PIPP's partitioning scheme only approximates the partition targets and also suffers from con-

[3]No downsizing can occur, and therefore all partitions can never exceed their target sizes.

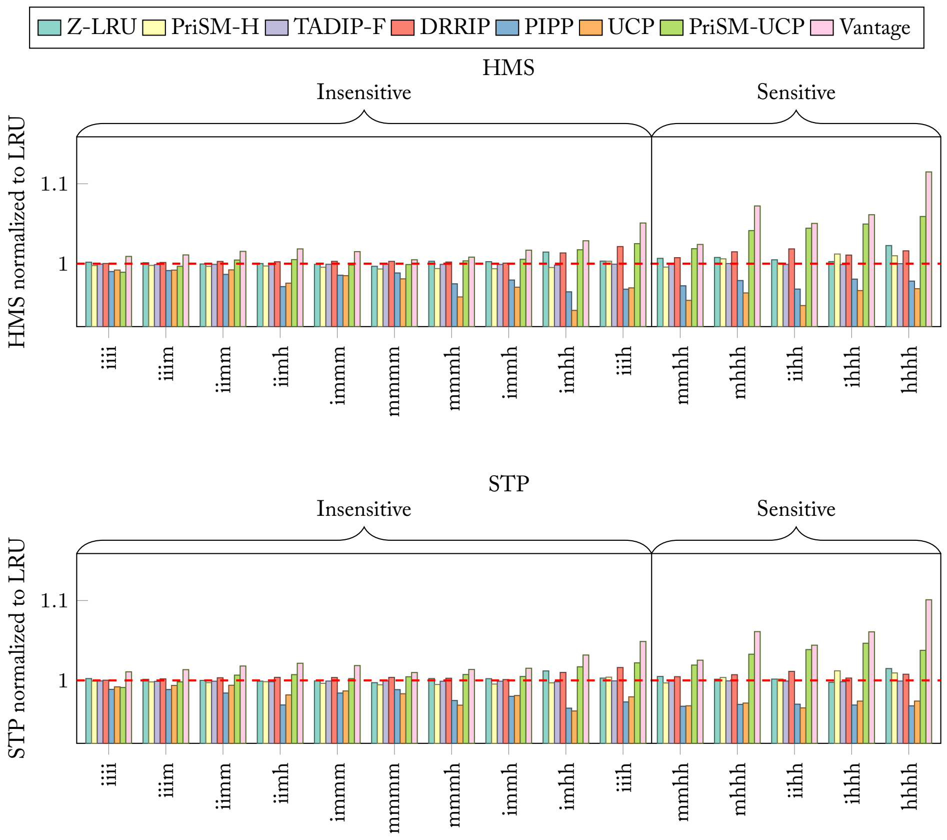


Figure 7.8: HMS & STP for all 15 workload classes, 16-cores

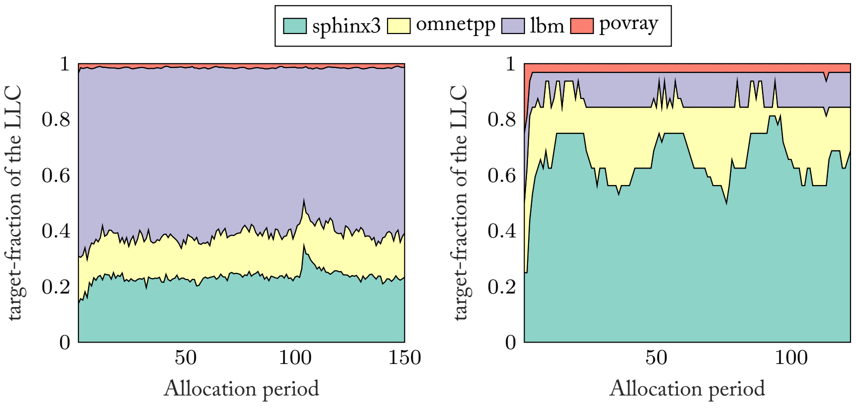
tention at the minimum insertion position, decreasing the performance of `povray` with respect to LRU. Although, in general, the partition allocations are too coarse, UCP can still cause speedups with respect to LRU due to its partitioning scheme. This results in higher STP's, but a more uneven distribution of speedups within a workload, for certain workloads, leading to the observed behavior.

8 | Case Studies

In this chapter we will provide several case studies to further explain several of the observed performance differences in our results.

This chapter starts with a case study on PriSM-H and PriSM-UCP, followed by a case study on the set-dueling mechanism as used in TADIP-F and DRRIP. This chapter is concluded with a case study on Vantage and PriSM-UCP.

8.1 PriSM-H does not Improve upon LRU



(a) PriSM-H target partitions (b) PriSM-UCP target partitions

Figure 8.1: Target Partition Sizes for PriSM for the 4-core Workload `immh-3`

From the results we observe that PriSM-H does neither improve STP nor HMS with respect to LRU. Although PriSM-H is in fact able to achieve speedups, those are balanced out by significant slowdowns on certain workloads. PriSM-UCP on the other hand, *is* able to consistently improve both STP and HMS at all core-counts. Since PriSM-H and PriSM-UCP share the same partitioning scheme, differences in performance are due to the different allocation policies.

During the experimental phase of this work, we found that the target allocations as given by PriSM-H, deviated significantly from those as given by UCP. This motivated

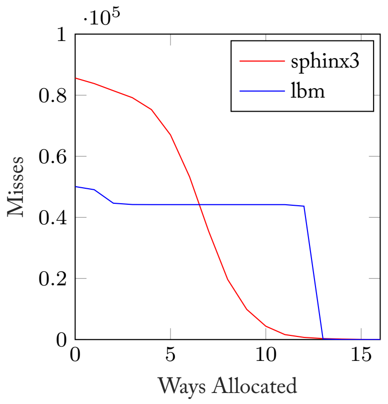


Figure 8.2: Miss Curves of `sphinx3` and `lbm` for the 4-core Workload `hhhh-3` - Calculated using Algorithm 1 in Section 6.6

the implementation of PriSM-UCP, such as to isolate the effect of the allocation policy and partitioning scheme.

We have selected the 4-core workload `hhhh-3` to illustrate this observed behavior. For this workload, the STP of PriSM-H is only 90% of that of PriSM-UCP.

Figure 8.1 shows the target allocations of both PriSM-H and PriSM-UCP. We observe that whereas PriSM-H allocates most space to `lbm`, PriSM-UCP allocates most space to `sphinx3`.

To find out why PriSM-UCP allocates most space to `sphinx3`, we have included Figure 8.2, which shows the miss curves during a representative partitioning period for this workload. We observe that `sphinx3` can save significantly more misses, also using less ways, than `lbm`. This explains why the look-ahead algorithm, as used by PriSM-UCP, allocates more space to `sphinx3` than to `lbm`.

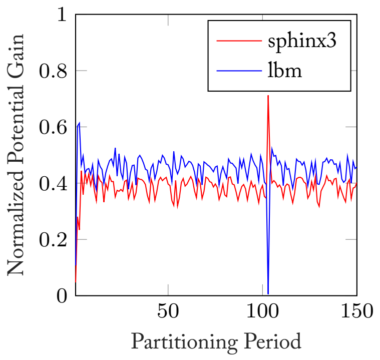
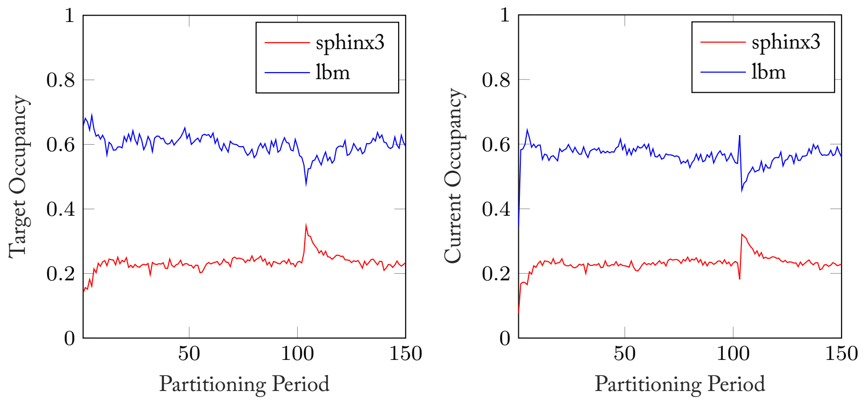


Figure 8.3: Normalized Potential Gain of `sphinx3` and `lbm` for the 4-core Workload `hhhh-3`



(a) Target Occupancies of `sphinx3` and `lbm` for the 4-core Workload `hhhh-3`

(b) Current Occupancies of `sphinx3` and `lbm` for the 4-core Workload `hhhh-3`

Figure 8.4: Current Occupancy and Target Occupancy of `sphinx3` and `lbm` for the 4-Core Workload `hhhh-3`

To find out why PriSM-H allocates more space to `lbm` than to `sphinx3`, we have included Figure 8.3, which shows the normalized potential gain:

$$\frac{\texttt{PotentialGain[i]}}{\texttt{TotalGain}},$$

for `sphinx3` and `lbm` as used by PriSM-H's allocation policy.

We observe that the normalized potential gain for `sphinx3` is slightly lower than that of `lbm`. Yet, the target allocation in Fig. 8.4a shows that `lbm` is allocated significantly more cache space. To find out why, we repeat the formula used to calculate the target occupancy:

$$T_i = C_i \cdot \left(1 + \frac{\texttt{PotentialGain[i]}}{\texttt{TotalGain}}\right).$$

We observe that the target allocations are linearly dependent on the current occupancy. Figure 8.4b shows why `lbm`'s target allocation is significantly larger: the occupancy of `lbm` is significantly higher than that of `sphinx3`. This means that with approximately the same normalized gain, `lbm`'s target partition size can still be significantly larger than that of `sphinx3`. This discrepancy persists throughout the whole simulation. This is due to `lbm`'s *initial* occupancy being relatively high. This prevents `sphinx3` from taking up more space throughout the simulation. The space taken up by benchmarks in the first partitioning period, can therefore have a significant effect on PriSM-H's overall performance.

Furthermore, unlike UCP's look-ahead algorithm, PriSM-H does not take the miss-curves into account. Instead, it calculates the potential gain of a core as the *total* amount of hits of this core in the ATD minus the hits of the core in the MTD. Not only does this couple the allocation to the current performance, but the amount of hits does, in general, not linearly depend on the cache space allocated. From Figure 8.2, we observe that `lbm` only experiences a significant reduction in misses when it is allocated 12 ways or more[1]. From Figure 8.4b we observe that its occupancy is approximately 0.6 on average. This corresponds to approximately 10 ways, which is not enough to attain this significant reduction. The potential gain of `lbm` therefore stays relatively high. `sphinx3`, on the other hand, directly starts to receive a significant amount of hits when its allocation is increased from 0 to more ways. This means that the difference in hits between `sphinx3` in the MTD and the ATD directly starts to decrease, limiting its potential gain. With an approximate occupancy of 0.2, this is enough to cause the potential gain of `sphinx3` to be approximately equal to that of `lbm` (see Figure 8.3).

[1]Since UCP halves the hit counters every period to take the history in account, hits can accumulate. This can lead to the impression that more misses can be saved than is actually the case. This curve is therefore only used to extract trends when discussing PriSM-H.

8.2 Set-Dueling Does not Accurately Capture Trashing Memory Access Patterns

Based on our main experiment, we observe that TADIP-F is not able to provide significant performance improvements over LRU: TADIP-F's STP and HMS are effectively the same as those of LRU.

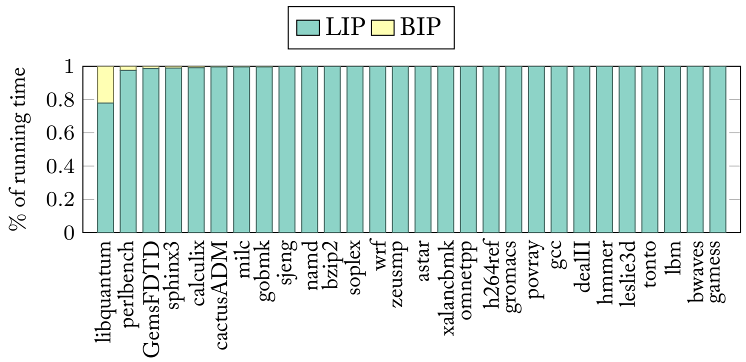


Figure 8.5: TADIP-F: Distribution of BIP and LIP Usage Among all Benchmarks

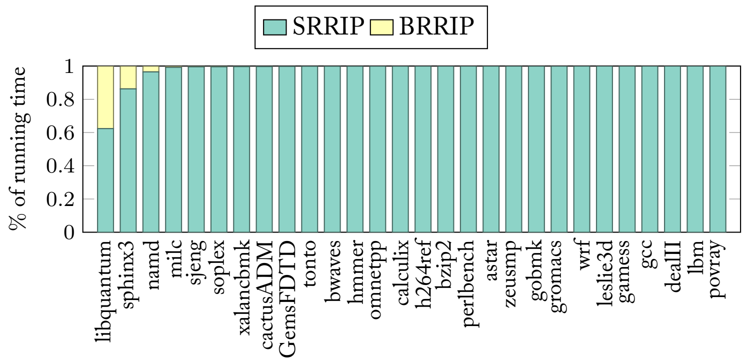


Figure 8.6: DRRIP: Distribution of BRRIP and SRRIP Usage Among all Benchmarks

We have therefore devised an experiment in which, for all 4-core workloads, the fraction of the running time spent in either BIP or LIP by TADIP-F has been tracked for each benchmark. A similar experiment has been performed for DRRIP, in which the fraction of the running time in SRRIP respectively BRRIP was tracked. The average results of those experiments are shown in Figures 8.5 and 8.6.

From Figure 8.5 we observe that for TADIP-F, `libquantum` uses the BIP insertion policy the most of all benchmarks. However, `libquantum` uses BIP, only 20% of its total running time. From its cache sensitivity profile (see Figure 6.1), we observe that the MPKI of `libquantum` stays the same up until 16MiB of cache space. This is

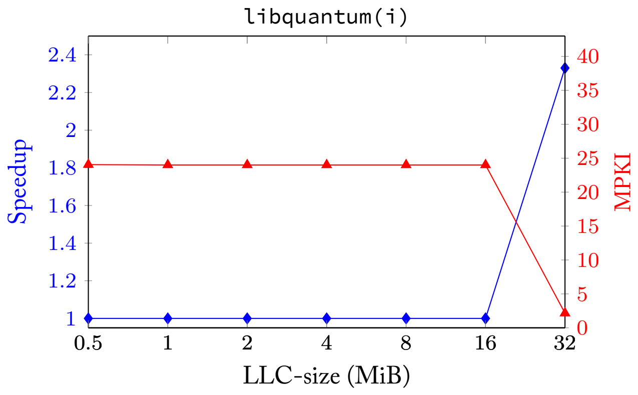


Figure 8.7: Extended Cache Sensitivity Profile of `libquantum` - 32MiB included

an indication that `libquantum` might trash the cache, at least up to 16MiB. Indeed, our extended cache sensitivity profile in Figure 8.7, which includes a 32MiB LLC, shows that `libquantum` is trashing the cache up to 16MiB. At 32MiB, the working set of `libquantum` fits in the cache, causing a significant performance improvement and reduction in misses.

From Figure 8.6 we observe that `libquantum` uses BRRIP for a little less than 40% of its running time.

Both when using TADIP and DRRIP, set dueling seems to be unable to consistently capture the trashing access pattern of `libquantum`. Other benchmarks that might be trashing in certain workloads, might suffer from the same problem.

One approach to increase the accuracy of the set-dueling mechanism would be to increase the amount of dedicated sets. However, as the amount of dedicated sets is increased, more sets are forced to use a suboptimal policy. This, in turn, can decrease overall performance. During the experimental phase of this work, we have not been able to improve the performance by altering the density of the dedicated sets. In our experiments we found that for example doubling the density of dedicated sets as used by Jaleel et al. [17], did not significantly alter the optimal detected policy and only resulted in a decrease in overall performance.

Although Figure 8.6 shows that DRRIP uses SRRIP most of the time, we have observed that DRRIP *is* able to significantly speed up certain classes of workloads. This can be explained by the use RRIP: Jaleel et al. [17] and Sanchez and Kozyrakis [33] show that the use of SRRIP alone can yield significant performance improvements over LRU.

8.3 Vantage Outperforms PriSM-UCP

Manikantan et al. [22] show that PriSM-UCP outperforms Vantage when used on a 16-way set associative cache. Our results, however, indicate that Vantage outperforms PriSM-UCP at all core counts. Unlike Manikantan et al. [22] we make use of the zcache architecture for the evaluation of Vantage. Based on our experiments, we observe that as core counts increase, the maximum speedups achieved by PriSM-UCP decrease more than those of Vantage. The root cause of this decrease is PriSM's inability to accurately enforce partition sizes as the core-count increases. This in turn leads to partition sizes drifting away from their targets, resulting in performance loss.

PriSM's partitioning scheme is based on the assumption that the case, in which the selected victim core has no block in the cache-set considered for replacement, is rare. Figure 8.8 depicts the average percentage of all evictions that evict data not belonging to the selected victim core, measured over all workloads. We observe that doubling the core count brings with it an 10% increase in wrongfully evicted data. This leads to

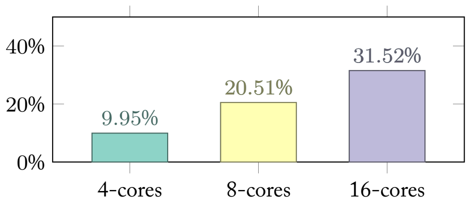
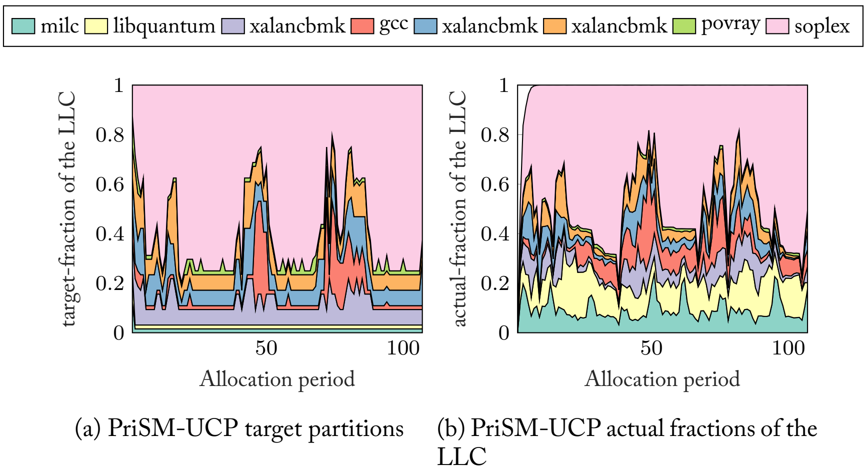


Figure 8.8: Percentages of Wrong Evictions Among all CMP Configurations for PriSM-UCP



(a) PriSM-UCP target partitions (b) PriSM-UCP actual fractions of the LLC

Figure 8.9: Target and actual partition sizes for PriSM-UCP for the 8-core workload `ihhh-4`

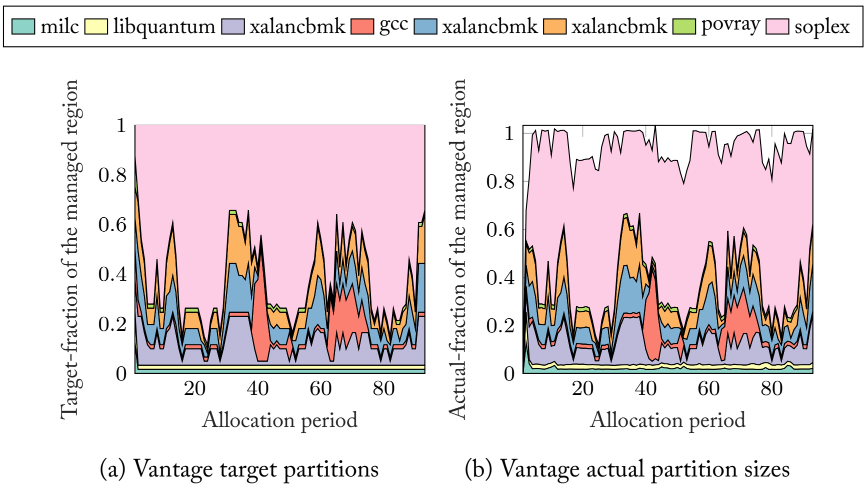


Figure 8.10: Target and actual partition sizes for Vantage for the 8-core workload `ihhh-4`

decreased accuracy in enforcing the partition sizes.

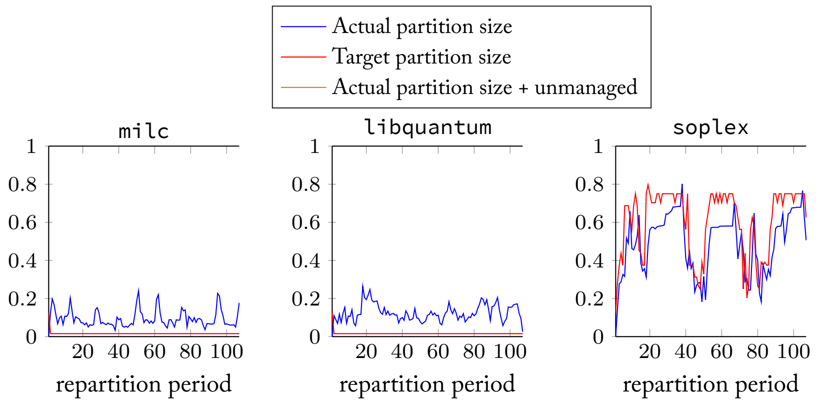
We have selected the 8-core workload `ihhh-4` to illustrate this. For this particular workload, on average, 41% of all evictions by PriSM-UCP evict data from other cores than the victim core. The STP achieved by PriSM for this workload is only 91.9% of that of LRU, whereas Vantage *improves* STP by 4.2% with respect to LRU. To get a better insight in the situation, we have depicted the target and actual partition sizes of PriSM-UCP in Figure 8.9. From Figure 8.9 we observe that the actual space that `milc` and `libquantum` use, is significantly larger than allocated to them by PriSM-UCP.

Figure 8.10 depicts the target and actual partition sizes for Vantage for the 8-core workload `ihhh-4`. We observe that Vantage enforces the partition sizes close to, or below their targets in the *managed* region. This means that Vantage's demotion system is working correctly and that the zcache is offering enough associativity for the demotion process.

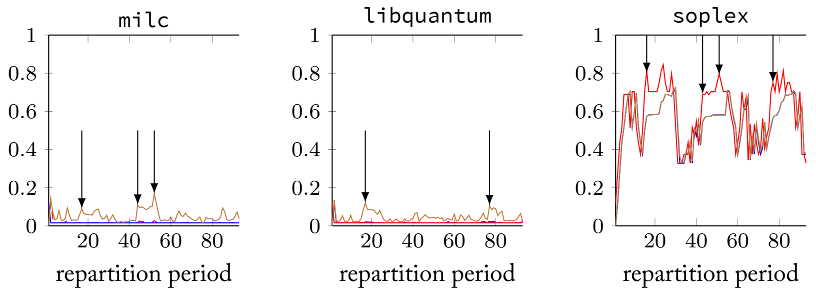
Whereas Vantage is able to control `milc` and `libquantum`'s partition size, PriSM-UCP is not. In the case of PriSM-UCP, more space is taken by `milc` and `libquantum` from the highly sensitive benchmarks. This leads to the observed performance difference.

So far, we have ignored the unmanaged region that Vantage uses. In our evaluation, the unmanaged region is 5% of the LLC-size. From Figure 8.10, we observe that the managed region can temporarily decrease to a size below its target size. This corresponds to an increase in size of the unmanaged region. We will now restrict ourselves to the benchmarks `milc`, `libquantum` and `soplex` in this workload to explain this behavior.

Figure 8.11 depicts the actual and target sizes of `milc`, `libquantum` and `soplex` for both PriSM-UCP and Vantage. Moreover, for Vantage we have included the space



(a) Partitions of `milc`, `libquantum`, `soplex` for PriSM-UCP for the 8-core workload `ihhh-4`. Depicted are the target and actual partition sizes for `milc`, `libquantum`. For clarity only the target of `soplex` is depicted



(b) Partitions of `milc`, `libquantum`, `soplex` for PriSM-UCP for the 8-core workload `ihhh-4`. Depicted are the target and actual partition sizes. `soplex` is depicted

Figure 8.11: Partitions sizes of `milc`, `libquantum`, `soplex` for PriSM-UCP and Vantage for the 8-core workload `ihhh-4`

usage including the blocks of each benchmark that have been demoted and reside in the unmanaged region.

We observe that overall, the space used by `milc` and `libquantum` is significantly higher for PriSM-UCP than for Vantage. This confirms that even with the data in the unmanaged region included, Vantage is better at restricting the amount of blocks that `milc` and `libquantum` store in the LLC.

Moreover, we observe that the space usage of `milc` and `libquantum` can exceed the 5% that has been allotted to the unmanaged region. Several of this points have been marked in Figure 8.11b. This is caused by *transient*-behavior as described by Sanchez and Kozyrakis [33]: when a partition is suddenly upsized and this partition is gaining space faster than the other partitions loose, the unmanaged region can grow temporarily. Or in other words: the overall demotion rate is higher than the eviction rate, resulting in growth of the unmanaged region. Indeed the points marked for `milc` and `libquantum` correspond the sudden, relatively large, increases in the target size of `soplex` as marked in Figure 8.11b.

Although the observed behavior makes it likely that this is indeed transient behavior, it might have been the case that the increase in space of the unmanaged region was caused by Vantage evicting too much data data from the managed region. By tracking the amount of evictions from the managed region, we found that the average probability of eviction of a block from the managed region is approximately $6.67 \cdot 10^{-3}$, indicating that the increase of the unmanaged region is indeed due to transient behavior. As it are the three partitions corresponding to `xalancbmk` that loose most of their size when `soplex`'s target is increased (see Figure 8.10), the cache space in the unmanaged region that `libquantum` and `milc` take up, is able to increase due to their high churns (see also their sensitivity profiles in Figure 6.1).

We therefore conclude that the observed discrepancies in the target and actual partition sizes for Vantage stem from transient behavior and are not caused by lack of associativity of the zcache. The discrepancies in PriSM-UCP's target and actual partition sizes, however, *are* the result of restricted associativity.

9 | Sensitivity Analysis

In this chapter we will present several experiments that explore the response of the evaluated cache-partitioning algorithms to changes in the target architecture. Section 9.1 describes an experiment exploring how Vantage and Z-LRU respond to changes in the amount of replacement candidates R of the zcache. Section 9.2 follows with an experiment exploring the response of LRU, PriSM-H, PriSM-UCP, TADIP, DRRIP and UCP to changes in the amount of ways of a set-associative cache. Section 9.3 and Section 9.4 are concerned with experiments that vary the LLC-size and DRAM-bandwidth respectively.

9.1 Sensitivity to Varying Associativity of the Zcache

To test the response to the additional associativity a zcache can provide, we have devised an experiment with all 4-core workloads for Z-LRU and Vantage. In this experiment the number of levels of the BFS-walk was varied from one level, up to four levels at a constant way count of 4. This means that we subsequently test with a zcache having 4,16,52 and 160 replacement candidates. The UMON's as used for Vantage used 16

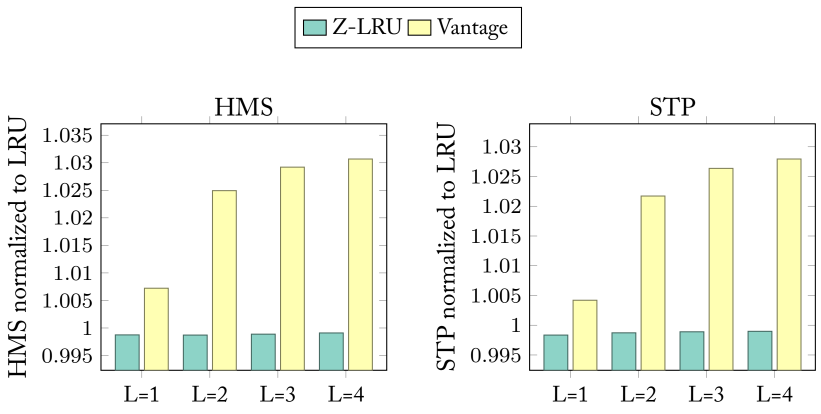


Figure 9.1: Performance of Z-LRU and Vantage for all 4-core workloads, for varying L of the zcache

ways for all configurations: in this way, we can isolate the effect of changing target allocations from the actual associativity change of the zcache.

The results of this experiment are normalized to the performance of LRU as obtained in the 4-core main experiment. The results are depicted in Figure 9.1. We observe that trends in HMS follow those in STP for both Vantage and Z-LRU. Moreover, we observe that as the amount of levels of the BFS-walk increases from one to two, the performance of Vantage increases considerably.

In Section 4.2.4 we showed that the minimum size u of the unmanaged region is given by

$$u \geq 1 - \sqrt[R]{P_{ev}},$$

with P_{ev} the worst-case probability of evicting a block from the managed region. Consequently, the minimum P_{ev} is given by $P_{ev} \geq (1-u)^R$. As P_{ev} increases, Vantages accuracy of enforcing the partition sizes decreases as it starts evicting more data from the managed region.

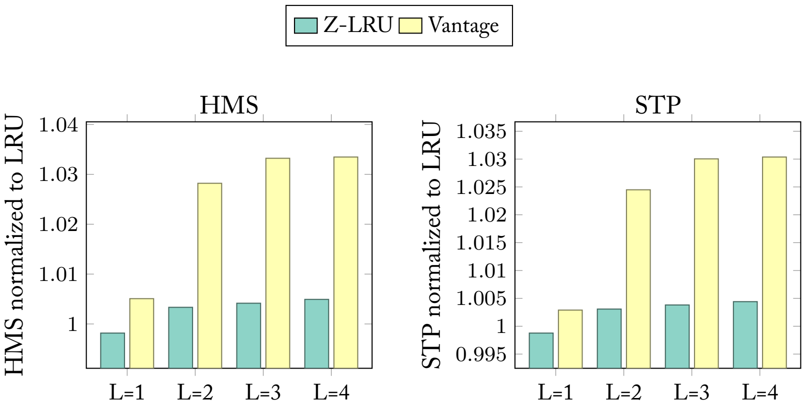


Figure 9.2: Performance of Z-LRU and Vantage for all 8-core workloads, for varying L of the zcache

Since $u < 1$, the minimum worst-case probability of evicting a cache-line from the managed region decreases exponentially with the amount of replacement candidates. At $L = 1$, Vantages demotion scheme has to work with 4 replacement candidates. With $u = 0.05$ in our evaluation, this results in $P_{ev} \geq 81\%$. For $L = 2$, 16 replacement candidates are available and $P_{ev} \geq 44\%$, almost halve. At $L = 3$ and $L = 4$, the minimum P_{ev} equals 6.7% respectively $2.7 \cdot 10^{-2}\%$. This explains the relatively large increase in performance from changing the amount of levels from one to two and the saturating performance as L further increases: from a certain point onward, performance is not limited by the accuracy of the partitioning scheme.

Z-LRU, however, is not able to profit from additional associativity. This is not completely unexpected: Sanchez and Kozyrakis [32] show that the difference between LRU-z4/16 and LRU-z4/52 is negligible. In our case, the difference between with LRU-z4/16 and LRU-z4/4 is also negligible. However, the results of Sanchez and

Kozyrakis [32] are for 32-core workloads, instead of 4-core workloads. The use of 32-cores can increases the amount of conflict misses with respect to 4-cores. When we look at the results of our main experiments, we notice that Z-LRU only provides benefits for 8- and 16-core workloads.

This is an indicator that at a core-count of 4, the amount of conflict misses is too low for Z-LRU to significantly profit from the additional associativity. The same experiment, but performed for all 8-core workloads is shown in Figure 9.2. Here we do observe a significant difference between LRU-z4/4 and LRU z4/16.

9.2 Sensitivity to Varying Associativity of the Set-Associative Cache

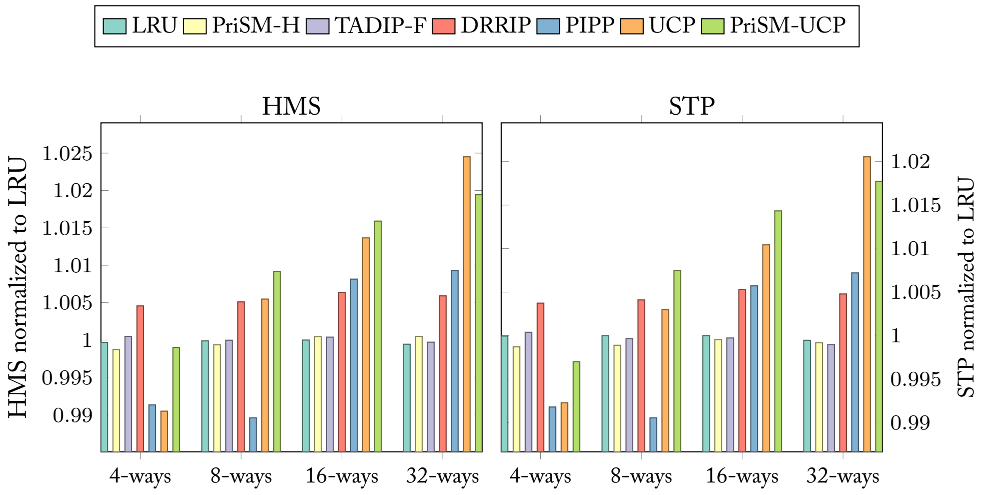


Figure 9.3: HMS & STP for all 4-core workloads, for varying way-counts of the set-associative cache

To test the response to varying associativity of a set-associative cache, we have devised an experiment for all 4-core workloads, in which the number of ways of the set-associative cache was varied from 4 up to 32 ways. Figure 9.3 depicts the results of this experiment and shows the overall STP and HMS normalized to the LRU on the default 4-core CMP configuration. We observe that the trends in HMS closely follow those in STP. For LRU, TADIP-F, DRRIP and PriSM-H we observe that the overall performance varies with less than 0.5% from the LRU-baseline. The performance of PriSM-H has been shown to mainly limited by its allocation policy. LRU, TADIP-F and DRRIP do not manage interference between cores, and thereby do not necessarily profit significantly from additional associativity to reduce conflict misses.

The HMS and STP of UCP, PIPP and PriSM-UCP have a maximum difference from the LRU-baseline of more than 1%. UCP and PriSM-UCP continue to benefit from an increase in associativity. PIPP in general as well, with the exception of the 8-core workloads, where it experiences a minor decrease in HMS and STP of approximately 0.1% with respect to the LRU-baseline.

These performance increases can be partially explained due to the increase of ways of the ATD's used by UCP, PIPP and PriSM-UCP. The increase in ways, allows the look-ahead algorithm to estimate the optimal target partition sizes with finer granularity. At the same time, the increase in ways also increases the granularity at which PIPP and UCP can partition the cache. We observe that at 32-ways PriSM-UCP is no longer the top-performer, but UCP. At 32-ways, the granularity at which UCP partitions the cache seems adequate and the probabilistic partitioning scheme of PriSM might not work as well as that of UCP.

9.3 Sensitivity to Varying LLC-Size

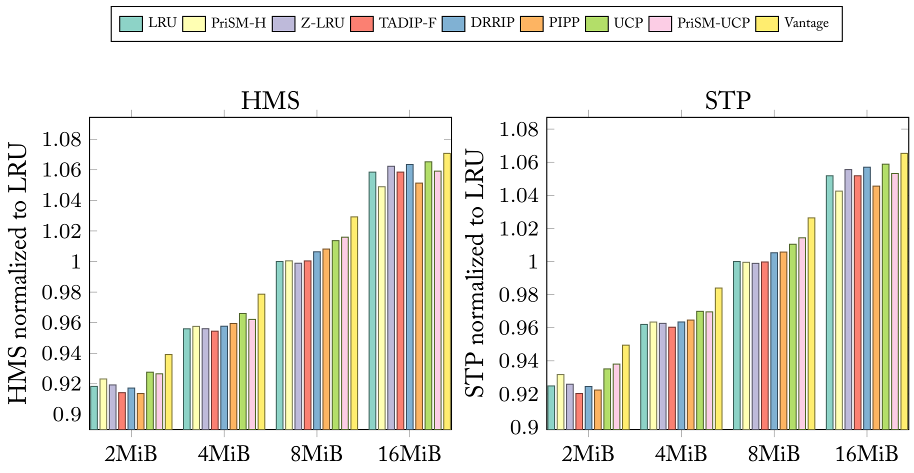


Figure 9.4: HMS & STP for all 4-core workloads, for varying LLC-sizes

To test the response of the cache partitioning algorithms to changes in the LLC-size, we have devised an experiment for all 4-core workloads, in which the size of the LLC was varied from 2MiB up to 16MiB. We expect that with increasing LLC-size, the performance difference between all algorithms becomes less, as the need for partition decreases: from a certain point the cache has enough space to accommodate the working sets of all benchmarks in a workload. This is indeed confirmed by the results of our experiment in Figure 9.4, which shows the overall STP and HMS normalized to the performance of LRU on the default 4-core configuration. We observe that the difference between LRU and the other algorithms becomes smaller with increasing cache size. Again, the trends in HMS follow the trends in STP closely. Furthermore, we observe that the ordering of the algorithms stays approximately the same for the cache sizes up to 8MiB. This indicates that only at 16MiB, the working sets of all benchmarks in a workload start to fit in the cache. We observe that *both* DRRIP and TADIP-F significantly decrease performance with respect to LRU at 2MiB: in this case cache space is too restricted and the use of dedicated sets affect performance negatively.

We furthermore observe that over all cache-sizes, Vantage is the top-performer, although the difference with LRU is the least significant at the largest cache-size tested, 16MiB. Furthermore, at a LLC-size of 16MiB we observe that PriSM-H, PriSM-UCP, PIPP decrease the HMS and STP with respect to LRU. This indicates that LRU is able to efficiently manage the cache and that the partitioning scheme of those algorithms hurt the performance.

9.4 Sensitivity to Varying DRAM Bandwidth

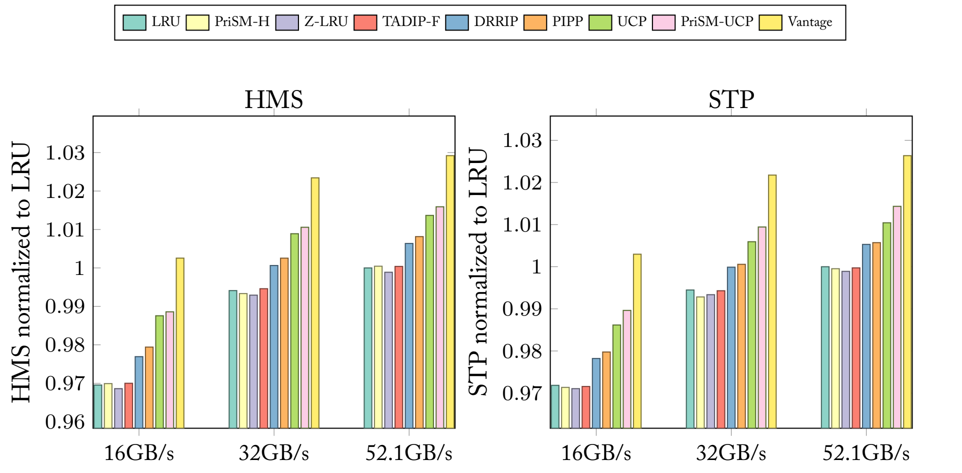


Figure 9.5: HMS & STP for all 4-core workloads, for varying DRAM-bandwidths

To test the impact of changing DRAM-bandwidth we have devised an experiment for all 4-core workloads in which we altered the configuration of the 4-core CMP by varying the DRAM-bandwidth. In this experiment we have made use of bandwidths of 16GB/s,32GB/s and 51.2GB/s respectively. Figure 9.5 depicts the overall STP and HMS as result of this experiment, normalized to the performance of LRU on the default 4-core configuration.

A global observation is that with increasing bandwidth, the STP and HMS of all algorithms increases. Furthermore, the trends in HMS are similar to those in STP. With increasing bandwidth to DRAM, the penalty of a miss in the LLC can decrease, as more accesses can concurrently request data from or write data to DRAM. As a result average performance increases. We furthermore observe that the ranking between performance of all algorithms is the same for all bandwidths. This indicates that all algorithms profit approximately equally from increasing bandwidth.

We furthermore observe that the differences in STP and HMS between 16GB/s and 32GB/s are bigger than the differences between 32GB/s and 51.2GB/s (approximately

0.4% respectively 0.1% for all algorithms). In order for the benchmarks to significantly profit from more bandwidth, it is necessary that the bandwidth of the DRAM controller is fully utilized. However, in Section 6.1 we have configured the bandwidth of our CMP's to 51.2 GB/s such as to *avoid* saturation of the available bandwidth. In our sensitivity experiment, as the amount of available bandwidth increased, the average bandwidth utilization decreased. This means that at 51.2 GB/s, not enough memory accesses are made to completely saturate the available bandwidth and that performance is mainly limited by latency instead of bandwidth. Continuous increase of bandwidth therefore does not result in a continuous increase in performance.

10 | Conclusion & Future Work

10.1 Conclusion

In this thesis we have given a theoretical overview of the implicit cache partitioning algorithms TADIP-F, DRRIP, Z-LRU and the explicit cache partitioning algorithms PriSM, UCP,PIPP and Vantage. To evaluate the performance of Vantage, a parallel software implementation of the zcache architecture and Vantage has been added to the simulation framework currently in use at NTNU.

We have used this simulation framework to evaluate the performance of caches managed by the aforementioned cache partitioning algorithms, with a conventional LRU managed cache as base-line. We find that in our evaluation:

F1: Overall impact on HMS and STP by partitioning the cache is minor. Overall, the highest improvements with respect to LRU are approximately 3%, both in the HMS and the STP. We have shown that this is due to improvements mainly being confined to classes of workloads that predominantly contain applications which are highly-sensitive to cache resources. In this group, performance improvements up to approximately 20% have been observed. However, as this group forms only 1/3th of all the workloads, their impact on the overall HMS and STP is not high enough to obtain significant overall improvements.

F2: The performance and scalability of explicit cache partitioning algorithms is limited by the associativity that a set-associative cache provides. We have shown that the explicit partitioning algorithms UCP and PIPP do not scale well with increasing core-count, as they partition the cache at a too coarse granularity. The granularity at which they partition the cache is directly proportional to the amount of ways of the set-associative cache (which is fixed in our evaluation), and thereby limits performance improvements for higher core-counts. PriSM-UCP on the other hand, does not show significant overall degradation in HMS and STP, but the maximum attained HMS and STP do decrease significantly as core-count increases. We have shown that this is, again, a consequence of the limited associativity of the set-associative cache: PriSM looses it ability to accurately enforce partition sizes when associativity is limited. Vantage uses the highly-associative, but unconventional, zcache architecture. By providing significantly more replacement candidates than the set-associative cache, Vantage is able to adequately enforce its partition allocations. This provides the necessary isolation to the cores of the CMP to attain the highest observed performance improvements.

F3: Explicit cache partitioning algorithms provide more significant performance improvements than implicit partitioning algorithms. We have shown that of all evaluated algorithms, the explicit cache partitioning algorithm Vantage provides the highest improvements for all CMP configurations that we have evaluated. All other explicit partitioning algorithms, although they do not scale with increasing core-count due to fixed-associativity of the cache, outperform all implicit cache-partitioning algorithms on a 4-core CMP configuration. The evaluated implicit cache-partitioning algorithms (Z-)LRU, TADIP-F and DRRIP target memory access patterns, but do not provide isolation to the cores of the CMP. This allows possible destructive interference to occur, which can limit their performance improvements. Furthermore, we have found that in our evaluation the set-dueling mechanism, as used by TADIP-F and DRRIP, does not accurately detect trashing memory access patterns. This further limits the performance improvements of TADIP-F and DRRIP.

10.2 Future Work

Although we have fulfilled all the listed requirements in Chapter 1, this does not mean that the research on cache partitioning ends here. On the contrary, there are several possibilities for further research.

As we have found that overall performance improvement is minor, further research is required to determine whether this is caused by the workloads that have been used. It might be the case that our workloads are not representative for a commercial consumer CMP.

Furthermore, during our work, TADIP-F has shown unable to yield significant benefits with respect to LRU. We have tracked this down to the set-dueling mechanism rarely detecting memory access patterns as trashing. During researching this problem, we found that certain benchmarks access only a select group of sets in the set-associative cache. This can result in most accesses missing the dedicated sets. At the same time, increasing the amount of dedicated sets to more accurately capture such memory access patterns, decreased the overall performance. Set-dueling has only been introduced with set-associative caches, but it would be interesting to see how the use of zcaches or skew-associative caches affect the accuracy: by using hash functions, accesses to the cache are spread out more uniformly over the cache (such a variant would be "block-dueling" rather than set-dueling). This could proof a possible solution to memory-access patterns that only access a limited amount of sets in a set-associative cache, but could also improve accuracy in general.

At last, based on our analysis in Chapter 8, we point to experimenting with PriSM on zcaches. Our results show that PriSM mainly suffers from decreased enforcement accuracy, as the set-associative cache does not offer enough replacement candidates. Zcaches can offer substantially more replacement candidates and could therefore increase the overall performance of PriSM.

Bibliography

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

[2] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, volume 4, pages 187–200, 2004.

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

[4] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[5] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[6] T. Carlson. Causality, December 2012. URL `http://snipersim.org/w/index.php?title=Causality&oldid=413`.

[7] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014. ISSN 1544-3566. doi: 10.1145/2629677.

[8] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of computer and system sciences*, 18(2):143–154, 1979.

[9] M. Chaudhuri. Pseudo-lifo: The foundation of a new family of replacement policies for last-level caches. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 401–412, Dec 2009. doi: 10.1145/1669112.1669164.

[10] Intel Corporation. Intel® xeon® processor e5-2680, 2012. URL `https://ark.intel.com/products/64583/Intel-Xeon-Processor-E5-2680-20M-Cache-2_70-GHz-8_00-GTs-Intel-QPI`.

[11] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation, June 2016. URL `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`.

[12] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974. ISSN 0018-9200. doi: 10.1109/JSSC.1974.1050511.

[13] Lieven Eeckhout. Computer architecture performance evaluation methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010.

[14] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008.

[15] W. Heriman. Understanding mshr in snipersim, April 2016. URL `https://groups.google.com/forum/#!topic/snipersim/-KoQxOPmZ28`.

[16] Mark D Hill and Alan Jay Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[17] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr, and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 208–219. ACM, 2008.

[18] 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.

[19] 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.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[21] R Manikantan, Kaushik Rajan, and Ramaswamy Govindarajan. Nucache: An efficient multicore cache organization based on next-use distance. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 243–253. IEEE, 2011.

[22] Raman Manikantan, Kaushik Rajan, and Ramaswamy Govindarajan. Probabilistic shared cache management (prism). In *ACM SIGARCH computer architecture news*, volume 40, pages 428–439. IEEE Computer Society, 2012.

[23] Cor Meenderinck and Ben Juurlink. (when) will cmps hit the power wall? In *Euro-Par 2008 Workshops-Parallel Processing*, pages 184–193. Springer, 2009.

[24] Runar Bergheim Olsen. Evaluation of cache management algorithms for shared last level caches. Master's thesis, NTNU, 2015.

[25] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3 (7):26–29, 2005.

[26] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 81–92. IEEE, 2004.

[27] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 318–319. ACM, 2003.

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

[29] 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.

[30] RM Ramanathan. Intel® multi-core processors. *Making the Move to Quad-Core and Beyond*, 2006.

[31] Sanchez and Kozyrakis. Response to "zcache skew-ered", June 2011. URL `http://www.eecg.toronto.edu/~enright/wddd/2011/papers/2011_06_05-wddd_zcache_response.pdf`.

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

[33] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 57–68. ACM, 2011.

[34] Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer Architecture News*, 41(3):475–486, 2013.

[35] Andre Seznec. *A new case for skewed-associativity*. PhD thesis, INRIA, 1997.

[36] André Seznec and Francois Bodin. Skewed-associative caches. In *PARLE'93 Parallel Architectures and Languages Europe*, pages 305–316. Springer, 1993.

[37] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[38] Kevin Skadron, Margaret Martonosi, David I August, Mark D Hill, David J Lilja, and Vijay S Pai. Challenges in computer architecture evaluation. *Computer*, 36 (8):30–36, 2003.

[39] Alan Jay Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[40] William Stallings. *Computer organization and architecture: designing for performance*. Pearson Education India, 2000.

[41] Ruisheng Wang and Lizhong Chen. Futility scaling: High-associativity cache partitioning. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 356–367. IEEE Computer Society, 2014.

[42] Yuejian Xie and Gabriel H Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 174–183. ACM, 2009.

A | 4-core Workloads

Table A.1: 4-core Workloads

Workload	Benchmarks			
iiii-0	calculix	milc	libquantum	dealII
iiii-1	namd	milc	libquantum	milc
iiii-2	dealII	bwaves	milc	calculix
iiii-3	GemsFDTD	dealII	leslie3d	GemsFDTD
iiii-4	GemsFDTD	bwaves	zeusmp	leslie3d
iiim-0	gromacs	namd	leslie3d	sjeng
iiim-1	calculix	zeusmp	gromacs	sjeng
iiim-2	calculix	milc	zeusmp	astar
iiim-3	zeusmp	milc	zeusmp	bzip2
iiim-4	GemsFDTD	calculix	namd	astar
iiih-0	GemsFDTD	calculix	milc	povray
iiih-1	GemsFDTD	gobmk	libquantum	gcc
iiih-2	gobmk	milc	gobmk	sphinx3
iiih-3	zeusmp	libquantum	milc	soplex
iiih-4	dealII	libquantum	GemsFDTD	soplex
iimm-0	zeusmp	zeusmp	h264ref	perlbench
iimm-1	gobmk	bwaves	perlbench	perlbench
iimm-2	dealII	bwaves	astar	tonto
iimm-3	milc	calculix	cactusADM	bzip2
iimm-4	GemsFDTD	zeusmp	astar	cactusADM
iimh-0	bwaves	leslie3d	perlbench	soplex
iimh-1	gromacs	namd	sjeng	povray
iimh-2	dealII	calculix	h264ref	omnetpp

iimh-3	bwaves	leslie3d	perlbench	gamess
iimh-4	bwaves	leslie3d	wrf	xalancbmk
iihh-0	leslie3d	GemsFDTD	povray	lbm
iihh-1	calculix	zeusmp	sphinx3	gcc
iihh-2	gromacs	zeusmp	sphinx3	xalancbmk
iihh-3	GemsFDTD	dealII	povray	lbm
iihh-4	libquantum	zeusmp	gamess	lbm
immm-0	dealII	tonto	cactusADM	hmmer
immm-1	calculix	perlbench	astar	perlbench
immm-2	gobmk	h264ref	bzip2	tonto
immm-3	calculix	wrf	tonto	perlbench
immm-4	dealII	sjeng	perlbench	sjeng
immh-0	calculix	h264ref	wrf	lbm
immh-1	namd	sjeng	cactusADM	omnetpp
immh-2	zeusmp	hmmer	h264ref	xalancbmk
immh-3	namd	bzip2	astar	lbm
immh-4	zeusmp	bzip2	cactusADM	lbm
imhh-0	gobmk	wrf	xalancbmk	omnetpp
imhh-1	zeusmp	tonto	xalancbmk	povray
imhh-2	bwaves	perlbench	gcc	povray
imhh-3	gromacs	hmmer	povray	omnetpp
imhh-4	calculix	perlbench	omnetpp	lbm
ihhh-0	GemsFDTD	xalancbmk	povray	soplex
ihhh-1	calculix	sphinx3	sphinx3	xalancbmk
ihhh-2	leslie3d	sphinx3	soplex	xalancbmk
ihhh-3	libquantum	povray	sphinx3	sphinx3
ihhh-4	zeusmp	gamess	omnetpp	povray
mmmm-0	perlbench	hmmer	cactusADM	hmmer
mmmm-1	h264ref	perlbench	perlbench	cactusADM
mmmm-2	hmmer	sjeng	hmmer	tonto
mmmm-3	cactusADM	tonto	perlbench	tonto
mmmm-4	cactusADM	perlbench	h264ref	sjeng
mmmh-0	astar	tonto	h264ref	gamess
mmmh-1	h264ref	perlbench	cactusADM	xalancbmk

mmmh-2	h264ref	wrf	sjeng	gcc
mmmh-3	tonto	bzip2	astar	lbm
mmmh-4	perlbench	wrf	perlbench	xalancbmk
mmhh-0	perlbench	cactusADM	sphinx3	xalancbmk
mmhh-1	astar	cactusADM	gamess	lbm
mmhh-2	cactusADM	sjeng	gamess	sphinx3
mmhh-3	h264ref	hmmer	lbm	sphinx3
mmhh-4	sjeng	perlbench	gamess	povray
mhhh-0	wrf	lbm	povray	gcc
mhhh-1	cactusADM	povray	lbm	lbm
mhhh-2	wrf	povray	povray	gcc
mhhh-3	h264ref	omnetpp	xalancbmk	omnetpp
mhhh-4	perlbench	soplex	gcc	sphinx3
hhhh-0	sphinx3	gcc	povray	gcc
hhhh-1	xalancbmk	povray	omnetpp	soplex
hhhh-2	sphinx3	gcc	xalancbmk	omnetpp
hhhh-3	sphinx3	omnetpp	lbm	povray
hhhh-4	lbm	lbm	gamess	gamess

B | 8-core Workloads

Table B.1: 8-core Workloads

Workload	Benchmarks			
iiii-0	libquantum	milc	GemsFDTD	leslie3d
	zeusmp	gobmk	zeusmp	GemsFDTD
iiii-1	bwaves	libquantum	zeusmp	dealII
	namd	libquantum	libquantum	libquantum
iiii-2	calculix	gobmk	bwaves	libquantum
	GemsFDTD	gobmk	namd	gromacs
iiii-3	dealII	dealII	namd	leslie3d
	GemsFDTD	namd	zeusmp	gobmk
iiii-4	namd	dealII	milc	zeusmp
	milc	dealII	bwaves	namd
iiim-0	namd	leslie3d	dealII	libquantum
	gobmk	dealII	sjeng	astar
iiim-1	dealII	zeusmp	bwaves	dealII
	libquantum	leslie3d	perlbench	bzip2
iiim-2	dealII	namd	milc	leslie3d
	zeusmp	dealII	astar	bzip2
iiim-3	bwaves	dealII	leslie3d	libquantum
	zeusmp	zeusmp	hmmer	perlbench
iiim-4	zeusmp	zeusmp	gromacs	libquantum
	bwaves	gromacs	cactusADM	sjeng
iiih-0	gromacs	gobmk	calculix	calculix
	libquantum	gobmk	omnetpp	lbm
iiih-1	namd	calculix	gromacs	leslie3d
	leslie3d	zeusmp	lbm	soplex
iiih-2	GemsFDTD	milc	namd	bwaves
	dealII	namd	sphinx3	lbm
iiih-3	bwaves	libquantum	libquantum	namd

	dealII	namd	povray	omnetpp
iiih-4	calculix	gromacs	namd	bwaves
	gobmk	gromacs	sphinx3	gcc
iimm-0	namd	dealII	gobmk	GemsFDTD
	cactusADM	sjeng	tonto	bzip2
iimm-1	calculix	milc	zeusmp	gobmk
	cactusADM	astar	h264ref	cactusADM
iimm-2	calculix	bwaves	milc	gromacs
	bzip2	hmmer	cactusADM	cactusADM
iimm-3	libquantum	namd	zeusmp	leslie3d
	wrf	cactusADM	h264ref	astar
iimm-4	namd	GemsFDTD	dealII	milc
	perlbench	bzip2	h264ref	bzip2
iimh-0	leslie3d	gromacs	zeusmp	leslie3d
	wrf	bzip2	soplex	sphinx3
iimh-1	dealII	namd	calculix	zeusmp
	bzip2	astar	omnetpp	soplex
iimh-2	GemsFDTD	gromacs	milc	calculix
	cactusADM	h264ref	gamess	gamess
iimh-3	calculix	namd	milc	calculix
	sjeng	h264ref	gamess	gcc
iimh-4	GemsFDTD	GemsFDTD	gromacs	dealII
	tonto	perlbench	lbm	omnetpp
iihh-0	gromacs	libquantum	calculix	zeusmp
	omnetpp	lbm	soplex	gamess
iihh-1	gromacs	namd	GemsFDTD	leslie3d
	povray	omnetpp	xalancbmk	soplex
iihh-2	calculix	leslie3d	namd	bwaves
	omnetpp	sphinx3	soplex	lbm
iihh-3	gobmk	calculix	libquantum	GemsFDTD
	omnetpp	sphinx3	soplex	sphinx3
iihh-4	milc	calculix	gobmk	bwaves
	povray	lbm	sphinx3	povray
immm-0	libquantum	namd	hmmer	tonto
	astar	sjeng	perlbench	h264ref
immm-1	calculix	namd	h264ref	sjeng
	tonto	tonto	wrf	sjeng
immm-2	milc	calculix	bzip2	h264ref
	cactusADM	astar	cactusADM	astar
immm-3	libquantum	milc	bzip2	cactusADM

	sjeng	tonto	tonto	sjeng
immm-4	leslie3d	gromacs	wrf	hmmer
	sjeng	h264ref	astar	tonto
immh-0	gobmk	gobmk	astar	bzip2
	tonto	tonto	povray	lbm
immh-1	gromacs	libquantum	tonto	wrf
	h264ref	perlbench	xalancbmk	omnetpp
immh-2	libquantum	leslie3d	hmmer	perlbench
	wrf	h264ref	lbm	omnetpp
immh-3	GemsFDTD	libquantum	sjeng	hmmer
	h264ref	hmmer	omnetpp	gcc
immh-4	bwaves	libquantum	h264ref	cactusADM
	perlbench	cactusADM	lbm	gamess
imhh-0	gromacs	gromacs	bzip2	cactusADM
	soplex	xalancbmk	povray	xalancbmk
imhh-1	gromacs	calculix	hmmer	astar
	omnetpp	gcc	sphinx3	gcc
imhh-2	gromacs	bwaves	hmmer	astar
	povray	sphinx3	gcc	sphinx3
imhh-3	dealII	GemsFDTD	astar	tonto
	lbm	xalancbmk	xalancbmk	gamess
imhh-4	libquantum	gobmk	h264ref	sjeng
	soplex	sphinx3	omnetpp	gcc
ihhh-0	dealII	milc	sphinx3	gcc
	xalancbmk	xalancbmk	xalancbmk	xalancbmk
ihhh-1	namd	calculix	gamess	omnetpp
	soplex	gcc	gamess	povray
ihhh-2	calculix	dealII	omnetpp	soplex
	xalancbmk	sphinx3	gcc	sphinx3
ihhh-3	gromacs	namd	sphinx3	gcc
	xalancbmk	sphinx3	gcc	sphinx3
ihhh-4	milc	libquantum	xalancbmk	gcc
	xalancbmk	xalancbmk	povray	soplex
mmmm-0	hmmer	perlbench	bzip2	astar
	tonto	hmmer	wrf	sjeng
mmmm-1	wrf	hmmer	sjeng	cactusADM
	wrf	sjeng	sjeng	astar
mmmm-2	astar	perlbench	hmmer	hmmer
	bzip2	h264ref	hmmer	astar
mmmm-3	cactusADM	bzip2	bzip2	wrf

	sjeng	hmmer	perlbench	astar
mmmm-4	sjeng	hmmer	hmmer	hmmer
	hmmer	perlbench	sjeng	h264ref
mmmh-0	h264ref	perlbench	h264ref	sjeng
	astar	cactusADM	omnetpp	sphinx3
mmmh-1	h264ref	sjeng	cactusADM	tonto
	wrf	hmmer	sphinx3	sphinx3
mmmh-2	h264ref	hmmer	tonto	h264ref
	tonto	astar	omnetpp	sphinx3
mmmh-3	hmmer	h264ref	cactusADM	perlbench
	h264ref	bzip2	sphinx3	gamess
mmmh-4	perlbench	tonto	cactusADM	tonto
	wrf	astar	soplex	gamess
mmhh-0	bzip2	astar	astar	bzip2
	xalancbmk	omnetpp	gamess	omnetpp
mmhh-1	perlbench	hmmer	sjeng	tonto
	gamess	lbm	gcc	gcc
mmhh-2	astar	tonto	h264ref	astar
	soplex	xalancbmk	soplex	sphinx3
mmhh-3	cactusADM	h264ref	wrf	astar
	sphinx3	xalancbmk	omnetpp	sphinx3
mmhh-4	cactusADM	cactusADM	h264ref	cactusADM
	gcc	omnetpp	gcc	xalancbmk
mhhh-0	sjeng	h264ref	povray	omnetpp
	xalancbmk	soplex	omnetpp	soplex
mhhh-1	perlbench	wrf	gamess	xalancbmk
	lbm	gamess	sphinx3	xalancbmk
mhhh-2	bzip2	astar	gcc	lbm
	gamess	soplex	soplex	xalancbmk
mhhh-3	perlbench	wrf	gamess	lbm
	sphinx3	povray	povray	xalancbmk
mhhh-4	astar	sjeng	povray	lbm
	gcc	gcc	soplex	povray
hhhh-0	omnetpp	gcc	xalancbmk	gcc
	xalancbmk	soplex	gamess	omnetpp
hhhh-1	omnetpp	omnetpp	povray	lbm
	sphinx3	soplex	xalancbmk	povray
hhhh-2	xalancbmk	soplex	xalancbmk	povray
	povray	soplex	povray	omnetpp
hhhh-3	sphinx3	xalancbmk	xalancbmk	povray

	omnetpp	gamess	povray	gamess
hhhh-4	povray	povray	lbm	lbm
	sphinx3	gamess	lbm	sphinx3

C | 16-core Workloads

Table C.1: 16-core Workloads

Workload	Benchmarks			
iiii-0	leslie3d	calculix	bwaves	leslie3d
	milc	dealII	zeusmp	gobmk
	dealII	gobmk	zeusmp	milc
	gromacs	calculix	zeusmp	gobmk
iiii-1	gromacs	milc	milc	GemsFDTD
	zeusmp	bwaves	dealII	GemsFDTD
	namd	GemsFDTD	leslie3d	milc
	zeusmp	namd	calculix	milc
iiii-2	dealII	zeusmp	GemsFDTD	zeusmp
	gromacs	zeusmp	libquantum	namd
	libquantum	dealII	zeusmp	calculix
	calculix	GemsFDTD	calculix	GemsFDTD
iiii-3	bwaves	zeusmp	calculix	zeusmp
	milc	libquantum	dealII	milc
	bwaves	gromacs	calculix	bwaves
	milc	namd	gromacs	bwaves
iiii-4	gromacs	GemsFDTD	dealII	leslie3d
	dealII	dealII	GemsFDTD	leslie3d
	gobmk	leslie3d	dealII	bwaves
	milc	namd	GemsFDTD	gobmk
iiim-0	zeusmp	gromacs	calculix	gromacs
	GemsFDTD	bwaves	gromacs	calculix
	calculix	GemsFDTD	libquantum	GemsFDTD
	sjeng	h264ref	bzip2	bzip2
iiim-1	leslie3d	leslie3d	GemsFDTD	gromacs
	calculix	libquantum	libquantum	GemsFDTD
	milc	leslie3d	gobmk	calculix
	cactusADM	wrf	h264ref	bzip2
iiim-2	namd	bwaves	milc	milc
	GemsFDTD	libquantum	milc	milc

	dealII	calculix	bwaves	calculix
	perlbench	cactusADM	perlbench	astar
iiim-3	dealII	namd	calculix	calculix
	dealII	zeusmp	namd	calculix
	namd	gromacs	namd	calculix
	tonto	h264ref	bzip2	tonto
iiim-4	zeusmp	namd	zeusmp	dealII
	namd	namd	GemsFDTD	gromacs
	leslie3d	calculix	gobmk	namd
	bzip2	astar	wrf	astar
iiih-0	bwaves	dealII	dealII	bwaves
	GemsFDTD	milc	milc	bwaves
	calculix	zeusmp	libquantum	GemsFDTD
	omnetpp	xalancbmk	gamess	gamess
iiih-1	gromacs	leslie3d	GemsFDTD	bwaves
	GemsFDTD	milc	namd	milc
	namd	libquantum	gobmk	libquantum
	xalancbmk	sphinx3	povray	xalancbmk
iiih-2	gromacs	namd	gobmk	GemsFDTD
	leslie3d	gobmk	zeusmp	bwaves
	zeusmp	libquantum	GemsFDTD	milc
	povray	sphinx3	omnetpp	lbm
iiih-3	gobmk	bwaves	zeusmp	gobmk
	gobmk	libquantum	gromacs	gobmk
	zeusmp	GemsFDTD	zeusmp	libquantum
	sphinx3	lbm	lbm	sphinx3
iiih-4	namd	calculix	namd	dealII
	calculix	milc	leslie3d	libquantum
	bwaves	leslie3d	gobmk	calculix
	xalancbmk	sphinx3	sphinx3	omnetpp
iimm-0	GemsFDTD	calculix	milc	dealII
	namd	GemsFDTD	GemsFDTD	dealII
	wrf	hmmer	cactusADM	h264ref
	hmmer	h264ref	cactusADM	tonto
iimm-1	GemsFDTD	calculix	namd	libquantum
	namd	dealII	zeusmp	milc
	sjeng	perlbench	h264ref	hmmer
	perlbench	sjeng	astar	wrf
iimm-2	leslie3d	milc	libquantum	gobmk
	dealII	zeusmp	GemsFDTD	leslie3d
	bzip2	hmmer	tonto	bzip2
	perlbench	tonto	wrf	sjeng
iimm-3	GemsFDTD	leslie3d	GemsFDTD	dealII
	gromacs	milc	calculix	milc

	tonto h264ref	cactusADM hmmer	tonto cactusADM	wrf perlbench
iimm-4	zeusmp namd perlbench hmmer	gobmk gromacs perlbench tonto	leslie3d leslie3d tonto cactusADM	bwaves bwaves sjeng sjeng
iimh-0	gobmk namd hmmer lbm	namd GemsFDTD astar povray	dealII gobmk perlbench gcc	gromacs leslie3d h264ref xalancbmk
iimh-1	gromacs gromacs tonto lbm	zeusmp libquantum astar gcc	milc bwaves cactusADM omnetpp	libquantum zeusmp bzip2 lbm
iimh-2	leslie3d libquantum sjeng povray	leslie3d libquantum cactusADM xalancbmk	bwaves leslie3d cactusADM povray	calculix libquantum sjeng sphinx3
iimh-3	namd namd hmmer omnetpp	milc bwaves astar omnetpp	milc dealII perlbench omnetpp	zeusmp libquantum wrf soplex
iimh-4	gobmk GemsFDTD wrf omnetpp	leslie3d calculix hmmer gamess	zeusmp gromacs sjeng omnetpp	leslie3d zeusmp astar gcc
iihh-0	GemsFDTD GemsFDTD lbm gamess	calculix dealII xalancbmk omnetpp	calculix GemsFDTD xalancbmk omnetpp	gobmk gobmk sphinx3 soplex
iihh-1	bwaves dealII omnetpp povray	gobmk gobmk lbm lbm	zeusmp milc omnetpp sphinx3	calculix bwaves gcc lbm
iihh-2	GemsFDTD gromacs soplex povray	dealII zeusmp sphinx3 omnetpp	gromacs zeusmp sphinx3 gcc	gobmk namd lbm xalancbmk
iihh-3	gobmk bwaves soplex gamess	milc gromacs sphinx3 soplex	bwaves zeusmp gcc lbm	GemsFDTD gromacs lbm omnetpp
iihh-4	gromacs bwaves	bwaves dealII	namd zeusmp	libquantum bwaves

	povray	sphinx3	lbm	gamess
	sphinx3	xalancbmk	gcc	omnetpp
immm-0	dealII	libquantum	bwaves	namd
	tonto	bzip2	perlbench	perlbench
	astar	bzip2	astar	wrf
	sjeng	hmmer	h264ref	h264ref
immm-1	gobmk	dealII	milc	milc
	cactusADM	h264ref	tonto	h264ref
	perlbench	tonto	sjeng	h264ref
	h264ref	astar	sjeng	perlbench
immm-2	bwaves	milc	calculix	gobmk
	sjeng	astar	perlbench	cactusADM
	h264ref	sjeng	sjeng	sjeng
	sjeng	h264ref	bzip2	perlbench
immm-3	milc	zeusmp	zeusmp	calculix
	h264ref	astar	wrf	wrf
	h264ref	h264ref	bzip2	astar
	bzip2	sjeng	h264ref	hmmer
immm-4	gromacs	bwaves	dealII	namd
	sjeng	cactusADM	wrf	wrf
	sjeng	wrf	wrf	hmmer
	hmmer	perlbench	h264ref	bzip2
immh-0	namd	calculix	namd	GemsFDTD
	cactusADM	tonto	sjeng	bzip2
	tonto	perlbench	sjeng	h264ref
	gamess	soplex	gcc	lbm
immh-1	dealII	zeusmp	milc	GemsFDTD
	wrf	wrf	perlbench	bzip2
	sjeng	sjeng	perlbench	astar
	soplex	xalancbmk	omnetpp	omnetpp
immh-2	gobmk	GemsFDTD	bwaves	milc
	perlbench	tonto	sjeng	hmmer
	h264ref	perlbench	h264ref	hmmer
	gcc	gcc	omnetpp	xalancbmk
immh-3	calculix	GemsFDTD	libquantum	gromacs
	cactusADM	cactusADM	sjeng	cactusADM
	sjeng	perlbench	tonto	sjeng
	xalancbmk	omnetpp	lbm	gamess
immh-4	leslie3d	bwaves	calculix	namd
	astar	wrf	bzip2	astar
	astar	sjeng	sjeng	h264ref
	omnetpp	gamess	soplex	omnetpp
imhh-0	dealII	gromacs	leslie3d	calculix
	tonto	h264ref	astar	tonto

	gamess	gcc	xalancbmk	lbm
	soplex	gcc	lbm	soplex
imhh-1	namd	gromacs	bwaves	milc
	perlbench	bzip2	sjeng	hmmer
	gcc	gamess	lbm	sphinx3
	sphinx3	xalancbmk	gamess	povray
imhh-2	GemsFDTD	namd	leslie3d	GemsFDTD
	hmmer	perlbench	h264ref	hmmer
	lbm	soplex	soplex	omnetpp
	lbm	gcc	gcc	lbm
imhh-3	gromacs	leslie3d	gromacs	leslie3d
	hmmer	astar	h264ref	bzip2
	povray	sphinx3	sphinx3	povray
	xalancbmk	gcc	gcc	lbm
imhh-4	bwaves	namd	libquantum	gobmk
	hmmer	sjeng	astar	astar
	povray	omnetpp	povray	sphinx3
	sphinx3	gamess	gcc	gcc
ihhh-0	dealII	libquantum	libquantum	gromacs
	gcc	lbm	povray	omnetpp
	gamess	povray	gamess	omnetpp
	gamess	gamess	sphinx3	lbm
ihhh-1	GemsFDTD	dealII	namd	libquantum
	sphinx3	xalancbmk	omnetpp	gcc
	gcc	soplex	lbm	povray
	soplex	povray	povray	xalancbmk
ihhh-2	namd	milc	zeusmp	GemsFDTD
	omnetpp	xalancbmk	gcc	sphinx3
	gamess	gcc	gcc	povray
	soplex	lbm	soplex	lbm
ihhh-3	dealII	leslie3d	namd	dealII
	omnetpp	gamess	sphinx3	soplex
	gamess	soplex	povray	sphinx3
	lbm	xalancbmk	lbm	povray
ihhh-4	bwaves	libquantum	leslie3d	libquantum
	lbm	gamess	xalancbmk	soplex
	gamess	omnetpp	lbm	lbm
	sphinx3	gamess	gamess	povray
mmmm-0	tonto	cactusADM	sjeng	bzip2
	tonto	bzip2	sjeng	cactusADM
	cactusADM	bzip2	tonto	h264ref
	h264ref	cactusADM	perlbench	sjeng
mmmm-1	tonto	wrf	perlbench	wrf
	bzip2	sjeng	h264ref	wrf

	wrf astar	h264ref hmmer	astar h264ref	wrf hmmer
mmmm-2	h264ref cactusADM wrf hmmer	tonto bzip2 perlbench h264ref	bzip2 wrf astar wrf	perlbench bzip2 tonto cactusADM
mmmm-3	perlbench sjeng sjeng perlbench	sjeng h264ref astar bzip2	sjeng bzip2 hmmer bzip2	hmmer hmmer tonto sjeng
mmmm-4	perlbench hmmer hmmer sjeng	wrf h264ref bzip2 wrf	astar hmmer tonto wrf	h264ref hmmer hmmer astar
mmmh-0	bzip2 perlbench wrf lbm	cactusADM perlbench hmmer xalancbmk	hmmer bzip2 cactusADM lbm	sjeng wrf wrf omnetpp
mmmh-1	perlbench bzip2 h264ref lbm	hmmer astar tonto povray	perlbench perlbench astar omnetpp	wrf hmmer wrf sphinx3
mmmh-2	astar astar perlbench gcc	sjeng h264ref wrf gamess	tonto bzip2 cactusADM povray	tonto bzip2 wrf povray
mmmh-3	h264ref hmmer hmmer gcc	h264ref sjeng tonto lbm	perlbench h264ref bzip2 povray	cactusADM tonto tonto omnetpp
mmmh-4	tonto wrf hmmer gamess	perlbench bzip2 hmmer xalancbmk	sjeng tonto tonto sphinx3	sjeng tonto perlbench gamess
mmhh-0	bzip2 bzip2 gcc povray	sjeng wrf gcc xalancbmk	cactusADM bzip2 sphinx3 xalancbmk	wrf sjeng gamess omnetpp
mmhh-1	tonto astar gcc lbm	tonto wrf xalancbmk omnetpp	h264ref tonto lbm soplex	sjeng perlbench gamess gcc
mmhh-2	bzip2 tonto	wrf perlbench	hmmer cactusADM	tonto astar

	sphinx3	xalancbmk	povray	sphinx3
	gcc	lbm	xalancbmk	omnetpp
mmhh-3	wrf	sjeng	bzip2	tonto
	astar	hmmer	perlbench	wrf
	gcc	povray	soplex	sphinx3
	lbm	lbm	gamess	soplex
mmhh-4	hmmer	h264ref	tonto	astar
	astar	sjeng	astar	bzip2
	lbm	soplex	povray	povray
	gcc	soplex	gamess	xalancbmk
mhhh-0	cactusADM	cactusADM	bzip2	cactusADM
	povray	soplex	soplex	lbm
	lbm	xalancbmk	gamess	gcc
	gamess	gcc	xalancbmk	omnetpp
mhhh-1	astar	hmmer	hmmer	sjeng
	gamess	sphinx3	lbm	povray
	xalancbmk	gcc	gcc	lbm
	lbm	soplex	gamess	xalancbmk
mhhh-2	astar	perlbench	bzip2	perlbench
	omnetpp	omnetpp	omnetpp	lbm
	sphinx3	sphinx3	lbm	omnetpp
	sphinx3	omnetpp	omnetpp	sphinx3
mhhh-3	bzip2	astar	tonto	perlbench
	sphinx3	gamess	lbm	soplex
	omnetpp	omnetpp	omnetpp	omnetpp
	sphinx3	povray	omnetpp	gamess
mhhh-4	sjeng	hmmer	perlbench	sjeng
	gcc	lbm	povray	gamess
	sphinx3	lbm	sphinx3	gcc
	sphinx3	povray	gcc	gamess
hhhh-0	lbm	soplex	lbm	soplex
	sphinx3	sphinx3	gcc	lbm
	lbm	soplex	gcc	lbm
	omnetpp	xalancbmk	xalancbmk	soplex
hhhh-1	sphinx3	omnetpp	povray	sphinx3
	omnetpp	povray	povray	lbm
	gamess	gamess	sphinx3	xalancbmk
	omnetpp	soplex	sphinx3	gamess
hhhh-2	gcc	sphinx3	gamess	povray
	sphinx3	povray	povray	lbm
	omnetpp	povray	povray	soplex
	povray	xalancbmk	omnetpp	soplex
hhhh-3	soplex	xalancbmk	xalancbmk	gcc
	lbm	sphinx3	xalancbmk	omnetpp

	sphinx3 povray	sphinx3 povray	lbm xalancbmk	sphinx3 gamess
hhhh-4	omnetpp xalancbmk xalancbmk gcc	lbm gamess povray xalancbmk	sphinx3 omnetpp gcc xalancbmk	soplex omnetpp lbm povray