Evaluation of Cache Management Algorithms for Shared Last Level Caches

Runar Bergheim Olsen

Master of Science in Computer Science
Submission date: June 2015
Supervisor: Magnus Jahre, IDI
Co-supervisor: Antonio Garcia Guirado, IDI

Norwegian University of Science and Technology
Department of Computer and Information 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.

In recent years, a large number of shared cache management schemes have been proposed to alleviate this problem. The main aim of this project is shed some light on the relative strengths and weaknesses of the different cache management techniques. The project must contain a review of recently proposed cache management techniques and identify similarities and differences. The student should implement at least one cache management technique and compare its performance to a conventional Least Recently Used (LRU)-managed cache. Additional cache management techniques should be implemented and evaluated if time permits.

Abstract

The performance gap between processors and main memory has been growing over the last decades. Fast memory structures know as caches were introduced to mitigate some of the effects of this gap. After processor manufacturers reached the limits of single core processors performance in the early 2000s, multicore processors have become common. Multicore processors commonly share cache space between cores, and algorithms that manage access to shared cache structures have become an important research topic. Many researchers have presented algorithms that are supposed to improve the performance of multicore processors by modifying cache policies. In this thesis, we present and evaluate several recent and important works in the cache management field. We present a simulation framework for evaluation of various cache management algorithms, based on the Sniper simulation system. Several of the presented algorithms are implemented; Thread Aware Dynamic Insertion Policy (TADIP), Dynamic Re-Reference Interval Prediction (DRRIP), Utility Cache Partition (UCP), Promotion/Insertion Pseduo-Partitioning (PIPP), and Probabilistic Shared Cache Management (PriSM). The implemented algorithms are evaluated against the commonly used Least Recently Used (LRU) replacement policy and each other. In addition, we perform five sensitivity analysis experiments, exploring algorithm sensitivity to changes the simulated architecture. In total data from almost 9000 simulation runs is used in our evaluation.

Our results suggest that all implemented algorithms mostly perform as good as or better than LRU in 4-core architectures. In 8- and 16-core architectures some of the algorithms, especially PIPP, perform worse than LRU. Throughout all our experiments UCP, the oldest of the evaluated alternative to LRU, is the best performer with an average performance increase of about 5%. We also show that UCP performance increases to more than 20% when available cache and memory resources are reduced.

Sammendrag

Ytelsesforskjellen mellom prosessorer og hovedminne har økt gjennom de siste tiår. Raske minnestrukturer kjent som hurtigbuffer vart introdusert for å redusere effekten av den økende forskjellen. Etter at produsenter møtte grensen for enkjerneprosessorytelse på starten av 2000-tallet har flerkjerneprosessorer blitt vanlige. Flerkjerneprosessorer deler vanligvis noe hurtigbuffer mellom kjernene, og algoritmer som kontrolerer kjernenes tilgang til det delte området har blitt et viktig forskningsområdet. Flere forskere har presenter algoritmer som skal kunne øke ytelsen til flerkjerneprosessorer ved å endre algoritmen som styrer hurtigbufferet. I denne avhandlingen presenterer og evaluerer vi flere nylig publiserte og viktige arbeid som omhandler kontroll av delt hurtigbuffer. Vi presenterer et simuleringssystem som vi bruker til å evaluerer flere algoritmer, basert på simuleringssystemet Sniper. Flere av de presenterte algoritmene er implementert; Thread Aware Dynamic Insertion Policy (TADIP), Dynamic Re-Reference Interval Prediction (DRRIP), Utility Cache Partition (UCP), Promotion/Insertion Pseduo-Partitioning (PIPP), og Probabilistic Shared Cache Management (PriSM). De implementerte algoritmene er evaluert mot den ofte brukte algoritmen Least Recently Used (LRU) og hverandre. I tillegg utfører vi fem sensitivitetseksperimenter, hvor vi utforsker algoritmenes sensitivitet til endringer i den simulerte arkitekturen. Totalt bruker vi data fra over 9000 simuleringer i vår evaluering.

Våre resultat viser at alle de implementerter algoritmene for det meste yter like bra eller bedre enn LRU på 4-kjerne arkitekturer. For 8- og 16-kjerne arkitekturer yter noen algoritmer, spesielt PIPP, dårligere enn LRU. Vi viser også at UCP, den eldste av de implementerte alternativet til LRU, gir best resultat på alle våre testprogrammer med gjennomsnittlig ytelsesøkelse på 5%. Vi viser også at UCP ytelsen øker til over 20% når vi begrenser tilgjengelig hurtigbuffer og minne resursser.

Preface

This thesis is submitted to the Norwegian University of Science and Technology.

Acknowledgement

The author would like to thank Associate Professor Magnus Jahre and Dr. Antonio Garcia Guirado at the Department of Computer and Information Science, Norwegian University of Science and Technology, for their excellent guidance and useful critique during the planning and realization of this thesis. The author would also like to thank his fiancée and family for all their support and encouragement throughout his studies. All of our experiments were run on the Stallo cluster under Nortur grant nn4650k.

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

ANTT Average Normalized Turnaround Time. 40

ATD Auxilliary Tag Directories. 13–15, 17, 20, 21, 24, 25, 29, 30

BIP Binominal Insertion Policy. 12–17, 25, 26

BRRIP Binominal RRIP. 17, 30

CLU Co-Optimizing Locality and Utility. 25

CMP Chip Multiprocessor. i, 2–6, 9, 32

CSMB Clock Skew Minimization Barrier. 50, 51, 60

DIP Dynamic Insertion Policy. 12–15, 17, 25

DRRIP Dynamic Re-Reference Interval Prediction. ii, iii, 16, 17, 29, 30, 44, 52, 53, 55, 57, 62

HMS Harmonic Mean of Speedup. 39–45, 50, 51, 57, 58, 60, 62, 63

ILP Instruction-Level Parallelism. 1, 2, 49

LIP LRU Insertion Policy. 12–16, 25, 26

LLC Last Level Cache. 5, 25, 37, 38, 40, 57, 58, 61

LRU Least Recently Used. i–iii, 7–9, 11–14, 16–21, 23–25, 29, 32, 35, 41–45, 51–56, 59, 62, 63

MPKI Misses Per Kilo Instruction. 40, 42–45, 51, 55–57

MRU Most Recently Used. 12–14, 17, 19, 21, 22, 24, 25

MSB Most Significant Bit. 13

MSHR Miss Status Holding Registers. 49

NP Normalized Progress. 39, 40

NRU Not Recently Used. 17

NTT Normalized Turnaround Time. 40

NUCache Next Use Cache. 18, 19, 25

PIPP Promotion/Insertion Pseduo-Partitioning. ii, iii, 21–25, 29, 30, 41, 42, 44, 51, 53–57, 62

PriSM Probabilistic Shared Cache Management. ii, iii, 23, 24, 29, 30, 44, 45, 52, 53, 56, 57, 62

QoS Quality of Service. 9, 24

SRRIP Static RRIP. 17, 30

STP System Throughput. 39–45, 51, 55–58, 60, 62, 63

TADIP Thread Aware Dynamic Insertion Policy. ii, iii, 14–17, 25, 29, 30, 44, 51, 53, 55, 57, 62

TADIP-F TADIP-F. 15, 16, 29

TADIP-I TADIP-I. 15

UCP Utility Cache Partition. ii, iii, 20, 21, 24, 25, 29, 30, 41–45, 52–54, 56–58, 62, 63

UMON Utility Monitor. 20, 21, 23, 29, 30, 53

Chapter 1

Introduction

In this chapter, we will first introduce chip multiprocessors and their memory systems. We will also introduce the role of a cache management algorithm, and introduce a set of common memory access patterns. Then, we will analyze our problem description and introduce a set of requirements the thesis has to fulfill. Finally, an overview of our contributions is given before we provide an outline of the thesis.

1.1 Chip Multiprocessors

Moore's Law [22], an observation made by G. E. Moore one of the co-founders of Intel, has been the driving force behind processor development in the last decades. The law is simply an observation; that scaling of transistors used to make integrated circuits will allow for approximately twice the number of transistors per die every 18 months. Up until the mid-2000s, manufacturers used these smaller transistors to increase single-core performance. Smaller transistors allowed for increased frequency, and more transistors per die allowed for increasingly complex processor cores. Features such as speculative and out-of-order execution were added to take advantage of Instruction-Level Parallelism (ILP) present in computer programs. By the mid-2000s, processor cores had become so complex and were running at such high frequencies that manufacturers had reached the limitation known as the power wall. The power wall entails that manufacturers were unable to continue increasing the frequency and the transistor count of each core, without also attaching high-performance cooling systems to counter the increased power usage and hence increased heat generation. Systems like water or even nitrogen cooling were needed to continue the single core performance increase [30]; these systems are not practical for personal computers.

Figure 1.1 shows the single core performance development from 1980 to 2010. The effect of the power wall is clearly visible from 2005 and out where we observe no improvement in single core performance. We also observe that in the five years leading to the power wall, the yearly performance improvement decreased. This

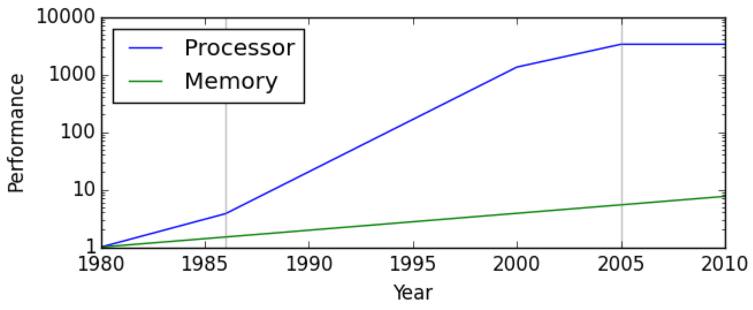


Figure 1.1: Comparison of (single core) processor and memory performance from 1980 to 2010 based on data collected by J. Hennessy and D. Patterson [12].

decrease is observed because improving single core performance became harder, as more advanced techniques for ILP exploiting where needed, ultimately resulting in the power wall.

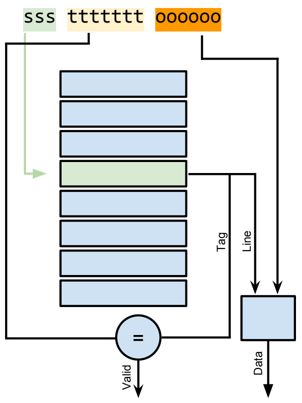
By 2005, most manufacturers had abandoned their plans for increased single core performance and were all working towards Chip Multiprocessors (CMPs) [30]. CMPs are built using several simpler processing cores without many of the aggressive ILP utilization features that were added to single core processors in the late 2000s to increase performance. This development is based on Pollack's rule [3], which states that performance increases proportionally to the square root of the area, and that power usage increases proportionally to the area. Consider having two cores, one large highly complex core, and a simple core at half the size. The performance of the larger core will by Pollack's rule be worse than the combined performance[1] of the two smaller while having roughly the same power consumption.

Moving to simpler and smaller cores allowed manufacturers to continue overall performance improvements while staying below the power wall. With an increasing number of transistors due to scaling, even more processing cores can be added to CMPs. Today CMPs are the de-facto standard, being used in everything from embedded computers [1] and mobile phones [13] to commercial and high-performance computing [31, 15].

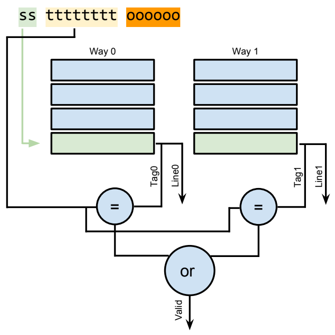
1.2 CMP Memory System

Memory is a vital component of any computing system, without memory we are unable to store our programs and computations. Traditionally the technology used to create memory have differed from the technology used in processors [32]. As processor performance increased, memory performance did not increase proportionally. This development has resulted in what is known as the processor-memory gap [32]. Figure 1.1 shows how the gap in processor and memory performance has developed

[1]Single thread performance is only increased by creating more complex cores; multithreading techniques are required in software to take advantage of CMPs performance increase.



(a) Direct mapped cache. Each set contains only one block. The three most significant bits used for set addressing, 7-bit tags.



(b) 2-way associative cache. Each set contains two blocks. Two bits used for set addressing, 8-bit tags. Additional hardware (not shown) required to select output between line0 and line1.

Figure 1.2: Simplified architecture of a direct and 2-way cache under a read operation. With 64B lines and byte addressing.

from 1980 to 2010. The figure shows that memory performance has had a constant development over the period of about 7% increase yearly. The processor-memory gap made itself visible in 1986 when processor performance started increasing by about 52% each year. If not handled, the increasing processor-memory gap would become the limiting factor in processor performance, as processors are dependent on the memory system providing both code and data. In response to the growing gap small and fast memory structures known as caches were introduced. Caches are produced using different techniques than main memory, proving faster access. This comes at the cost of higher production cost and power usage. Today, most single-core processors and CMPs have one or more caches integrated on the processor die. The caches filter all memory request made by the processing core. If one of the caches has a copy of the requested data, it will stop the memory request from going to main memory and respond with the correct data. This is known as a cache hit. If a cache does not have the value of the requested address, it is known as a cache miss. Because caches are faster and are located closer to the processor than main memory they can to an extent hide the processor-memory gap, assuming a high hit rate in the caches.

Traditionally there are three main ways of organizing cache memory; direct mapped, set associative and fully associative. Caches used in commercial CMPs today are set associative memory structures [31, 15, 1, 13]. A cache is organized as a 2d array, where each row is called a set, and each set consist of one or more blocks, sometimes known as cache lines. A block is the minimum unit of data a cache stores, this is typically 64B. The cache divides all memory addresses into three portions, the set address, block tag and block offset. The set address is used

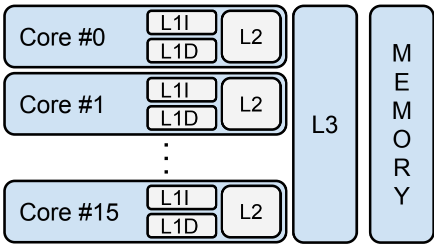


Figure 1.3: Generic Chip Multiprocessor Architecture.

to determine which set is responsible for caching a value. Within a set, all blocks are valid storage locations. Each block that contains valid data also stores the block tag of the data it contains. During a cache lookup, all valid blocks will be scanned looking for one containing the block tag from the address. In a direct mapped cache, each set contains only one block, making the block scan trivial. Fully associative caches have all blocks in a single set, and the block scan require that every block be checked. This is expensive both measured in hardware required, and the critical path of the cache. Set associative caches, also known as n-way caches, are a middle ground organization that stores n blocks per set. Figure 1.2 shows both a direct mapped and a 2-way cache under a read operation. The figure does not show a fully associative cache organization but given the 2-way cache organization a fully associative cache is created by duplicating the block scan hardware until each column has only one row. In this thesis, we will show that increasing the number of ways can improve performance of a cache via increased hit rate. However, the access time and complexity of a cache also increases as the number of ways increases. As a result, first level caches are often limited to 2- or 4-ways [27], because the access latency must be short to prevent the CPU stalling while it waits for data. Even third level caches rarely exceed 16- or 32-ways. Alternative cache organizations have been proposed, such as zcache [27], that allows for higher associativity while providing access times comparable to traditional caches. However, these techniques are not currently used in commercial CMPs and hence are not considered further in this thesis.

When data not present in the cache is requested by a processing core, a request is sent to the next cache level, or possibly the main memory. Once the cache receives a response, it normally stores the data to provide faster access times in case of a re-reference. Caches also normally store data that is written to memory by a processing core, speeding up the write operation. For a set-associative or fully associative cache, there are multiple valid storage locations for a data block. An algorithm known as the cache management algorithm decides which of the valid blocks are used to store the value. If the block that was chosen by the algorithm already contains valid data the cache removes or evicts the existing data.

In the memory hierarchy, smaller means faster. For instance, main memory is much faster than disks, but disks can store much more data. The same is valid for caches, a smaller cache has a shorter critical path and hence a lower access time compared to a larger cache. Figure 1.3 shows an example 16-core CMP architecture with three cache levels and main memory. For each cache level, both the size and access time increases. First level caches are the smallest while the third level caches are largest. To save area on CMPs, and also to provide an easy mechanism for data sharing between cores, it is common to have at least one level of shared cache. In Figure 1.3 each core has a private L1 and L2 cache, and a shared Last Level Cache (LLC), the L3 cache. While sharing cache can improve performance, and improve overall utilization, it also makes the memory system exposed to destructive interference that potentially can hurt the performance of all processing cores. The cache management algorithm, running on the shared cache, may either ignore the effects of interference or it may attempt to reduce it by some form of prioritization.

1.3 Common Memory Access Patterns

In this section, we will define four memory access patterns [17]; recency-friendly, trashing, streaming and combined access patterns. These patterns will later be used to describe the strengths and weaknesses of the presented algorithms, and to explain algorithm performance in our experiments.

1.3.1 Recency-friendly

Several cache management algorithms make an assumption known as the recency-assumption. This is the assumption that recently accesses memory addresses have a higher probability of re-use that less recently accesses addresses. A recency-friendly access pattern is one that causes no misses under an algorithm that always keeps the most recently used blocks in the cache. Figure 1.4 illustrates an example memory access pattern that is said to be recency-friendly. Because the pattern repeats and k is less than the number of ways we know that the least recently used block will always be referenced before being evicted, if we assume a recency-assumption based algorithm. Hence, the algorithm will be able to provide hits for every access after the initial repetition, and the pattern is recency-friendly.

$$p_{rf} = (a_0 a_1 ... a_{k-1})^N$$

Figure 1.4: Recency-friendly access pattern (k <= number of ways, N > 1).

1.3.2 Trashing

A trashing memory pattern is one that repeats in a similar manner to a recency-friendly pattern, but with more blocks per cycle than the number of cache ways. Figure 1.5 shows an example of one such pattern, the only thing separating this from the previous pattern in the value of k. Under a recency-assumption based algorithm, an access pattern similar to this will never hit because all blocks are evicted before they are re-referenced. A better management algorithm for these patterns might keep some of the working set in the cache, providing hits for parts of the access pattern.

$$p_{rf} = (a_0 a_1 ... a_{k-1})^N$$

Figure 1.5: Trashing memory access pattern (k > number of ways, N > 1).

1.3.3 Streaming

A streaming memory access pattern is one that has no re-references, or where the period is so large that no pattern is detectable. Figure 1.6 illustrates one such pattern, where k is infinite. No caching algorithm can provide hits for such an access pattern, simply because there are no re-references.

$$p_{rf} = (a_0 a_1 ... a_{k-1})$$

Figure 1.6: Streaming memory access pattern (k = ∞).

1.3.4 Combined

In reality, memory access patterns can be more complex than the simple examples shown above. A single program might, for instance, behave both in a recency-friendly and streaming fashion. An example would be a program performing a reduction over a large dataset. Most accesses would be streaming as the program iterates over the dataset, but some accesses will exhibit recency-friendly behavior like when the program stores temporary results to memory during the iteration.

For shared caches, the observed pattern is the union of accesses from all cores. A particular non-optimal situation for a cache assuming recency-friendly behavior is when multiple cores execute recency-friendly applications, and one single core execute a streaming application. In this case, the streaming application will constantly clear the cache, degrading performance of the recency-friendly applications. An algorithm that could detect the streaming application and handle it as a special case could potentially increase the performance of the recency-friendly applications without affecting the performance of the streaming application. Some of the algorithms we cover in the following sections will attempt to detect streaming applications.

1.4 Requirements

By analyzing the problem description, we have been able to extract a set of requirements that this thesis has to fulfill:

R1 Introduce CMPs, their memory system, and the role of a cache management algorithm.

R2 Present recent and important work in the cache management field. Compare similarities and differences of the various proposed algorithms.

R3 Create a framework for evaluation of various cache management algorithms.

R4 Implement at least one of the presented algorithms and compare against a conventional Least Recently Used (LRU)-managed cache.

Based on the problem description and discussions with the thesis advisors, we list an additional set of optional requirements that the thesis may fulfill:

O1 Implement additional algorithms and evaluate them.

O2 Compare performance of the implemented algorithms against each other.

O3 Investigate algorithm sensitivity to changes in L2 cache size, L3 cache size, and/or memory bus bandwidth.

1.5 Contributions

In our work with this thesis we have made the following contributions:

- We have created a curated list with detailed descriptions of several recently published algorithms in the cache management field. The list has been limited to only consider algorithms that target conventional caches and optimize for cache miss minimization.
- A framework for evaluation of cache management algorithms has been built on top of the Sniper [5] simulation system.
- Several of the algorithms presented in our list of existing works have been implemented and tested within our simulation framework.
- We performed several sensitivity experiments further exploring the properties of our simulation framework and the implemented algorithms.
- The implemented simulation framework and all our results will at the end of this thesis be made available to NTNU and the CARD [4] research group for future research.

1.6 Outline

The outline of the rest of this thesis is as follows:

- Chapter 1 introduces the thesis by putting it in a historical context, presenting important background knowledge and summarizing the contributions made, fulfilling requirement **R1**.

- Chapter 2 presents a selection of cache management algorithms and provides a theoretical comparison of them, fulfilling requirement **R2**.
- Chapter 3 presents our simulator and the framework built on this simulator to enable evaluation of cache management algorithms. It also presents the subset of algorithms we have implemented. This fulfills requirement **R3** and **O1**.
- Chapter 4 contains a description of our simulated processor model and explains all metrics we later use to evaluate our experiments.
- Chapter 5 presents an experiment where we compare all implemented algorithms against an LRU managed cache. Also, we compare all implemented algorithms against each other. This fulfills requirement **R4** and **O2**.
- Chapter 6 presents five different experiments all exploring either simulation framework or algorithm sensitivity to various architectural changes, fulfilling requirement **O3**.
- Chapters 7 and 8 contain a discussion of our results and a conclusion based on these. Also, an overview of future work is given.

Chapter 2

Cache Management Algorithms

A cache management algorithm manages the storage space in a cache. It decides where to store new data blocks and which of the existing blocks are evicted to make room for new blocks. Some algorithms are thread-aware and geared towards shared caches. Others are thread-agnostic and work both for shared and private caches. Some have advanced optimization goals such as Quality of Service (QoS) while others use simpler metrics like miss minimization. Algorithms proposed for shared caches may in general be divided into two groups, those that explicitly divide storage space between cores sharing the cache and those that do not. The term cache replacement algorithm is often used to describe algorithms that do not divide the storage space while the term cache partitioning algorithm describe algorithms that do divide the space. Throughout this paper, we will use the two terms interchangeably.

The field of cache management is well researched, and there exists a large number of proposed algorithms. In this thesis, we present a few recently proposed algorithms and compare their performance. We will also present LRU, an algorithm that is thread-agnostic and widely used both in private and shared caches today. Table 2.1 lists the selected algorithms. Only algorithms that optimize for fewer cache misses are included. This metric is easy to measure and also makes it easy to compare the various algorithms. Also, we only consider algorithms that target conventional caches, as they are designed in CMPs today. This limitation makes the comparison of results from different algorithms easier. Also, we avoid having to extend our simulator with a new cache type, which would be unfeasible given the time constraints of this thesis.

Name	Year	Thread aware	Repl. policy	Insert. policy	Promo. olicy	Hardware overhead[1]	Partition
DIP	2007	No	LRU	LIP/ BIP	Promote to MRU	1 counter, set dueling	No
TADIP	2008	Yes	LRU	LIP/ BIP	Promote to MRU	1 counter per core, set dueling	No
DRRIP	2010	Yes	LRU approx.	DRRIP/ BRRIP	Stepwise promotion	1 counter per core, set dueling	No
NUCache	2011	No	LRU + Deli-Ways	LIP	Promote to MRU	NUTrack	No
UCP	2006	Yes	Per core LRU	LIP	Promote to MRU	UMON, 1 ATD per core	Yes
PIPP	2009	Yes	LRU	Utility position	Stepwise promotion	UMON, 1 ATD per core, random generator	Yes
PriSM	2012	Yes	Per core LRU	LIP	Promote to MRU	1 ATD per core, random generator	Yes
CLU	2014	Yes	LRU	LIP/ BIP	Promote to MRU	UMON ˜3 ATDs per core	Yes

Table 2.1: Overview of Cache Management Algorithms.

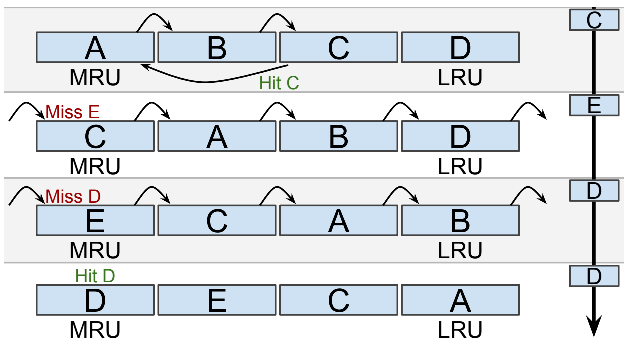


Figure 2.1: LRU managed 4-way cache set.

It is possible to divide all algorithms included in this evaluation into three distinct policies:

- The replacement policy specifies which block a cache set evicts when inserting a new block into that set.
- The insertion policy specifies the state of new blocks after insertion into the cache set.
- The promotion policy specifies how the state of a block changes following an access from a processor core.

In the following sections, we will explain how each of the selected cache partitioning algorithms work, with an emphasis on this division to make comparisons easier.

2.1 Cache Replacement Algorithms

This section covers cache replacement algorithms, or algorithms that do not explicitly divide the available cache space between cores.

2.1.1 LRU

LRU replacement, or some simplification of LRU, is one of the dominant cache management algorithms in hardware today. As a result, LRU is normally used as the baseline for comparisons when presenting new cache management algorithms [17, 25, 26].

The LRU algorithm relies on the temporal locality of data accesses; it assumes that recently accessed data has a higher reuse frequency than less recently accessed data. Theoretically one can envision a cache set managed by LRU as a stack, where recently accessed cache blocks are near the top and less recently accessed blocks are near the bottom. The bottom position of the stack is the LRU position, and the top

[1]Simplified hardware overhead compared to an LRU managed cached.

position is the Most Recently Used (MRU) position. In a hardware implementation, the blocks are not stored in a sorted fashion, but additional storage bits are used to keep track of LRU positions. The replacement policy of LRU is to evict the least recently used cache block, the one on the bottom of the LRU stack. The insertion and promotion policy of LRU is the same; a inserted or accessed block is always moved to the MRU position unless it is already there.

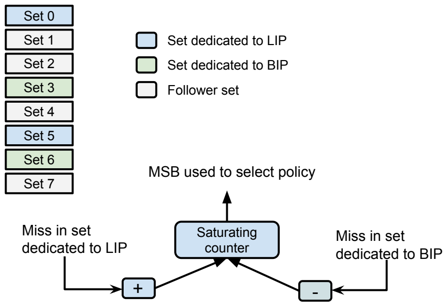
Figure 2.1 shows how a 4-way cache set managed by LRU replacement handles four requests. Initially, the set contains four blocks; A, B, C and D. A is in the MRU position while D is in the LRU position. The first request is for block C; this is a hit, and that causes the block to move to the MRU position, pushing both block A and B one step closer to the LRU position. The second request is made to block E; this block is not present in the cache. The LRU algorithm evicts block D at the LRU position and then places block E at the MRU position. Then a request for D follows, this is a miss and B is evicted. Finally, another request is made to block D. Nothing changes since the block already is at the MRU position.

One important result of the LRU insertion, promotion, and replacement policies is that an LRU managed cache satisfies the *stack property*. Given a 4-way LRU managed cache with counters that count the number of hits in each way; then we know that the number of misses in a 3-way LRU managed cache equals the misses in the 4-way cache plus the number of hits in the 4th cache way. This effect occurs because any request that hits in the 4th way of the 4-way cache would miss in the 3-way cache, but in both caches the requested block is then moved to the MRU position. No blocks can enter the cache set at any other position than the MRU. As a result, the state of the three first ways of the caches is always identical given an identical memory request sequence. The same argument holds for a 2-way cache, by summing the hits in the third and fourth way of the 4-way cache we get the additional misses in a 2-way cache. This property generalizes; we can find the relative miss rate of any LRU managed cache with 1 to n ways by having a single n-way cache with access counters per way. If we additionally have a total miss counter in the n-way cache we can also find the absolute number of misses for each cache size.

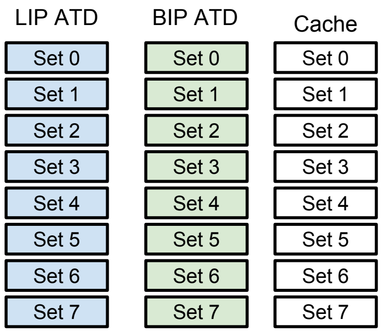
LRU is a simple replacement algorithm that is usable in both private and shared caches. In shared caches, LRU will favor access frequency, giving cores that issue many cache requests more cache space than those who issue fewer requests. In some cases, this might be an acceptable solution. However, as we will discover, several thread-aware replacement algorithms claiming to outperform LRU exists.

2.1.2 DIP

DIP [26] was originally proposed in 2007. The DIP algorithm views the cache set as a stack, as in LRU. Replacement and promotion policies are equal to LRU, DIP evicts the block at the LRU position, and following a cache hit a block moves to the MRU position. In contrast to LRU, DIP is a combination of two insertion policies, the standard LRU Insertion Policy (LIP) and Binominal Insertion Policy (BIP). LIP inserts new blocks at the MRU position. BIP inserts new blocks either at the LRU position or with a small probability, $p = \frac{1}{32}$, at the MRU position. The



(a) Set-dueling architecture.



(b) ATD architecture.

Figure 2.2: Alternate Dynamic Insertion Policy (DIP) organizations.

overall DIP algorithm switches between the two insertion policies by always using the one that is expected to cause fewer cache misses.

By mostly inserting at the LRU position the BIP insertion policy can theoretically handle trashing memory access patterns. When most new blocks enter at the LRU position, the upper parts of the LRU stack can contain blocks that have been re-referenced. In a trashing access pattern, this results in part of the working set residing in the upper part of the stack while the rest are inserted at the LRU position and evicted at the next miss. By sometimes inserting at the MRU position BIP will give blocks not referenced by the next miss a chance to stay in the cache. Inserting at the MRU position will also force stale cache blocks in the upper part of the stack to move towards the LRU position.

The authors of DIP present several methods to detect the best of the two replacement algorithms, one of them is set-dueling. Set-dueling is implemented by having some sets of the cache always use BIP and some always use LIP. A counter tracks the performance of the dueling sets. Misses in LIP sets will increment the counter and misses in BIP sets will decrement the counter. The Most Significant Bit (MSB) of the counter can then be used to select the best performing of the two algorithms. If the MSB is one, an overweight of misses in LIP sets are occurring, and BIP is the best performing algorithm. If the MSB is zero, then an overweight of BIP misses are occurring, and LIP is the best performing algorithm. Figure 2.2a shows the set dueling and algorithm selection architecture. In the figure sets 0 and 5 are dueling sets for LIP while 3 and 6 are dueling sets for BIP. All other sets are follower sets, meaning that they utilize the algorithm indicated by the selection logic.

Another solution is to utilize two Auxilliary Tag Directoriess (ATDs), as shown in Figure 2.2b. An ATD is equal to the cache's tag directory; it keeps track of blocks present but does not store any data. ATDs are, for this reason, cheaper than a full cache, but still requires more storage than duel-sets that use the existing cache. As the figure shows, the two ATDs run one algorithm each and all operations on the main cache execute in parallel on the ATDs. The same counter architecture

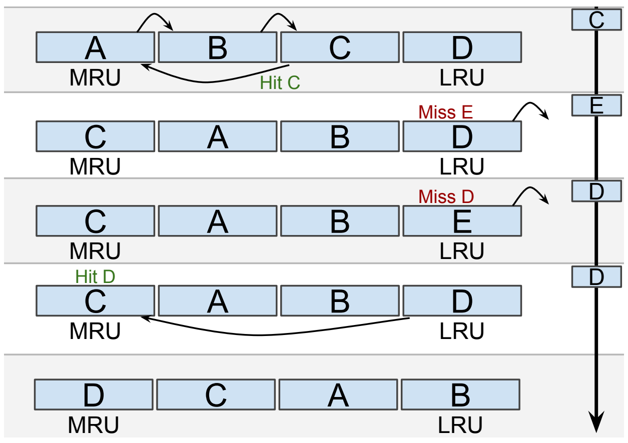


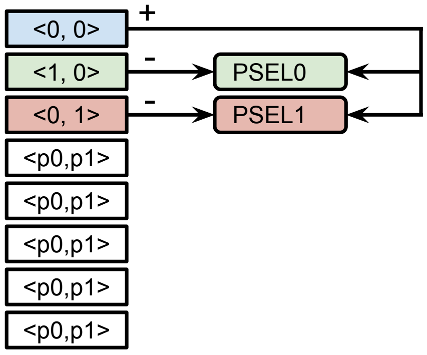
Figure 2.3: DIP managed 4-way cache set. (Assuming BIP insertion)

controlled by misses in either ATD is used to select the best performing algorithm for the main cache. The main advantage of using an ATD is that all available information is used when selecting between BIP and LIP. Also, the entire cache will always use the best algorithm while in set-dueling a fraction of the sets will always run the worst performing algorithm. The difference between using an ATD and cache dueling sets in terms of misses were shown to be small in the original paper. On their benchmarks, they measured an average decrease in misses by 22.3% using ATDs, compared to a 21.3% decrease when using 32 duel-sets [26], on a 4096 set cache.

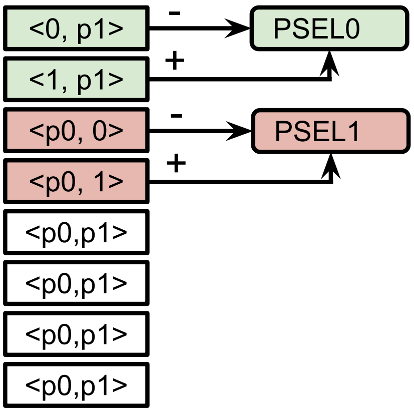
Figure 2.3 shows an example cache set managed by DIP. In the example, we assume BIP insertion with no insertions at the MRU position. In the initial state, there are four blocks; A, B, C, and D. A is at the MRU position and D is at the LRU position. The first request is for C; this is a hit, and C is promoted to the MRU position, A and B are pushed towards the LRU position. Then follows a request for E, which is a miss. DIP evicts D at the LRU position, and E is inserted in its place. Then follows two requests to D; the first request is a miss causing E to be evicted and D to be inserted at the LRU position. The second request is a hit and promotes D to MRU, pushing all other blocks one step towards the LRU position.

2.1.3 TADIP

TADIP [16] proposed in 2008 is a thread-aware extension of DIP [26]. The main issue with DIP that TADIP counters, is that DIP does not consider which core initiates a cache access. In a workload with multiple benchmarks, some might be recency-friendly while others are not. In a shared cache managed by DIP, the algorithm choice is made based on the sum of the cache accesses and then applied equally to all cores. The authors of TADIP recognized that improvements in performance could be achieved by selecting the DIP policy on a per-core basis



(a) Cache managed by TADIP-I (TADIP-I).



(b) Cache managed by TADIP-F (TADIP-F).

Figure 2.4: Alternate duel set organizations for Thread Aware Dynamic Insertion Policy (TADIP).

when utilized in a shared cache.

When selecting the best performing algorithm per core, the ATD technique requires two ATDs per core sharing the cache. This solution can quickly become too expensive to be practical. Set-dueling in DIP requires a minimum of two sets, one running LIP (1) and one running BIP (0). With two cores, the number of combinations rises to four (00, 01, 10, 11). When the number of cores increases this also seems to be an impractical solution. Based on this observation, the authors of TADIP suggested two new selection techniques based on set dueling, which reduce the number of duel sets required. Both solutions have one saturating counter per core sharing the cache. This counter is used to select the best performing policy for that core.

TADIP-I has one set per core running BIP for that core and LIP for all others. In addition to these N sets, a single set runs LIP for all cores. A miss in the LIP set will increment all the core counters while a miss in the core specific set will decrement the counter for the specific core. For a large N, this solution requires significantly fewer duel sets compared to having one per combination ($N + 1 << N^2$). This solution assumes that all other cores run LIP, and, therefore, cannot fully capture the effect of interactions between cores. Figure 2.4a shows an example of a cache managed by TADIP-I. In the figure, PSEL0 is the saturating counter used to select the best performing policy for core 0. Within a set; $< 0, 0 >$ indicates that both cores run LIP while $< 1, 0 >$ indicates that core 0 runs BIP while core 1 runs LIP. The variables p0 and p1 represent the current best performing policy for core 0 and 1 respectively.

TADIP-F attempts to reduce the error caused by the assumption of other cores

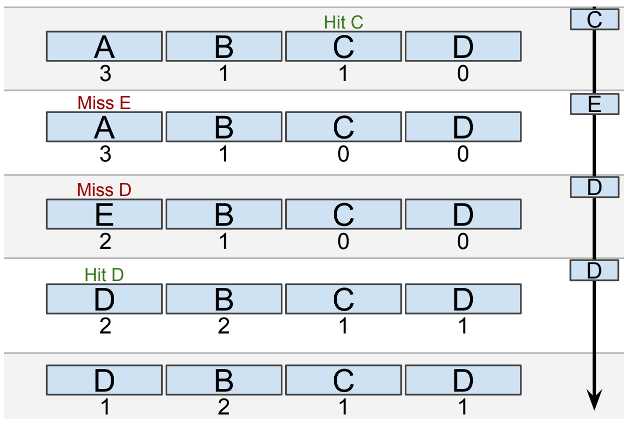


Figure 2.5: DRRIP managed 4-way cache set. (M=2, static insertion)

by having two sets per core, a total of 2N. A cache managed by TADIP-F is illustrated in Figure 2.4b. For each of the cores, one duel set use LIP and the other use BIP. Any insertions from other cores into the duel sets use the current best performing algorithm for that core. Like in the other policies, a miss in the LIP set for a core will increment that core's counter and a miss the BIP set will decrement the counter. For the remainder of this thesis when we refer to TADIP we assume TADIP-F unless otherwise stated.

When implementing TADIP, some mechanism is required to select which sets are duel sets, and which are follower sets. The authors of TADIP provide a simple hash function that can be used to select dueling sets, shown in Algorithm 1. This algorithm assumes a 4096 set cache. In the algorithm, set index is a number from 0-4095, core_id is the zero-indexed id of the requester core and cores is the total number of cores sharing the cache. If BIP or LIP is true, then the set is a duel set for the given core, and the policy forced to either BIP or LIP. If both BIP and LIP are false, then the set is a normal follower set and utilizes the current best performing algorithm for the given core. It follows from the algorithms that the original authors use a total of 32 duel sets spread evenly throughout the cache.

Algorithm 1 TADIP duel set selection.

```
1: $LIP \leftarrow set\_no[11:7] + core\_id == set\_no[6:0]$
2: $BIP \leftarrow set\_no[11:7] + core\_id + cores == set\_no[6:0]$
3: $FOLLOWER \leftarrow !LIP + !BIP$
```

2.1.4 DRRIP

Dynamic Re-Reference Interval Prediction (DRRIP) [17] was proposed in 2010. DRRIP does not utilize the concept of an LRU stack as done by LRU and TADIP. In DRRIP, each cache block has a number associated with it, called re-reference interval. The re-reference interval is a relative measure of when the algorithm expects a block to be re-referenced. Given two blocks with different re-reference intervals, then the block with a lower interval is expected to be re-referenced before the other block. A value between 0 and $2^M - 1$ is used to represent the re-reference interval. M is a configurable variable usually in the interval $[2, 5]$ [17]. The value of 0 indicates a *near* re-reference interval, the algorithm expects the block to be re-referenced in the near future. The value $2^M - 1$ indicates a *distant* re-reference interval while the value of $2^M - 2$ indicates a *long* re-reference interval. Multiple blocks may have the same re-reference interval. Hence, blocks are not strictly ordered as in the LRU stack. By setting $M = 1$, DRRIP degrades into the Not Recently Used (NRU) [20] algorithm, which among others is used on the UltraSPARC T2.

The replacement policy of DRRIP is to scan all blocks and evict the first one found with a distant re-reference interval. If no blocks have a distant re-reference interval the re-reference interval of all blocks is incremented by one and the scan restarts. This process repeats until the algorithm finds a victim block. If multiple blocks are potential victims, the algorithm uses the scan order as a tie-breaker. In the original paper, the authors specify that the leftmost potential block, the one with a lower block index, is the victim in the case of a tie.

DRRIP's promotion policy is to decrement the re-reference interval of the accessed block. By doing this DRRIP utilize access history rather than access time when calculating the re-reference interval. Hence, to reach a near re-reference interval a block has to be accessed multiple times. This promotion policy is different compared to LRU and TADIP, where a block will move to the MRU position following a hit, independent of the previous access history.

The insertion policy of DRRIP, like DIP and TADIP, is composed of two different policies and a selection mechanism. Static RRIP (SRRIP) will always insert new blocks with a long re-reference interval. Depending on the state of the cache, there might be existing blocks with a higher re-reference interval than the blocks inserted by SRRIP. This gives the newly inserted blocks a chance to see a re-reference before being replaced. Binominal RRIP (BRRIP) is analog to BIP in DIP. BRRIP with either insert new blocks with a distant re-reference interval or, with a small probability, insert like SRRIP with a long re-reference interval. Like BIP, BRRIP will allow trashing access patterns to keep some of the working set in the cache and hence improve performance over SRRIP. Selecting between the two insertion policies can be done using set dueling or ATDs, similar to what was described for TADIP. The authors use set-dueling in their original paper, and we opt to do this in our implementation as well.

Figure 2.5 shows an example cache managed by DRRIP. In the example, M is set to 2, making the distant re-reference interval 3 and the long re-reference interval 2. Also, we assume static insertion throughout the example. Initially, there are four blocks A, B, C and D with re-reference intervals 3, 1, 1 and 0. First an access hits the C block, and its value decrements to 0. Next a miss to block E occurs, block A has a re-reference interval of 3 and is evicted, E is assigned a re-reference interval of 2. Then a miss to block D occurs, as no blocks have a re-reference interval of 3

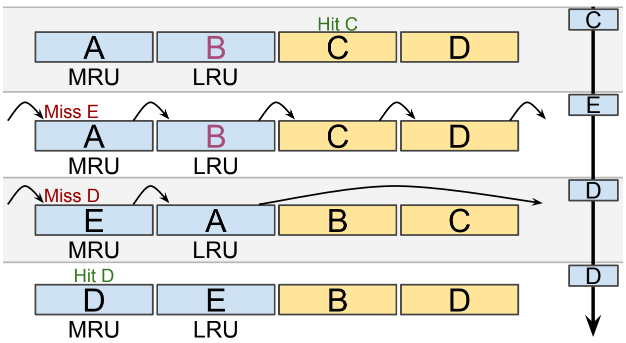


Figure 2.6: NUCache managed 4-way cache set. (M=2)

all values are incremented by one. After one incrementation E now has an interval of 3 and is evicted. Then follows a hit to block D, causing its re-reference value to decrease by one. The last row contains the final state of all blocks.

2.1.5 NUCache

Next Use Cache (NUCache) [19] was first proposed in 2011. NUCache does not partition the cache by examining each core's access pattern separately like many of the other algorithms. Instead, NUCache uses the concept of delinquent PCs. A delinquent PC is the PC value of a memory instruction that often causes cache misses. By evaluating the properties of the delinquent PCs, NUCache selects a set of PCs and allocates more cache space to blocks loaded by these instructions. Because all applications running may contain one or more delinquent PCs, NUCache will implicitly share the cache between the applications.

To detect delinquent PCs, NUCache uses a novel DeliTrack structure. The DeliTrack is a storage structure, indexed by PC that stores a miss count, insertion time and a next use histogram. LRU is used to manage the DeliTrack, which naturally ensures that PCs causing many misses are kept while others are replaced. The next use histogram counts the next use value of blocks loaded into the cache by the given PC in buckets of 8 from 0 to 64. This histogram is later used when a set of prioritized delinquent PCs are selected.

The next use distance of a block is defined as the number of misses observed by the cache between the time the block was evicted and the next time it is loaded due to a cache miss. This number is then scaled by the number of sets in the cache to get the set relative next use distance. An additional storage structure, NUTrack, is used to generate the next use histogram in the DeliTrack. NUTrack is a set-associative structure indexed by block address. Each row in the NUTrack stores a evicted bit, eviction time, and PC. When a new block belonging to a PC in the DeliTrack is inserted into the cache, an attempt is made to insert a new row in the NUTrack. A row is inserted iff there is a valid replacement target in the NUTrack. Two valid replacements exists; an unused row, or a row with the eviction bit set to true and an eviction time older than the maximum tracked next use value (64). When a block is evicted from the main cache, the eviction bit and eviction time is set in the corresponding NUTrack row, if it exists. On insertion in the main cache, the NUTrack searches for a matching row. If a matching row exists the next use distance is calculated and if the value is lower than the max value (64) the corresponding row in the DeliTrack histogram is incremented.

NUCache divides the ways in each cache set into two groups, MainWays, and DeliWays. The MainWays are managed by LRU while the DeliWays are simply first in first out. The value M defines the number of DeliWays. NUCache attempts to reduce misses by not evicting blocks from selected delinquent PCs when they are evicted from the MainWays, but rather let them enter the DeliWays. By using the size of the DeliWays and the next use information in the DeliTrack structure, the algorithm periodically selects a set of PCs that are allowed to use the DeliWays. The selection is done using a greedy algorithm that attempts to ensure that each block entering the DeliWays will receive a hit at least once before they are pushed out by other blocks. DeliWays and MainWays are implemented by having two extra bits per cache block, one indicating if the block can enter the DeliWays, another indicating if the block is the DeliWays. On insertion, all blocks inserted into to the MainWays. When the LRU block in the MainWays is about to be replaced, the algorithm checks if it is marked to enter the DeliWays. If the block is allowed to enter the DeliWays, it will not be evicted but rather moved from the MainWays. If, after moving the new block into the DeliWays, the number of DeliWays blocks has exceeded M the oldest block is removed. Otherwise, the new LRU block in the MainWays is evaluated. Because of this implementation the MainWays may use the entire cache if no DeliWays are in use, at the same time the DeliWays cannot exceed M. This allows for an efficient use of every cache set.

Figure 2.6 shows an example cache set managed by NUCache with M set to 2. Initially, there are four blocks, A, B, C, and D. A and B are in the MainWays, indicated by the blue background. While, C and D are in the DeliWays, as indicated by the yellow background. Block B is eligible for insertion into the DeliWays, no other blocks in the example are eligible for the DeliWays. The first request is for block C; this is a hit. C is not promoted as it is a part of the FIFO managed DeliWays. Next is a request for E, and this is a miss. The cache first attempts to evict B, but B is eligible for DeliWays and is not evicted. After B enters the DeliWays, it contains a total of 3 blocks, this is one more than the upper limit of 2, and hence the first in, D, is evicted. E is then inserted at the MRU position. Next is a request for D; this is also a miss, and A at the LRU position is evicted. The final request is also for D, causing no change as D is already at the MRU position.

2.2 Cache Partitioning Algorithms

This section covers cache partitioning algorithms. In contrast to the replacement algorithms, these algorithms explicitly assign a set number of blocks in each cache set to each core.

2.2.1 UCP

Utility Cache Partition (UCP) [25] was first presented in 2006. UCP uses the concept of utility when assigning ways to a core. Using a Utility Monitor (UMON), UCP divides the ways in the cache between the cores. UCP then uses the same insertion and promotion policy as LRU. The replacement policy is as in LRU but with two modifications: First if the number of blocks owned by the requesting core is less than the number of ways assigned to it, then the least recently used block that is not assigned to the requester core is replaced. If however the number of blocks owned is greater than or equal to the number of assigned ways the replacement algorithm selects the least recently used block of those owned by the requester. This replacement policy ensures that the division between cores in each set move toward the global allocation following cache misses. At the same time, a core may use more blocks that it is currently assigned, given that the space is not claimed by any other core.

The UMON is the core of the UCP algorithm. It consists of one ATD per core sharing the cache. The ATD is managed by normal LRU replacement and has one access counter per way. Whenever a cache request hits in the ATD, the access counter representing the way the block was in is incremented. In other words, UMON uses the stack property of LRU, as explained in Section 2.1.1, to find the hit rate of all valid partition sizes. In addition to the ATDs, there is a monitor circuit that uses the access counters to calculate a new global partition at set intervals. In the original paper, the authors recalculate the partitioning every 5M cycles.

The original paper proposes several algorithms for determining optimal partitioning based on the counter data. One of them is the Lookahead Algorithm suitable when there are more than two cores sharing a cache. The Lookahead Algorithm assigns ways based on an increase in marginal utility; it is given in Algorithm 2. While there are more ways to distribute, the algorithm calculates the maximum marginal utility achievable by each core. The core with the highest value wins and is assigned as many ways as needed to achieve the increase. The algorithm continue until all ways have been assigned. Lines 27-28 calculate the marginal utility. First the number of misses prevented by increasing the allocation from a to b is found. Due to the stack property of LRU, this is simply done by summing the access counters for ways a to $b-1$. The number of misses is then divided by the number of sets introduced, to find the marginal utility. The rest of the algorithm is simply a greedy algorithm selecting the highest marginal utility at each iteration. After a reallocation of cache ways, the ATD counters are all halved. By doing this, the UMON will keep historical data for future decisions while prioritizing data from the current period.

Because the lookahead algorithm is greedy, finding a case where it makes a non-optimal choice is rather easy. Consider a case with two cores, and two ways left to assign. By assigning one way to core 0 it will receive 10 more hits, 2 ways offers no improvement. Assigning one way to core 1 causes no additional hits, but if given two ways it will receive 18 hits. In this case, the marginal utility is 10 for core 0 and 9 for core 1. The algorithm assigns one way to core 0, and in the next

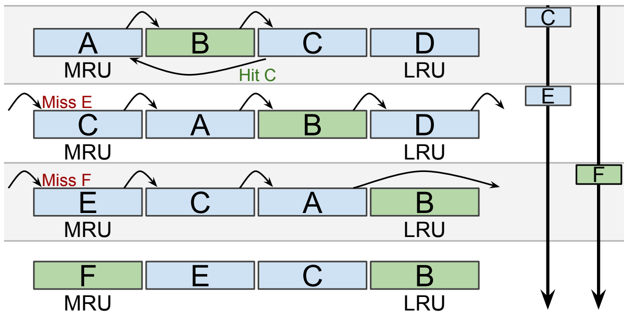


Figure 2.7: UCP managed 4-way cache set. (Two cores each allocated two blocks)

iteration both have zero utility and it does not matter which is assigned the last way. In this case, the algorithm saved 10 misses while it could have saved 18. In order to guarantee optimal decisions the algorithm would have to do an exhaustive search of the solution space, but this is infeasible as it requires a lot more resources than the simplified greedy algorithm.

Figure 2.7 show an example cache set managed by UCP shared by two cores. We assume that both cores are allocated two blocks. Initially, core 0 has three blocks in the cache as indicated by the blue background color; A, C, and D. The first request is by core 0 for block C; this is a hit causing the block to be promoted to MRU, pushing A and B towards the LRU position. Next is a request for E by core 0; this is a miss. Because core 0 already has more than the number of allocated blocks in the cache, the block closest to LRU owned by core 0 is evicted, D in this case. Finally, a request for F is made by core 1; this is also an miss. Because core 1 has less than the number of allocated blocks in the cache, a block not owned by core 1 is to be evicted. As a result, B owned by core 1 at the LRU position is saved, and rather A at the next to LRU position is evicted.

2.2.2 PIPP

Promotion/Insertion Pseduo-Partitioning (PIPP) [34] proposed in 2009 is an algorithm based on a slightly modified UMON circuit and a novel insertion and promotion policy. The UMON changes are to enable stream detection. Where the UCP algorithm only handles streaming applications indirectly, by assigning few ways because of a low hit rate in the ATDs, PIPP's UMON actively detects streaming applications. Stream detection is implemented by adding a counter that counts the total number of cache misses in the ATD. An application is then deemed to be streaming if either the number of misses or the miss rate in a single allocation period is above a threshold.

PIPP like UCP views the cache set as an LRU stack. The replacement policy is as in LRU, but the insertion and promotion policy is novel. The insertion policy inserts new blocks π_n blocks from the LRU position. Here π_n is the number of ways assigned to the n^{th} core. In a 4-way cache dual-core setup where both cores are

Algorithm 2 UMON Lookahead Algorithm.

```
1: $balance \leftarrow N$ /* Number of ways */
2: $allocations[i] \leftarrow 0$ /* for each core $i$ */
3: while $balance$ do
4:     for all $cores\ i$ do
5:         $alloc \leftarrow allocatations[i]$
6:         $max\_mu[i] \leftarrow$ GET_MAX_MU$(i, alloc, balance)$
7:         $blocks\_req[i] \leftarrow$ min blocks to get max_mu[i] for i
8:     end for
9:     $winner \leftarrow$ application with the maximum value of max_mu
10:    $allocations[winner] += blocks\_req[winner]$
11:    $balance -= blocks\_req[winner]$
12: end while
13: return alloactions
14:
15: function GET_MAX_MU$(i, alloc, balance)$
16:     $max\_mu \leftarrow 0$
17:     for $ii = 1; ii <= balance; ii++$ do
18:         $mu \leftarrow$ GET_MU_VALUE$(p, alloc, alloc + ii)$
19:         if $mu \geq max\_mu$ then
20:             $max\_mu \leftarrow mu$
21:         end if
22:     end for
23:     return $max\_mu$
24: end function
25:
26: function GET_MU_VALUE$(p, a, b)$
27:     $U \leftarrow$ change in misses for application p when number of blocks assigned to
        it increases from a to b
28:     return $\frac{U}{b-a}$
29: end function
```

assigned two ways, PIPP will insert all new blocks from either core in the second to last position in the stack. In this situation, the two top positions in the cache stack can only be reached by a cache block through promotion. On a cache access, a block has a chance, $p_{prom} = \frac{3}{4}$, to move one position upwards in the stack unless it is already at the MRU position.

On insertion, the PIPP policy does not consider how many blocks are owned by the requesting core, this is unlike UPC's insertion policy that prevents a core from claiming more ways that what it is assigned. However, cores with more ways assigned to it will insert its blocks higher up in the stack. The core with the highest number of ways assigned will not have any insertion competition pushing its blocks out of the cache. The only way blocks from this core can be pushed out is by other blocks from the same core, or by blocks from other cores that are

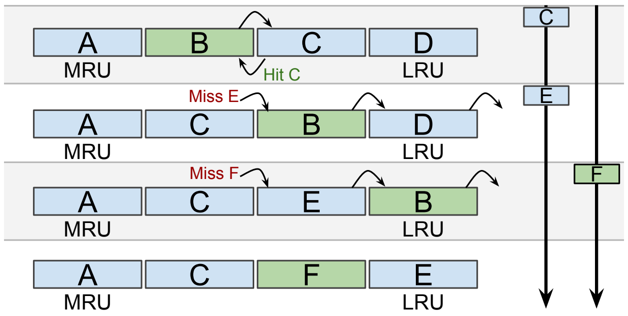


Figure 2.8: PIPP managed 4-way cache set. (Two cores each allocated two blocks)

re-referenced repeatably. Two cores with the same allocations will both have an equal chance of keeping their blocks in the cache, as they both insert at the same position. Statistically a core with a lower allocation, inserting at a lower position in the stack, should also on average own fewer blocks in the cache compared to a core with a higher allocation. This way PIPP obtains what the original authors call pseudo partitioning, where overall a higher allocation will statistically result in more cache space. However, the access frequency of cores can cause a core with a low allocation to own most of or all blocks in the cache if the other cores have a much lower access frequency.

When the UMON detects a core that is streaming PIPP will no longer insert blocks from this core at the position given by the allocation. A special insertion position, π_{stream}, is used for all streaming cores. π_{stream} is set to the number of cores currently streaming. By inserting at this fixed position, PIPP attempts to limit the interference the streaming core has on the non-streaming cores. Blocks from streaming applications have a reduced chance of promotion after an access, $p_{stream} = \frac{1}{128}$. In the case where all cores are streaming, and there are no cores to protect, PIPP inserts all blocks at the LRU position.

Figure 2.8 shows an example cache set managed by PIPP shared by two cores. Both cores are allocated two blocks, and initially core 0 owns three blocks in the set; A, C, and D. In this example we assume all hits cause a block promotion. The first request is for C by core 0; this is a hit, and the block is promoted, effectively swapping block B and C. Next is a request for E by core 0; this is a miss. E is inserted at the second to last position of the cache set, as the core is allocated two blocks, causing D to be evicted and B to move to the LRU position. Finally, a request for F is made by core 1; again it is inserted at the second to last position, evicting B. Note that after the initial request for C, the state of the two upper blocks do not change.

2.2.3 PriSM

Probabilistic Shared Cache Management (PriSM) [18] was first presented in 2012. PriSM is a framework for cache management with optimization algorithms targetting multiple performance goals. The original paper presents hit maximization, fairness and QoS goals. We will focus on the hit maximization algorithm , or miss minimization algorithm, as all other algorithms in this thesis also targets this goal. PriSM utilizes ATDs to estimate private cache performance for each of the cores. The ATD will keep track of total misses and hits. It will not track hits per cache way like the ATDs in UCP and PIPP. In addition to the ATDs, the algorithm requires three counters per core tracking hits, misses and number of blocks owned by the core in the actual cache. PriSM utilizes the same insertion and promotion policies as LRU, but the replacement policy is optimized based on the ATD and the optimization target.

The replacement algorithm of PriSM utilizes eviction probabilities, E_i ($\sum E_i = 1$), assigned to each core when selecting a victim block. On replacement, a victim core is first selected by a random draw using the eviction probabilities. The LRU block owned by the victim core within the cache set is the eviction target. In the case where the selected target does not own a block in the set, all blocks owned by cores with $E_i > 0$ are considered, and the LRU of these is the eviction target. At set intervals, an optimization algorithm determines the eviction probability, E_i, for each core. The original paper recalculated E_i values at every 10000 cache miss.

The insertion and promotion policy of PriSM is equal to LRU. On insertion, a block is promoted to the MRU position, and on any subsequent accesses the block is again promoted to MRU unless it already has that position.

Selecting an eviction probability E_i for each core is done by considering how the eviction probability will effect a core's usage of the cache. Consider an interval of W misses where each core contributes a fraction of the misses, M_i. At the start of the interval the blocks owned by $core_i$ equals a fraction C_i of the total number of blocks in the cache. If we do not evict any blocks owned by $core_i$ during the interval, then at the end of the interval the core owns a fraction T_i of the cache. T_i is known as the target allocation, and is expressed by $T_i = C_i + M_i * W/N$. Here $M_i * W$ is the number of misses caused by $core_i$ during the interval, which also is the number of blocks inserted by the core. N is the total number of blocks in the cache, and the fraction $M_i * W/N$ equals the fraction of the cache claimed by $core_i$ during the interval. If the core has a non-zero eviction probability, then this formula extends into $T_i = C_i + (M_i - E_i) * W/N$. As noted, PriSM defines three optimization targets, each one of these is responsible for calculating the optimal T_i that will fulfill the optimization target. Rearranging the above formula for E_i yields: $E_i = (C_i - T_i) * N/W + M_i$. Algorithm 3 shows how T_i values are calculated for hit maximization. It is a relatively simple algorithm that will adjust the target occupancy based on the current occupancy and the potential for gaining more hits.

While we have presented PriSM based on LRU replacement, as done in the original paper, it should be noted that PriSM is not dependent on this underlying replacement algorithm. Any algorithm is usable, as long as it is augmented to prioritize the selected victim during replacement. The algorithm run on the ATDs has to be the same as the underlying algorithm in the PriSM implementation.

Algorithm 3 PriSM Hit Maximization.

```
1: $N$ /* Number of cores */
2: for all $cores\ i$ do
3:     $PotentialGain[i] \leftarrow StandAloneHits[i] - SharedHits[i]$
4: end for
5: $TotalGain \leftarrow \sum PotentialGain$
6: for all $cores\ i$ do
7:     $T_i \leftarrow C_i * (1 + \frac{PotentialGain[i]}{TotalGain})$
8: end for
9: $T_i = \frac{T_i}{\sum T}$ /* Normalize target occupancy */
```

2.2.4 CLU

Co-Optimizing Locality and Utility (CLU) [36] was first presented in 2014. The authors of CLU recognize that recent research in LLC partitioning has followed two distinct directions. Some publications optimize for access locality and attempt to improve performance by changing the lifetime of blocks in LRU managed caches. DIP, TADIP, and NUCache are three such solutions that use novel methods to reduce or extend the lifetime of blocks in an elsewise LRU managed cache. Other publications recognize the usefulness of utility and do way-partitioning between cores based on their utility values. Examples here are UCP and PIPP. Both UCP and PIPP are forced to use LRU as the underlying algorithm because they both depend on the stack property of LRU to do utility calculations [25, 34].

The authors of CLU present a novel approach for calculating the utility curve of a BIP managed cache. BIP, as covered earlier, is one of the two insertion policies under DIP and TADIP. BIP violates the stack property of LRU by mostly inserting new rows at the MRU position, or at a low probability in LRU position. To correctly measure the utility curve of a BIP managed k-way cache, one needs k ATDs; ATD(1), ATD(2), ... ATD(k). Where ATD(x) is an x-way ATD. In contrast, the utility curve of an LRU managed cache can be found using one ATD, due to the stack property. Having k ATDs per core sharing the LLC is not a realistic goal due to the required overhead. The authors of CLU propose a simplification where there are $m = log_2 k$ ATDs; ATD(1), ATD(2^1), ..., ATD(2^m). A linear increase between the sample points is assumed when calculating the final utility curve. It should be noted that the storage overhead of m ATDs in total is less than twice the overhead of the single ATD(k) required to sample the LRU curve.

CLU uses the two curves first to allocate ways to each core using the same algorithm as shown for UCP in Section 2.2.1. The only difference is that the algorithm uses either the LRU or BIP value when estimating utility given an allocation, depending on which algorithm performs best. During runtime, CLU works like UCP. The only exception is that the core's ways are managed by either LRU or BIP, depending on which algorithm has the best utility value for the number of ways currently assigned to that core.

Figure 2.9 is an example LIP and BIP utility plot for a core sharing a 16-way

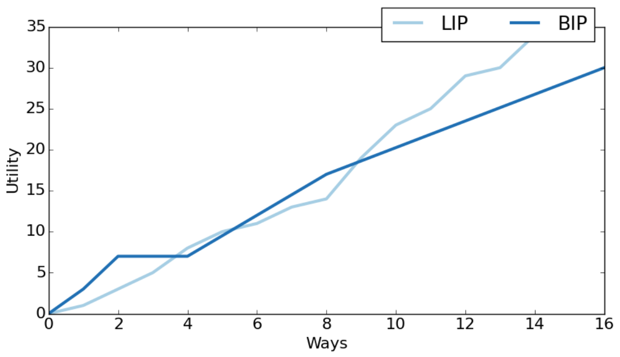


Figure 2.9: BIP and LIP utility plot in a 16-way cache.

cache. For each way value, the maximum achievable utility is the maximum of the LIP and BIP line. Hence, when the lookahead algorithm is used to assign ways to each core, the maximum of the LIP and BIP value is used. If the sample core were assigned 2 ways it would use BIP replacement, this follows from the fact that BIP has a higher utility at 2 ways. However, if the core were assigned 12 ways, it would use LIP replacement because LIP has a higher utility at that point.

Chapter 3

Framework

This chapter presents our selected simulator and shortly outlines the structure of the simulator's memory simulation. We present the selection of algorithms we have implemented and show how they relate to each other and the simulator. Finally, we provide details regarding any assumptions or changes that were necessary to implement working versions of the selected algorithms.

3.1 Simulator

There are several different simulators used in computer architecture research today [5, 2, 21, 24]. Of the simulators used today, we evaluated two possible candidates for this work, Sniper [5] and gem5 [2]. These two candidates are mainly chosen because they are both in active use by the CARD [4] research group at NTNU where this thesis is performed. Gem5 is also an obvious candidate because it is the simulator used in most computer architecture research today [6]. It is a cycle accurate simulator meaning that it can theoretically simulate real hardware perfectly. This accuracy comes at the cost of simulation time. The other candidate, Sniper, is based on interval simuation [8]. Interval simulation allows it to simulate benchmarks significantly faster than gem5 [5, 23] at the cost of reduced accuracy.

Sniper is multithreaded, and each simulated core runs in a separate simulation thread. This separation allows Sniper to take advantage of today's multicore processors to speed up the simulation. On the other hand, gem5 performs all simulation in a single thread. By having multiple simulation threads, there is a chance of clock skewing [5] during simulations in Sniper. Clock skewing is when one core simulates faster or slower than the others, making the clock values in each core different. When this happens, the simulator cannot correctly simulate inter-core interactions, such as access to the shared cache. There are however techniques implemented in Sniper that attempt to reduce errors caused by clock skewing.

Gem5 is an execution-driven simulator. It takes the benchmark binary as input and simulates it by correctly executing instructions like a real processor. Sniper, which is trace-driven, does not need the benchmark binary, but rather a trace of

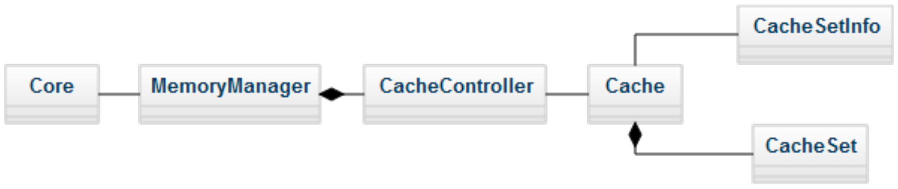


Figure 3.1: Partial core memory model in Sniper.

instructions from a previous run. The instruction trace is generated by running the benchmark ahead of time, and dumping committed instructions as the benchmark progresses. As a result, the instruction trace does not contain instructions that were executed along miss-predicted execution paths and later reverted. Because of this, Sniper cannot correctly estimate the cost of a branch miss prediction, as this may vary depending on the effect of the wrongly executed instructions. Gem5 will execute the wrong execution path just like a real processor, and hence correctly estimates this penalty. By using traces, Sniper does not have to simulate the effect of each instruction correctly, it only needs to estimate the time it will take to execute. This makes the simulator code base smaller, and it becomes easier to extend it with new algorithms.

Based the authors previous work [23], we chose Sniper over gem5 because the simplicity and speed of Sniper outweigh the reduced accuracy compared to gem5. Because we know that clock skewing in Sniper might be an issue for our work, we will perform an experiment investigating this error source further in Section 6.2.

3.2 Implementation

Figure 3.1 shows an overview of the classes involved in the Sniper memory simulation. The core class simulate the execution of instructions; this also includes simulating memory accesses to get and store instructions and data. Each core has a memory manager instance which again, in our case, has four cache controllers. A cache controller represents a single cache. For shared caches, the cache controller instance in all but core 0 is a proxy, any calls to the proxy cache controllers is directed to the main controller at core 0.

When a core issues a memory request to the memory manager, the manager will issue the request to caches in order until it finds the data. If none of the caches have the requested data, the request is handled by the main memory simulation, which is not in the scope of this simplified overview. Each cache controller knows its response latency and handles updating of the simulation time during each request. In other words, the classes from Core down to CacheController handle a mix of functional and performance simulation.

Each cache controller has a cache instance, which is a purely functional cache model. The cache class has methods for reading and writing to the cache, and each method returns whether the request was a hit or a miss. This return value is then used by the CacheController to update the performance simulation. Each

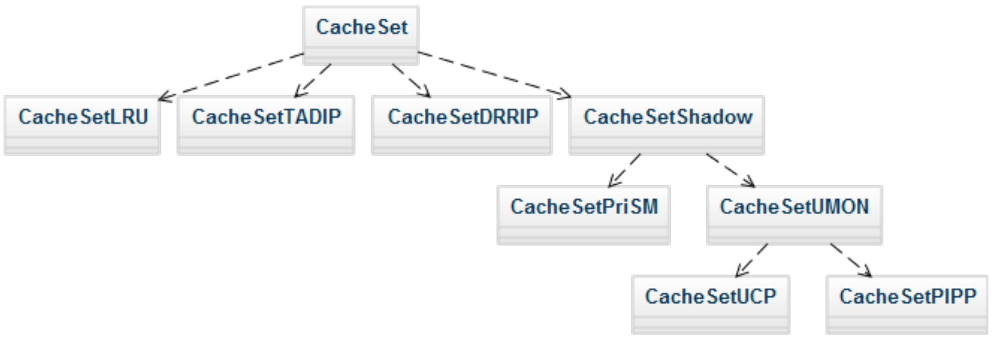


Figure 3.2: Implemented algorithms and their relations.

cache is built from several CacheSet instances, each representing a single set, and possibly a single CacheSetInfo instance. By modifying how the CacheSet operates, cache management algorithms can be implemented. The CacheSetInfo instance is available to all cache sets and enables implementation of schemes that share data between cache sets.

In the original Sniper implementation, only the block tag requested is made available to the CacheSet. While this is enough to implement simple schemes, such as LRU and TADIP, we need more information to implement cache partitioning schemes such as UCP. We have therefore modified Sniper by adding a data structure that is sent from the core when it initiates a memory request, all the way down to the cache set. This data structure allows us to pass arbitrary data from the core to the cache set, allowing for more complex schemes.

Of the schemes presented in Chapter 2, five have been implemented and tested in our simulation framework; LRU, TADIP-F, DRRIP-3, PriSM, UCP, and PIPP. Figure 3.2 shows how the five implemented algorithms relate to each other, and the base CacheSet class. LRU, which is included in Sniper, and the two set dueling schemes TADIP and DRRIP, are all implemented as direct subclasses of the CacheSet class. Based on the CacheSet class we have also implemented a cache with support for ATDs, also known as shadow tags. By default, the ATDs do full sampling of the cache, but we made this configurable. PriSM, which requires access statistics for each core, is implemented on top of the ATD implementation. Also, we create a UMON implementation on top of the ATDs; this adds the utility calculating and block assigning functionality needed by both UCP and PIPP. UCP and PIPP, are then implemented on top of the UMON base class.

Our implementation using inheritance reduces the number of code lines required when we implement multiple algorithms that share several properties. By reducing the number of code lines required, we improve implementation time, and we reduce the chance of bug causing issues in our simulations. Implementing additional algorithms in this framework requires only an understanding of the algorithm, as all of the simulator groundwork is already in place. This makes the framework strongly suitable for future research.

3.3 Algorithm Details

Some of the original papers explaining the algorithms makes unstated assumptions in their implementation. This section covers the assumptions we have had to make when implementing algorithms in our simulation framework.

In the DRRIP paper, the authors specify that they make use of set dueling to choose between SRRIP and BRRIP. The authors do not specify how they select which sets are dueling sets and which are follower sets. We choose to use the same algorithm for set classification as used by the authors of TADIP, shown in Section 2.1.3. Our implementation of DRRIP uses 3-bit counters per cache block, hence DRRIP-3.

As mentioned in Section 3.2 we have built a generic ATD, or shadow tag, implementation. We choose to use full sampling in all algorithms that make use of this implementation, PriSM, UCP, and PIPP. While previous work has shown that the estimation error when using dynamic sampling with a sufficient number of sets [16] is negligible, we still opted for full sampling in all our experiments. By doing full sampling, we eliminate the error source caused by imperfect data and allow the focus of our experiment to be on the decisions made by the algorithm instead.

The UMON implementation, as covered in Section 2.2.1, allows a core to be assigned zero ways. Sniper simulates inclusive caches, which requires a block to be stored in all cache levels when it is first loaded. As a result, we cannot allocate less than one block per core. To achieve this, we have modified the first two lines of the UMON allocation algorithm, as shown in Algorithm 2. Algorithm 4 shows the modification that ensures that each core is assigned at least one way. Several of the previous works also did this modification to support inclusive caches [25, 34].

Algorithm 4 Snip: Modified UMON Lookahead Algorithm.

```
1: $balance \leftarrow N - cores$ /* Number of ways minus number of cores */
2: $allocations[i] \leftarrow 1$ /* for each core $i$ */
```

In the original PIPP paper [34], there are two conditions that cause an application to be marked as streaming; miss rate and miss count. The original paper uses dynamic sampling in the shadow tags while we use full sampling. Because the miss count condition scaled badly with full sampling, it was removed in our experiments. Additionally, the original paper used a miss limit of 0.125 as the classification limit. However, using our workloads we observed that most benchmarks always were marked as streaming, causing poor performance. As a result, we increased the value to the next power of two, 0.25. This change resulted in most benchmarks switching between streaming and not streaming as expected.

In addition to PIPP, we have implemented a variation we named PIPP-min8. This variation works exactly like the normal PIPP algorithm, except that the insertion policy always adds eight to every position. The theory is that blocks in PIPP-min8 has a longer lifetime, and this may improve performance. Previous works have mentioned this modification [18] as well.

The original PriSM paper states that the eviction probabilities, E_i, for all cores should sum to 1 [18]. However, we discovered that this is not the case, at least for the miss minimization algorithm. Simply normalizing the eviction probabilities is not a viable solution. One or more cores may have an eviction probability of 1, indicating that they are always the eviction target, while other cores have $E > 0$. If we normalize, the core(s) with $E = 1$ are not guaranteed to be selected as the eviction target breaking this assumption. As a result, we use a compound victim selection algorithm;

1. If one or more cores have an eviction probability of 1 we choose a victim at random between these cores.

2. If no core has an eviction probability of one or the selected core has no blocks in the set; we select one of the cores with $0 < E < 1$.

3. If neither of the two selected cores has blocks in the cache set, the algorithm selects a random block owned by a core with $E > 0$ as in the original algorithm.

This modification allows us to prioritize cores with $E = 1$, which is not possible via normalization.

Chapter 4

Methodology

In this chapter, we introduce the processor model used in our experiments. We also introduce our benchmarks, and how we built workloads of various sizes based on these benchmarks. Finally, we introduce the various performance metrics used to evaluate the implemented cache partitioning algorithms.

4.1 Processor Model

Processor core	3GHz, OOO, 6 inst. dispatch width, 128 rob entries, 4 inst. commit width, 3 Int. ALU, 1 FP MUL/DIV, 1 FP ADD, 2 Int. SSE ALU, 1 Int. SSE MUL
Private L1 inst.	32kB, 64B block-size, 4-way, 8 MSHRs, LRU
Private L1 data	32kB, 64B block-size, 8-way, 8 MSHRs, LRU
Private L2 unified cache	128/256/512/1024kB, 64B block-size, 8-way, 12 MSHRs, LRU
Shared L3 cache	4/8/16/32MB, 64B block-size, 24 MSHRs, 32-way, varying replacement algorithm
Memory controller	6.4GB/s, 100ns access latency
Clock Skew	100 cycle barrier synchronization

Table 4.1: Model properties.

Throughout this thesis, we utilize a CMP model simulated on Sniper [5]. In our model, each processing core has two levels of private cache, the L1 data and code caches and a unified L2 cache. Additionally there is a third cache level, L3, which is shared by all cores. The private caches are managed by LRU, and the replacement policy of the third cache level varies throughout our experiments. Figure 4.1 shows an overview of the simulated architecture and Table 4.1 contains an overview of the system properties.

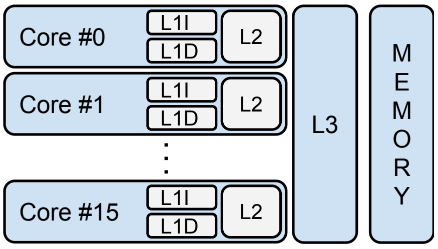


Figure 4.1: Processor model architecture.

As the sniper simulation system implements a Nehalem core model, all our processor core properties are based on Intel's Nehalem architecture [31]. The first level cache size is also selected based on the Nehalem architecture. We do note that in more recent architectures such as Intel's Haswell [15] the core properties have changed compared to the older Nehalem, but the size of the first level cache remains the same. The reason being that with increasing cache size the access latency also increases. Increased latency can cause problems if the first level cache is unable to feed the processor pipeline with a sufficient stream of instructions and data.

For our second cache layer, we have selected four different size configurations. We will use these configurations to evaluate how private cache size affects shared cache behavior. Our third level cache has three configurations. During our experiment, we will choose a configuration based on the number of cores used. For 4-, 8- and 16-core simulations we will respectively use a 4MB, 8MB, and 16MB L3 configuration.

Sniper's memory model is very limited compared to models in cycle-accurate simulators, such as gem5. In our processor model each memory request completes in a constant 100ns, there is no simulation of row and column access delays. Because inter-core interactions may be out-of-order, the memory bus model is a statistical model that attempts to estimate queuing delays based on a history window. Both of these limitations introduce an error source in our simulations, but previous work [5, 23] has shown that results are still comparable to cycle-accurate simulations. To reduce clock skewing caused by having multiple simulation threads, we utilize a 100 cycle barrier synchronization. Barrier synchronization implies that all simulated cores must wait every 100 cycles for all other cores to catch up. In Section 6.2 we cover an experiment where we measure the effect of various synchronization barriers on our results.

4.1.1 Cache Models

In the following sections, we describe how CACTI [28] was used to estimate the access latency for each of our caches. When using CACTI we model each cache with parallel access to the tag directory and data. We also opted for high-performance storage cells over the power saving ones that CACTI also supports. Finally, we specify an optimization goal with a preference for lower access latencies. To get a model that is as accurate as possible we have estimated the access latency for each configuration in both the L2 and the L3 caches. As a result, we will correctly observe increasing access latencies with increasing cache sizes.

L1 Code and Data Caches

	Data	**Instruction**
Size	32kB	32kB
Block size	64B	64B
Associativity	8	4
Banks	1	1
Technology	32nm	32nm
Access time (Tag)	0.16ns	0.16ns
Access time (Data)	0.38ns	0.32ns
Access cycles (Tag)	1	1
Access cycles (Data)	2	1

Table 4.2: L1 cache properties.

When choosing a first level cache size, we must consider that first level caches are on the critical path of the processor core. Having a hit latency of more than 2-3 cycles in a first level cache will be a limiting factor in the overall processor design. We can observe that first level cache sizes have remained constant between Intel's Nehalem [31] and the newer Haswell [15] architectures. We have for this reason opted to simulate only a single size configuration at this cache level. Both the first level caches have a size of 32kB, divided into sets of 4 and 8 blocks for the instruction and data caches respectively. Both caches have 64-bytes long cache blocks. Table 4.2 summarizes these values as well as the best bank count and access latency for the tag directory and data as estimated by CACTI. We define the best bank count to be the one that provides lowest access latency for data and tag directory measured in cycles.

We convert the access latency to cycles assuming a period of 0.33ns, equal to a clock speed of 3GHz. For example, a tag access latency of 0.16ns equals one cycle while a data access latency of 0.38ns equals two cycles.

L2 Cache

For the unified second level cache we have modeled four different sizes; 128kB, 256kB, 512kB, and 1024kB. In all configurations, there are 8 blocks in each set and each block is 64-byte long. Using CACTI, we have found the best number of banks per size configuration and the corresponding access latency. Table 4.3 summarises these values. Again we have converted access times to cycles assuming a 0.33ns period. As expected we observe an increased access latency as the cache size increases.

Size	128kB	256kB	512kB	1024kB
Block size	64B	64B	64B	64B
Associativity	8	8	8	8
Banks	1	2	8	8
Technology	32nm	32nm	32nm	32nm
Access time (Tag)	0.28ns	0.29ns	0.26ns	0.32ns
Access time (Data)	0.57ns	0.66ns	0.88ns	0.95ns
Access cycles (Tag)	1	1	1	1
Access cycles (Data)	2	3	3	3

Table 4.3: L2 cache properties.

L3 Cache

Size	2MB	4MB	8MB	16MB
Block size	64B	64B	64B	64B
Associativity	32	32	32	32
Banks	2	4	4	4
Technology	32nm	32nm	32nm	32nm
Access time (Tag)	0.56ns	0.52ns	0.71ns	0.88ns
Access time (Data)	1.42ns	1.81ns	2.11ns	2.69ns
Access cycles (Tag)	2	2	3	3
Access cycles (Data)	5	6	7	9

Table 4.4: L3 cache properties.

Like the previous level we have three different size configurations for the L3 cache; 4MB, 8MB, and 16MB. Unlike the previous two cache levels, that all use a standard LRU replacement policy, we will vary the replacement policy of the third level. Many of the algorithms we are experimenting with in this work resemble some form of way-partitioning by assigning some ways (or cache blocks) per cache set to each core. For this reason, the third level cache has 32 cache blocks per cache set, in contrast to the 16 cache blocks per set in the original Nehalem architecture. Giving us an average of 8/4/2 sets per core during our 4/8/16 core experiments. The block size is set to be 64-bytes as in the previous levels. Again using CACTI we find the best bank count and the corresponding access latencies. Table 4.4 summarises the cache properties for the various cache sizes.

4.2 Benchmarks and Workloads

In this section, we will present the benchmarks used to evaluate cache partitioning algorithms in this thesis. We explain how we extracted simulation traces from benchmarks and how we classified those traces based on their sensitivity to changes in available cache space and memory bandwidth. Finally, we explain how we created 4-, 8-, and 16-core workloads based on those benchmark traces.

4.2.1 Benchmarks and Sample Extraction

In all our experiments, we are utilizing benchmarks from the SPEC CPU2006 [29] benchmark suite. We choose this suite because it is the newest of the CPU benchmark suites from SPEC, and it is specifically designed to test the performance of various computer architectures with benchmarks based on real user applications. Unless otherwise stated all benchmarks use the first reference input set. Because simulating an entire benchmark is a time-consuming process, we choose to extract a sample interval used to represent each benchmark. There are multiple ways of extracting sample intervals. The naive way is to specify an offset and a length and use this across all benchmarks. However, several more advanced methods of sample selections exists, two examples are SimPoint [9] and SimFlex/SMARTS [11, 33].

SimPoint analyzes the benchmark and divides it into basic blocks. It then divides the dynamic instruction stream of the running benchmark into intervals of a set size. Each interval is classified by which of the basics blocks in the benchmark the interval executes. Finally, the intervals are clustered using k-means, and the algorithm selects the intervals closest to the centroid in each of the k-clusters to represent the benchmark. The value of k is configurable. SimPoint is run ahead of time and after it has been run an external simulator is used to simulate each of the SimPoint selected intervals.

SimFlex also divides the dynamic instruction stream into intervals, but unlike SimPoint this is done while the simulator is simulating the benchmark. Some intervals are simulated in detail while others are simulated in a low detail fast forward mode. SimFlex uses statistics to calculate the variation of simulation results based on the intervals simulated in detail. The simulation ends after a minimum number of intervals has been simulated, and the result variation between the intervals is within a set limit.

Both solutions utilize the same idea of simulating only parts of the dynamic instruction stream. The main difference is the selection criteria and the separation between ahead of time processing and integration with the simulator. J. Yi et al. [35] have shown that there is little variation in accuracy between SimPoint and SimFlex selected intervals. SimPoint also integrates well with our existing simulation framework. As a result, we chose to utilize SimPoint in our work.

Ideally a few sample intervals are needed per benchmark to get accurate simulation results. To simplify the simulation, we chose to extract one larger interval per benchmark, in place of multiple smaller ones. By doing this, we are willingly increasing the error [10] between simulating our interval and the results obtained by simulating the entire benchmark. In our experiments, we are interested in observing performance change in our simulated intervals due to architectural changes. Producing results comparable to the results of the full benchmark run is not required to achieve this. Also, it is not obvious how to correctly combine performance metrics used in this thesis from multiple simulated intervals. Therefore, we choose to use only one interval per benchmark.

The length chosen for our intervals may also affect the final results. As caches are empty when the simulation starts the cold cache effect, caused by compulsory misses at simulation start, may skew our results if the simulated interval is short. Additionally, as we are experimenting with cache partitioning algorithms, we believe that a certain number of instructions are needed for our results to stabilize. Finally, by increasing the number of instructions we are also increasing the time required to simulate a benchmark. We choose to extract 250M instruction intervals using SimPoint. This number of instructions will make the cold cache effect negligible [9, 10, 23] while we keep the simulation time relatively low. We generate an instruction trace using Sniper for each SimPoint extracted interval. All later experiments utilize these traces in place of the actual benchmark executable.

4.2.2 Benchmark Classification

Cache	**Bandwidth**	**Cache & Bandwidth**	**Compute**
astar	bwaves	bzip2	GemsFDTD
gobmk	cactusADM	mcf	calculix
h264ref	gcc	omnetpp	dealII
hmmer	lbm	soplex	gamess
perlbench	libquantum	sphinx3	gromacs
	milc		leslie3d
	wrf		namd
	xalancbmk		povray
	zeusmp		sjeng
			tonto

Table 4.5: Benchmark Classifications

To better understand our simulation results, we perform a benchmark classification experiment on each of the previously generated traces. This experiment is intended to detect various properties in each trace that may affect how they behave on our simulated architecture with various cache partitioning algorithms. We choose to categorize traces based on their sensitivity to the size of the LLC and the bandwidth of the bus connecting the LLC and the DRAM.

The system model used in this experiment is as shown in Table 4.1 with the smallest of the L2 configurations, 128KB. The size of the L3 cache and the speed of the memory bus is varied as shown in Table 4.6. By reducing the size of the L3 cache and the speed of the memory bus, we intend to simulate a situation where the benchmark has reduced access to resources due to contention.

L3 Cache size	256kB, 512kB, 1024kB, 2048kB, 4096kB
Memory Bus Speed	1.6GB/s, 3.2GB/s, 6.4GB/s, 12.8GB/s

Table 4.6: Model properties.

We simulate each benchmark for each combination of LLC size and memory bus bandwidth, in total 20 simulations per benchmark. Then we evaluate how changes to the architecture affected the benchmarks performance using the reported IPC. The evaluation is done by organizing the IPC measurements in a 2d data table with LLC size on one axis and bandwidth on the other. Between each data pair along each axis, we calculate the performance reduction. Using the arithmetic average of reductions along each axis, we classify each benchmark as either sensitive or not sensitive to changes in that axis property. The standard deviation is used to increase the required limits on the average reduction if a benchmark shows high variability along an axis. We observed this in cases where there is little change in performance except for one point where performance is significantly worse, dragging the average value down. We define four groups for our benchmarks as described below. For each group, we define rules used to detect benchmarks in that group.

- **Cache sensitive** (ca) benchmarks are in general benchmarks with memory access patterns that are recency-friendly. We required an average performance reduction along the cache axis of at least 4% and a standard deviation of less than 11% to classify a benchmark as cache sensitive. If the deviation is higher than 11%, a performance reduction of at least 13% is required to be classified as cache sensitive.

- **Bandwidth sensitive** (bw) are benchmarks with no to little temporal locality, often streaming access patterns. We required an average performance reduction along the bandwidth axis of at least 8% and a standard deviation of less than 11% to classify a benchmark as bandwidth sensitive. If the deviation is higher than 11%, an average performance reduction of at least 20% is required to be classified as bandwidth sensitive.

- **Cache- and Bandwidth sensitive** (cabw) are benchmarks with trashing memory access patterns that will benefit from more cache (less trashing) and more bandwidth (faster loading of previously trashed data). To be classified as cabw both the requirements of the ca and the bw groups as described above must be satisfied.

- **Compute sensitive** (co) are benchmarks limited by the processing power of the simulated processor. These benchmarks do not satisfy the requirements of the ca nor the bw groups as described above.

We set the classification limits based on manual observation of benchmark behavior. We noted that benchmarks that are bandwidth dependent, in general, had a higher performance loss when we halved the available bandwidth compared to cache sensitive when we halved the cache size. We also noted that one sample point dominates the average performance drop for some benchmarks, in these cases the standard deviation also rose. To prevent false positives, we applied higher cutoff limits when the standard deviation was high compared to the general case. Table 4.5 lists all benchmarks and their classification according to the above rules.

4.2.3 Workloads

Based on the classified benchmarks we generated 4, 8 and 16 core workloads. The 4 core workloads come in five classes. One class per benchmark group, these contain workloads only from that particular group. Also, one class with benchmarks picked from all groups. The ca and cabw groups contain 5 workloads while the bw and co groups contain 10 workloads. Table A.3 contains an overview of the 50 4 core workloads and their short names used throughout the report. There is only a single class of 8 and 16 core workloads, as there are not enough benchmarks per group to make workloads of this size. The ten 8 and 16 core workloads contain benchmarks from across all benchmark groups. Table A.1 and A.2 contains an overview of the 8 and 16 core workloads.

All workloads are generated randomly, but with a few predefined rules. No benchmark can occur twice within the same workload. This because we suspect that running two instances of the same benchmark, issuing the same memory operations in lock step, might cause unwanted interference that could skew our results. Also, we require that all benchmarks eligible for a workload set must be present in at least one workload in that set.

4.3 Performance Metrics

When we simulate our workloads, we expect destructive interference between benchmarks to cause slowdowns. Performance metrics are needed to quantify the performance of workloads and to compare the performance of different cache partitioning algorithms. This section defines two metrics; System Throughput (STP) [7] and the Harmonic Mean of Speedup (HMS) [7].

Two concepts are needed to define STP and HMS, private mode execution time and shared mode execution time. Shared mode execution time is the simulation time of a benchmark when run as a part of a workload. Private mode execution time is the simulation time of a benchmark when run alone on the same processor model. By definition, we expect a benchmark to execute slower in shared mode than in private mode.

Based on private and shared mode execution time we can define Normalized Progress (NP), or speedup, as $NP_i = \frac{T_i^P}{T_i^S}$. Here T_i^P and T_i^S is respectively the private and shared mode execution time for benchmark i. Normalized progress is a measure of benchmark progression in shared mode. A perfect value of 1 indicates that the benchmark progresses just as fast in shared and private mode. While a value of 0.5 indicates that the benchmark progresses at half the rate in shared mode compared to private mode. STP is defined by L. Eeckhout [7] as the sum of NP for all benchmarks in a workload, as shown in Equation 4.1. By definition, a perfect STP value equals the number of benchmarks in a simulation, in our case either 4, 8 or 16.

$$STP = \sum_{n=1}^{k} \frac{T_i^P}{T_i^S} \tag{4.1}$$

We also define HMS as the harmonic mean of NP values, as shown in Equation 4.2. The summation kernel in HMS is also known as Normalized Turnaround Time (NTT). HMS is therefore by definition the reciprocal of Average Normalized Turnaround Time (ANTT) [7], and hence HMS has a system level meaning relating to the benchmark's average normalized turnaround-time.

$$HMS = \frac{k}{\sum_{n=1}^{k} \frac{1}{\frac{T_i^P}{T_i^S}}} = \frac{k}{\sum_{n=1}^{k} \frac{T_i^S}{T_i^P}} \tag{4.2}$$

In general an increase in system throughput is also expected to improve turnaround time. However, there are cases where this is not true. Given a workload where most benchmarks are performing well while one is performing badly. The one benchmark performing badly may pull the STP value down, indicating a badly performing workload. At the same time, HMS could show another picture, as the well-performing benchmarks may dominate the average performance. As a result, both STP and HMS are needed to get an insight into system performance.

In addition to HMS and STP, we will use Misses Per Kilo Instruction (MPKI) when evaluating cache partitioning algorithms. MPKI is defined as the total number of misses in the LLC, per 1000 instructions. MPKI is an important metric in our experiments, especially because we are simulating an out of order core. Due to latency hiding in the processor we cannot assume that a reduction in MPKI necessarily will increase performance nor that a performance increase must be caused by a reduction in MPKI. As a result, all three metrics are required to gain full insight into the effects of various cache partitioning algorithms.

Chapter 5

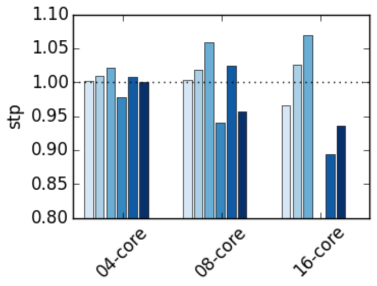
Results

This chapter describes an experiment that will compare the various implemented cache partitioning algorithms both against LRU and against each other. When executing this experiment, we utilize the base system configuration as previously detailed in Table 4.1. The L2 cache size is set to 128kB per core, and the L3 cache size is set to 4MB, 8MB or 16MB for respectively 4-, 8- and 16-core workloads. Each workload is simulated until all benchmarks in that workload have completed at least once. The first time a benchmark completes we store its statistics. After completion, a benchmark will be restarted unless it is the last benchmark to complete in which case we end the workload. We generate reference statistics for each benchmark by executing it in private mode. In private mode, we use the same system as in this experiment but with only a single core, the L2 and L3 sizes are set equal to the 4-core workloads; 128kB L2 and 4MB L3.

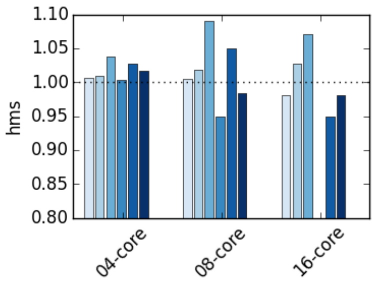
This chapter is divided into two sections. In the first section, we present average algorithm performance by core count. We then take a closer look at the results of the five 4-core workload groups.

5.1 Overall Results

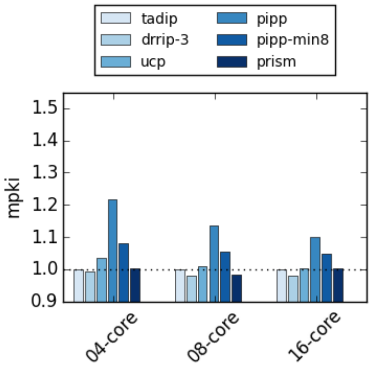
Figure 5.1a shows the average speedup of all workloads normalized to LRU performance grouped by workload size. We observe that most of the implemented algorithms perform close to LRU for the four core workloads. UCP gives the best speedup of 2.1% while PIPP performs badly with a 2.4% performance decrease. The modified version of PIPP, PIPP-min8, performs as good as LRU. When considering the harmonic mean of speedups as shown in Figure 5.1b we observe that all algorithms perform as good or better than LRU. Most noticeable is PIPP, which in terms of HMS is equal to LRU. As explained in Section 4.3, STP is a measure of the overall speedup of all benchmarks in the workload, and a decrease indicates that completing all of them is slower. HMS, however, measures the average speedup of each benchmark. Because PIPP is as good as LRU measured in HMS, it would indicate that individual benchmarks on average runs equal under PIPP and LRU.



(a) STP (not shown; PIPP 0.55).



(b) HMS (not shown; PIPP 0.61).



(c) MPKI

Figure 5.1: Average STP, HMS and MPKI normalized to LRU for all workloads, grouped by number of cores.

Figure 5.1c shows L3 cache misses, and as expected there is a significant increase, 20%, in misses for PIPP compared to LRU, which explains the bad performance. The modified PIPP algorithm has a lower increase of 6.7%. UCP, which is the highest performer in terms of STP and HMS, gives the third highest miss increase at 3.2% more misses than LRU.

The increase in both misses and performance for UCP could be an artifact of the lookahead algorithm, as shown in detail in Section 2.2.1. If there is a core, which has relatively few cache accesses per allocation period and also only accesses a small number of different blocks. Then, this core will have a high initial marginal utility as only a few ways are needed to provide hits for most accesses. Then on the other side of the spectrum there might be a core with many accesses, spread across all cache ways. This core, which causes more misses in total than the first one, may still have a lower initial marginal utility. If this is the case, UCP will first allocate ways to cache all the blocks accesses by the first core, before allocating any to the other. On the other hand, LRU would have prioritized the second core because it has a higher access frequency. By shielding the blocks of the first core UCP saves all misses caused by this core, but the other core will miss more compared to the

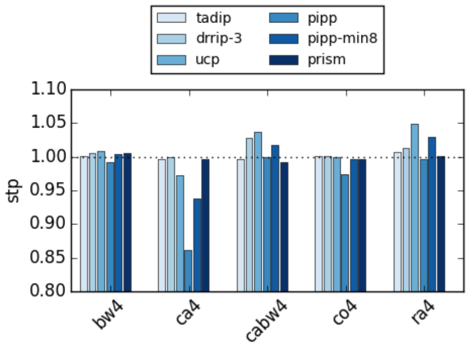
LRU case. In total UCP might cause more misses than LRU. The overall speedup might, however, be positive if the first core gains more from having fewer misses than the other core loses from having more.

With increasing core count, we increase the size of the L3 cache, but the associativity is unchanged. As a result, even more cores have to share the 32 blocks in each cache set. For some algorithms, especially PIPP, this increased cache set pressure significantly degrades performance. At 8-cores, PIPP has a 7.2% performance decrease measured in STP compared to LRU. The modified PIPP-min8 outperforms PIPP, and even slightly outperforms LRU by 2.2%, in the same situation. This is an indication that blocks inserted by PIPP do not stay in the cache for long enough to see much re-use. The modified algorithm seems to counteract this problem by inserting with an offset of 8 blocks higher than normal PIPP. In the 16-core case, this effect is even more visible, with PIPP performing 45% worse than LRU measured in STP and PIPP-min8 at only 7.6% worse than LRU. DRRIP and UCP, the two best performers in the 4-core case, continue to perform well for both 8- and 16-cores. UCP beating LRU by 5.7% and 6.9% measured in STP in 8- and 16-core workloads, and DRRIP at 1.8% and 2.6%. TADIP and PriSM, which both perform equal to LRU in the 4-core case, lose some traction when core count increases. TADIP performs equal to LRU for 8-cores, but 3.6% slower for 16-cores. PriSM cannot keep up for more than 4-cores, and performs 4.7% and 7.6% slower for 8- and 16-cores. As the number of cores increase, it might be tempting to blame TADIP's performance loss on an increased fraction of duel-sets, more duel-sets means more sets forced to use a non-optimal policy. However, since we scale the shared cache size linearly with increased core count while keeping the associativity static, the number of sets increase with the core count. Hence, the fraction of duel sets is equal in all cases. Neither TADIP nor PriSM caused an increase in misses, which is a good result considering they target miss-minimization. The fact that UCP can increase STP while increasing misses, and TADIP and PriSM decreases STP without affecting miss count is an important result that shows that miss minimization does not directly imply a speedup.

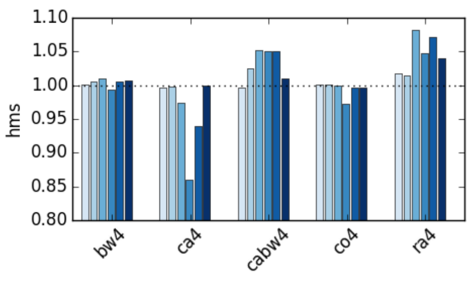
5.2 4-core Workload Results

Our 4-core workloads consist of five distinct groups, where four of the groups contain benchmarks with a specific characteristic. Section 4.2 lists each group and explain their specific characteristics. Figure 5.2 shows average STP, HMS and MPKI normalized to LRU for these five groups. Exploring the result from each of these groups individually is useful as it will show how various algorithms react to specific workload characteristics.

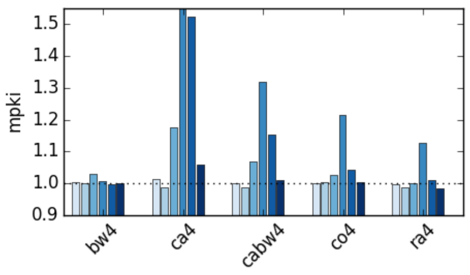
Bandwidth bound workloads contain benchmarks that do not benefit from increased cache space. These are benchmarks with mainly streaming access patterns. As expected the results show that none of the algorithms can significantly improve performance compared to LRU. As seen in Figure 5.2c UCP causes 3% more misses than LRU, and in return increases STP by 4.8% compared to LRU. We expect that even our bandwidth bound benchmarks will have phases with memory re-references,



(a) STP



(b) HMS



(c) MPKI (not shown ca4 pipp 1.9)

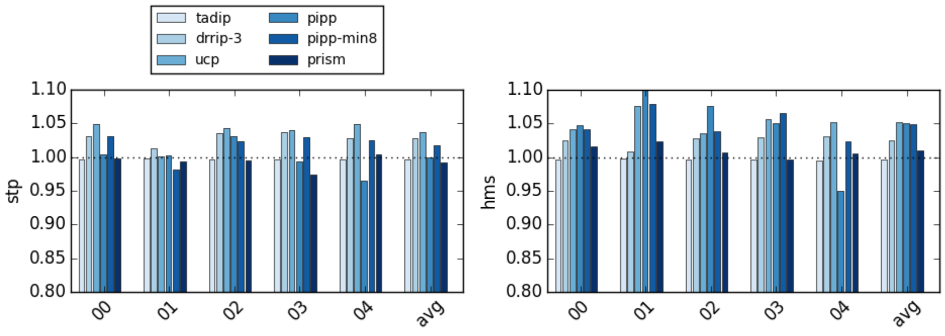
Figure 5.2: Average STP, HMS and MPKI normalized to LRU for all 4-core workload groups.

which means that their utility will increase. Based on our results, UCP seems to detect these phases and prioritize benchmarks correctly. While PIPP in theory also should be able to detect such changes, our result shows it does not. A possible explanation to this is that PIPP uses both utility and streaming flags. While an application may periodically have increased utility causing UCP to prioritize it, PIPP might still consider it as streaming due to a high miss-fraction, if this is the case, PIPP will ignore the increased utility.

Cache bound workloads contain benchmarks that are sensitive to changes in available cache space. In general these benchmarks have recency-friendly access patterns. Our results from these workloads show two main trends. First, as expected, LRU performs well, and none of the other algorithms increases performance or significantly reduce misses. Secondly, UCP and PIPP, the two algorithms that perform way partitioning, both reduce performance and cause a significant miss increase. While TADIP and DRRIP, which both mimic LRU and PriSM, which performs a variant of block level partitioning, performs as good as LRU in terms of performance. From this, we see that way partitioning is not beneficial if all benchmarks are recency-friendly. This is an expected result, as way-partitioning is designed to improve performance by shielding recency-friendly access patterns from thrashing caused by other cores. When all applications are recency-friendly, it seems that having the cores dynamically share the cache based on access frequency is a better solution. PriSM, which does block level partitioning, confirms this assumption as it performs as good as LRU in terms of STP and HMS. It does, however, cause a small increase in misses.

The performance of compute-bound workloads is expected to be mostly unaffected by the partitioning algorithm. Our results support this assumption, with the exception of PIPP, which again causes increased misses and a slight performance decrease. Once again, PIPP-min8 seems to remedy this, pointing to an issue with short block lifetimes in a PIPP managed cache, causing more misses.

Both cache and bandwidth bound workloads and the random workloads show results that concur with the overall averages discussed earlier. One interesting fact to note is that both versions of PIPP and UCP are equally good and also the best performers when measuring in HMS in cache and bandwidth bound workloads. This result points to PIPP being able to provide speedups of individual benchmarks that are good enough to raise the average while still performing as good as LRU measured in STP. Most likely this indicates that applications marked as streaming are performing badly while those shielded are performing so good their performance increase raises the average.



(a) STP (b) MPKI

Figure 5.3: STP and MPKI normalized to LRU for cache and bandwidth bound workloads.

In all our previous findings, we have observed that UCP is raising performance while also increasing misses. To ensure that this result is not just an artifact of result averaging, we show the per workload STP and MPKI for the cache and bandwidth bound workloads in Figure 5.3. As expected, the STP measurements show UCP providing a speedup in four out of five workloads. In the fifth workload, cabw01, UCP performs as good as LRU. When considering the MPKI measurements, we observe that UCP increase misses by at least 30% in the four workloads where performance is best. We also note that in the case where UCP performs as good as LRU, it also causes the least number of misses. These same trends are also visible in our other results. Based on this, we conclude that UCP can increase performance while also increasing misses.

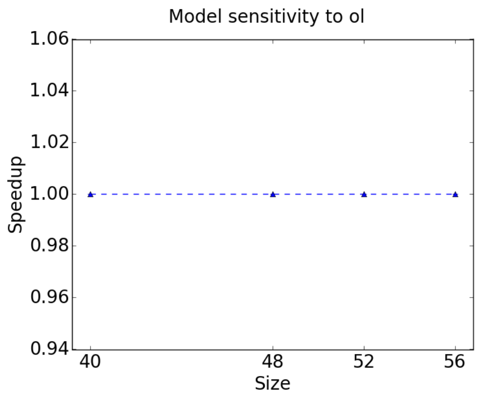
Chapter 6

Sensitivity Analysis

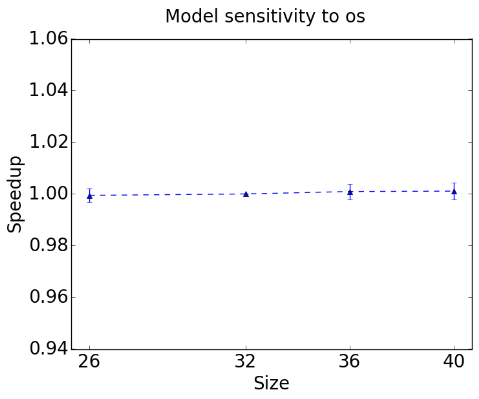
This chapter outlines a total of five experiments exploring the sensitivity of our simulated system and our cache partitioning results to various changes in the simulated architecture. The first experiment, covered in Section 6.1, will investigate the stability of our processor model and attempt to uncover and remove any bottlenecks. Then Section 6.2 explores how simulation clock skew in Sniper affects the outcome of our experiments. Sections 6.3 and 6.4 explore how algorithm performance is affected by the size of the L2 and L3 cache respectively. Finally, Section 6.5 explores how changing the bandwidth of the memory controller affects algorithm performance.

6.1 Processor Model Parameter Sensitivity

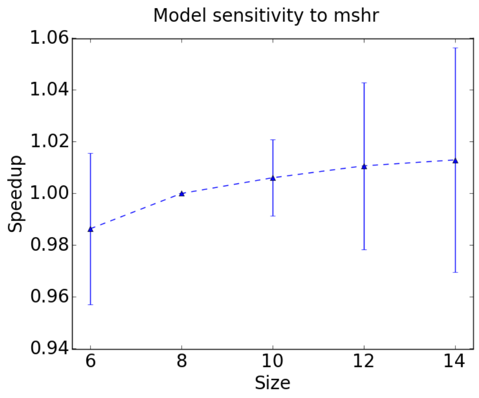
As detailed in the processor model overview, in Section 4.1, we have based our simulated processor core on the Nehalem [31] architecture. An experiment was devised, with the goal of gaining a better understanding and improved confidence in this model. Of the properties used to define the model, we selected a total of five we believe to have an important impact on the performance of the simulated core. We then varied each of the properties in isolation, keeping the others constant. For each parameter combination, we ran all our benchmarks and calculated the average speedup relative to the base configuration. The result of this experiment will show us how sensitive our simulated core is to configuration changes. If we observe a significant performance variation when changing one or more of the selected properties, we might have uncovered a bottleneck. In this case, it might be necessary to tweak the model until we find one that is less sensitive to change.



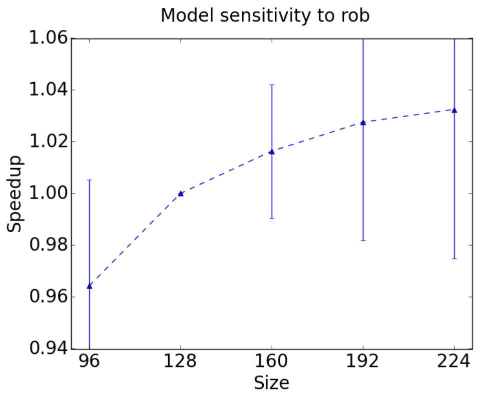
(a) Outstanding Loads.



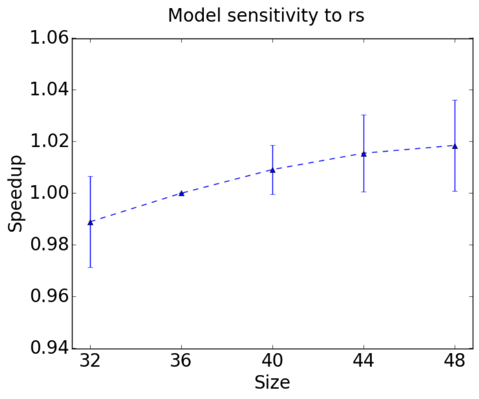
(b) Outstanding Stores.



(c) L1 Miss Status Hold Register.



(d) Re-order Buffer.



(e) Reservation Station.

Figure 6.1: Core model property sensitivity.

Figure 6.1 shows the average speedup of all benchmarks when we vary our five selected properties; Outstanding loads (ol), outstanding stores (os), L1 Miss Status Holding Registers (MSHR), re-order buffer size (rob) and reservation station entries (rs). Outstanding loads and stores specify the number of outstanding memory requests the core can have active in the rob. The number of L1 MSHRs decide how many outstanding cache misses the first level cache can handle before it has to block on a miss. The size of the rob and rs together decide how many instructions can be live during execution. Increasing the number of live instructions can increase the amount of ILP the processor can extract from the program while possibly increasing the cost of a branch miss prediction.

For two of the memory related properties; os and ol, we observe no improvement nor decrease in performance for the values we explored. When we increase the last memory related parameter, MSHR, we do observe a slight performance change. With an increasing number of MSHRs, the cache and hence the core can handle more outstanding memory requests. As a result, the core will be able to exploit more ILP, and a slight performance improvement is observed. Unlike the MSHRs, we do not expect the value of os and ol to affect performance. If the core is to gain performance from supporting more outstanding loads there has to be more than 48 loads among the 128 instructions that fit in the rob. Equally there must be more than 32 stores per 128 instructions for an increased os limit to be beneficial. Also, both os and ol are limited by the number of memory requests the memory system can handle, and the total number of L1 MSHRs is less than the size of both os and ol. The observed performance gain when increasing the number of MSHRs in the first level caches, as seen in Figure 6.1c, is less than 1% with a 50% storage increase. We also observe a standard deviation of more than 3%.

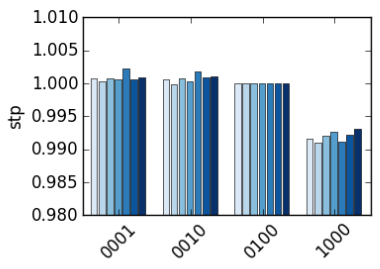
When Increasing the size of the rob and the number of rs entries, we observe a slight increase in performance. Figures 6.1d and 6.1e show an average performance increase of about 2% with more rob entries, and about a 3% increase with more rs entries. We observe that these increases come at the cost of a 75% and 50% storage increase respectively. Also, we observe that the standard deviation in both cases is about the same as the average performance increase.

When reviewed, these results lead us to conclude that the processor model we have presented, based on the Nehalem architecture, is stable and that we have no obvious performance gains from small adjustments. As a result, we decide to continue using this model for the rest of our experiments without making any adjustments.

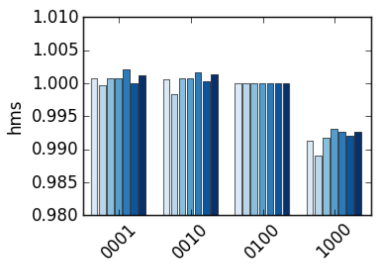
Considering that we base our model on an actual architecture and that our simulator strives to simulate the core model of that same architecture, it is not a far-fetched result observing little sensitivity to property changes. During the design process of the architecture, it is natural to expect that the designers made a conscious choice between speed and area using a similar analysis. The final properties would then most likely have been selected to provide a stable middle ground, which we see reflected in our simulation results.

6.2 Clock Skew Barrier Sensitivity

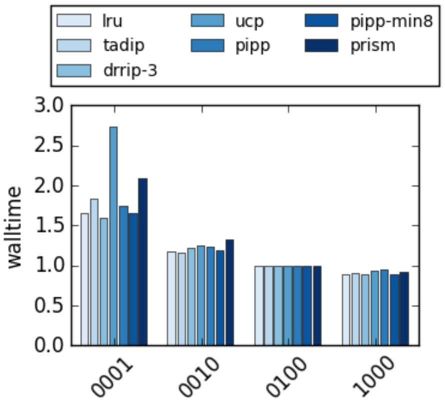
As explained in Section 3.1 one of the techniques that make Sniper faster than conventional cycle-accurate simulators, such as gem5, is the use of multiple simulation threads that each simulate one processor core. A method that keeps the simulation threads in sync is required to simulate inter-core interactions correctly. The method used to keep the threads in sync affects both simulation accuracy and simulation time. By having a relaxed synchronization method, one can improve simulation time, at the cost of simulation accuracy. In our experiments, we have used barrier synchronization with a barrier width of 100 cycles. Any inter-core interactions that occur between two successive barriers are not guaranteed to occur in the correct order, but events separated by a barrier will be simulated in the correct order. In other words, within a single barrier there is the possibility that all simulation threads run sequentially. For our work, this implies that there is a possibility that memory requests within a barrier is sent to the cache sorted by the core id, rather than by time. Because of this possibility we expect that changing the Clock Skew Minimization Barrier (CSMB) value could have a noticeable effect on our experimentation results.



(a) STP sensitivity to CSMB.



(b) HMS sensitivty to CSMB.



(c) walltime sensitivity to CSMB.

Figure 6.2: STP, HMS and walltime sensitivity to size of CSMB.

We devised an experiment to investigate how much the choice of synchronization barrier width affects our results, and also how much it affects simulation time. In the experiment, we vary the value of the CSMB and compare average STP, HMS and MPKI values for all 4-core workloads. Figure 6.2 contains plots for both STP and HMS relative to the default 100 cycle barrier. From the graph, it is apparent that lowering the value below 100 cycles causes negligible variations in our average results. The most noticeable is PIPP; that varies by about 0.2% with a tighter barrier interval. Increasing the interval to 1000 cycles results in a more noticeable difference in measurements. For both HMS, shown in Figure 6.2b, and MPKI, not shown, the trends are the same.

The variance in simulation walltime when we vary CSMB values, as shown in Figure 6.2c, is as expected. When lowering the barrier interval we measure an increase in average walltime. Increasing the barrier value causes a slight decrease in walltime. We observe that the performance gain by increasing the barrier is small compared to the result variation. When decreasing the barrier, the opposite is true; the result variation is small compared to the walltime increase. These observations suggest that a barrier width of 100 cycles is a good trade-off between accuracy and walltime.

6.3 L2 Cache Size Sensitivity

In this section, we investigate how increasing the size of the private cache affects the performance of the cache partitioning algorithms. We ran the same experiment as in Section 5.1, but with varying L2 sizes. The L2 configurations are as shown in Table 4.3, to summarize we utilize cache sizes of 128kB, 256kB, 512kB, and 1024kB. As in previous experiments, we set the L3 size depending on the workload size. In this experiment, we only utilize the three random workload groups. We do this to be able to aggregate and compare 4-core results to 8- and 16-core results. We omit the 4-core workloads with specific traits because they would bias the overall 4-core averages. Also, we have only included plots of the STP results, this because HMS and MPKI results did not add any additional insight in this experiment.

Figure 6.3 shows the average number of L3 accesses for random workloads with varying L2 cache size. As can be seen from the graph, by increasing the size of the L2 cache we are decreasing the number of accesses to the L3 cache. In other words, the L2 caches are hiding an increasing amount of memory requests from the shared level. We expect this increased filtering of requests to have an impact on the performance of the implemented algorithms.

Figure 6.4a shows the speedup of TADIP normalized to LRU measured in STP. As seen previously, TADIP performs as good as LRU in both 4- and 8-core workloads with a 128kB L2 cache. With increasing L2 cache size TADIP steadily outperforms LRU with between 0.1% and 0.6% depending on the configuration. At 16-cores, TADIP underperforms compared to LRU, as previously shown. We note that, in this case, increasing the L2 size seems to cause a further decrease in TADIP performance, while the opposite is true in the 8-core case. We expect that TADIP will react slower to changes in application phases as memory filtering increases,

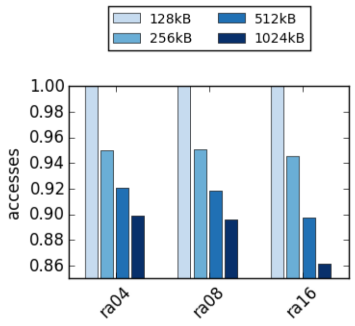
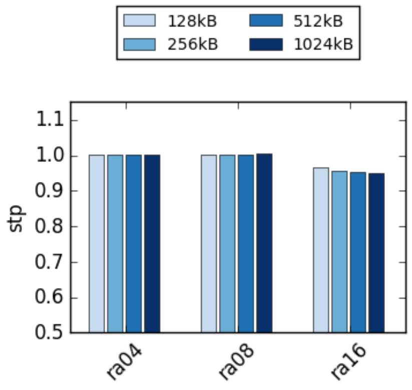


Figure 6.3: Relative number of accesses to L3 cache with varying L2 size.

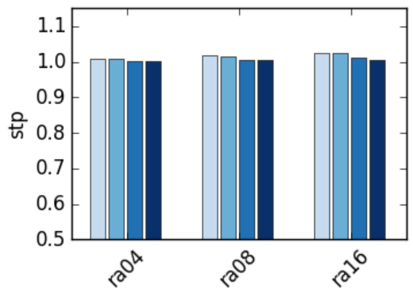
because of the counter architecture used to switch between algorithms. This effect does not seem to have a noticeable impact on results for the 4- and 8-core runs, but we assume it is causing the visible decrease in performance for the 16-core runs.

DRRIP as already covered outperforms LRU, Figure 6.4b confirms this. The figure also shows that increasing the L2 size causes a reduction in DRRIP performance. We know that DRRIP uses a step-wise promotion policy where each successive access promotes a block one position. Naturally less information about successive accesses will be available to the shared level as filtering in the private levels increase. It is consequently not unexpected that DRRIP suffers from increased filtering by private cache levels. From the figure, we note that DRRIP seems to be slightly less sensitive to small changes in L2 size with increasing core count, but in all cases a 1024kB L2 causes DRRIP performance to mimic LRU performance.

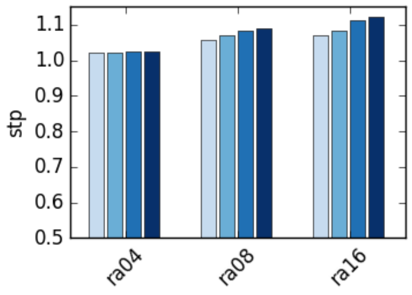
In contrast to the previous algorithms, UCP performance increases with L2 cache size in all workloads, as seen in Figure 6.4c. We know that UCP uses a utility algorithm as the mechanism for allocating ways to cores. The input to this algorithm changes when we increase filtering of requests to the shared cache level. As a result, the allocation of ways to cores is also expected to change, but this is the intended mechanism of UCP and should not negatively affect performance. UCP uses LRU to manage replacement for each core, but UCP under normal circumstances only allows a core to evict one of its own blocks. We have already covered that this is why UCP outperforms LRU in the base configuration, in Section 5.1. As filtering increases at the private level, we notice that UCP increases its performance compared to LRU. We expect that this is because, with increased private cache, more requests from recency-friendly applications can be satisfied by the private levels and less information reaches the shared level. Trashing and streaming applications will still have its requests propagate to the shared cache, largely independent of the size of the private cache. Hence with increasing private cache size we expect LRU to make worse decisions by prioritizing trashing and streaming patterns due to their access frequency. UCP with utility-based way-partitioning will not suffer as much from the lack of information about recency-friendly applications, and as the results state, can take advantage of increasing private cache size.



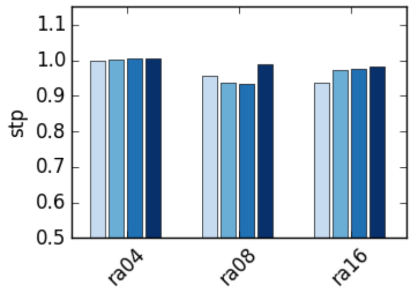
(a) Speedup of TADIP normalized to LRU.



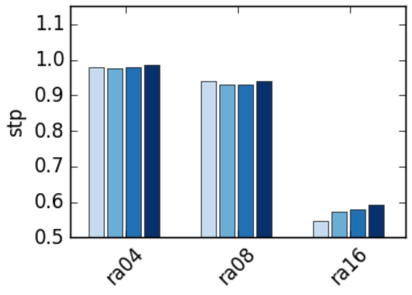
(b) Speedup of DRRIP normalized to LRU.



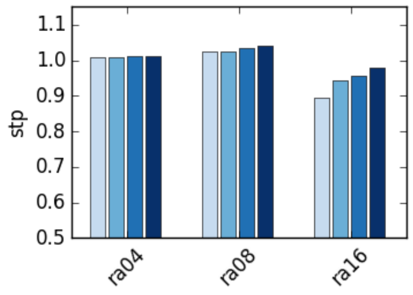
(c) Speedup of UCP normalized to LRU.



(d) Speedup of PriSM normalized to LRU.



(e) Speedup of PIPP normalized to LRU.



(f) Speedup of PIPP-min8 normalized to LRU.

Figure 6.4: Speedup of cache partition algorithms normalized to LRU with increasing private L2 size

PriSM calculates target allocations for each core with the goal of reducing misses. This technique bears some resemblance to the utility calculation done by the UMON. As with UCP, we expect PriSM to be able to increase its performance compared to LRU with increased private cache size because it will continue to limit the cache use of streaming and trashing applications. We find this expectation reflected in our results. For both 4- and 16-core workloads we observe an increase in performance compared to LRU as the size of private cache increases. In the 8-core results we see the same trend between the smallest and largest L2 configuration, but we unexpectedly observe a performance drop for 256kB and 512kB configurations. It is unclear what causes this performance drop, and further work is required to analyze this.

Finally Figure 6.4e show the performance of PIPP, and Figure 6.4f shows the performance of the modified PIPP algorithm. Since PIPP uses the same utility algorithm as UCP and also aims to achieve the same allocations as UCP, we expect them to show similar trends. This expectation somewhat holds true for the 4-core case, where there is a slight upward trend with increasing L2 size. However, PIPP underperforms compared to LRU in all workloads, and with increasing core count performance drops significantly. We expect the short lifetime of blocks in PIPP managed caches to be the cause of this, as covered in Section 5.1. The modified PIPP algorithm shows a performance development much closer to what is expected, at least for 4- and 8-core workloads. We observe the same increase in performance with increased L2 cache size as seen in the UCP case. In the 16-core workloads, the performance trend is still as expected, but the modified algorithm performs worse than LRU. This performance reduction for larger core counts has also been observed in previous research [18].

6.4 L3 Cache Size Sensitivity

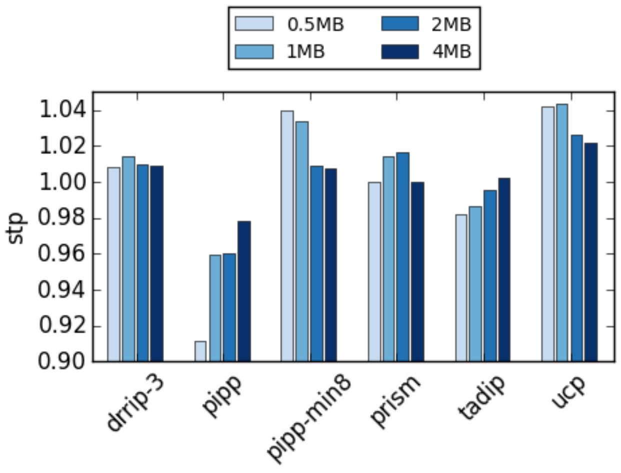


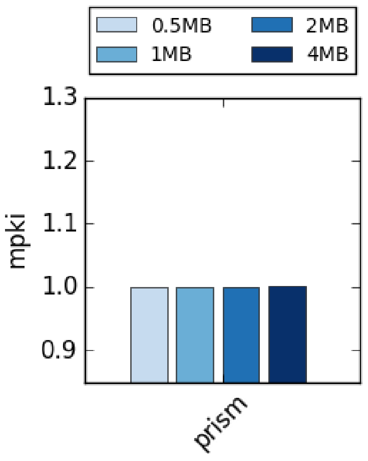
Figure 6.5: Speedup of cache partition algorithms normalized to LRU with decreasing shared L3 size

In this section, we cover an experiment where we run all 4-core workloads with varying L3 cache size. While we have already shown in Section 4.1 that our simulated model is realistic compared to current processor architectures, we want to explore how algorithm performance changes when we constrain available L3 cache. This experiment uses the same simulated system as in the cache partitioning experiment, Section 5.1. We use four different L3 sizes; 4MB, 2MB, 1MB, and 0.5MB. When we reduce the size of the shared cache level, we keep the associativity constant. As a result, we have fewer sets in the cache and hence, more addresses map to the same set. Table 4.4 shows detailed information about the two larger configurations. The details for the two smaller configurations are equal to the L2 configurations of the same size, shown in Table 4.3, but with an associativity of 32. Fewer sets cause increased pressure on each set. We expect to see some of the algorithms further their improvement over LRU in this situation. Also, we expect PIPP, which already has shown bad performance compared to LRU, to continue this trend.

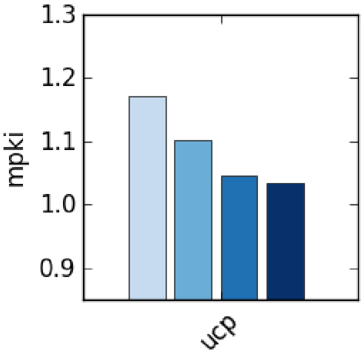
Before showing the results of this experiment, we will briefly discuss a special case that arises in this experiment. When we set the L3 cache to 0.5MB, we have a situation where the sum of the L2 caches equals the L3 cache. This is an extreme case in an inclusive cache architecture because when the L2 caches are fully utilized there is no spare room in the L3 cache. In a real processor, it would not make sense to have an L3 of this size. We still experiment with this configuration, as interesting results may arise in situations where the L2 caches are not fully utilized, such as in a mixed workload with streaming and recency-friendly applications.

Figure 6.5 show the speedup of all algorithms normalized to LRU for varying shared cache size. DRRIP is the algorithm that shows least variation across the various shared cache sizes. The figure shows DRRIP performing comparable to LRU in all cases, with a negligible increase of 0.3% in the 1MB case. TADIP that in the baseline scenario performs as good as LRU seems to suffer from the increased set pressure, with increasingly worse performance as the cache size decreases. In our implementation, we scale the number of duel-sets relative to the total number of cache sets. Hence, for both DRRIP and TADIP the fraction of duel sets is constant across the various L3 configurations.

As expected the performance of PIPP decreases as the set pressure increases. We have previously, in Section 5.1, postulated that the potentially short lifetime of blocks in a PIPP managed cache may be the cause of the performance decrease compared to LRU. This experiment further shows that when the number of accesses to a single set increases the performance of PIPP further decreases compared to LRU. The MPKI in the 0.5MB cases, not shown here, is over 50% worse than the LRU case, compared to only 20% worse in the 4MB case. PIPP-min8, a modified version of PIPP, have previously been shown to improve performance over normal PIPP replacement. This is also the case when reducing shared cache size. Figure 6.5 shows that PIPP-min8 not only performs as good as LRU in the base experiment, but with increased set pressure actually performs better than LRU. With a 0.5MB L3 cache, the modified PIPP algorithm performs 4% better than LRU measured in STP. In the same configuration, the unmodified algorithm performs about 18% worse compared to LRU. This result clearly shows the advantage of the extended



(a) PriSM



(b) UCP

Figure 6.6: MPKI normalized to LRU with decreasing L3 cache size.

block lifetime in the modified PIPP algorithm, and at the same time points a fundamental performance problem with PIPP.

Next, we have PriSM, which shows a slight performance increase with the 2MB and 1MB cache. At both 4MB and 0.5MB PriSM performs as good as LRU. Figure 6.6a shows the MPKI for PriSM. From this figure, we observe that PriSM in all configurations causes the same number of misses as LRU. This is true even when PriSM shows an performance increase measured in STP. Finally, we observe a performance increase by UCP in Figure 6.5. In previous sections (5.1 and 6.3) we have shown that UCP is the top performer of our algorithms when measured in STP. This is also the case in this experiment. We observe that UCP increases performance compared to LRU in both the 2MB and 1MB case, in the 0.5MB case is comparable to the 1MB case. Interestingly, Figure 6.6b shows that while UCP increases performance compared to LRU it also causes more misses, shown by an increase in MPKI. Previous experiments have also shown this effect, as seen in Section 5.1.

6.5 Memory Bus Speed Sensitivity

In this experiment, we explore how changes to the memory bus bandwidth affect the performance of the cache partitioning algorithms. With lower bandwidth, we expect that memory requests will take longer to complete, also we expect memory queue times to increase in periods with high utilization. However, we know that increased memory latency does not necessarily imply lower performance, as the OOO-core may be able to hide the increased memory latency. We also know that

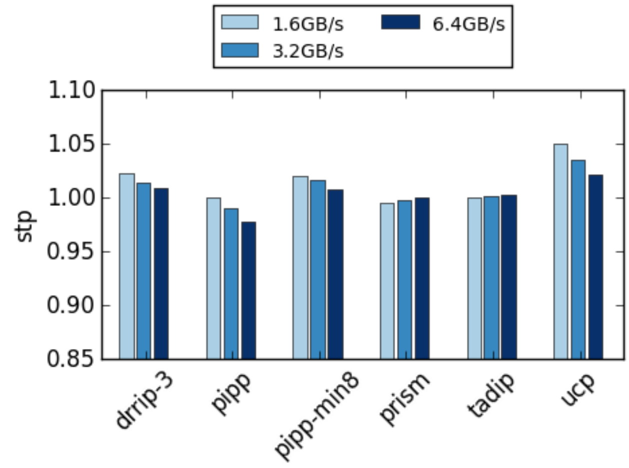
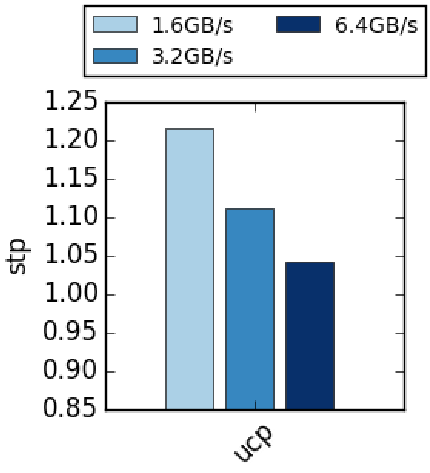


Figure 6.7: Speedup of cache partition algorithms normalized to LRU with decreasing memory bus bandwidth

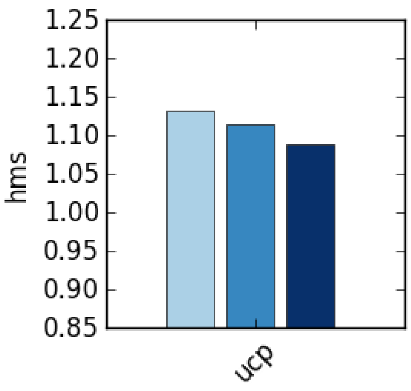
increasing the memory latency may have an impact on the amount of speculative execution performed by the cores. As a result, we might see a change in the number of memory requests and also MPKI in our experiments. For this experiment, we use the same base system as in all previous experiments, and we utilize all our 4-core workloads. The memory bus bandwidth is varied, from the standard 6.4GB/s down to 3.2GB/s and 1.6GB/s.

Figure 6.7 show average STP of all 4-core workloads for each algorithm. TADIP shows no sensitivity to reduced memory bandwidth, both the STP measurements shown in the figure and the MPKI measurements are about equal in all cases. The results for DRRIP show a slightly better performance compared to LRU with reduced memory bandwidth. UCP has the best speedups in this experiment, increasing from about 4.8% in the base case to about 7% with reduced memory bandwidth. The only algorithm that shows a decline in performance measured in STP is PriSM. However, the HMS measurements for PriSM show a steady increase from about 2% better than LRU in the base case to about 4% in the most constrained case. This indicates that some of the benchmarks are seeing a performance improvement, while others are slowed down enough to affect the STP measurements negatively. For the two final algorithms, PIPP, and PIPP-min8, we see a slight performance improvement. Most notably we see that PIPP performs as good as LRU in the most constrained case. The performance development in PIPP-min8 mimics that of PIPP, which has been shown to be the case in several previous experiments. PIPP-min8 outperforms both PIPP and LRU.

In all our experiments, we have shown that UCP outperforms the other implemented algorithms. Both when reducing the available memory bandwidth in this experiment, and when we reduced the size of the LLC in Section 6.4, UCP has show improved performance over LRU. Because of this, we ran an additional



(a) STP.



(b) HMS.

Figure 6.8: Speedup of UCP normalized to LRU with reduced L3 cache.

experiment varying the memory bandwidth, but this time utilizing the smallest L3 cache configuration used in Section 6.4, 0.5MB. Because of the time constraints on this thesis, we only have results for UCP compared to LRU. Figure 6.8 shows the speedup for UCP compared to LRU with varying memory bus bandwidth. These results show that when we combine the high cache pressure caused by reduced LLC size, with increased memory latency, UCP can outperform LRU greatly. Our results show UCP performing more than 21% better than LRU measured in STP in the most constrained case. Also, UCP shows a 15% increase in HMS compared to LRU in the most constrained case.

Chapter 7

Discussion

In this chapter, we discuss various shortcomings in our implementation, our simulator framework, and our processor model. We explain why various choices were made and what might have influenced our final results.

7.1 Parameter Fitting and Lack of Implementation Details

All of the algorithms we have implemented and evaluated in this thesis have one or more parameter controlling its operation. As Chapter 3 explains, we have mainly opted to use parameter values from the original papers. Each of the original papers has attempted to show how their algorithm performs better that LRU and the current best theoretical algorithm. It is fair to assume that the authors have chosen parameter values that are fitted to provide good results on their benchmarks. Because none of the papers uses the same benchmark set, and none of the benchmark sets matches the one we used in this thesis, there is a potential optimization opportunity here.

By performing experiments selecting the property values that give the best overall performance on our workloads we could potentially improved performance of several algorithms. Even though it might seem like an unfair comparison, comparing algorithms with nonoptimal parameter values, we still choose to not fit parameters to our benchmarks. The reasoning behind this is that an algorithm has to not only perform well under benchmarking, but also under a real world scenario. In this case, one cannot reliably fit parameters because the workload is not known ahead of time. Hence, running the algorithms using the authors selected parameter values on an unknown workload set might give results closer to a real life scenario. Also by fitting parameters, one might end up overfitting. In this case, the parameters result in a good performance on the workload set, but performance on all other workloads decreases. We feel this would also be an unfair comparison.

As mentioned in Section 3.3 we also had to make some assumptions in our implementations. These assumptions were necessary due to missing or unclear implementation details in the original papers. While we made all attempts to keep our implementation identical to the algorithm described by the original authors, there is still a chance some of our algorithms function slightly different due to one of these assumptions. As a result, our performance evaluations results may not follow those of the original papers.

7.2 Clock Skew Synchronization Barrier

One of the advantages of Sniper is the use of multiple simulation threads [5]. We detailed how this could improve performance in Section 3.1, and also how this speed improvement could result in decreased accuracy. In the context of this thesis, there are two effects of having multiple simulation threads that could bias our results. The problem relates to the ordering of memory events. With multiple simulation threads, the only guarantee the simulator gives regarding the ordering of memory events from different cores are that events from different intervals execute in the correct order. The order of events from different cores within the same interval depends on the OS scheduler. In the worst case the OS scheduler may schedule all simulation threads serially, this will cause the ordering of memory events to be by core id and not by time. All the implemented cache partition algorithms assume that memory requests are in order, and this inaccuracy will break this assumption. In addition to the effects on the cache partitioning, memory bus scheduling is hard to estimate when requests come out of order, this was an issue covered in the author's autumn project [23].

Section 6.2 presents an experiment where we attempted to vary the CSMB interval, and we evaluate the performance using changes in measured STP and HMS. The results of this experiment showed that there is little change when lowering the CSMB from the default value of 100 cycles all the way to 1 cycle. This would indicate that although the potential issues outlined above are serious, they seem to not affect results when we synchronize every 100 cycles. The cache used in this experiment and all other four core experiments is 4MB with 32 blocks per set. In other words, there are 2048 sets. For out of order memory accesses to have an impact on our accuracy there would have to be at least one of the 2048 sets accessed by two cores within the same interval. Additionally the order of the requests have to be reversed. For this to have a serious effect, we expect there would have to be a significant larger number of requests resolving the same set within the same interval. This seems to be an unlikely situation, and our experiment supports this assumption.

The experiment only did a sensitivity analysis of 4-core workloads. We chose to do this because repeating it for 8- and 16-cores would have been unpractical given the time constraints of this work. However, as we scale the number of sets in the cache linearly with the number of cores, there is no reason to doubt that the results would have been similar also for 8- and 16- cores.

We note that in the most constrained case in the in the L3 sensitivity experiment, in section 6.4, there are only 256 sets in the cache. With this reduced number of cache sets the effects of the clock skew, as investigated in section 6.2, might become more visible. We still however expect the effect to be minimal because multiple memory requests must issued out of order within the 100 cycle barrier and resolve to the same set, for the effect to be noticeable.

7.3 Uniform LLC Access

In Section 4.1 we present the processor model used on our experiments. Our architecture, as shown in Figure 4.1, assumes a uniform access latency to the LLC. All the algorithms presented in this thesis also assume uniform access latency to the LLC or rather does not take into account non-unform access latency when assigning cache space. This assumption of uniform access latency is reasonable for 4- and 8-core processors and requires a uniform access method between the L2 caches and the LLC. One possible solution is a crossbar. However, scaling this architecture to 16 cores is somewhat unreasonable as the crossbar grows exponentially. This is visible in the newer 18 core Intel i7 where the last level cache is distributed around the chip and interconnected using ring interconnects [14]. This solution results in non-uniform access to the LLC. Due to this fact, we chose not to simulate 32 and higher core workloads, even thought our simulation framework supports it.

Chapter 8

Conclusion

8.1 Conclusion

In this thesis, we have curated a list of important and recent work in the cache management research field. We have presented a theoretical explanation of all algorithms and compared them with each other. A simulation framework based on Sniper was created, and several of the algorithms presented have been implemented within this framework. We have presented some potential error sources within Sniper; most notably is the lack of precision when simulating inter-core interactions. However, experiments were conducted to investigate the severity of this inaccuracy, and based on the results we conclude that Sniper is a viable choice for this research.

Figure 8.1 is a highly simplified view of 4-core random workload results, showing average STP and HMS for all algorithms by publication year, normalized to LRU performance. Throughout our work, UCP has proven to be the best performer providing up to 5% speedups measured in STP for our main experiment. With constrained resources, we observed more than 20% performance improvement with UCP. PIPP has shown comparable performance to LRU in 4-core random workloads, but has been unable to compete with LRU in most other cases. We suspect this is due to a short lifetime of cache blocks in a PIPP managed cache. Both DRRIP and TADIP, which are arguably simpler schemes compared to UCP and PIPP, have shown to outperform LRU. Neither has been able to achieve speedups comparable to UCP. In terms of misses, we have observed that DRRIP, TADIP, and PriSM have succeed in reducing the number of misses compared to LRU. UCP has proven to cause an increased number of misses. We suspect that this may be an artifact of the greedy lookahead algorithm, which prioritizes high marginal utility and that might not always equal fewer misses. Section 5.1 covers in detail how this might result in both increased miss count and performance in the same workload.

Independent of the performance metric used, none of the implemented algorithms has proven to beat UCP on average. Our results make it is tempting to conclude that UCP is the best solution. In sections 6.4 and 6.5 we demonstrated that changes in the simulated architecture can greatly affect algorithm performance,

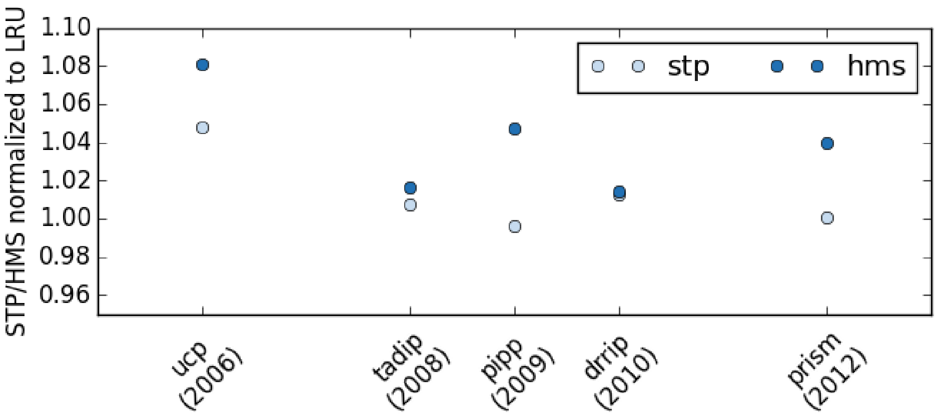


Figure 8.1: Average algorithm performance measured in STP and HMS normalized to LRU by year of publication.

and in all cases UCP had a positive trend. While we claim that UCP is the best solution in our setup, this might not be the case in all architectural configurations. LRU is still the dominating algorithm in hardware implementations, even if UCP has existed for nine years, and several publications have shown its improvement over LRU. Compared to LRU, UCP requires more hardware to implement. In a real design process, this additional hardware will come at the cost of reduced area for some other functionality. In the end, this might cause the improvement of cache performance to be reverted by a decrease in performance of some other component.

8.2 Future Work

The time constraints put on this thesis has limited the number of algorithms presented. Given more time we would like to continue exploring the vast research field that is cache management, and bring further algorithms into this comparison. Implementing more algorithms within the simulation framework would also allow for more interesting results.

The Sniper simulation system has integrations to various power estimation tools. Given that power is a concern when designing processor cores today, it would make sense to extend the simulation framework to provide power estimations as well. With this data at hand, we would be able to use not only performance but also increased power consumption when comparing algorithms. The potential for implementing this was not considered during this thesis, mainly because of the time constraint.

Bibliography

[1] ARM. Cortex-A5. 2010.

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

[3] Shekhar Borkar. Thousand Core Chips—A Technology Perspective. pages 746–749, 2007.

[4] CARD. https://www.idi.ntnu.no/grupper/dm/start, 2015.

[5] Trevor E. Carlson, Wim Heirmant, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, 2011.

[6] TNJM Chen-Han and HK Sankaralingam. gem5, GPGPUSim, McPAT, GPUWattch," Your favorite simulator here" Considered Harmful. *research.cs.wisc.edu*, 2014.

[7] Lieven Eeckhout. *Computer Architecture Performance Evaluation Methods.* Morgan & Claypool Publishers, 2010.

[8] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. *... Computer Architecture (...*, pages 1–12, January 2010.

[9] Greg Hamerly and Erez Perelman. SimPoint 3.0: Faster and More Flexible Program Phase Analysis. *Journal of Instruction-Level Parallelism*, 7(7):1–28, 2005.

[10] Greg Hamerly, Rez Perelman, and Bard Calder. How to use simpoint to pick simulation points. *ACM SIGMETRICS Performance ...*, pages 25–30, 2004.

[11] Nikolaos Hardavellas, Stephen Somogyi, Thomas F Wenisch, Roland E Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C Hoe, and Andreas G Nowatzyk. SIMFLEX : A Fast , Accurate , Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture. *Performance Evaluation Review*, 31(4):31–35, 2004.

[12] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2012.

[13] Joshua Ho, Brandon Chester, Chris Heinonen, and Ryan Smith. The iPhone6 Review, 2014.

[14] Joel Hruska. Intel's new 18-core Haswell Xeon chips will try to preempt the ARM server onslaught, 2014.

[15] Tarush Jain and Tanmay Agrawal. The Haswell Microarchitecture - 4th Generation Processor. 4(3):477–480, 2013.

[16] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, and Joel Emer. Adaptive insertion policies for managing shared caches. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, page 208, 2008.

[17] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). *Proceedings of the 37th annual international symposium on Computer architecture - ISCA '10*, page 60, 2010.

[18] R Manikantan, K Rajan, and R Govindarajan. Probabilistic shared cache management (PriSM). *ACM SIGARCH Computer ...*, 00(c), 2012.

[19] R. Manikantan, Kaushik Rajan, and R. Govindarajan. NUcache: An efficient multicore cache organization based on next-use distance. In *Proceedings - International Symposium on High-Performance Computer Architecture*, pages 243–253, 2011.

[20] Sun Microsystems. UltraSPARC T2 Processor. (8):1–2, 2007.

[21] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald Iii, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite : A Distributed Parallel Simulator for Multicores. (January), 2010.

[22] Gordon E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[23] Runar Bergheim Olsen. Comparison of Cycle-accurate Simulation and Analytical Modelling for Multi-core Memory System. 2014.

[24] Michael Pellauer, Michael Adler, Michel Kinsy, Angshuman Parashar, and Joel Emer. HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing. *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 406–417, February 2011.

[25] MK Qureshi and YN Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. *Proceedings of the 39th Annual IEEE/ACM . . .*, pages 423–432, December 2006.

[26] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381, June 2007.

[27] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pages 187–198, 2010.

[28] Premkishore Shivakumar and Norman P Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. *Computer*, (2001/2), 2001.

[29] SPECCPU. http://www.spec.org/cpu2006/, 2006.

[30] H Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, pages 1–9, 2005.

[31] ME Thomadakis. The architecture of the Nehalem processor and Nehalem-EP SMP platforms. *Resource*, 2011.

[32] Maurice V. Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, 29(1):2–7, 2001.

[33] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003.

[34] Yuejian Xie and GH Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. *ACM SIGARCH Computer Architecture News*, pages 174–183, 2009.

[35] J.J. Yi, S.V. Kodakara, R. Sendag, D.J. Lilja, and D.M. Hawkins. Characterizing and comparing prevailing simulation techniques. *11th International Symposium on High-Performance Computer Architecture*, 2005.

[36] Dongyuan Zhan, Hong Jiang, and Sharad C. Seth. CLU: Co-optimizing locality and utility in thread-aware capacity management for shared last level caches. *IEEE Transactions on Computers*, 63(7):1656–1667, 2014.

Appendix A

Workloads

Group	Workload	Benchmarks			
ra	ra8-0	tonto	bzip2	gobmk	h264ref
		soplex	astar	hmmer	mcf
	ra8-1	sjeng	omnetpp	GemsFDTD	calculix
		perlbench	xalancbmk	gcc	mcf
	ra8-2	h264ref	sjeng	gromacs	milc
		tonto	libquantum	povray	astar
	ra8-3	calculix	omnetpp	gromacs	mcf
		gobmk	leslie3d	xalancbmk	soplex
	ra8-4	dealII	leslie3d	povray	gamess
		wrf	sphinx3	cactusADM	perlbench
	ra8-5	lbm	xalancbmk	libquantum	sjeng
		cactusADM	zeusmp	hmmer	povray
	ra8-6	h264ref	lbm	mcf	wrf
		omnetpp	hmmer	perlbench	gamess
	ra8-7	libquantum	leslie3d	h264ref	namd
		cactusADM	astar	perlbench	dealII
	ra8-8	astar	sjeng	povray	h264ref
		calculix	gcc	gamess	wrf
	ra8-9	gcc	gamess	milc	gromacs
		tonto	hmmer	wrf	sphinx3

Table A.1: 8-core workloads

Group	Workload	Benchmarks			
ra	ra16-0	bzip2	calculix	astar	omnetpp
		wrf	gamess	gobmk	povray
		leslie3d	xalancbmk	namd	milc
		hmmer	GemsFDTD	lbm	gcc
	ra16-1	wrf	gobmk	zeusmp	leslie3d
		perlbench	hmmer	milc	cactusADM
		calculix	tonto	bwaves	povray
		omnetpp	h264ref	gromacs	bzip2
	ra16-2	wrf	GemsFDTD	libquantum	tonto
		omnetpp	dealII	perlbench	soplex
		lbm	leslie3d	bwaves	calculix
		xalancbmk	milc	gamess	namd
	ra16-3	soplex	povray	h264ref	leslie3d
		namd	gobmk	hmmer	sjeng
		astar	omnetpp	gcc	lbm
		gamess	wrf	sphinx3	GemsFDTD
	ra16-4	astar	bwaves	cactusADM	zeusmp
		h264ref	omnetpp	namd	gromacs
		GemsFDTD	libquantum	sphinx3	hmmer
		xalancbmk	leslie3d	gcc	gamess
	ra16-5	sphinx3	GemsFDTD	soplex	milc
		libquantum	astar	zeusmp	omnetpp
		namd	bwaves	bzip2	lbm
		povray	cactusADM	h264ref	dealII
	ra16-6	bwaves	hmmer	sjeng	leslie3d
		astar	mcf	GemsFDTD	gromacs
		libquantum	sphinx3	bzip2	omnetpp
		gcc	milc	cactusADM	zeusmp
	ra16-7	omnetpp	calculix	tonto	hmmer
		h264ref	astar	wrf	sjeng
		soplex	namd	zeusmp	povray
		leslie3d	gromacs	gamess	perlbench
	ra16-8	lbm	h264ref	soplex	gobmk
		cactusADM	tonto	zeusmp	GemsFDTD
		calculix	gromacs	bzip2	gcc
		sphinx3	mcf	sjeng	namd
	ra16-9	lbm	sjeng	sphinx3	tonto
		milc	mcf	wrf	omnetpp
		bzip2	leslie3d	gcc	perlbench
		gamess	calculix	namd	xalancbmk

Table A.2: 16-core workloads

Group	Workload	Benchmarks			
ca	ca4-0	perlbench	h264ref	gobmk	hmmer
	ca4-1	perlbench	gobmk	astar	h264ref
	ca4-2	perlbench	hmmer	h264ref	astar
	ca4-3	perlbench	astar	gobmk	hmmer
	ca4-4	h264ref	gobmk	hmmer	astar
bw	bw4-0	wrf	gcc	milc	libquantum
	bw4-1	libquantum	milc	gcc	xalancbmk
	bw4-2	libquantum	cactusADM	milc	zeusmp
	bw4-3	xalancbmk	libquantum	lbm	milc
	bw4-4	bwaves	gcc	lbm	xalancbmk
	bw4-5	xalancbmk	wrf	cactusADM	libquantum
	bw4-6	libquantum	gcc	xalancbmk	bwaves
	bw4-7	cactusADM	milc	zeusmp	gcc
	bw4-8	wrf	gcc	bwaves	cactusADM
	bw4-9	libquantum	gcc	milc	lbm
ca-bw	cabw4-0	omnetpp	sphinx3	soplex	bzip2
	cabw4-1	sphinx3	mcf	soplex	omnetpp
	cabw4-2	sphinx3	mcf	soplex	bzip2
	cabw4-3	bzip2	sphinx3	omnetpp	mcf
	cabw4-4	soplex	bzip2	omnetpp	mcf
co	co4-0	sjeng	calculix	namd	dealII
	co4-1	namd	GemsFDTD	sjeng	tonto
	co4-2	povray	gamess	calculix	namd
	co4-3	calculix	tonto	gromacs	povray
	co4-4	gromacs	namd	leslie3d	calculix
	co4-5	namd	dealII	sjeng	tonto
	co4-6	sjeng	povray	gromacs	dealII
	co4-7	leslie3d	gromacs	povray	dealII
	co4-8	GemsFDTD	dealII	calculix	povray
	co4-9	dealII	namd	calculix	gromacs
ra	ra4-0	libquantum	namd	perlbench	gcc
	ra4-1	calculix	gobmk	GemsFDTD	cactusADM
	ra4-2	astar	mcf	omnetpp	gcc
	ra4-3	lbm	povray	sphinx3	gamess
	ra4-4	calculix	dealII	lbm	perlbench
	ra4-5	libquantum	h264ref	GemsFDTD	gamess
	ra4-6	astar	sphinx3	lbm	libquantum
	ra4-7	sphinx3	soplex	povray	perlbench
	ra4-8	sphinx3	bwaves	gobmk	bzip2
	ra4-9	milc	calculix	astar	povray
	ra4-10	lbm	hmmer	calculix	gamess
	ra4-11	bwaves	soplex	GemsFDTD	milc
	ra4-12	mcf	hmmer	cactusADM	wrf
	ra4-13	sphinx3	leslie3d	namd	tonto
	ra4-14	sphinx3	libquantum	calculix	povray
	ra4-15	cactusADM	perlbench	povray	gromacs
	ra4-16	astar	libquantum	calculix	soplex
	ra4-17	tonto	povray	libquantum	cactusADM
	ra4-18	xalancbmk	mcf	cactusADM	sjeng
	ra4-19	h264ref	soplex	astar	libquantum

Table A.3: 4-core workloads