NTNU – Trondheim
Norwegian University of
Science and Technology

A Comparative Analysis of Shared Cache Management Techniques for Chip Multiprocessors

Christian Vik Grøvdal

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Magnus Jahre, 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 also investigate how different memory system design choices impact performance and throughput with the SPEC2006 benchmarks and the gem5 simulator. The student should implement at least one cache management technique and compare its performance to a conventional LRU-managed cache and a statically partitioned cache. Additional cache management techniques should be implemented and evaluated if time permits.

Abstract

In this thesis we present a comparative analysis of shared cache management techniques for chip multiprocessors. When sharing an unmanaged cache between multiple cores, destructive interference can reduce the performance of the system as the cores compete over limited cache space. This situation is made worse by stream-like applications that exhibit low locality of reference but has high cache demands. Several schemes for dynamically adjusting cache space available to each core has been suggested, and in this work we evaluate 3 such schemes as well as static partitioning and conventional LRU.

We deploy a well defined simulation methodology to analyze the performance of the cache management techniques. The gem5 simulator is used to simulate the ARM ISA, and the SPEC2006 benchmark suite is used to create multi-programmed workloads. The simulator has been extended to support cache management schemes and provide detailed simulation statistics. We implement UCP, PIPP, PriSM and static partitioning, and simulate dual core, quad core and 8 core workloads.

Our results show that destructive interference is a real issue in many workloads. Static partitioning can work well in scenarios where applications have similar cache demands, by creating private areas in the cache for each core. UCP improves on static partitioning by dynamically adjusting the size of each partition during runtime. PIPP performs decently by trying to maintain a specific cache occupation for each core without strictly enforcing a partition, but does not quite achieve the desired occupation and thus its performance suffers. PriSM fails to perform well, as its effort to determine a target cache allocation and maintain it does not work successfully for our workloads.

Sammendrag

I denne oppgaven presenterer vi en analyse av teknikker for å håndtere delt hurtigminne (cache) i flerkjerne prosessorer. Når man deler et hurtigminne mellom flere kjerner kan destruktiv interferens redusere ytelsen til systemet fordi flere prosesser konkurrerer om begrenset minneplass. Strømmende applikasjoner som har lav referanselokalitet men samtidig høyt hurtigminnebruk gjør dette problemet enda større. Det har blitt foreslått flere teknikker for å dynamisk justere hvor mye hurtigminne hver kjerne skal få, og i denne oppgaven har vi evaluert 3 slike teknikker opp mot konvensjonell LRU og statisk partisjonering.

Vi bruker en veldefinert simulasjonsmetodologi for å analysere ytelsen for hver av teknikkene. Vi bruker gem5 simulatoren til å simulere en ARM ISA, og SPEC2006 benchmark suite til å skape applikasjonsgrupper (workloads) som bruker flere kjerner. Simulatoren har blitt utvidet til å støtte håndteringsteknikker for hurtigminne, og presentere detaljert informasjon fra hver simulering. Vi implementerer UCP, PIPP, PriSM og statisk partisjonering, og simulerer 2, 4 og 8-kjerners arkitekturer med hver av disse teknikkene.

Våre resultater viser at destruktiv interferens er et reelt problem for mange applikasjonsgrupper. Statisk partisjonering kan fungere bra i tilfeller hvor applikasjonene har like store krav til hurtigminne, ved å skape private områder i hurtigminnet for hver kjerne. UCP forbedrer ytelsen til statisk partisjonering ved å dynamisk justere størrelsen til hver partisjon under kjøring. PIPP får grei ytelse, ved å prøve å beholde en gitt hurtigminnefordeling uten å strengt partisjonere hurtigminnet mellom hver kjerne, men klarer ikke helt å nå den riktige fordelingen og taper dermed litt ytelse. PriSM har dårlig ytelse, i stor grad fordi dens forsøk på å beregne en optimal hurtigminnefordeling og opprettholde denne ikke fungerer for våre applikasjonsgrupper.

Acknowledgements

I would personally like to thank my advisor Magnus Jahre for excellent help and guidance with this thesis[11] .

I would also like thank the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) for their help, and for providing resources used in this work.

Finally I would like to thank The Norwegian Metacenter for Computational Science (NOTUR) for providing the computational resources used, granting access to the supercomputer Stallo and 150,000 CPU hours [2]. The work has been performed under the project number NN4650K .

Christian Vik Grøvdal
June 2013

Contents

Nomenclature

Application a single threaded program running on a single core. The word application is used interchangeably with program and benchmark, describing the code running on the core.

ATD Auxillary Tag Directory. An extra tag store for each core in each set, which contains the tags of the data that would have been in the cache if the core had the entire cache to itself.

Benchmark Synonym for application, but specifically refering to applications from the SPEC2006 benchmark suite.

CMP Chip Multiprocessor. A single-chip processor with multiple processing cores, capable of simulataneous execution of several threads or processes.

Core A single processing unit in a CMP, capable of running a single thread or process at a time.

CPU A single processing core in a CMP, equivalent to a core.

DSS Dynamic Set Sampling. A method to reduce storage overhead in Shadow Tag Stores (STS), by placing ATDs in a subset of the sets. This approximates cache usage by assuming some uniformity of the cache accesses across sets.

LRU Least Recently Used. Refers to either A) A cache replacement policy, B) The least recently used block, equivalent to the lowest position on the stack.

MRU Most Recently Used, refers to the last used block in a conventional LRU cache, or the highest priority position block in PIPP.

PIPP Promotion/Insertion Pseudo-Partitioning. A shared cache management scheme used in this work.

PriSM Probabilistic Shared Cache Management. A shared cache management scheme used in this work.

STS Shadow Tag Store. A monitoring component used to gather information about each cores use of a shared cache. Contains Auxillary Tag Directories and recency counters.

UCP Utility-based Cache Partitioning. A cache partitioning scheme used in this work.

UMON Utility Monitor. Equivallent to Shadow Tag Store (STS).

Workload A set of benchmarks, equivalent in size to the number of cores on the CMP. A workload defines what is run on each core.

Chapter 1

Introduction

This work aims to shed some light on the strength and weaknesses of proposed cache management techniques for chip multiprocessors. In particular we look at frequently cited cache partitioning schemes that claim to improve performance over the common LRU cache. In this Chapter we present the motivation behind this work, discussing the Chip Multiprocessor and the memory system. We introduce our research questions and list the contributions of this work, and outline the rest of the thesis.

1.1 Chip Multiprocessors (CMPs)

Chip Multiprocessors (CMPs) has become the norm for modern computing, leaving behind the single core era of the early 2000's and before. Up to the mid 2000's the improvement in performance mostly came as an effect of increased clock speeds made possible with shrinking transistor sizes. Eventually increasing clock speed further caused significant problems with heat and energy consumption, and hardware designers met the power wall, preventing further improvements using this technique. But as just as continued improvements following Moore's Law looked less likely, the focus shifted to adding multiple cores per processor [4]. This kept the aggregated performance and transistor counts increasing at a rate similar to what Moore's Law predicts. In 2013, quad cores are common (like Intels i7 series [6]), and the core count appears to be increasing. The introduction of multicore computing led to many new challenges in hardware architecture, amongst them how to perform cache management.

By CMP, we mean a single chip with several processing cores on it (Figure 1.1). It is also commonly known as a multicore processor, although CMP is a more precise term, indicating that the whole processor is located on a single chip. Having several processing cores allows it to run multiple programs or threads concurrently,

increasing the amount of work that can be done per unit of time. A single process can be separated into several threads to provide simultaneous execution, or independent processes can share the cores between them.

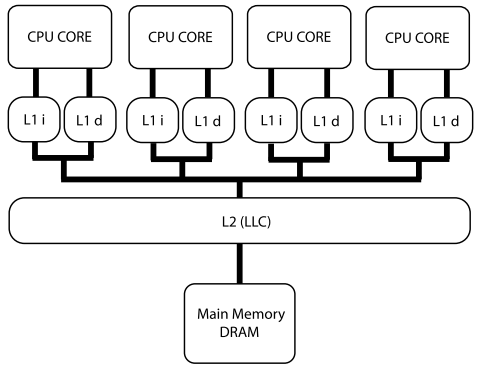

Figure 1.1: An illustration of a Chip Multiprocessor with private and shared caches.

1.2 CMP Memory Systems

Caches are vital to todays high performance in CPUs. Processing power has increased at a much higher rate than memory speeds, and this has led to a performance difference called the memory gap [12]. Figure 1.2 shows the historic difference between processor and memory performance. The memory gap has been the cause of much research for a long time, as many techniques have been tried to help bridge the difference. One of the most important techniques to mitigate the memory gap is the memory hierarchy. Figure 1.3 shows the basic form of a memory hierarchy, where smaller and faster memories are placed closer to the CPU while larger and slower memories are used further down. The higher up the chain the data can be found, the lower the access time will be. After registers, caches are the fastest type of memory available, and deciding what data to place in the cache is crucial to the systems total performance.

When multiple applications in a CMP try to use the same cache resource simultaneously, they can have adverse effects on each others performance. Certain applications can take up large amounts of space in the cache without using it efficiently, whereas others may only require a few kilobytes but can have frequent accesses to it. The typical replacement policy is Least Recently Used (LRU), but this policy provides no isolation between applications. One application can cause the eviction of another applications cache lines, reducing the reuse of data stored in the cache. This is called destructive interference, and is the principal motivator behind the push for more utilization-aware shared caches.

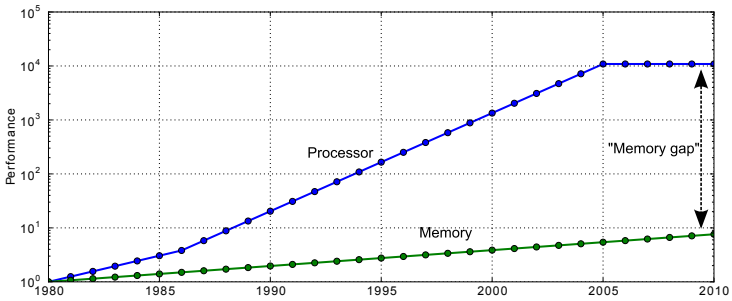

Figure 1.2: The memory gap. Graph based on data from [12].

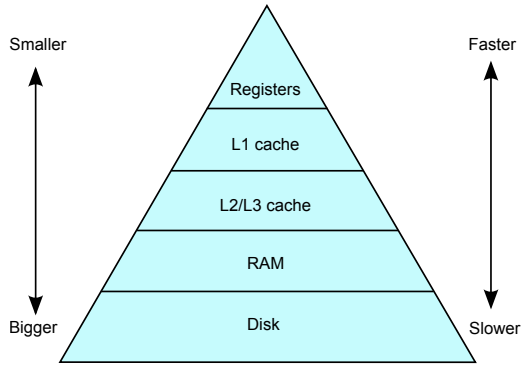

Figure 1.3: The memory hierarchy. A general memory hierarchy is shown, many more levels can be identified depending on the architecture.

In this work we will focus on the efforts of optimizing performance for shared caches in a CMP, in particular by preventing destructive interference between applications. We will look at selected cache management schemes for CMPs, that propose various forms of cache partitioning. Some schemes use strict isolation like UCP [18] or static partitioning, while others use probability distributions (PriSM [16]) or promotion/insertion strategies (PIPP [23]) to achieve their goals. These different strategies have varying strengths and weaknesses, improving or reducing performance depending on the workload. All the schemes except static partitioning depend on extra monitoring circuitry, that monitors the cache utilization and gives input to partitioning algorithms. We will also briefly evaluate the overhead of these circuits, making sure that a hardware implementation of the suggested scheme is feasible.

1.3 Research questions

The main research question that motivates this work is:

How can performance be improved when sharing a cache between multiple applications in a CMP?

This is a very broad question, spanning several aspects of computer architecture and design. In our work we focus on cache management schemes and how they impact the performance of a system. We are interested in evaluating the strengths and weaknesses of these schemes, and to see under what conditions they perform well or poorly. We will use simulation results to draw conclusions regarding each schemes performance and evaluate its usefulness. In particular, we are trying to answer the following questions:

- How much does LRU performance degrade when using multicore workloads, due to interference between the cores?
- What performance do the proposed schemes have in comparison to LRU and static partitioning?
- What are the limitations of each cache partitioning scheme, and how much impact does this have on its usefulness?
- What limitations are there in the simulation methodology, and can these skew the results in favor of any of the schemes?

1.4 Contributions

The main contributions of this work are:

- A comparison of several cache management schemes performance
- Several case studies to provide deeper understanding of each schemes strengths and weaknesses
- Multiple extensions to the gem5 simulator that enable shared cache management schemes and auxiliary functions
- A framework to perform multicore experiments on a distributed supercomputer

This thesis tries to take an impartial look at shared cache management schemes. Authors proposing new cache management schemes do include their own performance analysis, but these may not be directly comparable to other works. Usually the methodology differs significantly between each analysis, making comparisons unfair or impossible. Therefore it is useful to look at this topic in a unified sense, with the same methodology across the testing of all cache management schemes.

This thesis draws conclusions regarding the strengths and weaknesses of the individual cache schemes. We analyze outlier cases where performance is particularly good or poor for some of the schemes, to shed light on what makes the schemes respond differently. This supports the arguments presented about the cause of a schemes performance, and deepens the understanding of how each scheme works.

In addition, a framework for running multicore experiments using the gem5 simulator has been developed. The gem5 simulator has been extended to support CPU-aware cache accesses, a prerequisite for all shared cache management techniques. The simulator has been extended with implementations of 4 cache partitioning schemes, and a general purpose Shadow Tag Store implementation. Additions to the simulator have been made to dump statistics multiple times during a single simulation.

We have developed tools for merging checkpoints from multiple applications to a combined checkpoint, allowing precise resuming of workloads with arbitrary combinations of applications. A framework for managing jobs on a supercomputer and triggering simulations have been created, based on previous work done at IDI, NTNU. Finally, multiple tools for aggregating data from a large number of experiments been developed, allowing rapid analysis of a massive amount of data. The framework and tools will be transferred to IDI, to serve as a starting point for future work.

1.5 Outline

The rest of this work is organized as follows: In Chapter 2 we present background information on caches, and some proposed cache management techniques. In Chapter 3 we talk about modeling the CMP and the details of our simulated architecture. Chapter 4 contains methodology, explaining our work with benchmarks and workloads, the simulation methodology, and how the results are produced. In Chapter 5 the results are presented, organized by number of cores. We also perform case studies of some workloads to show how the different cache schemes perform. A discussion of the results and our methods then follow in Chapter 6. We round off this work with a conclusion in Chapter 7, including a brief look at future work. Extra details about workloads and results are included in Appendix A and B.

Chapter 2

Background

In this chapter we will present some background information about caches and cache management schemes. We introduce a conventional LRU cache, and how this is used in an unmanaged way in shared caches. We then present methods of managing shared caches, using a variety of methods. Static partitioning, UCP, PIPP and PriSM are introduced as schemes we will perform experiments on, while Vantage is presented as a different scheme that we unfortunately did not include in our work.

2.1 Caches

This section describes a traditional cache structure, similar to an LRU cache. There are other cache structures that function differently, but they are less common and will be described where necessary.

Caches are used to store recently used data closer to the CPU so the data can be accessed faster than if they were located in main memory [12, 13]. To keep cache accesses fast, the caches need to be small. Larger caches have higher access latency, so the fastest caches have low capacity and are located close to the CPU. Although larger caches are slower, they have a greater probability of containing the data we are interested in. For this reason, caches are organized in hierarchies, named by their level, usually L1, L2 and sometimes L3. L1 is the first level cache, the fastest and closest to the CPU. If a piece of data is not found in the L1, the request goes to the L2 cache which is larger. Some systems also have another level of cache, the L3, which is even larger and slower. The Last Level Cache is referred to as LLC.

The mapping from an address to a cache location is illustrated in Figure 2.1. The storage available in a cache is split into a number of sets. Data with a given address can only reside in a single set, thereby limiting the number of places that needs to

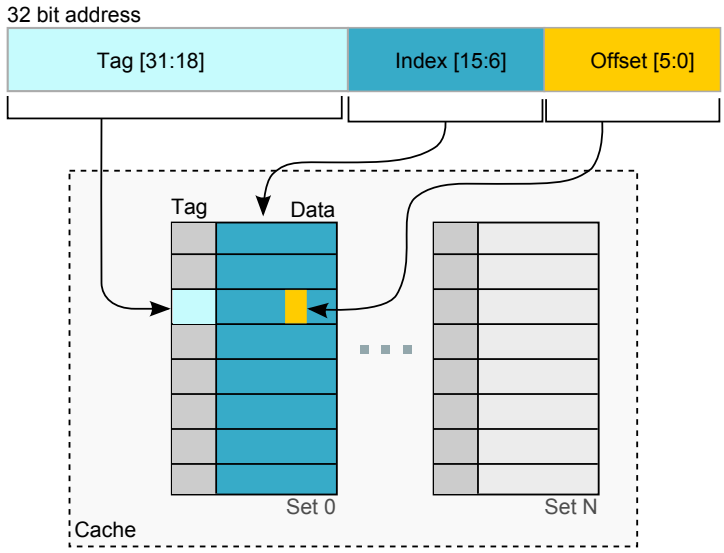

Figure 2.1: Address mapping in caches. This cache has a block size of 64 bytes, 1024 sets and 8 way associativity.

be searched. The set is determined by the index bits of the address, so the number of index bits depend on the number of sets. Within a set, there are several cache ways, each of which contains one cache block of data. The number of cache ways per set is called the associativity of the cache, and tell us how many possible location a piece of data can be in. Each of these ways have to be searched for the correct tag, to determine if the data is present. A higher associativity allows us more flexibility of where to place data and what data to keep in the cache, but increases the access time since more searching is required. Finally, after determining the set and the way, the correct data is extracted from the cache block based on the offset bits in the address.

For caches with associativity larger than 1, in other words where we have more than one possible location for a block of data to be inserted, we require a replacement policy. This policy determines what block should be evicted to make room for the new data being requested if the set is already full. The most common policy is Least Recently Used (LRU). We visualize LRU as a stack, where the least recently used block is on the bottom, and the most recently used is on the top. In practice it is implemented using counters. The LRU policy always evicts the block in a set that is least recently used. When a block is accessed, it is moved to the top of the stack, moving the other blocks one step closer to the least recently used position. Other policies include Least Frequently Used (LFU) which takes the block with the lowest use frequency as the victim, and Pseudo-LRU which is cheaper to implement than true LRU but does not always select the least recently used as the eviction candidate.

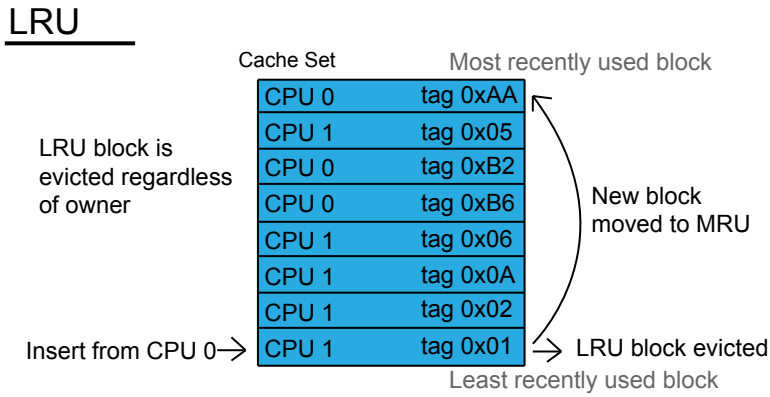

Figure 2.2: An unmanaged shared cache, with LRU as the eviction policy.

2.2 Cache Management Techniques

A vast number of cache management techniques for shared caches have been proposed. In this section we will provide a review of some recently proposed cache management techniques that are relevant for this work. Static partitioning, UCP, PIPP and PriSM are the management schemes evaluated later in this work. Vantage is introduced as additional background information.

2.2.1 Unmanaged caches

The simplest management technique for shared caches is not to manage the cache at all. All cores can use the shared cache freely, just as it was their own cache. Cache accesses from the cores are serialized, and there is not control over how much of the cache each core can occupy. The core with the highest cache demand will occupy the largest portion of the cache, regardless of its ability to reuse any of the data. Figure 2.2 shows an unmanaged cache using LRU as the replacement policy.

The major problem with this approach lies in interference between the cores. When a core has a private cache, it can be fairly certain that a block will remain in the cache if the working set is small enough so it never gets evicted. In an unmanaged shared cache a core does not know how the other cores are using the cache. If their working sets are large, they could cause evictions of the first cores blocks. This would be detrimental to the first cores performance, and there is nothing it can do to limit this destructive interference.

However, there are benefits to unmanaged caches. They are significantly easier to implement since they require very few modifications from a standard private cache. There are also no extra overhead involved in cache management, neither in form of data gathering or enforcement methods to prevent interference. A single core can

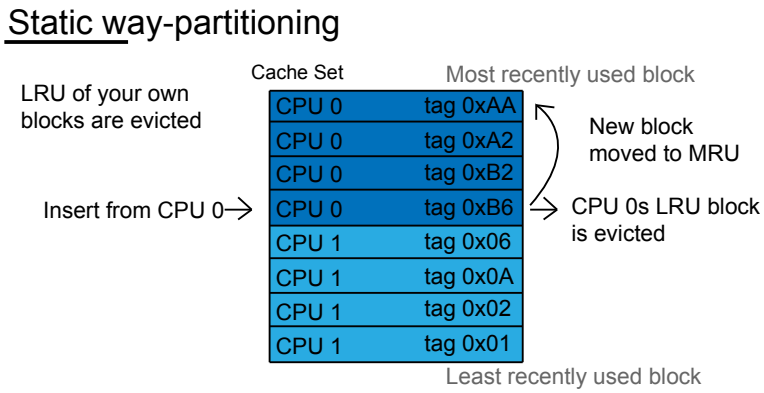

Figure 2.3: Static way-based partitioning.

also utilize the entire cache, without being limited by an allocation policy. All in all it is a more open approach, with fewer guarantees but more opportunities.

Whether it is beneficial to keep caches unmanaged is completely dependent on the workloads running on the system. As we will see later, certain types of workloads like those with rapidly changing characteristics or small working sets will achieve good performance with unmanaged caches, while applications with uneven cache needs may benefit from having stricter guarantees about the cache.

2.2.2 Managed caches

Most CMP cache management schemes utilize cache partitioning in one way or another. Cache partitioning allocates areas of the cache to a specific process or core. The partitioning policies and implementations that decide the allocation and enforces it vary between the schemes suggested. Allocating areas to cores limits interference between the cores, and attempts to optimize the use of the available space.

One way of partitioning a cache is way-based partitioning, where each core is allocated a specific number of ways. The number of sets available to a core is then constant, but the associativity varies. The simplest way of doing way-based partitioning is by statically partitioning the cache as shown in Figure 2.3. A fixed number of ways is allocated to each core for the entire execution. This effectively creates a smaller private cache available for each core, but has no flexibility in rearranging the allocation based on utilization.

One can do better by changing the allocation of the ways during the execution. The cost of doing way-based partition individually for each set is usually prohibitive, so the common solution is to partition the ways globally. In other words, a core has the same number of ways available in all sets. The way that is allocated to a core

cannot be used by any other core. This is considered a coarse grained approach to cache partitioning. Since there are only small number of ways, the number of possible allocations is also low. And crucially, the number of cores must not exceed the number of ways, or else the cache will be unavailable to one or more of the cores. UCP [18] utilizes way-based partitioning.

To provide a finer grained partitioning, new techniques have been proposed that do not enforce a strict partitioning of the cache. The entirety of the cache is available to all cores, but the cache management scheme decides which fraction of the cache should be occupied by each core. It then tries to keep the occupancy close to the target allocation by using enforcement policies. PriSM [16] does this by choosing what block will be evicted based on an eviction probability for each core. By adjusting this eviction probability, the cache occupancy can be controlled without strictly enforcing an allocation. Similarly, PIPP[23] inserts blocks at lower priorities in the cache than the head, and uses a different promotion strategy to achieve the same goal.

Vantage [21] has been proposed as a shared cache management scheme, that uses a different sort of cache than the usual set/way based ones. ZCaches [20] is a cache design where extra associativity is achieved by increasing the potential number of eviction candidates past that which is possible with a standard type cache. It uses hashing to find potential candidates for eviction, and can chain a lookup to extend the associativity further. This makes Vantage a fine grained partitioning schemes that can provide strict partitioning of the cache.

2.2.3 Shadow Tag Store

To decide on the target cache allocation, the management scheme needs information on how each core is utilizing the cache. By obtaining usage information from the cache, the allocation can be adjusted to meet specific performance criteria, such as minimizing the number of misses or maintaining fairness. The most common method of tracking cache usage is with a Shadow Tag Store [8].

The Shadow Tag Store (STS)[1] is an extra monitoring circuit to maintain tag information and hit counters for shared caches. It is used in addition to the tags that identify the data in each set. Using the STS gives us useful information about the usage of the cache for each of the applications, by determining the number of hits in each recency position. There are two components to the STS; the Auxiliary Tag Directories (ATDs) and the recency hit counters. Depending on the desired accuracy of the cache monitoring, there may be one ATD and one set of recency hit counters for each set in the cache.

[1] In UCP [18] terminology, the STS is called Utility Monitor (UMON). To stay consistent with other work, we refer to it as a Shadow Tag Store. We refer to a single tag store in one of the sets as an Auxiliary Tag Directory (ATD).

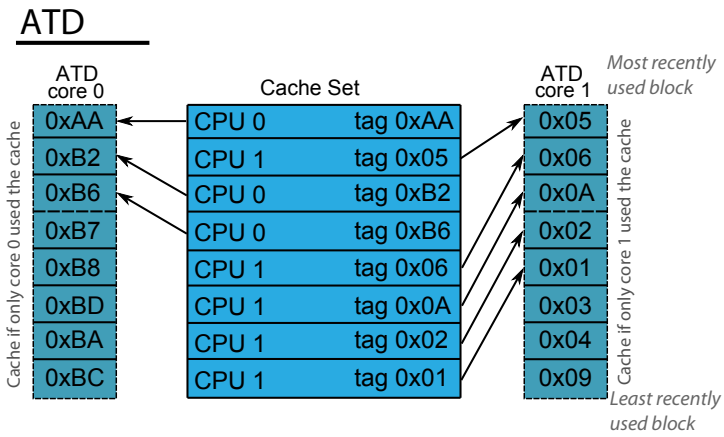

Figure 2.4: Two ATDs in a dual core system. Each ATD shows what the cache contents would have been if the core had the cache to itself.

2.2.3.1 Auxiliary Tag Directories

The Auxiliary Tag Directory (ATD) is a LRU-managed list of the most recently accessed tags. It operates exactly as a private cache set, except it contains no data, only tags. When a new tag is used, the least recently used tag is removed and the new tag is inserted at the head of the stack. There is one ATD per core for each set of the cache. This is in contrast to the tag store of the data, of which there is only one per set. Having an ATD per core means we can determine what the contents of the cache would have been if the core had the entire cache by itself. Figure 2.4 shows a cache set and the corresponding ATD contents.

2.2.3.2 Recency hit counters

A cache blocks recency position is its position in the LRU stack when a cache access causes a hit. A hit for the block in the most recently used position in the cache is a hit in the first recency position. And a hit for the block in the least recently used position is a hit in the last recency position. Consider a cache with only 1 set that is being accessed with the addresses 0xA, 0xB, 0xC, 0xA, 0xB, 0xC, 0xA... and so on, as shown in Figure 2.5. The first access to 0xA will insert it into the cache at the MRU position. The next access to 0xB will insert this to the MRU, pushing 0xA down to the second recency position. Inserting 0xC will bring 0xA down to the third recency position. The next access to 0xA will cause a hit, as we find 0xA in the third recency position.

A hit for 0xA causes the hit counter for the third recency position to be incremented. This tells us that this application requires 3 ways of cache to achieve a hit on this access. Notice how if this cache set only had 2 ways, 0xA would have

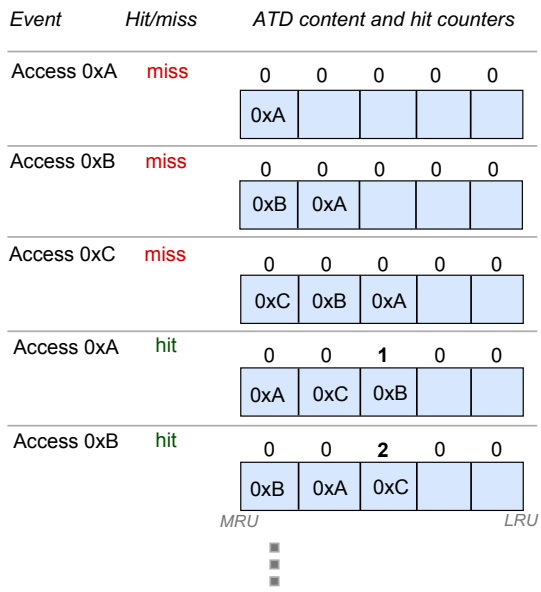

Figure 2.5: The content of the Shadow Tag Store and the hit counters. Notice that the hit counter is tied to the recency position, not the tag.

been evicted before it was accessed. The next access, 0xB, also hits in the third recency position, incrementing the counter to 2. In fact, this cyclic access pattern of 0xA, 0xB, 0xC, 0xA... will only cause hits in the third recency position. From this knowledge, the STS will know that there will be 0 hits unless this core gets at least 3 ways available to it.

2.2.3.3 Dynamic Set Sampling (DSS)

Unfortunately, the storage overhead of keeping an ATD for each core per set makes an implementation of the naive Shadow Tag Store unfeasible. It would require significant amounts of data to store the tag information for so many ATDs. Doing this naively for a 16-way associative cache with a tag size of 40 bits and 16 bits of counter values would require $(40+16)*16 = 896$ bits for each core in a set. Using a common 64 bytes per line this set uses $(40+(64*8))*16 = 8232$ bits for its data and tags. Each core would then add 11% on the storage requirements. For an 8 core system, half the storage necessary would be for ATDs, which is not feasible.

To avoid this issue, Dynamic Set Sampling (DSS) [17] is used. DSS is used to approximate cache utilization by sampling only a few of the sets in the cache. This reduces the storage overhead significantly, by allowing us to have ATDs in only a few of the cache sets. The accuracy of the estimate depends on the number of sets sampled and the uniformity of the accesses across sets. If each application accesses

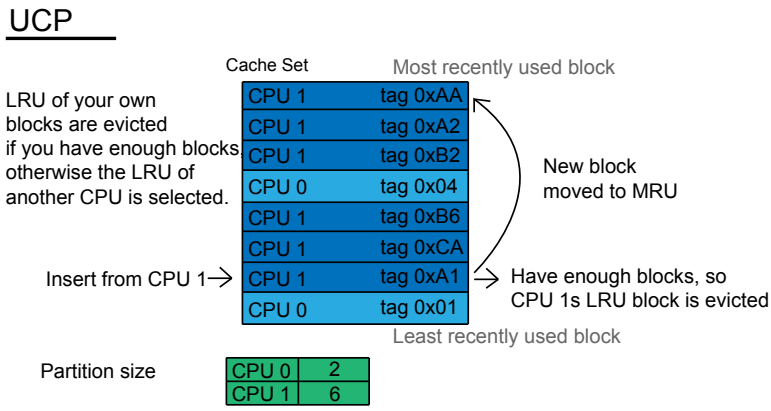

Figure 2.6: Utility-based Cache Partitioning.

each set in the same pattern, a single set would be sufficient to determine the total utilization. But in practice, access patterns vary between sets, especially as the number of sets increase. UCP [18] samples 32 sets total. These are chosen by a simple static pattern, using an ATD on every $N/32$ cache set.

2.2.4 UCP: Utility based cache partitioning

Utility Based Cache Partitioning[18] was proposed in 2006 by Moinuddin K. Qureshi and Yale N. Patt from the University of Texas at Austin. The original paper proposed solutions that have been used by later cache partitioning schemes, including the Utility Monitor (UMON) which is equivalent to the Shadow Tag Store [8].

The overall idea behind UCP is to partition the cache by ways. The number of ways allocated to each core depends on its cache utilization. The UMON circuits are used to monitor the utilization, and a partitioning algorithm allocates a number of ways to each core to minimize the number of cache misses. A way belongs exclusively to a core and cannot be evicted by any other core. This prevents interference between the cores that could lower the hit rate of the cache. The partitioning is illustrated in Figure 2.6.

The Shadow Tag Store tells the partitioning algorithm how many hits an application would have if it had N number of ways allocated to it, as illustrated in Figure 2.7. As the number of ways available is reduced, the number of misses increase. With multiple applications, there exists an optimal partitioning of ways between the applications, which minimizes the number of misses. This is equivalent to finding the partitioning that provides the maximum number of hits. Unfortunately, finding the optimal solution turns out to be NP-Hard [19]. The possible partitions increase with both the number of cores and number of ways, making it an exponential problem to evaluate them all. This exhaustive search, called EvalAll, is

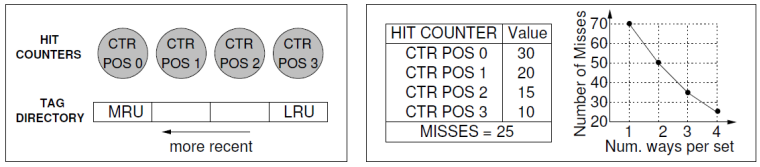

Figure 2.7: Shadow Tag Store and hit counters. Any reduction in allocation will increase the number of misses according to the table. Illustration from [18].

Algoritme 2.1 Greedy Partitioning Algorithm, from [18]

```
balance = N # Num blocks to be allocated
allocations[i] = 0 for each competing application i

while(balance):
    for i,a in enumerate(application): # get utility for next 1 block
        alloc = allocations[i]
        Unext[i] = get_util_value(i, alloc, alloc+1)
    winner = application with maximum value of Unext
    allocations[winner] += 1
    balance -= 1
return allocations

def get_util_value(p, a, b):
    U = change in misses for application p when the number
    of blocks assigned to it increases from a to b
    return U
```

unfeasible for anything but dual-core architectures, where it reduces to a trivial search.

The authors of UCP suggest two simpler partition algorithms that do no guarantee an optimal solution but have much better performance than EvalAll. A Greedy Algorithm (Algorithm 2.1) can be shown to be optimal if the utility curves of all UMONs are convex, in other words the number of hits for each cache way is decreasing for each step. The utility graph in Figure 2.7 is convex. If the utility curve is non-convex, the greedy algorithm will not give the correct partitioning. It will not consider the benefit of allocating one low-gain way in order to be able to allocate a high-gain way next, thereby not finding the optimal solution.

The second algorithm, called the Lookahead Algorithm (Algorithm 2.2), tries to improve on the Greedy Algorithm without adding too much computational complexity. It uses the notion of marginal utility (MU), defined as the difference in misses when it receives a and b ways, divided by the distance between a and b. This lets it look further ahead than the greedy algorithm. In our comparisons, the Lookahead Algorithm is used as the partitioning algorithm for UCP.

For each step of the Lookahead Algorithm, a number of ways are allocated to the application with the highest marginal utility. If the utility graph is convex we will

Algoritme 2.2 Lookahead Algorithm, from [18]

```
balance = N /* Num blocks to be allocated */
allocations[i] = 0 for each competing application i

while(balance):
        foreach application i: /* get max marginal utility */
                alloc = allocations[i]
                max_mu[i] = get_max_mu(i, alloc, balance)
                blocks_req[i] = min blocks to get max_mu[i] for i
        winner = application with maximum value of max_mu
        allocations[winner] += blocks_req[winner]
        balance -= blocks_req[winner]
return allocations

def get_max_mu(p, alloc, balance):
        max_mu = 0
        for(ii=1; ii <= balance; ii++):
                mu = get_mu_value(p, alloc, alloc+ii)
                if( mu > max_mu ) max_mu = mu
        return max_mu

def get_mu_value(p, a, b):
        U = change in misses for application p when the number
        of blocks assigned to it increases from a to b
        return U/(b-a)
```

allocate 1 way at a time, reducing the lookahead algorithm to the greedy algorithm. Calculating the highest marginal utility can be done in parallel for each core, with complexity N each. In the worst case the main allocation part allocates only 1 way at a time, which gives a total time complexity of

$N + (N-1) + (N-2)... + 1 = N(N-1)/2 \approx N^2/2.$

When evicting blocks from the cache, a check is made to see if this core fills its target allocation. If it is equal to or over its allocation, its least recently used block is selected for eviction. If it is below its target allocation, the least recently used block from one of the cores that exceed their quota is selected instead. The new block that is being inserted gets moved to MRU. This method prevents interference and makes the occupation always approach the allocation.

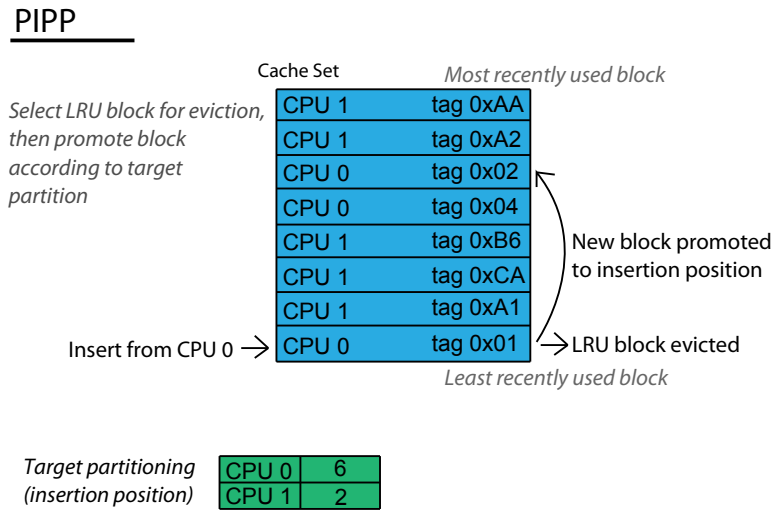

Figure 2.8: Promotion/Insertion Pseudo-Partitioning.

2.2.5 PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches

PIPP[23] was proposed in 2009 by Yuejian Xie and Gabriel H. Loh from Georgia Institute of Technology. In contrast to UCP that uses strict partitioning between the cores, PIPP implements an implicit partitioning by regulating where cache blocks are inserted in the LRU stack. Figure 2.8 shows how PIPP works for replacements.

When evicting blocks, PIPP works similar to LRU and evicts the least recently used block. However, the new block that is being inserted is not moved to the most recently used position in the LRU stack. Instead it inserts it according to the partition algorithm, as shown in Figure 2.9. The blocks of an application that is being allocated more of the cache will be inserted closer to the MRU. This means that it is less likely to be evicted soon, and thus increases this cores occupation of the cache. An application who gets a smaller allocation will have its blocks inserted closer to the LRU, and thus more likely to be evicted earlier. This leads to a pseudo-partitioning of the cache, where the occupancy of the cache regulates itself based on the insertions.

As the number of cores increase, the average partition size decreases. This causes the insertion positions to become closer to the least recently used position in the cache. Inserting close to LRU will in turn lead to quicker evictions, and is a known drawback with PIPP. Unless a new block is accessed very soon after insertion, it is highly likely that it will be quickly evicted.

On a cache hit, the block is promoted up the stack step-wise, not straight to the MRU position. At its basic form, this promotion is simply a shift 1 step towards

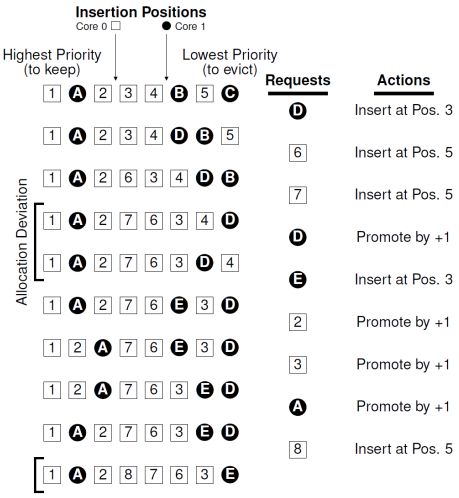

Figure 2.9: Insertion and promotion policy for PIPP. Figure from [23].

MRU. In addition to this, the block is only promoted with a specific probability p_{prom}. The probability can be tuned to adjust the rate of which blocks climb the stack, helping to maintain the target occupation.

PIPP relies on a Shadow Tag Store to determine the target partition sizes. The STS keeps track of each cores hit counters if it was allocated more or fewer ways. This information is then used as input to the partition algorithm to regulate insertions and promotions. The insertion position is set equivalent to the cache allocation. If a core is allocated 1 way, its blocks are inserted at the LRU position, and similarly if a core is allocated 5 ways, its blocks are inserted 5 steps above LRU.

PIPP also suggests a method for dealing with applications that exhibit low locality of reference, specifically those who have a stream-like behavior. If PIPP detects that an application would have more than m_i misses even with the whole cache available, it assumes the application is stream-like. This changes its insertion position for new blocks to π_{stream}, independent of its target partitioning. This π_{stream} is equal to the number of streaming applications, effectively inserting it very close to the LRU of the cache. In addition, promotion probability is reduced to p_{stream} which is significantly lower than p_{prom}. This combination tries to reduce the interference from streaming applications on applications that can utilize the cache better, by making it more likely that the streaming applications blocks are quickly evicted.

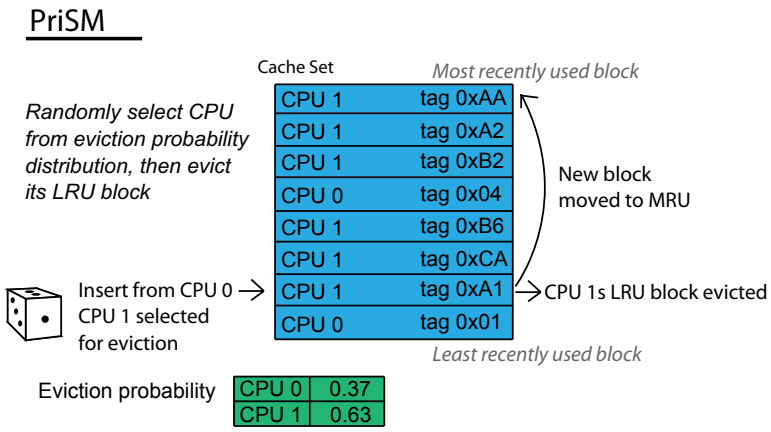

Figure 2.10: Probabilistic Shared Cache Management.

2.2.6 PriSM: Probabilistic Shared Cache Management

The article detailing PriSM[16] was published in 2012, by R. Manikantan, Kaushik Rajan and R. Govindarajan of the Indian Institute of Science, Bangalore, in cooperation with Microsoft Research India. It utilizes probabilities to maintain a cache occupation determined by a partitioning algorithm.

PriSM uses eviction probabilities to control the cache occupancy in each set. Each core has an eviction probability E_i such that $\Sigma E_i = 1$. When a miss occurs, this probability distribution is used to choose the victim block. First, a core is randomly selected according to the eviction distribution. A core may have an eviction probability of 0, which means it will never be selected for eviction unless all other options are out. After a core is selected, the LRU block of this core is chosen as the victim. If this core does not have any blocks in this set, the LRU block of any non-zero eviction probability core is selected. There is no change in hit policy from LRU, a hit causes the block to be moved to the most recently used position. The basics of PriSM is illustrated in Figure 2.10.

Adjusting the eviction probabilities is the job of the allocation policy. The authors of PriSM suggest three different policies depending on the desired functionality of the cache. The Hit Maximization policy does what the name suggests, it attempts to maximize the overall speed of the system by getting the highest number of cache hits. It will allocate more cache space to those cores than can best utilize it. PriSM uses a Shadow Tag Store without the recency hit counters to evaluate the cache utilization of the different cores. Instead of the recency counters it only stores the total hits per core in the ATD, a more coarse grained approach than PIPP and UCP uses.

The Fairness policy tries to equalize the slowdown between the cores when sharing a cache. If a core is slowed down more than the rest, this creates an unfairness even

though the total performance of the system might be better. To use this policy, PriSM requires access to the performance counters for CPI, instructions committed and cycles, in addition to the standard shadow tag information.

The final policy is Quality of Service, which tries to ensure a specified minimum level of performance for a given core. The authors use maximum slowdown in IPC as their target. This can be useful in situations with prioritized applications, for example applications that have specific timing needs.

PriSM changes the eviction probabilities at specific intervals, given in number of cache misses. Once the threshold has been passed, the allocation policy determines new eviction probabilities that will be used until the threshold is reached again.

2.2.7 Vantage

Vantage[21] was proposed by Daniel Sanchez and Christos Kozyrakis from the Electrical Engineering Department at Stanford University in 2011. It breaks with the previous cache schemes we have seen, by using a different cache structure and allowing for a more fine grained partitioning and still having strict isolation between the cores.

Unlike LRU, UCP, PIPP and PriSM, Vantage uses a highly-associative cache type called ZCache [20]. It differs from ordinary caches in how it is organized. In an ordinary cache, the index bits of an address uniquely identifies the set, and then the associativity of that set determines the possible block candidates. ZCaches uses multiple hashing functions on the address to determine its possible location in the cache. On an insertion, the new block is inserted to the position given by one of the hash functions. If the block selected for eviction should not be evicted, we can hash its address, and move it to another location where we find a new eviction candidate. This process can be repeated until a desired block is found, which is then finally evicted. To keep the search finite, there is a depth limitation. The total number of potential eviction candidates is then

$$R = \sum_{d=0}^{D-1} (W-1)^d$$

where W is the number of hash functions and D is the max depth.

Vantage separates the cache into two parts, a managed region and a smaller unmanaged region that is about 15 % of the total cache size. A target allocation is given to each of the cores. The cores are then allowed to slightly outgrow their allocations by borrowing size from the unmanaged region. On average the partition should stay at its target allocation, by having an equal number of insertions and evictions. New blocks are inserted into the managed region, then demoted to the unmanaged region, and then either evicted from the cache or promoted back into the managed region if it gets a hit.

The evictions, demotions and promotions are controlled by an eviction priority. Each block has an eviction priority, which changes while the cache is being used. This priority can be any ordering of the cache blocks, an LRU stack is a good example of such an ordering. In the LRU stack, the MRU has the lowest eviction priority while the LRU has the highest.

To keep the size of the managed and unmanaged regions under control, the number of promotions and demotions need to be equivalent on average. This is done by setting an eviction threshold called an Aperture on the managed region, causing every block in the top Aperture percent of eviction priorities to be demoted to the unmanaged region. The aperture is set to $A = 1/(R \cdot m)$, where R is the number of replacement candidates and m is the fraction of the cache desired as the managed region.

The partition size for each core is given by an allocation policy, typically UCP or a software policy. The allocation policy can have different performance targets, the common being hit maximization, but it can also try to optimize for other metrics such as fairness or Quality of Service.

Vantage and ZCaches are very different cache schemes than the others we have looked at. A deeper look at these is outside the scope of this work, in particular since we did not implement Vantage for our experiments.

Chapter 3

Modeling a CMP

3.1 ISA and multicore architecture

We simulate a CMP with an architecture similar to what you would find being produced today, with 2 levels of cache and 2 GHz core clock speed. We use the ARM ISA because it is well supported in our simulator, and is an important ISA that is experiencing significant growth of use [15].

We chose a two level cache organization for two reasons. First, we want to exercise the shared cache as much as possible, thus we want few other caches between the CPU and the shared LLC. Modern Intel processors like the i7 have 3 levels of cache, where L1 and L2 are private and L3 is shared between the cores[6]. We do not use 3 levels of cache because it would require even more simulation time to reach a sufficient number of accesses to the LLC. A cache that is being underutilized by not being accessed enough provides very few insights into the performance of a cache scheme. Simulation time is valuable, and instead of simulating longer periods we can use a 2 level structure. 2 cache levels are also common for ARM based chips.

Second, the 2 level cache is in line with what the proposed cache schemes use in their evaluation [23, 16, 18]. Hence it is natural for us to maintain this, so we do not stray too far away from their methodology.

The architecture used is specified in Table 3.1. This is the baseline on which most of the experiments in the next section is based upon. We are interested in the performance benefits gained from using different schemes in the shared L2 cache, so the other parameters are kept the same as much as possible.

CPU Cores			2/4/8 (ARM)	
Core Clock			2 GHz	
Cache levels			2	
L1i	32 kB	8 way	6 MSHR	LRU
L1d	32 kB	8 way	6 MSHR	LRU
L2	1/2/4/8 MB	32 way	16 MSHR	[Various schemes]
Main memory	2/4/8 GB	-	64 rd/wr queue slots	-

Table 3.1: A typical architecture used. The L2 (LLC) is used with different cache schemes to evaluate performance.

Cache Size	8 MB
Line Size	64 bytes
Associativity	16
Number of banks	4
Technology node	32 nm
Access time	3.5 ns
Response time	0.5 ns

Table 3.2: The cache parameters and corresponding access and response times.

3.2 Cache and cache latency

In order to have as realistic simulations, we are using cache latency specifications from CACTI [14]. Cacti is an integrated cache and memory access model, both for timing, leakage and power estimation. In our work we are interested in the timing values for caches, which we can use in the simulation.

We obtain the cache latency parameters used for simulation by calculating them using CACTI for a baseline cache. This baseline cache is an 8 MB 16-way cache, with 4 banks. It has a line size of 64 bytes, and is on the 32 nm technology node. The CACTI output for this is shown in Table 3.2. This configuration gives us an access time of 3.5 ns, equivalent to 7 CPU cycles at 2 GHz. The additional response time for a miss notice to be sent back to the CPU is 0.5 ns.

The cache latencies will accentuate or limit the impact of cache hits and misses on the total performance. The optimal size-to-latency of a cache is very workload dependent, and is an additional variable to consider when optimizing cache systems

	L1i	L1d
Hit Latency	1 cycle	2 cycles
Response latency	1 cycle	2 cycles
Block size	64 bytes	64 bytes
MSHRs	2	6
Write buffers	-	16
Size	32 kB	32 kB
Associativity	2	2

Table 3.3: Baseline parameters for the L1 caches. These values are defaults that come with the gem5 simulators ARM caches.

for multicore workloads. To keep the number of variables at a manageable level, we have decided to fix the cache latencies independent of the cache size. This is an obvious simplification, as a 1 MB cache will be faster than an 8 MB cache. The access time for a 1 MB cache is 2.9 ns, compared to 3.5 ns for 8 MB. But keeping the latencies fixed lets us compare experiments that would otherwise be incomparable. As these values will differ for every real implementation, it is more important that they are equal between experiments than completely accurate. This is also in line with previous work [16].

An unfortunate situation was discovered near the end of this work. As we started simulating our experiments, we used 32-way associative caches without updating the latency values from 16-way models. The access time for a 32-way 8 MB cache is 5.7 ns, compared to 3.5 ns for the 16-way cache. This error was not caught in time to redo all simulations, thus the results presented are using the latencies from a 16-way cache. The simulations are still fair as all experiments have had the same baseline, but the cache latencies do not match as closely to CACTI as they could have.

Baseline L1 specifications are shown in Table 3.3, and L2 specifications are shown in Table 3.4. The values are derived from the output of CACTI in Table 3.2.

The majority of the results presented will use 1 MB L2 cache for dual and quad core results, and 4 MB L2 for the 8 core results. This was chosen experimentally, to keep the cache small enough to ensure contention, yet large enough to have some impact on performance.

Connecting the caches are gem5s CoherentBus, which ensures that caches stay coherent during the simulation. It is 32 bytes wide, and has the same clock speed as the CPU core. The bus takes care of any concurrency issues during the simulation, and serializes accesses to the cache. This is the default bus in the gem5 simulator. Unfortunately there was no available information on what interconnect is used in

	L2
Hit Latency	7 cycles
Response latency	2 cycles
Block size	64 bytes
MSHRs	16
Write buffers	8
Size	1/2/4/8 MB
Associativity	32

Table 3.4: Baseline parameters for an L2 cache.

similar work. But as the interconnect remains the same for all schemes, we consider this a fair approach.

3.3 Main memory

The main memory size is set to 1 GB per benchmark in the workload, to allow sufficient space for all SPEC2006 benchmarks. This means 2 GB for dual core workloads and 4 GB for quad core workloads. The size of the main memory does not affect its latency or any other specification other than size itself.

For the main memory model, we use the SimpleDRAM module from gem5. It is a single-channel single-ported DRAM controller model that aims to model the most important system-level performance effects of a DRAM without getting into too much detail of the DRAM itself. It will model row and column operation and refresh cycles, and other effects such as write-to-read switch delays. Some important specifications are shown in Table 3.5.

These specifications approximate a DDR3-1066 DRAM with 7-7-7 timing, with a 1 GHz main memory bus. DDR uses both rising and falling edge of the clock, giving an actual clock rate of 500 MHz. This results in a clock period of 2 ns. In a 7-7-7 timing, the numbers indicate clock cycles for tCAS (latency to access a certain column), tRCD (delay between an row address and a column address) and tRP (latency to open a row). A fourth parameter, rRAS (row active time) is not modeled in gem5. A 7 cycle latency means delays of $2\,ns \times 7 = 14\,ns$, which is equivalent to our timing parameters.

The DRAM parameters have a huge performance impact on the workloads throughput. If the RAM has to do a random memory access, it will take 2 cycles missing in L1, 9 cycles missing in L2 and then 28 ns from accessing the main memory. At 2 GHz clock speed this adds up to 67 cycles. In the same time you could access

	SimpleDRAM
Write buffer size	32
Read buffer size	32
RAS to CAS delay	14 ns
CAS delay	14 ns
Row precharge time	14 ns
Refresh cycle time	300 ns
Refresh command interval	7.8 μs
Write to read switch time	1 ns

Table 3.5: DRAM model parameters.

L2 almost 7 times, and L1 a total of 67 times. Out-of-order execution help utilize this waiting period a bit better, but it still affects performance significantly. If we were to decrease the performance of DRAM, the performance difference in the results would increase. The main cost of a cache miss is due to the access latency to the DRAM, and thus a scheme that has fewer misses will benefit further when the DRAM is slower.

3.4 Hardware and computational overhead of cache management schemes

The suggested schemes all add hardware to perform cache partitioning and management. These additions monitor cache usage, calculates optimal use of the cache and enforce the partitioning schemes. However, all cache schemes claim that their implementation require a negligible overhead, and that no additional latency occur in the cache due to this overhead. In our simulations we maintain this assumption.

3.4.1 Maintaining a partitioned cache

The actual partitioning of the cache has a very low overhead. Depending on the number of cores N in the system, you will need $log_2(N)$ bits per cache line to indicate ownership of the cache line. Cache lines already have this sort of information to mark valid data, dirty lines, and so on. With a common 64 byte cache line size, an 8-core CPU will require 3 extra bits per line to partition the cache. This is an increase of at most 0.4 % .

For a physically tagged cache, no additional circuitry needs to be added to perform

checks on data accesses, as the tag uniquely identifies the data and thus the owner core. Only on replacements is it necessary to know the owner of the data.

3.4.2 Allocation algorithms

UCP and PIPP depend on a target partitioning given by some allocation policy. This policy could come from many places, e. g. the operating system, but is commonly given by a hit maximization algorithm. This algorithm takes the hit information from the shadow tag store and computes the best partitioning of the cache. The optimal solution to this problem is unfortunately NP-hard [19], although trivial for dual core CMPs. To get a good but not necessarily optimal partitioning, the Lookahead Algorithm is used instead. This is easily implementable and is reasonably fast, having a worst case time complexity of $N^2/2$, where N is the number of ways.

For PriSM, the target partitioning algorithm is even simpler. It only uses the total number of hits in the shadow tag store to evaluate the potential gain if an application had the entire cache for it self. The target partition is then adjusted based on what application has the highest potential gain. It requires approximately 20 arithmetic operations to compute the new target partition for a 4 core system, and grows linearly with the number of cores. The arithmetic operations do include floating point, used when computing the eviction probabilities required by PriSM. Some of these can be implemented as fixed point arithmetic to reduce overhead. Despite this, PriSM has the lightest partitioning algorithm of the implemented cache schemes.

All three cache management schemes assume no additional latency to calculate the target partitions.

3.4.3 Enforcement algorithms

Enforcement of the allocation differ between cache schemes. In UCP, accesses work as in a LRU cache, the only difference is in replacement handling. When searching for an eviction candidate, UCP first has to see if it occupies the target amount of ways. If it does not, it needs to find a block from one of the other cores that exceeds its quota, preferably the one that exceeds it the most. A block is then selected, and the rest works as a normal eviction/replacement. This process requires a small counter and access to the target partition values, none of which produce a significant overhead.

Enforcing PIPP requires more computation, in particular on an access hit. Unlike LRU and UCP, a block is not promoted on a hit in all cases. PIPP needs a 4 bit random number, and promotes the block if the number is not 0, giving a 3/4 probability of promotion. In the case of a streaming application a 7 bit random number is required, and the block is promoted if it is equal to 0, giving a probability

of 1/128. Random numbers may not be readily available, and as such could incur a small overhead. Evictions in PIPP add no extra overhead compared to LRU, the lowest priority block is always evicted.

PriSM does not require changes for the access methods. The only changes are when searching for an eviction candidate. When choosing an eviction candidate, PriSM requires a random number that follows a probability distribution. The number of bits required depend on how fine grained the cache partitioning has to be. The higher the number of bits, the better the eviction candidates will match the eviction priorities. The authors of PriSM report that between 6 and 12 bits will ensure similar behavior to using floating point numbers. After selecting an eviction core, the least recently used block of this core will be evicted, if it has a block. In the rare cases when it does not, a LRU block from a core with a non-zero eviction probability is evicted instead. This selection process does add a small overhead, but as it is done during eviction, time is not as limited as during a hit.

Again we simplify our model and assume no additional latency to enforce the cache management schemes.

Chapter 4

Methodology

4.1 Simulation methodology

4.1.1 Simulator

We use the Gem5 Simulator System [1] to simulate the architecture and cache systems. This simulator is designed to simulate a wide range of ISAs, including ARM, ALPHA and x64. We are using the ARM ISA, since this is a highly relevant ISA and has good support for detailed simulations in gem5. The internal CPU model is arm_detailed, for all parts of the experiment, the most detailed simulation model available. Gem5 can be configured in almost any way with varying number of cores, cache architectures and so on. In this work we are mainly interested in the cache features of gem5. Additions and extensive modifications have been done to simulate the various management schemes described in this work.

Out of the box gem5 comes with only LRU as a cache model. This work extends the simulator with extra models; static way-based partitioning, UCP, PIPP and PriSM. Modifications have been made to the simulator to make private caches aware of what CPU they belong to, and that all cache requests to shared caches include the requesting CPUs identifier. These changes are necessary to implement partitioning schemes, and is a common requirement for all the proposed cache schemes to work [23, 16, 18].

4.1.2 Single core checkpointing

Each individual benchmark is run to 15 billion instructions, and then checkpointed. This checkpointing takes a snapshot of the current memory, as well as storing all open file pointers, the program counter, register state, the page table, and so on. The whole state of the machine is preserved, with the exception of cache contents.

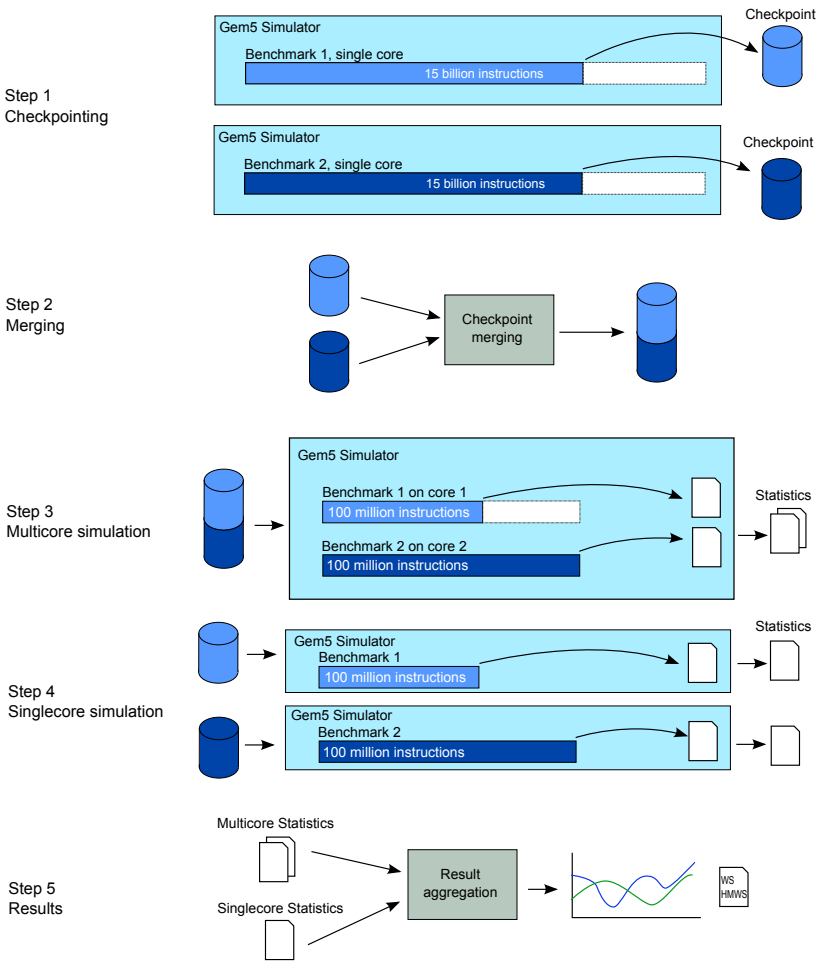

Figure 4.1: An overview of the simulation methodology. Notice how in multicore simulation, the benchmarks finish 100 million instructions at different times. The other benchmarks then continue to execute to simulate resource contention.

4.1.3 Multi core simulation

To start the simulation, a set of benchmarks (i. e. 4 benchmarks) is collected, one benchmark for each core. The memory of each of these benchmarks are merged together to form one larger memory snapshot. This is done using a checkpoint aggregation tool, which also resizes the size of the memory snapshot to N GB where N is the number of benchmarks. File pointers are restored for each of the new cores. At this point we have a known starting point for all benchmarks on all cores, and we are ready to perform the simulation.

Starting from this checkpoint, we simulate the desired number of instructions for all benchmarks in the workload. Every time a benchmark reaches its target number of instructions, statistics are dumped. When all benchmarks have simulated sufficient instructions, the simulation is complete. The whole process is illustrated in Figure 4.1.

This methodology allows us to know exactly where each benchmark inside each workload starts every time, and still lets us combine our benchmarks arbitrarily. This is important to be able to calculate the metrics comparing single core performance to multi core. Statistics are sampled when benchmarks reach their target number of instructions. Once it has reached this target, it need to continue simulating until all the others benchmarks have completed to ensure fair conditions for all. This continued execution then keeps up the resource contention between the benchmarks.

4.1.4 Checkpoint merging details

A set of python tools had to be developed to merge checkpoints from individual benchmarks into a common workload checkpoint. Each checkpoint is a dump of the memory and CPU state at the time of checkpointing. In order to successfully merge these together, we need to make sure that none of the addresses overlap. Thanks to virtual addressing, this is fairly simple. The page table dictates the translation from virtual to physical addresses, so by shifting the physical addresses we can separate the processes in memory. This is illustrated in Figure 4.2. After altering the page table, we can concatenate the memory dump of each of the processes into a single memory image, which is then used by the simulator.

Shifting the physical address mapping can either be done with a constant shifting factor, by 1 GB ($0x40000000$) as shown in Figure 4.2, or just shifted sufficiently to make the benchmarks go clear of each other. The latter option saves space in the file that contains the merged memory dump. It is therefore the method we use in our merging, but the effect is the same.

The Translation Lookaside Buffer (TLB) is used to speed up the virtual to physical address mapping, by acting as a cache for translations. Since we are modifying the page table during this merge, we invalidate the entire TLB when resuming from the checkpoint. This ensures that there are no old translations in the TLB that could contain translations that are no longer valid.

4.1.5 Computing resources

To simulate a large number of benchmarks, and architectures in many configurations, we were given access to the supercomputer Stallo[22] at University of Tromsø, Norway. This supercomputer has 304 nodes, each with two Intel Xeon 2.6 GHz 8-core CPUs. Each node has 32 GB of RAM available, and 500 GB of storage.

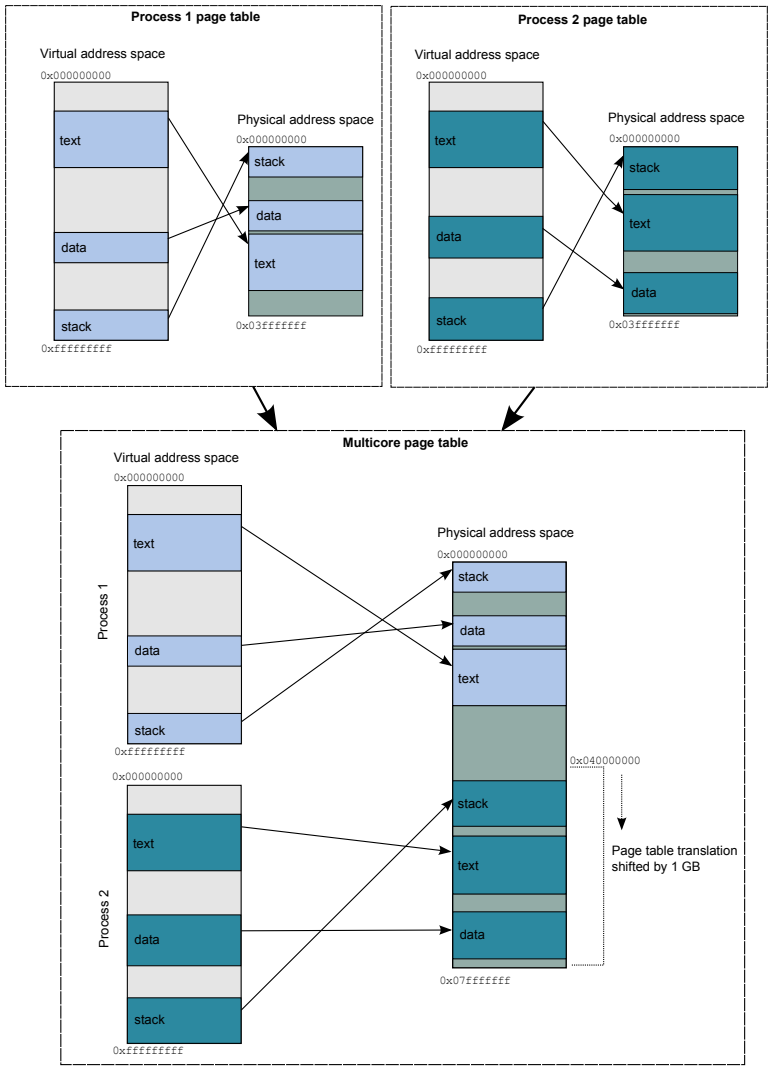

Figure 4.2: Merging page tables for two processes. This allows us to merge the memory dump of two processes without altering the programs themselves.

	Aggregated	Per node
Peak performance	104 Teraflop/s	332 Gigaflop/s
Nodes	304 x HP BL460 gen8 blade servers	1 x HP BL460 gen8 blade servers
CPUs / Cores	608 / 4864	2 / 16
Processors	608 x 2.60 GHz Intel Xeon E5 2670	2 x 2.60 GHz Intel Xeon E5 2670
Total memory	12.8 TB	32 GB (32 nodes 128 GB)
Internal storage	155.2 TB	500 GB (32 nodes 600GB raid)
Total storage	2000 TB	2000 TB
Interconnect	Gigabit Ethernet + Infiniband	Gigabit Ethernet + Infiniband

Table 4.1: A summary of the Stallo Supercomputer resources. Data from the Stallo User Documentation.

The node interconnect is both Gigabit Ethernet and QDR Infiniband. A summary of the specifications are shown in Table 4.1.

We were allocated 150,000 CPU hours for use in this work and related projects. This is equivalent to 17 CPU years. Of the 150,000 hours allocated, we spent 35,000 hours on simulating and development, 23 % of the quota.

For our purposes, we run separate experiments on each node without the need for communication between the nodes. With 16 cores on each node, the limiting factor on how many experiment you could run on each node is the RAM available. The SPEC2006 benchmarks require 1 GB of RAM each, so a 4 core simulation requires 4 GB of RAM. In addition, the simulator itself requires some RAM. 4 simulations were run on each node at a time, to ensure that sufficient memory was available.

4.2 Performance metrics

To evaluate the performance of the different schemes, we use well-known metrics to quantify results. Eyerman et al. [9] takes a more in depth look at measuring performance in CMPs and offer some advice in selecting appropriate metrics. The metrics used in this work are selected based on these recommendations.

In related work regarding computer architecture we find a wide range of metrics used to quantify performance. Various metrics can emphasize certain properties and hide others. It is thus important to select the metrics carefully and be mindful of the limitations. When you are condensing the notion of performance down to a single number, there is significant loss of information regardless of the metric used.

Using several metrics can help mitigate this somewhat, but there will always be tradeoffs.

4.2.1 Single core

Instructions per cycle (IPC) is a commonly encountered term, and is the basis of most performance metrics. It is trivially defined as

$$IPC = Instructions/Cycles$$

and tells us the average throughput of an application.

When looking at the change in performance of a single application, the most commonly used metric is Speedup.

$$Speedup = \frac{IPC_{new}}{IPC_{old}} = \frac{Execution\,time_{old}}{Execution\,time_{new}}$$

Both IPC and speedup is a bigger-is-better metric, an increase of these values is a positive change. Speedup is often denoted using a value followed by an x, (e. g. "1.5x"). A speedup of 1.5x is equivalent to 50 % increase in performance. A speedup of 0.8x is a reduction of 20 % in performance.

4.2.2 Multicore

There are several options for evaluating the performance of a multicore system. There are two straightforward approaches, the Weighted Speedup and the sum of IPCs.

$$Weighted\,Speedup\,(WS) = \sum \frac{IPC_i}{Standalone\,IPC_i}$$

$$Sum\,of\,IPCs = \sum IPC_i$$

The weighted speedup is the sum of IPCs divided by the IPC of the same workload if it is executed alone, without interference from other workloads. This is a common approach to quantifying performance, and is used by some of the articles written about cache partitioning [18, 23]. Weighted speedup as a metric is recommended by Eyerman et al. [9], as it tells us the system throughput. It quantifies the number of jobs completed per unit of time.

The sum of IPCs is a metric of throughput of the system as a whole. This is less interesting for the end user, who is mostly interested in speedup of each of

the applications. This metric also does not quantify fairness in any way, as an improvement for a high throughput application will improve the metric more than a low throughput workload, even if the total execution time is higher. Therefore sum of IPC is a rarely used metric, and is best avoided.

A variant of WS is the Harmonic Mean of Weighted Speedup (HMWS). The effect of this metric is that it will give less significance to a small number of outliers than the arithmetic sums. This may in some situations give a better picture of the data. For example, if only one application in a multicore system has a large speedup while the other applications stay the same, a harmonic mean will show less of a change than the arithmetic mean. HMWS is defined as

$$\textit{Harmonic Mean Weighted Speedup}\,(HMWS) = \frac{N}{\sum \frac{Standalone\,IPC_i}{IPC_i}}$$

In the article detailing PriSM [16], the principal metric used is Average Normalized Turnaround Time (ANTT), which is the inverse of HMWS. It is a lower-is-better metric, and tells us the user-perceived turnaround time slowdown from multiprogram execution. ANTT ranges between 1 and N, where 1 is perfect sharing without any resource contention. As ANTT goes towards N, the performance goes towards that of a serial execution of a program. ANTT is defined as

$$ANTT = \sum \frac{Standalone\,IPC_i}{IPC_i}/N$$

In this work, we use the metrics Weighted Speedup (WS) and Harmonic Mean of Weighted Speedup (HMWS). We present the averages for each set of workloads in these three metrics, and the detailed results for each workload using Weighted Speedup.

4.3 Benchmarks

4.3.1 SPEC2006 benchmark suite

As the benchmark suite we used SPEC2006 [7], from the Standard Performance Evaluation Corporation. This suite has long been the most used set of benchmarks for computer architecture research. Today there are other benchmark suites that also would be appropriate, but SPEC2006 was chosen as it was readily available, and also already compiled for ARM during an earlier project[10]. There are 29 benchmarks in SPEC2006. 28 of them were successfully cross-compiled for ARM, 24 of these successfully started in the simulator. Of these 24, 22 could be run up to the desired number of instructions, as two of the benchmarks crashed early in the simulation. And finally 20 of the 22 remaining benchmarks could be successfully resumed from their checkpoints. A summary can be found in Table 4.2.

Used (20)	**Not used (9)**
400.perlbench	403.gcc (Unable to compile)
401.bzip2	416.gamess (Unable to start)
410.bwaves	434.zeusmp (Unable to start)
429.mcf	436.cactusADM (Unable to start)
433.milc	454.calculix (Unable to resume)
435.gromacs	459.gemsFDTD (Unable to start)
437.leslie3d	465.tonto (Unable to resume)
444.namd	470.lbm (Stops during simulation)
445.gobmk	482.sphinx3 (Stops during simulation)
447.dealII	
450.soplex	
453.povray	
456.hmmer	
458.sjeng	
462.libquantum	
464.h264ref	
471.omnetpp	
473.astar	
481.wrf	
483.xalancbmk	

Table 4.2: Summary of the SPEC2006 benchmarks used.

With the limited time available, we prioritized getting results from those benchmarks that worked rather than to spend too much time trying to fix those that did not work. Some of the benchmarks did not run due to limitations in the simulator, although it was hard to narrow down the exact reason for each of them. Others can probably be fixed with a better understanding of gem5 and crosscompiling. Gem5 requires libraries to be statically linked, which could also be a source of issues for certain libraries.

4.3.2 Benchmark profiling

To create useful sets of benchmarks for our multicore experiments, the SPEC2006 benchmarks were individually profiled for various cache configurations. This gave us a better understanding of how each benchmark reacts to limited cache resources. The benchmarks were run on a single core architecture equivalent to that presented in section 3.1.

Benchmarks whose running time is not impacted by reducing the cache available to it are called cache insensitive benchmarks. These programs are typically CPU intensive, and have low memory requirements. A high L1 hit rate with few requests that needs to be serviced further down the hierarchy will keep the sensitivity low. Another type of cache insensitive applications are those with memory access patterns that do not utilize the any locality of the data. This leads to a large number of misses regardless of the allocated cache space, as caches are only useful if the data placed there is reused later. We call these streaming or stream-like applications.

Correspondingly, benchmarks that have a significant worsening of their runtimes when the cache resources are limited are said to be cache sensitive. Memory access patterns that take advantage of spatial and temporal locality will increase the caches sensitivity. The programs with this property are efficient when allocated enough resources so that the majority of the working set fits inside the cache. Reducing the cache allocation for a program like this will cause the working set to be moved outside of the cache, and thus incur more frequent misses. This in turn drives up the CPI, lowering the performance of the system.

To simulate a way-based partitioning scheme (such as UCP [18]), it is interesting to look at how the benchmarks react to reduced associativity and size at the same time. This corresponds to what happens in a multicore environment when a core is assigned a specific number of ways. If a core gets half of the available ways, it is equivalent to the associativity being reduced by half while the number of sets remains the same. This also reduces the cache space by half. In Figure 4.3 we see how the miss rate is reduced as the number of cache ways available increases. For some benchmarks, this lower miss rate can be seen as a reduction in CPI. These are the cache-sensitive benchmarks that were described earlier, for example astar and bzip2. Other benchmarks, like dealII, have large reductions in miss rate but little or no impact on the CPI. These benchmarks are cache-insensitive.

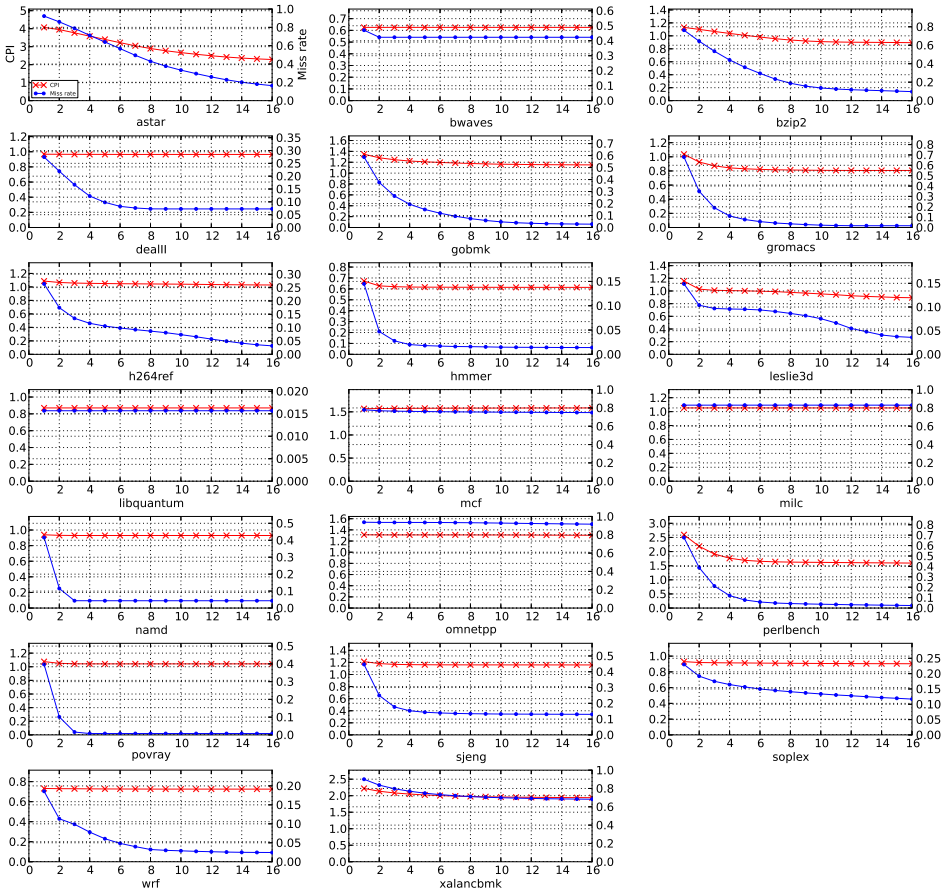

Figure 4.3: L2 cache miss rate (blue dotted) and CPI (red crossed) for SPEC2006 benchmarks with varying number of ways available. Ways not allocated to the benchmark are turned off. This simulates a strict way-based partitioning scheme.

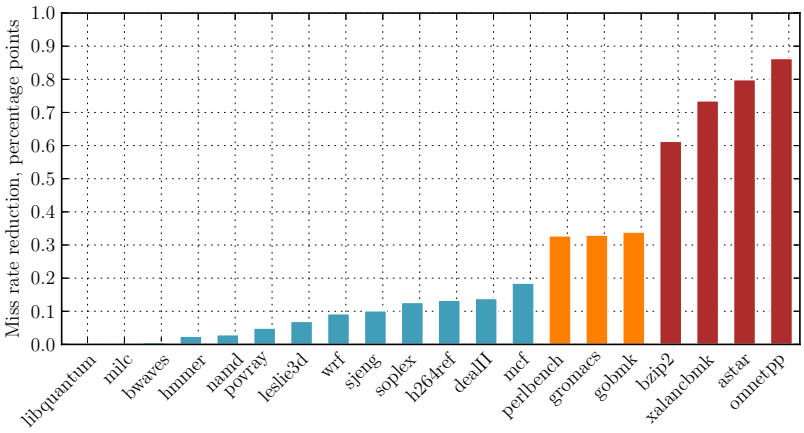

Figure 4.4: The reduction of miss rate in percentage points when increasing the cache size from 128 kB to 8 MB. High sensitivity benchmarks in red, medium sensitivity benchmarks in orange, low sensitivity in blue.

To simulate interesting workloads that cover many use cases, we group the benchmarks based on their cache sensitivity. We calculate the miss rate reduction when the cache size is increased from 128 kB to 8 MB. This is a significant increase in cache size, which lets us see all the high and medium sensitivity benchmarks clearly separated from the low sensitivity ones. Calculating this miss rate reduction leads to Figure 4.4.

Based on the data presented in Figure 4.4 and 4.3 we can create the grouping shown in Table 4.3. We decide on limits between the groups. Low sensitivity benchmarks are defined as those with less than 20 percentage points reduction. Medium sensitivity benchmarks have between 20 and 50 percentage points reduction, while high sensitivity benchmarks have over 50 points reduction.

Even though a benchmark might not be cache sensitive itself, it can have a high cache demand. Demand is defined by the number of unique addresses accessed in a given interval. High demand benchmarks with low cache sensitivity are interesting to pair up with high-sensitivity benchmarks, as they can have a significant negative impact on the other benchmark by stealing cache resources. Figure 4.5 shows the number of accesses to the L2 for each of the benchmarks. Particularly interesting are benchmarks such as leslie3d and mcf, high cache demand applications that do not benefit from the resources they use. These are the types of situations that have good potential for speeding up the execution by partitioning the cache.

It is worth noting that the profiling is only valid within the 100 million instructions that is being simulated. Beyond this, the benchmarks can and do change characteristics considerably. Some benchmarks go through cycles of low and high

Low sensitivity	Medium sensitivity	High sensitivity
bwaves	gobmk	astar
dealII	gromacs	bzip2
h264ref	perlbench	omnetpp
hmmer		xalancbmk
leslie3d		
libquantum		
mcf		
milc		
namd		
povray		
sjeng		
soplex		
wrf		

Table 4.3: Grouping the SPEC2006 benchmarks based on their cache sensitivity. The metric used is speedup from 128kB to 8MB cache.

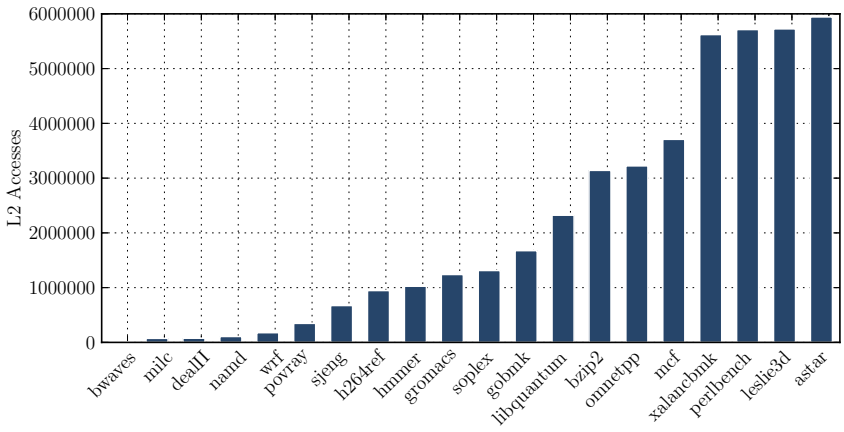

Figure 4.5: Number of L2 accesses for each of the benchmarks.

cache demand during their execution, and similar changes happens with the cache sensitivity. Thus a benchmarks characteristic cannot be defined by a single value for the entire execution. It is however a good starting point to create workloads and perform simple analysis of a simulation.

4.4 Workloads

We define workloads as sets of benchmarks, consisting of as many benchmarks as there are cores in the system under test. To test the cache schemes under different conditions, we put together workloads that will strain the cache in different ways. At the same time we do not want to bias the measurements by hand crafting the workloads to suit a particular need. To do this in the fairest way possible, we randomly combine benchmarks into workloads, but in two groups.

4.4.1 Dual core workloads

With 20 benchmarks the total number of possible dual core workloads is $\binom{20}{2} = 190$. This is slightly outside our simulation budget, so we select a subset of these. The natural approach is to select workloads at random. However, our initial findings indicated that many of the individual benchmarks did not utilize the cache enough to create visible differences between the cache schemes. This was caused by the low-demand benchmarks not causing any cache contention.

To get a higher percentage of workloads where differences can be seen, we perform a separation of the benchmarks into two groups and create workloads from these. We first select the 10 benchmarks with the highest number of L2 accesses, and pair these into workloads. These benchmarks guarantee some cache contention, and will more clearly show differences between cache schemes. Selecting from these 10 benchmarks result in $\binom{10}{2} = 45$ workloads, a manageable number. We can therefore simulate all combination of benchmarks for the most cache intensive benchmarks. These benchmarks are prefixed with the name 2H.

To cover all benchmarks, we also select 50 at random from the 190 total possible combinations. We ensure that none of the selected workloads overlap with the previous 45 workloads selected, so they are all unique. These 50 workloads are prefixed 2A. Doing this ensures that all benchmarks are represented in the results, and that we are likely to cover as many use cases as possible.

The dual core workloads are listed in Appendix A. 2A workloads are listed in Table A.2 and 2H workloads in Table A.1.

4.4.2 Quad core workloads

Combining the 20 available benchmarks into quad core workloads in all possible ways leads to $\binom{20}{4} = 4845$ workloads. We would like to simulate 100 workloads, and

perform the same division as before. The top 10 cache demanding benchmarks are used to create 50 high demand workloads. There are a total of $\binom{10}{4} = 210$ possible combinations possible when selecting 4 benchmarks from a set of 10, and we select our 50 workloads from these at random. These workloads are named 4H-workloads

We also select 50 workloads from the set of all 4845 possible combinations. We make sure none of the selected workloads overlap with any of those previously selected, so we get 100 unique workloads. The workloads selected from all benchmarks are named 4A-workloads.

The 4H workloads are listed in Appendix A, Table A.3, and the 4A workloads in Table A.4.

4.4.3 8 core workloads

For 8 core workloads, we selected a lower number of total workloads to stay within our simulation budget. The workloads are constructed from randomly selected benchmarks, without any of the limitations used in dual or quad core workloads. The reason for this is that the cache demand with this number of applications will be sufficient, as there will be a high probability of selecting at least a few high intensity benchmarks. We generate 25 workloads at random, and denote these at 8A-workloads. They are listed in Appendix A, Table A.5.

4.5 Implementation of cache management schemes

4.5.1 Overview

To as large extent as possible, we have followed the original implementation specification given by the cache schemes. However, some specifications are incomplete or unavailable, at which point we have made our own assumptions and modifications. The target has been to ensure fairness between the schemes to best be able to compare them. We present our implementation details in the following sections. Specifications that are not given here are described in Chapter 2 and in the original articles for UCP [18], PIPP [23] and PriSM[16].

4.5.2 Shadow Tag Store

The Shadow Tag Store is based on the one detailed in the UCP article [18]. The Auxiliary Tag Directory is a LRU stack with tags for each core in each set. The Shadow Tag Store is the same for all implementations in this work. Evictions are done from the LRU position, insertions are done at MRU. Hit counters are

incremented on each access that hits, and the counters are halved after each partitioning period. Invalidation calls to the parent cache has no effect on the Shadow Tag Store.

DSS was implemented, but not used. We decided to grant all schemes access to accurate information about the cache usage. This removes an additional variable in the analysis, and helps to analyze the cache schemes fairly. Using DSS would have added an extra dimension to our comparison, namely the number of sets sampled. Also, UCP have shown how DSS approximates the real cache behavior well[18]. Our results can be seen either to rely on these approximations to be correct, or as an analysis of how cache schemes operate when given accurate information.

4.5.3 UCP

UCPs specifications are sufficiently detailed in the original article to implement it, so no assumptions on our part have been made. The partition period is set to 5 million cycles. We use the Lookahead Algorithm to compute the partitioning, even for dual core workloads where it would computationally feasible to calculate the optimal partitioning.

4.5.4 PIPP

PIPP suggests a large number of variants of its algorithm. In this work we have used a variant that is as close to the original article as possible. On a hit we use a promotion of 1 position, with a probability $p_{prom} = 3/4$. In case of a streaming application, we use a promotion of 1 position with probability $p_{stream} = 1/128$. UCP is used to calculate the partitioning of the cache, using the Shadow Tag Store and the Lookahead Algorithm. The Shadow Tag Store is detailed above, but it is worth noting that on a hit, tags are promoted to MRU, not by one position as in PIPP. Oddly enough the partition period is not specified, but is assumed to be equivalent to that of UCP, 5 million cycles.

To detect if an application is streaming, a miss ratio threshold and miss count threshold is used. If the application exceeds these thresholds, it is considered streaming. In our implementation we use only the miss ratio $\frac{m_i}{A_i}$, and check that it is larger than a threshold θ_m. We do not use the miss count, as the period is not specified. The article suggests using a miss rate threshold of 0.125, in our findings this is too low and causes too many applications to be considered streaming, overall reducing performance. Instead we use the threshold $\theta_m = 0.25$.

4.5.5 PriSM

PriSM uses the Shadow Tag Store, the same as the other schemes use, but only use the total number of hits, not the recency information. The partitioning period is

given in number of misses, but this number is not specified in the article. A figure in the article does show fraction of replacements not from desired core for various periods, for the periods 32K, 64K and 128K[16] . We chose the middle ground, and selected a partition period of 64K misses for our implementation.

Chapter 5

Results

5.1 Introduction

A large amount of data has been produced from simulating different cache configurations and workloads. To avoid overwhelming the reader, much of the data has been placed in the appendix. In addition to aggregated results, we present some deeper analysis of selected workloads. We have focused on those workloads that do not perform as well as the rest. The workloads that do perform well are less interesting, as their operation is described in the background work. The poorly performing workloads often break assumptions that the cache schemes make, and thus gives us insight into the limitations of each scheme. We provide such analysis for UCP, PIPP and PriSM.

Unfortunately, 4 out of the 975 simulations did not complete successfully. Even with significantly more simulation time than the other benchmarks, they did not complete for various reasons. Pairing high IPC benchmarks with low IPC ones will require significant amounts of instructions to be simulated, and some cache schemes can cause starvation. These and other reasons have caused the simulations to not complete. This will appear as missing bars in the performance bar plots later. It is an unfortunate situation, but we chose to present these results instead of selectively removing workloads from the full set of results. The value for an incomplete workload is not included when calculating a schemes average performance.

We present the results in the following sections, grouped by number of cores.

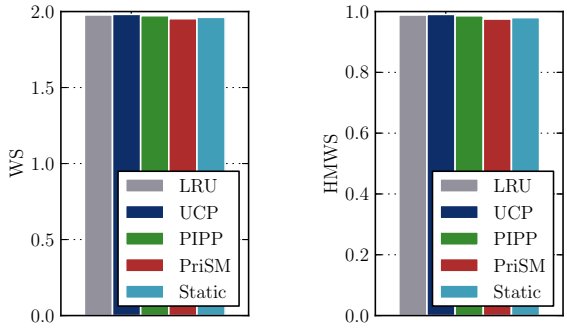

Figure 5.1: Arithmetic averages of WS, HMWS and ANTT for 2A workloads. Note that ANTT is a smaller-is-better metric.

5.2 Dual core

5.2.1 Performance overview

We start by looking at the results of dual core workloads. For the 2A workloads, the results show that there often is no impact of the cache sharing. When one or both of the benchmarks in a workload is a low intensity application, the performance difference between the schemes is close to 0 and the speedup is almost perfect. The arithmetic averages of WS, HMWS and ANTT for 2A workloads is shown in Figure 5.1.

The 2H workloads contain benchmarks that are more cache intensive than the 2A workloads. The differences between the cache schemes now come into play, as cache contention is increasing. The arithmetic averages of WS, HMWS and ANTT is shown in Figure 5.2. UCP has a WS increase over LRU of 1.4 %, while PIPP barely increases it with 0.3 %. PriSM reduces WS by 3.5 %, and has the lowest performance of the four cache schemes.

To look at how the performance is distributed across workloads, Figure 5.3 shows the WS of each workload in ascending order. Each cache scheme is ordered individually, which illustrates the fraction of workloads which are below or above a certain WS. PriSM can be seen to fall significantly in performance in its worst 5 % of workloads. The top 20 % of workloads have fairly low utilization of the cache, so here the schemes converge and all obtain the optimal WS of 2. Keep in mind that individual workloads can not be compared from this figure.

Static partitioning has the second lowest performance of the group. It loses performance particularly in workloads that consist of 1 high demand application and one low demand. These workloads have little cache contention, and the static partitioning reduces the available space for the high demand application, lowering the performance.

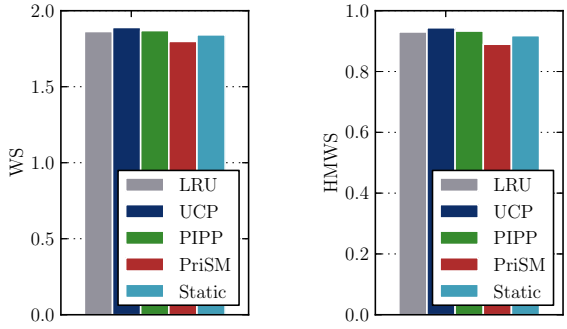

Figure 5.2: Arithmetic averages of WS, HMWS and ANTT for 2H workloads. Note that ANTT is a smaller-is-better metric.

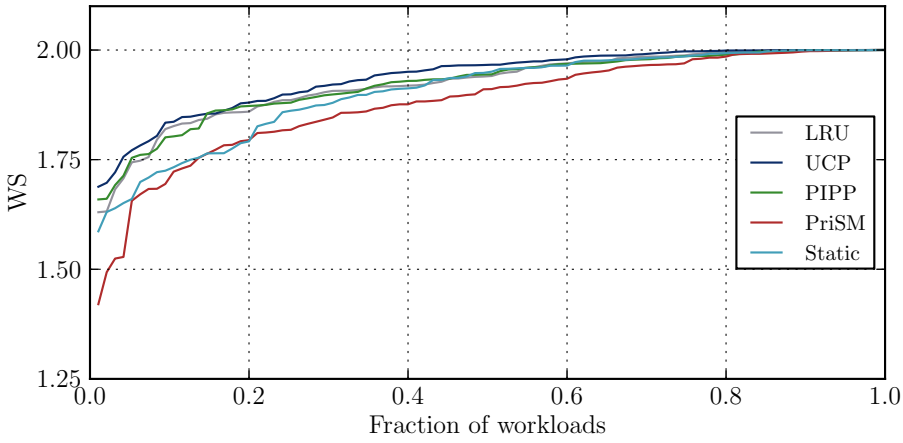

Figure 5.3: WS of dual core workloads, in ascending order.

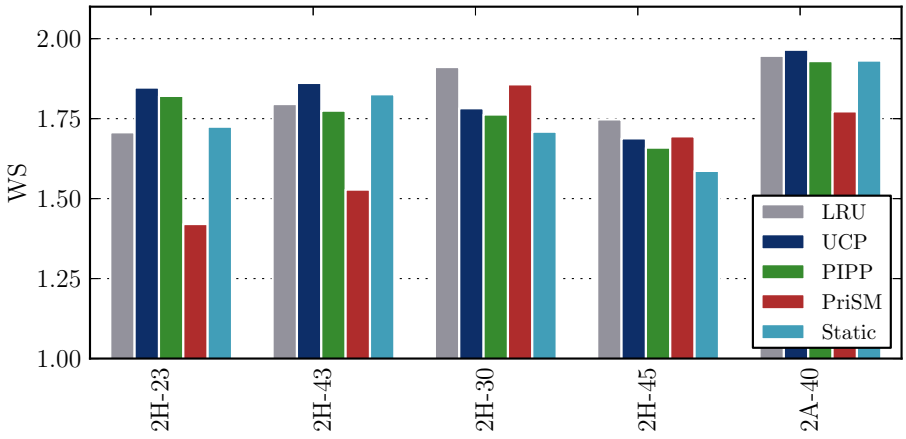

Figure 5.4: WS of some selected workloads described in this section. Remaining workloads are shown in Appendix B.

Performance values for all workloads are shown in Appendix B, Section B.1. These figures present information about each schemes performance, and allows comparisons for individual workloads. Due to the amount of data, these have been placed in the appendix, and instead we show some selected workloads in Figure 5.4 that will be discussed later as we look at the performance of schemes on specific workloads.

For the 40 workloads with the worst LRU performance (Figure B.1 and B.2), UCP has the strongest performance. PIPP also does well, in many cases outperforming LRU. PriSM does have some good workloads, but is significantly worse than the others in other workloads, e. g. in 2H-23 and 2H-43 which can be seen in Figure 5.4.

The remainder of this section is dedicated to provide a deeper analysis of the performance of UCP, PIPP and PriSM on specific dual core workloads.

5.2.2 UCP performance analysis

On average, UCP has good performance compared to LRU in the dual core workloads simulated. It improves on LRU for most high-demand workloads, and it rarely hurts the performance in workloads where there is little cache contention. However, it reduces the performance compared to LRU in some workloads. The workload 2H-30 astar-bzip2, previously shown in Figure 5.4, is an example of this. We will take a more thorough look at this workload to determine why it has lower performance.

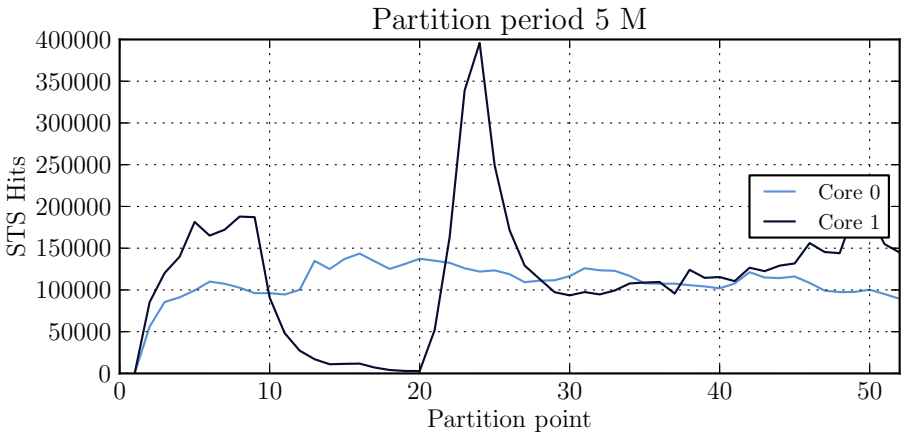

Figure 5.5: Shadow Tag Store hits for each of the cores, with the default partition period of 5 million cycles.

5.2.2.1 Case Study: Workload 2H-32

This workload contains the benchmarks bzip2 and astar, both high-intensity and high-sensitivity benchmarks. They utilize large caches efficiently, and when they share a cache, contention is high.

However, the largest factor affecting the performance of this workload is the partitioning interval. During execution, we have significant changes in cache utilization for bzip2 on core 1. These utilization changes have very short periods, and with the default partitioning interval of 5 million cycles they simply disappear. In Figure 5.5 we see the Shadow Tag Store hits for each core, using the default interval. Each sample is taken at the start of each partition period, and is the sum of all hits in the STS this period plus the remaining hits from previous periods. At the start of a new period, the current hit counters are halved. This ensures some significance to historic hits, but most consideration to newer occurrences. The curve for core 1 changes with the major cycle in the application, and between the peaks the curve is relatively smooth.

By reducing the partition period to 1/10th of this, 500,000 cycles, a new pattern emerges. The hits in the STS with this partition period is shown in Figure 5.6. The hits for core 1 is now suddenly much more unevenly distributed, going up and down in a cyclic fashion. Particularly important is that this change causes it to have alternating higher and lower hits than core 0. This means UCP can partition the cache differently to achieve better performance, by giving core 1 more or less cache space depending on its utilization.

It is now easy to see why LRU performs much better than UCP in this situation.

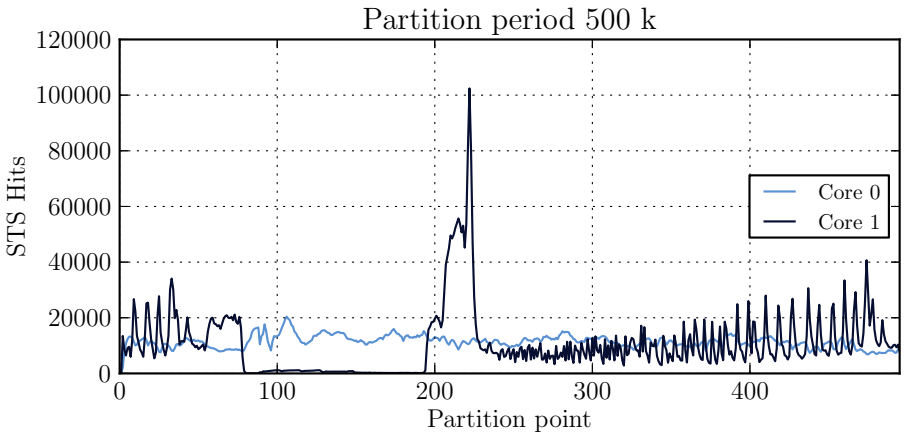

Figure 5.6: Shadow Tag Store hits for each of the cores, with a shorter partition period of 500,000 cycles.

LRU has no periods where it is locked to a specific partitioning. This lets it respond very quickly to changes in the access patterns, utilizing the cache as best it can. Even though it does not prevent interference between the applications, it leads to good performance for this specific workload.

By simulating UCP with different partition periods we analyze the change in performance, shown in Figure 5.7. LRU has a baseline WS of 1.782. By decreasing UCPs partition period to 2.5 million cycles, its performance becomes 0.9 % better than LRU, as opposed to being 3.7 % worse at 5 M cycles. Further decreasing the partition period increases the performance more, up to a WS of 1.86 at $50,000$ cycles per period.

However, lowering the partition period increases the number of times the cache is partitioned. Performing the partitioning is not without cost. It requires adding many hit counters, running the Lookahead Algorithm to determine the next target partitioning and halving all the STS counters. All this consumes energy and time, and thus a tradeoff between frequency and cost must be made. In our simulations there are no costs associated with partitioning, and we have unfortunately not devised a way to model this cost. Therefore we can not determine the best tradeoff in this situation, only conclude that the default period leads to a performance worse than that of LRU for this specific workload.

A potential source of poor performance can also be the global set partitioning enforced by UCP. Consider the case where the global partitioning splits a 32-way cache evenly, 16-16 ways for each core in every set. But for a given set, the optimal partitioning might be 25-7, as one application accesses blocks with a specific index more often than the other core. This core can not utilize the cache ways belonging

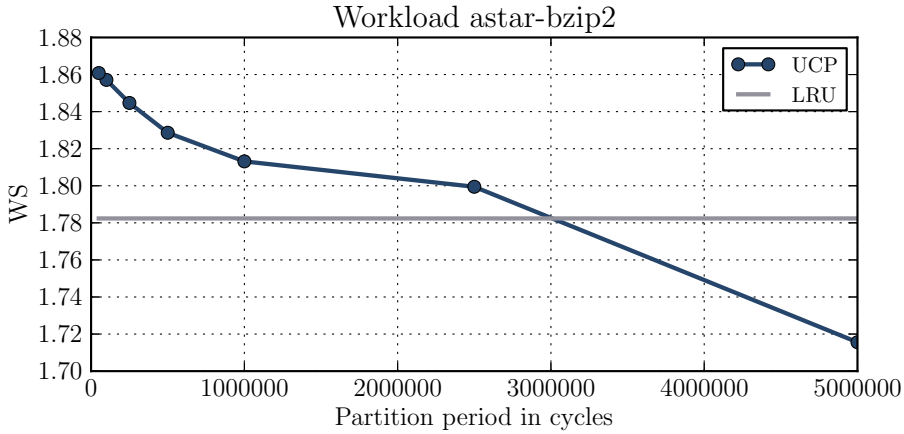

Figure 5.7: WS for different partition periods. When the period is shorter than 2.5 M cycles, UCP performs better than LRU.

to the other core, and the full potential of this cache set is not used. One could imagine a situation where a local partitioning would split half the sets 31-1 and the other half 1-31, but the global partitioning partitions it 16-16. In this scenario, the cache would be severely underutilized.

In our workload 2H-32 astar-bzip2, we see some of these effects. We can measure the allocation difference between a local and a global partitioning, to see the impact of this coarse grained approach. In Figure 5.8 we attempt to visualize this difference. First, UCP calculates the global partitioning, indicated by the solid lines. For each set in the cache, it then also calculates the best local partitioning. We graph the positive difference between local and global partitioning, averaged per set [1]. The formula used is

$$D_i = \frac{\sum_{set=0}^{numSets} \begin{cases} Local_{set} - Global_i & , if\ Local_{set} > Global_i \\ 0 & , otherwise \end{cases}}{numSets}$$

When D_i is high, there exists many sets where local partitioning would increase the allocation of $core_i$ at the cost of the other core. D_i is higher here than in other workloads where UCP performs well. So in our workload 2H-30 astar-bzip2, using a local partitioning would increase the number of ways allocated to core 1 (bzip2) in many of the sets.

[1] We could not have used the average of the absolute difference between global and local allocation here ($(\sum |Local_{set} - Global|)/N$), as this would have led to equal D_i for both cores. Lowering one cores allocation increases the other cores allocation. Regardless of the metric used,

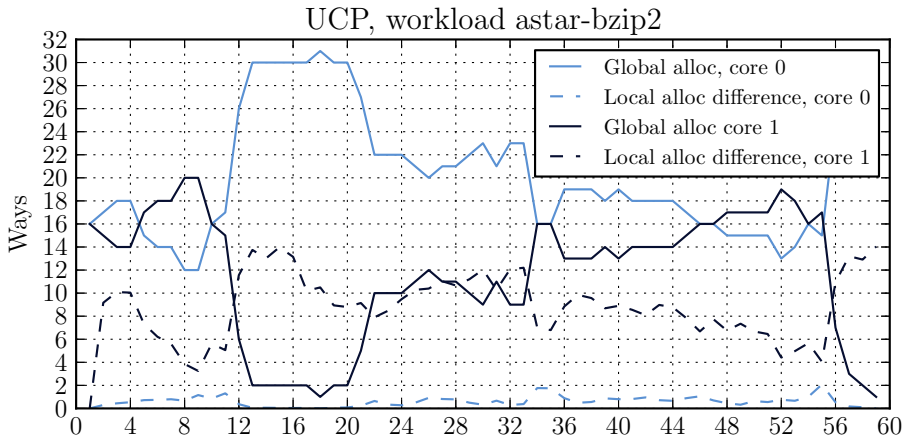

Figure 5.8: The difference in allocation between a global allocation policy and a local allocation policy. The dashed lines indicate the absolute difference in ways between the global and local allocation, averaged to the number of sets.

The coarseness of UCPs partitioning thus results in a less than optimal partitioning. Utilizing local instead of global partitioning will increase the performance of bzip2, but will reduce the performance of astar. Unfortunately, local partitioning is unfeasible in practice, due to its large implementation overhead. Local partitioning would prevent UCP from using DSS to minimize storage, and would require the Lookahead Algorithm to run for every set in the cache. Another problem with local partitioning is that the unevenness of allocation makes it even more susceptible to changes in the program behavior and the partition period as we saw before.

5.2.3 PIPP performance analysis

Taking a closer look at one of the workloads where PIPP performs poorly is useful to see how it operates. We look at workload 2H-45 astar-leslie3d, from Figure 5.4. Here PIPP has the worst performance of all the cache schemes.

PIPP utilizes the same shadow tag store as UCP to calculate a target allocation. The allocation is based on the optimal number of ways that each core has to obtain to maximize the total number of hits in the entire cache. PIPP then uses the number of ways allocated to a core as the insertion position when blocks are inserted to the cache. The concept is that this policy, combined with a promotion strategy, will keep the actual allocation very close to the target allocation, without strictly enforcing a partitioning of the cache.

---

the goal is to illustrate the unevenness of optimal allocation between sets.

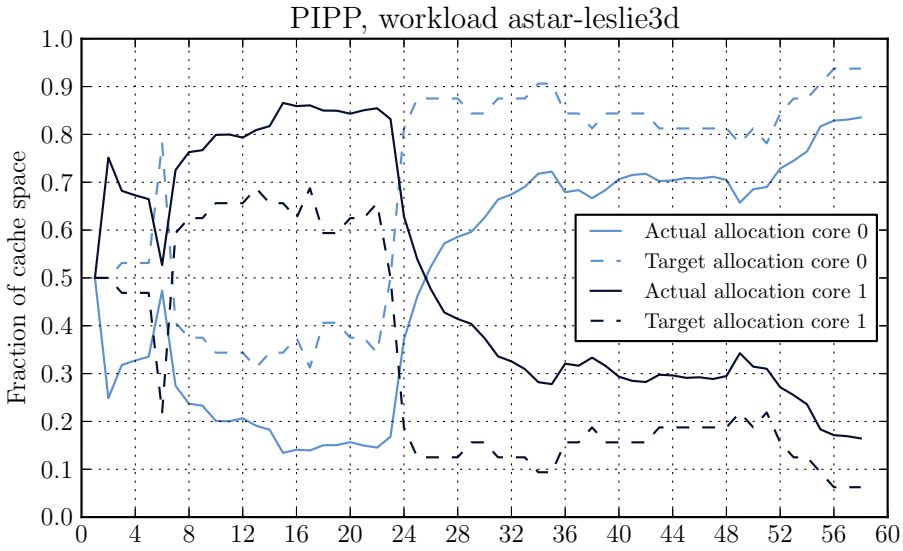

Figure 5.9: Actual and target allocation for PIPP during workload simulation. Notice the change in behavior for core 1. The increase in allocation for core 0 is a response to the reduced utility of core 1. Also note how the actual allocation does not follow the target allocation very well.

5.2.3.1 Case Study: 2H-45

In this workload, one of the two benchmarks changes its memory access characteristic during the simulation. Leslie3d goes from being a high-intensity workload to requiring much less cache around halfway through the workload. This can be clearly seen in Figure 5.9. After partition point 21 we see a significant drop in core 1s target allocation (the dashed line). This is a response to its reduced utilization of the cache, which causes the allocation policy to allocate less space for it. This is the intended functionality, where astar on core 0 is more capable of utilizing the bigger cache space.

What is also clear is that PIPP is not able to match the actual occupancy of the cache to the desired target. Ideally, the solid lines in Figure 5.9 would follow the dashed ones closely, indicating that the partitioning scheme works as intended. Instead it follows it only loosely, normally staying at around 10 percentage points off the target allocation. This results in a less-than-optimal use of the cache.

In particular when the target allocation changes significantly, we see that PIPP takes a long time to re-adjust the occupation of the cache. It takes almost 10 partition periods, from partition point 24 to partition point 34 in Figure 5.9, for the cache to stabilize with its new allocation (and this is still not precisely the

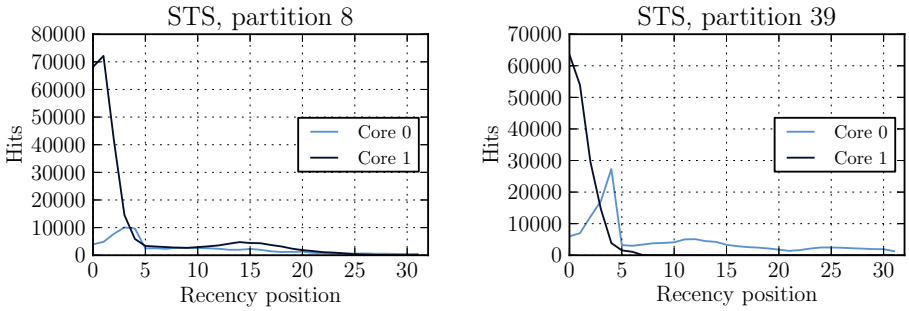

Figure 5.10: Hit counters per recency position in the shadow tag store, for two of the partition points from Figure 5.9.

desired occupation). This is partly due to the slower promotion approach, which in turn means that it takes longer for a block near MRU to fall all the way to LRU.

In order to understand why the target allocation changes, we will look at two points in time to see how the workload changes its behavior. The shadow tag store monitors the cache, and tells us at which recency position accesses causes hits in the cache for each core. The STS hit counters for the two points 8 and 39 from Figure 5.9 are shown in Figure 5.10. Partition position 8 in Figure 5.9 is representable for the first part of the execution, where leslie3d on core 1 has better hit numbers than astar for a large part of the recency range. This leads to the allocation seen before, in this case 62.5 % for core 1 (20 ways of 32) and the remaining 37.5 % for core 0.

Then, at around partition point 24, we have a significant drop in utilization of the last recency positions. This is effectively a sign that the working set of leslie3d has now been reduced, fitting in a much smaller amount of cache. Now it only utilizes the first 7 recency positions, and then has no use for any later location. Although leslie3d still has significant use of the cache in term of number of hits, it needs less of the cache space to achieve the same hit rate. This leads to a reversal of the optimal cache allocation, with 84.3 % (27 ways) of the cache going to astar on core 1, and 15.6 % going to leslie3d.

5.2.4 PriSM performance analysis

PriSM has the lowest performance of the schemes in our dual core experiments. It has a few particularly poor workloads, where its performance is far below that of the other schemes. We will study one of these in detail, workload 4A-40. The WS of this workload can be seen in Figure 5.4.

5.2.4.1 Case Study: Workload 2A-40

Workload 2A-40 consists of bzip2-hmmer, and is a workload where PriSM performs the poorest. In Figure 5.11 we see the next target occupation and current occupation for PriSM. Although the lines follow each other, it is not due to the current occupation following the target allocation as one would expect. Instead it is due to the next target allocation being based on the current occupation. This causality ultimately makes the allocation fall too much in favor of core 1, negatively impacting the performance of core 0.

Bzip2 (on core 0) is a high sensitivity application with a high number of L2 accesses, and hmmer (on core 1) is a low-sensitivity application with a fewer number of L2 accesses. Hmmer has about 1/3 of the cache accesses of bzip2. Intuitively this would lead us to believe that bzip2 should be allocated more of the cache, as it can utilize it better. And indeed as we see from UCP in Figure 5.11, most of the cache space is allocated to bzip2, leading to good performance for UCP.

We can look at a single partitioning point to see how PriSM does its allocation, point 3 in Figure 5.11. Here PriSM suggests a next target allocation T_{core} of 0.308 for core 0 and 0.692 for core 1. But on the next point, point 4, the current occupation for core 0 has sunk to 0.04, and for core 1 it has risen to 0.96. Thus the target allocation has not affected the cache occupancy in the desired way.

The target allocation is calculated by the formula

$$T_{core} = C_{core} \cdot (1 + (PotentialGain[core]/TotalGain))$$

and then normalized:

$$T_{core} = T_{core} / \sum T_{core}$$

The potential gain is given by

$$PotentialGain[core] = StandaloneHits[core] - SharedHits[core]$$

Unlike UCP and PIPP, PriSM simplifies its optimal allocation analysis. Instead of analyzing the potential gain of a adding certain amount of extra cache, it only

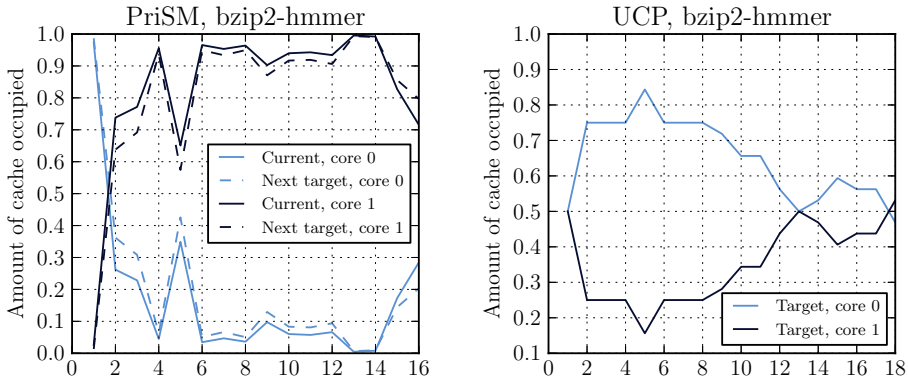

Figure 5.11: Target allocations for UCP and PriSM. UCP outperforms PriSM in this workload. UCP partitions the cache fairly between the two cores, while PriSM prioritizes core 1. This leads to starvation for core 0, reducing the total performance. Note that the partition points are not corresponding, UCP and PriSM partitions at different times during the execution, hence they are not plotted in the same graph.

evaluates the gain if the entire cache belonged to a core. This is a very coarse grained approach that only approximates what more cache space would achieve. The counter for standalone hits comes from the Shadow Tag store, shown in Figure 5.12. The total number of standalone hits is 102691 for core 0 and 60521 for core 1. PriSM then knows that core 0 can utilize the cache best, as it has the higher standalone hit count. But it does not take into account that core 1 has significant amounts of hits only on its first few recency positions. This means that although it has a target allocation of 0.692, it will only require 8 ways to achieve 98.5 % of its hits, the rest of the cache is useless to it. The remaining ways could be better utilized by core 0.

As we have seen so far, the highest potential gain belongs to core 0, yet core 1 is allocated almost all of the cache. The reason for this is that the target allocation T_{core} is a function of the current occupancy of the cache C_{core}. As the occupancy of a core goes towards zero, the potential gain gets reduced in the multiplication: $T_{core} = C_{core} \cdot (1 + (PotentialGain[core]/TotalGain))$. The target allocation is severely shifted by the contents of the cache, not just the possible hit increase. This seems like an odd design choice. If the current occupation is shifted without PriSM being able to interfere successfully, it will also alter the target allocation for the next period, regardless of whether this will improve performance or not.

After calculating the new target allocation, PriSM reaches the step where it calculates the eviction probabilities. It is done according to the equation

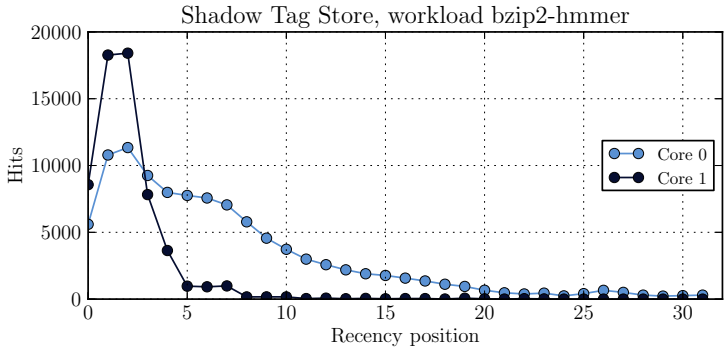

Figure 5.12: Hits from each recency position in the shadow tag store, from partition point 3 in Figure 5.11. The total number of hits are 102691 for core 0 and 50621 for core 1.

$$E_i = \begin{cases} 0 & (C_i - T_i) \cdot N/W + M_i < 0 \\ 1 & (C_i - T_i) \cdot N/W + M_i > 1 \\ (C_i - T_i) \cdot N/W + M_i & otherwise \end{cases}$$

where C and T are the current occupation and target allocation, N is the total number of cache blocks, W is the number of misses in the partition interval and M is the fraction of misses caused by this core. M is an important factor at this step, as PriSM tries to adjust the eviction probabilities so that they take into account the extra replacements caused by a high-miss application. The factor N/W determines the impact of the difference between current and target allocation on the eviction probability. If the allocation period is shorter, the eviction probabilities needs to be adjusted more to reach the target more quickly.

Overall, PriSM is unable to reach its initial target allocation. As it then proceeds to set its new target based on the earlier failed attempt, the allocation is skewed further. It does not manage to impact the cache occupancy in the desired way, thus leading to a very uneven distribution of the cache space, finally resulting in very poor performance.

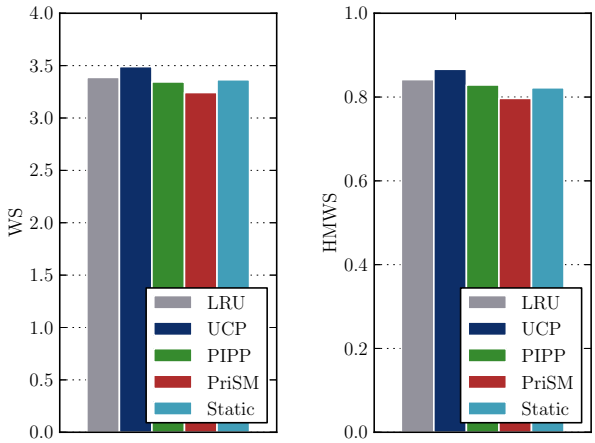

Figure 5.13: Arithmetic mean for WS, HMWS and ANTT for quad core workloads. Note that ANTT is a smaller-is-better metric, WS and HMWS are bigger-is-better metrics.

5.3 Quad core

5.3.1 Performance overview

For these quad core results, we use a cache size of 1 MB to keep contention high between the applications. Again we compare the three suggested cache schemes to LRU and static partitioning, using the metrics WS, HMWS and ANTT. The arithmetic averages of these metrics across quad core workloads is shown in Figure 5.13. UCP takes the lead with the best performance, followed by LRU and PIPP. PriSM performs the poorest of the four. UCP improves on LRUs WS with 3.0 % on average. PIPP has a WS decrease of 1.1 %, while PriSMs reduction is 4.4 % compared to LRU. Static partitioning reduces the WS by 0.6 % compared to LRU.

We further analyze the WS performance by ordering the workloads in ascending order, individually for each cache scheme, as shown in Figure 5.14. This illustrates the fraction of workloads below a given WS for the schemes. Again note that schemes can not be compared for individual workloads in this figure. We can however see the distribution for each scheme by itself. All the schemes have a relatively similar distribution, in other words none of the schemes perform significantly better or worse for a larger subset of their workloads. PriSM does converge with the other workloads a bit later than the rest, indicating that it does perform poorly for a slightly larger amount of workloads than PIPP and UCP.

For completeness, we present the weighted speedup for each of the quad core workloads in Appendix B, Section B.2.

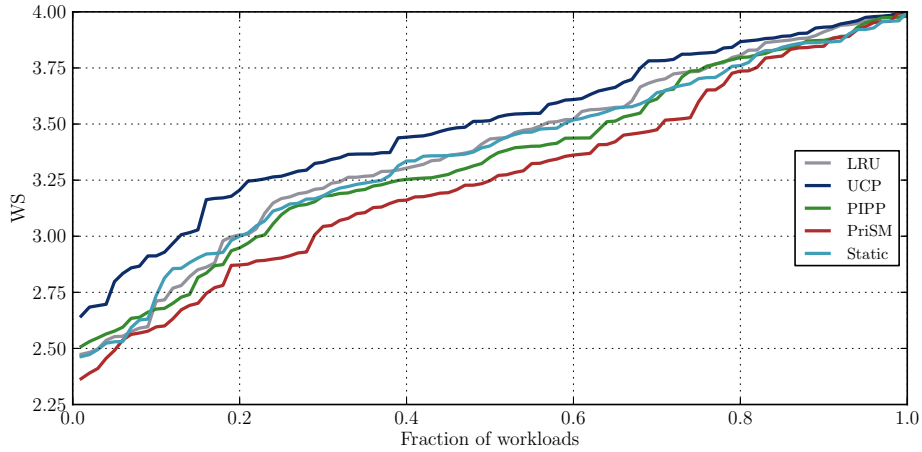

Figure 5.14: WS of 4 core workloads (both 4H and 4A), in ascending order.

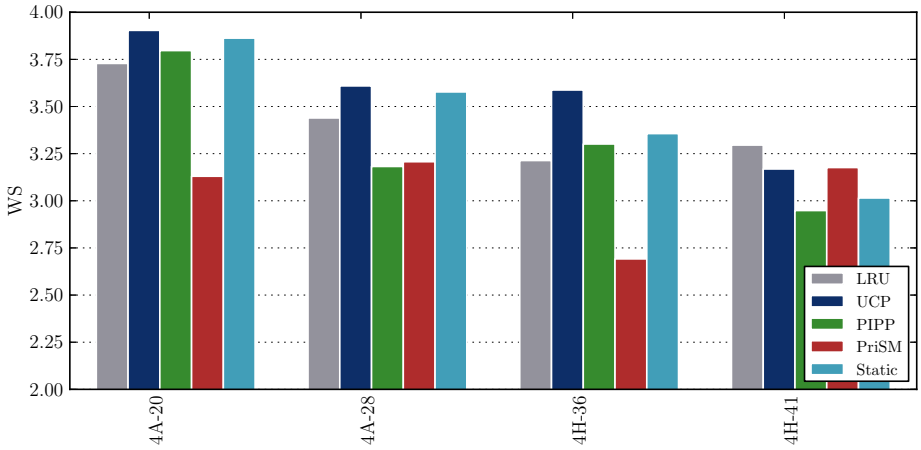

Figure 5.15: WS of some selected workloads that will be discussed in this chapter. The remaining workloads are shown in Appendix B, Section B.2.

By looking at workloads where LRU performs poorly , we can evaluate the schemes performance in high congestion workloads. UCP performs significantly better than LRU here, improving on it by 6.6 % for the 40 workloads with the lowest LRU performance, and by 8.2 % when only looking at the lowest 20. PIPP performs as good as LRU (less than 0.1 % difference) in the bottom 40 workloads, while PriSM has a 4.5 % slowdown compared to LRU.

For the 20 workloads with the best LRU performance, (Figure B.10 in Appendix B), the difference between the schemes is much smaller. UCP, PIPP and LRU have very similar performance, all within 0.1 % of each other. These are workloads that do not stress the cache very much, as we are very close to a perfect speedup by running these applications in parallel. The low congestion of the cache means that there are few conflicts and all the cache schemes work well. PriSM performs slightly worse than the other schemes, only beating static partitioning. Static partitioning works very poorly here, as it limits the available cache space to 1/4th, even for those benchmarks with good cache utility.

Figure 5.15 shows the WS of some workloads that we will discuss in the next sections. We will take a closer look at the performance of UCP, PIPP and PriSM, and again perform a case study of a workload where each scheme performs poorly.

5.3.2 UCP performance analysis

In the quad core workloads, UCP over all has very good performance. It is by far the most consistent performer, and has the highest performance in 74 of the 100 workloads. Especially in the workloads where LRU has poor performance, UCP is able to improve on this significantly. Another important factor is that it rarely decreases performance with regards to LRU. This indicates that it might be a viable policy that works well for a broader range of workloads, not just a specific type of benchmark combinations.

5.3.2.1 Case study: Workload 4H-41

We perform a deeper analysis of 4H-41 (Figure 5.15), one of the few workloads where UCP performs worse than LRU. This workload consists of astar, perlbench, mcf and bzip2. These are all high-intensity workloads, accessing the L2 frequently. Bzip2 and astar benefit from getting large cache allocations, and so does perlbench although to a slightly lesser degree. Mcf is a stream-like application, with little benefit for extra cache space despite its high cache access rate.

The allocation graph is shown in Figure 5.16. Mcf gets low amounts of ways allocated, as expected for a stream-like application. Perlbench also maintains a fairly steady allocation, averaging 8 ways. The varying factor in the allocation is the relationship between bzip2 and astar. This is the same situation encountered in the dual core situation with these two benchmarks.

The allocation is calculated from the hit distribution among the applications. UCP uses the recency hit counters to optimize the number of ways allocated to each core. The sum of the hit counters in each core is shown in Figure 5.17. The changing behavior of bzip2 is clearly visible, changing between high and low cache utility. Perlbench also has variations in its number of hits, while mcf and astar are relatively stable. These values are sampled at an interval of 5 million cycles, the default UCP

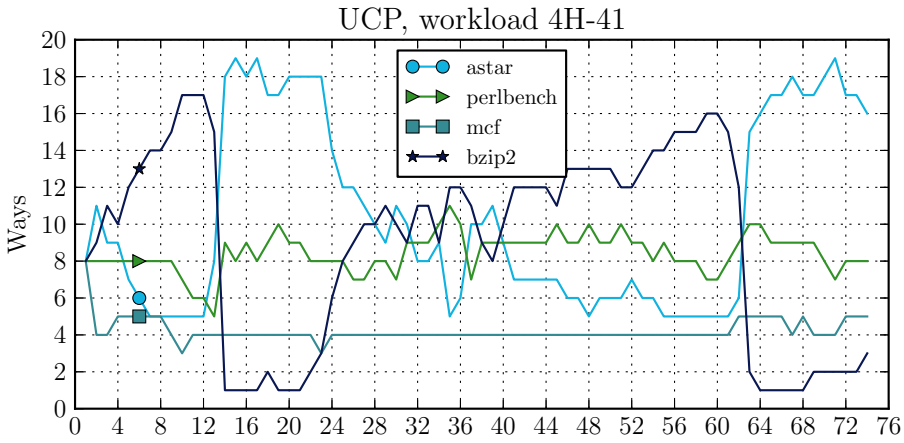

Figure 5.16: The allocation graph for 4H-41, for each partition point during simulation.

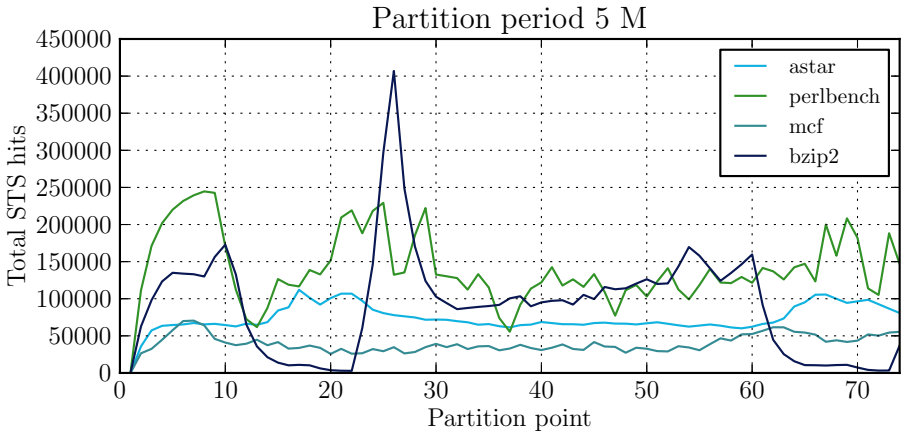

Figure 5.17: STS hit counters for each core with a partition period of 5 million cycles.

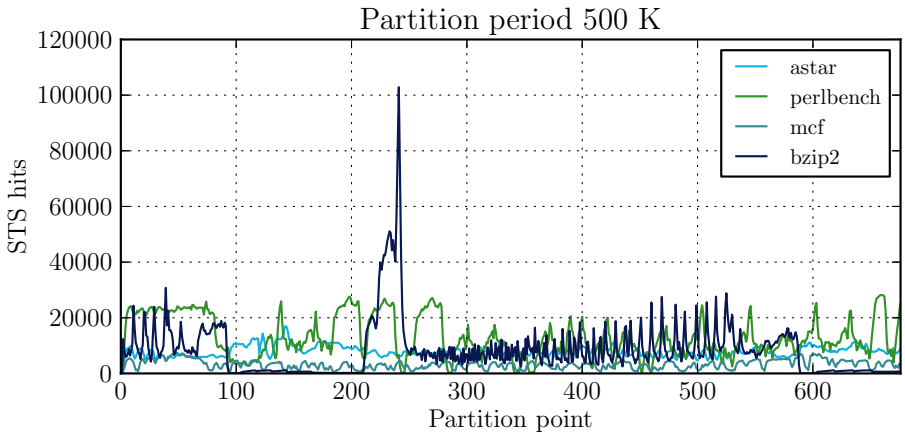

Figure 5.18: STS hit counters for each core with a partition period of 5 million cycles.

partition interval. From earlier experimentation we know that this interval could mask more high frequency changes in behavior of the benchmarks.

By reducing the partition interval, UCP is able to more accurately estimate the current utilization of the cache for each application. The cache can be repartitioned to account for changes in behavior that only last for short periods of time. This allows for better utilization of the cache. Figure 5.18 shows the STS hits when using 500,000 cycles as the partition interval. This finer grained monitoring of the applications reveals dramatic shifts in the cache utilization that goes unnoticed with larger intervals. Bzip2 can be seen to have a high frequency high/low pattern, which we discussed earlier in the dual core experiments. Perlbench also has a similar pattern, with a longer period. In fact the period is almost so long that it shows with a 5 million cycle interval. However, the effect is much clearer with a shorter partitioning interval.

Reducing the partition interval improves the performance and brings it closer to LRU, as can be seen in Figure 5.19. The two shortest intervals, 50,000 and 100,000 cycles, perform slightly better than LRU, but the difference is tiny. At 50,000 cycles, the WS of UCP is 3.2995 while LRU has 3.2968. However the trend is clear, UCP suffers in performance due to quickly changing behavior patterns that it does not respond to quickly enough. In a quad core configuration, this becomes even more of an issue as several applications will have changing behaviors, constantly changing the optimal distribution of the cache. In these scenarios, LRU will be a superior choice, as it quickly responds to increased or decreased activity. The effects of interference between the cores are then less of an issue than allocating sufficient amounts of data in the cache for short periods of time.

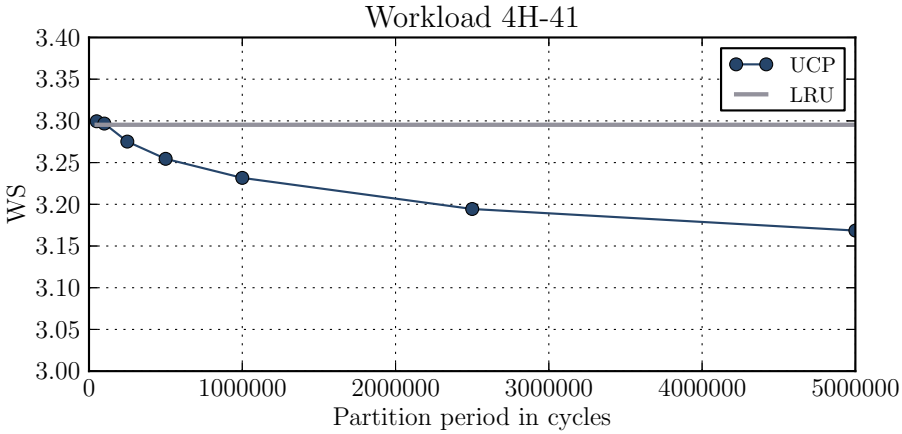

Figure 5.19: Weighted speedup of workload 4H-41 with UCP and different partition periods.

5.3.3 PIPP performance analysis

The performance of PIPP is on average slightly worse than LRU. The average WS is 1.1 % lower for PIPP than LRU, when looking at all the quad core workloads. PIPP is a pseudo-partitioning scheme, and therefore does not give strict guarantees that the target cache occupation is achieved, or even approached. Analysis of the benchmarks where PIPP has poor performance reveals that the target occupation is not achieved. The reasons behind this are complex, but the combination of insertion and promotion policies does not ensure a good allocation in many of the workloads. We study a specific workload to get a deeper understanding of what happens.

An interesting pattern can be seen in many of the workloads where PIPP performs poorly. These workloads often contain the benchmark leslie3d. From our benchmark profiling we know that leslie3d is a low sensitivity benchmark with a high cache intensity, similar to that of a streaming application. But this is an overly simplistic image of leslie3d. It can utilize cache at times during its execution, especially as it goes past the 100 million instruction mark. And the sheer number of L2 accesses means that it will often get a large cache allocation when other benchmarks are not able to utilize the cache more efficiently. This large number of accesses means that it often goes beyond the target allocation when PIPP is used, hurting the performance of other benchmarks. This was observed in the dual core case study of PIPP, and will be seen again in the next section.

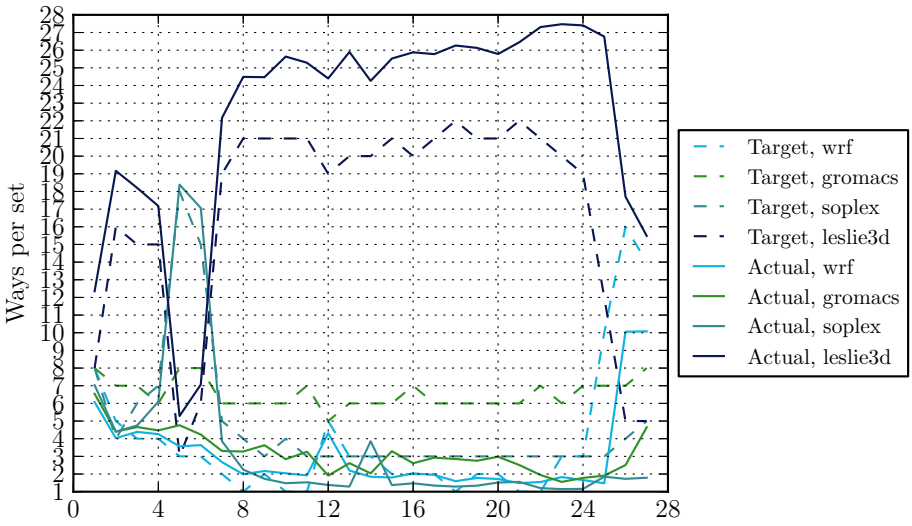

Figure 5.20: Target allocation and actual occupation of the cache in workload 4A-28, using PIPP.

5.3.3.1 Case study: Workload 4A-28

There are many interesting workloads that may be analyzed, in this section we take a look at 4A-28 (Figure 5.15). This workload consists of wrf, gromacs, soplex and leslie3d. It differs from many other workloads by having low performance for PIPP, while UCP and LRU performs well. The WS of PIPP is 7.5 % lower than that of LRU, a significant difference.

The target allocation for the cache is calculated by the Lookahead Algorithm, based on data from the Shadow Tag Store. The target should then be achieved during the next partition period of 5 million cycles. Some variation is expected from set to set, this helps deal with unevenly distributed access patterns and is a strength of PIPP compared to a stricter partitioning scheme. However the average occupation should be close to the target to keep the performance as high as possible. Figure 5.20 shows the target number of ways per set as dashed lines, and the corresponding applications actual occupation as solid lines. Leslie3d outgrows its target partition quite significantly, which causes in particular gromacs to get a lower occupation than its target. This reduces gromacs' performance and impacts the total system throughput.

It is worth noting that none of these benchmarks trigger the stream detector in PIPP. The miss rate on each benchmark are all lower than 11 % for the entire simulation.

The average deviation from the target occupancy varies between the benchmarks. Wrf and soplex are fairly close to their allocations, with wrf getting 0.67 ways and soplex 1.08 ways less per set on average. Gromacs gets 3.56 ways fewer than desired, while leslie3d takes 5.31 ways more than intended. This means leslie3d takes 16.6 % more of the cache than the optimal situation. This is the cause of the lower performance of PIPP, the inability to control leslie3ds occupation.

5.3.4 PriSM performance analysis

PriSMs performance is the poorest of the evaluated cache schemes in the quad core simulations. Particularly noteworthy is its large variation in performance. While it in many cases performs as good as the other schemes, in other workloads it performs significantly worse, up to 16 % performance loss compared to LRU in workload 4A-20 (Figure 5.15).

The main cause for PriSMs performance is how it manages the target allocation. The use of eviction policies to maintain and change the cache occupation does not work in a large number of the workloads. Applications outgrow their desired amount of cache space, or do not achieve the target occupation despite the eviction policies. As the cache contents slide further away from the optimal, the performance is reduced.

5.3.4.1 Case Study: Workload 4H-36

Workload 4H-36 (Figure 5.15) consists of perlbench, mcf, bzip2 and libquantum. The WS of this workload when using PriSM is 2.69, compared to LRUs 3.21 and UCPs 3.59. This is a significant decrease in performance, dropping 16 % compared to LRU.

Figure 5.21 shows the cache occupation for each partition step, as well as the next target allocation given by PriSMs hit maximization algorithm. From this graph, it is immediately apparent why this workload has low performance. From the benchmark profiling we know that libquantum is a high intensity stream-like application with very little use for cache space. Despite this, it manages to occupy around 80-90 % of the cache for the first half the execution. This is very detrimental to the performance of this workload, as large amounts of cache space is now unavailable for the rest of the applications.

As the cache occupation of a core increases, so does the chance of it to maintain a large target allocation in the next partition period. This has the unfortunate strengthening effect of keeping leslie3ds occupation high, even though it has very low potential gain. The target occupation is calculated by the formula

$$T_{core} = C_{core} \times (1 + PotentialGain_{core}/TotalGain)$$

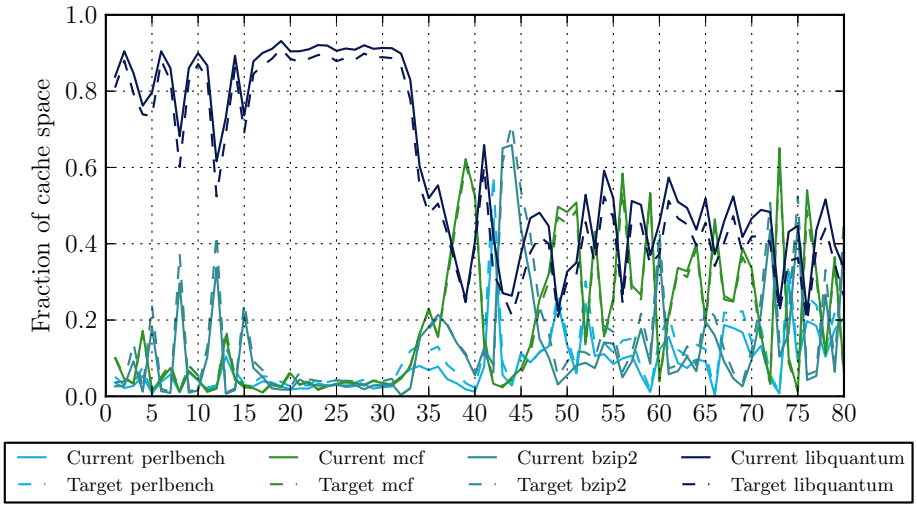

Figure 5.21: Current and target allocations for each partition point, workload 4H-36.

where C_{core} is the current occupation of the cache for this core. As one cores occupation goes towards 1, the other cores occupation goes towards 0. This reduces their target allocation for the next partition period, and the current occupation fraction is likely to remain as it is. The same effect was visible in the dual core experiments, where we explored the subject further.

To ensure that the low performance is not just due to bad sampling like we saw in our analysis of UCP, we attempted this simulation with a wider range of partition interval. In PriSM, the partition interval is defined in number of misses, making it respond quicker to changes in program characteristics that would increase the hit rate. Changing the partition interval did not change the performance significantly, in particular compared to the much better performance of UCP and LRU. Figure 5.22 shows the WS of PriSM for different partition periods, and LRU performance for reference.

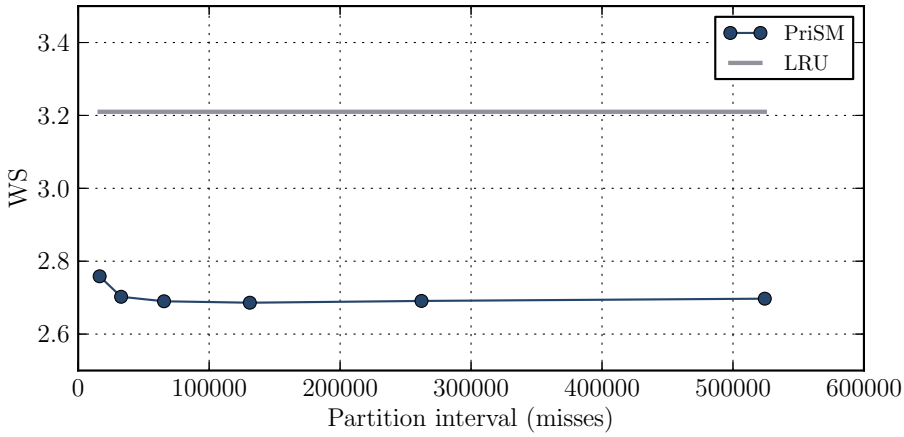

Figure 5.22: WS of PriSM for different partition intervals. Intervals for PriSM are given in number of misses.

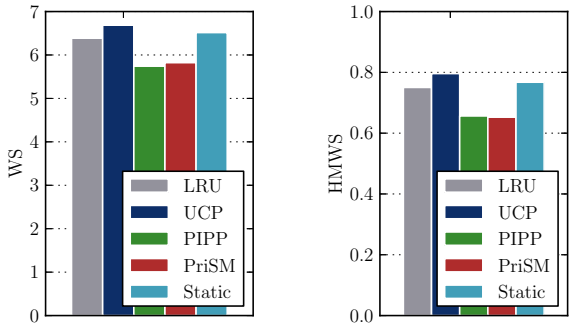

Figure 5.23: Arithmetic mean for WS, HMWS and ANTT for 8 core workloads. Note that ANTT is a smaller-is-better metric.

5.4 8 core

5.4.1 Performance overview

Unfortunately, our 8 core simulations was considerably more demanding than anticipated. Uneven workloads meant that there was significant difference in execution pace of the benchmarks in each workload. The difference increases the number of instructions that needs to be simulated, preventing many of the workloads from finishing even when simulating up to 5 days. 24 of 125 simulations did not complete, causing only 17 of the 25 workloads to complete simulation for every scheme. Extra unfortunate is that the failing simulations are likely to be outlier cases with poor or uneven performance, making them more interesting than those that do complete. Therefore the results of the 8 core experiments are incomplete and may not provide a correct picture of the actual performance for all cases. In the evaluation that follows, we only include the 17 of 25 workloads that completed for all cache schemes, to prevent further bias of the results.

Figure 5.23 shows the average performance values for the schemes in the 8 core configuration. There is an increasing difference between the schemes compared to what we have seen with lower core counts, and UCP again has the best performance. UCP increases the WS compared to LRU by 5.0 %.

PIPPs performance is comparably lower for 8 cores than it was 4. As PIPP uses the allocated number of cache ways as the insertion position in the cache, the average insertion position becomes lower as the number of cores increase. This means more insertions closer to LRU, where blocks will be quickly evicted again if they are not reused within a short period of time. This is a well known issue with PIPP as we mentioned in Chapter 2, making it unsuitable for higher core counts. Ironically, a streaming application in an 8 core 32-way configuration will have a higher insertion position than the average non-streaming application. PIPP inserts

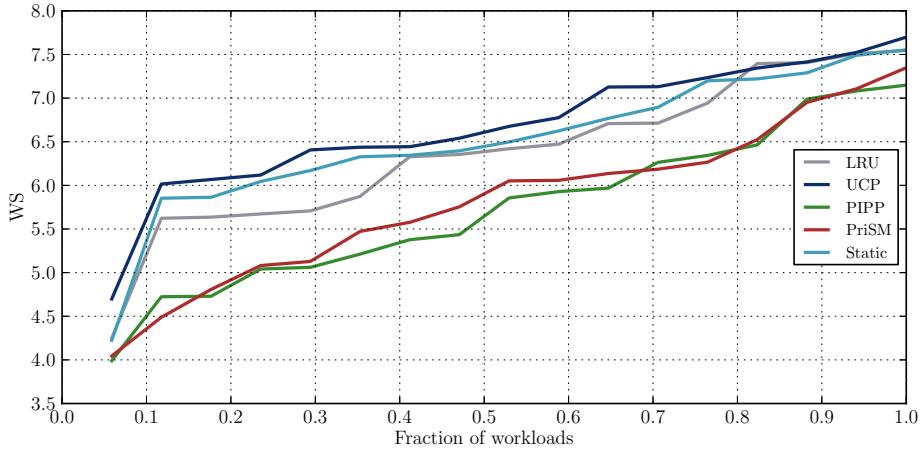

Figure 5.24: WS of 8 core workloads, in ascending order.

streaming applications blocks in the position equivalent to the core count, 8 in this case. While the average allocation (which is used as the insertion position) for each core is $32/8 = 4$ for a non-streaming application. PIPP does however reduce the promotion probability for streaming applications.

PriSM has similarly poor performance as PIPP, reducing the WS by 8.8 % compared to LRU. It suffers from the similar issues we have seen in the previous case studies, being unable to properly maintain a target occupation and then setting the new allocations to less than optimal amounts.

Static partitioning is performing (surprisingly) very well in our 8 core simulations. It improves on LRUs WS by 2 % on average, and by 8.1 % in its best workload. This tells us that destructive interference in LRU is an issue for 8 core workloads, to such an extent that benchmarks are better off having their available cache space reduced to 1/8th rather than share the cache in an unmanaged fashion.

Figure 5.24 shows the WS distribution for the workloads. Compared to the 4 core results (Figure 5.14 in the previous section), PIPP now has a lower performance, while the relative difference between LRU, UCP and PriSM remains largely the same. Static partition improves its performance somewhat compared to the dual and quad core experiments.

A full set of results for all workloads, including those that are incomplete, are shown in Appendix B, Section B.3. We do not perform any case studies for the 8 core results. The performance differences are caused by the same reasons as we have explored for dual and quad core in previous sections, only exaggerated by the increasing number of benchmarks per workload.

Chapter 6

Discussion

In this chapter we present a discussion of various topics from this work, as well as discussing the results. We attempt to take a critical look at our own methodology and the challenges we faced, and take a higher level look at our findings.

6.1 Selecting benchmarks and workloads

The workloads and benchmarks that are used to obtain results for a cache management scheme can have significant impact on the observed performance. We chose the SPEC2006 benchmark suite due to its availability. Once this was determined, the challenge was to cross-compile the benchmarks for ARM, statically linking the libraries and ensuring that the benchmarks ran on the simulator. Only 20 of the 29 SPEC2006 benchmarks was ultimately used. This is a rather small benchmark sample, ideally one would like it to be bigger. The only benefit from having a small number of benchmarks was that it simplified the workload analysis, as we had multiple workloads where each benchmark was present.

In addition, one would like to combine benchmarks into workloads in an unbiased and fair manner. Unfortunately, objectively selecting workloads is near impossible. Ideally, you would like a set of workloads that are representable for the majority of uses. But programs span a massive range of types, with very different characteristics, even within a single program. There is no such thing as a typical program or a typical execution. Instead one should try to test as broad a range of programs as possible, in different workload combinations, to determine strengths and weaknesses of a system. A larger range of workloads will give more confidence in a schemes performance for all workloads, as long as the workloads are diverse.

In our work, we have attempted to avoid selectively picking benchmarks and workloads. We did however have to create a set of workloads taken only from the top

half of benchmarks when ordered by cache demand. This had to be done to avoid having workloads where the majority of the benchmarks have little or no use of the cache, something that would not produce any difference between the cache schemes. Despite this, the same patterns can be seen in the workloads selected from all benchmarks as in the workloads selected from the high-demand benchmarks, only with lower differences between the schemes. We believe this validates our method, at least given the benchmarks available.

6.2 Limiting simulation to 8 cores

In this work we limited our multicore experiments to 8 cores, with the main focus being on dual and quad core simulations. This is done for two reasons, applicability and simulation time. First, high performance CPUs today are currently using 4 cores as the most typical configuration, i. e. the Intel i7 series [6] or the Qualcomm Snapdragon 600 series[3]. The validity of this argument can of course be debated, e. g. in the context of hyper-threading [5] or server CPUs which tend to have higher core counts.

Second, and most importantly for this work, simulation time affects the number of cores that we can simulate within a reasonable amount of time. Multicore simulations that combine low and high IPC applications will run for long periods of time before the lowest IPC application reaches its target number of instructions. This effect gets worse as the number of cores increase. With four cores, one needs to simulate at least 400 million instructions for each of the applications to reach 100 million instructions each.. But in practice we often have to simulate way beyond 1 billion instructions, because one application takes much longer to reach its target. The other applications then have to simulate until the slowest application reaches it target, to simulate contention Our 8 core simulations had even bigger problems, even when significantly reducing the number of simulated instructions. Our memory system was also slightly underspecified for 8 core simulations, extra memory ports would probably improve performance and lessen the unevenness. In the end, all these issues made our 8 core simulations problematic.

6.3 Performance of UCP

The performance we observe for UCP is overall quite good. On average it beats both LRU and the other cache schemes.

UCP is dependent on a high number of ways relative to the number of cores. This is due to its coarse grained approach to partitioning. Each core will always be assigned at least 1 way, and the remaining ways can then be distributed between the cores. The more ways available the finer the partitioning can be. PIPP and PriSM have claimed that this coarse grained approach is UCPs biggest challenge

[23, 16]. Our findings indicate that it is rather the lack of quick response to changes in the applications behavior that is holding UCP back, as we looked at in the case studies. In some workloads this will make LRU perform better, although UCP still beats it on average.

UCPs enforcement of the partitioning is one of its biggest strengths. When the new allocations are set, the cache will have a different occupancy than the target allocation. But for each replacement, UCP goes towards the target partitioning, by evicting blocks from cores that exceed their quota. Unlike PIPP and PriSM, a cores occupancy will never increase further when it exceeds its target quota. And similarly, a core that has too few blocks in the cache can only gain blocks, never lose them. UCP thus has the nice property of always converging on the target partitioning, and never going further away from it.

6.4 Performance of PIPP

PIPPs performance is on par with that of LRU, but is on average beaten by UCP.

In our results when using PIPP, we observe that the actual cache occupancy does not follow the target allocation closely. In the article detailing PIPP there are a large number of variants of the algorithm [23]. These include changing the probabilities for promotion, stream detection and distance of promotion. In our work we have used the values considered optimal in the original work, assuming that these would produce the best results. Some sensitivity analysis have been performed, but there are variants that are still unexplored. It is therefore plausible that by tweaking the algorithm one could get a closer cache occupancy to the target allocation than we were able to achieve. Using different promotion strategies and probabilities will change the behavior of PIPP, possibly towards better occupation control. This would in turn lead to better performance of PIPP, and thus explain the good results that the original article shows.

Despite this potential improvement, PIPP does respond poorly to large changes in target allocation. The insertion/promotion strategy does not allow any selective evictions of cores that are way outside their allocation, and this leads to slow changes in the occupancy of the cache. We are unable to reproduce the good results given in the original article. And unlike UCP, PIPP does not strictly converge towards the target allocation, an unfortunate chain of accesses could lead towards an incorrect occupation of the cache, regardless of which insertion and promotion policies are being used.

6.5 Performance of PriSM

The performance observed from PriSM is the lowest of the tested schemes. This is somewhat disappointing, especially considering the claims they made in the original

paper [16].

Looking more closely at the original paper reveals a few interesting pieces of information. Quad core results are the lowest core counts they are testing, where they report "[*PriSM] performs as well as them [UCP and PIPP] in the quad core scenario*". In many of the actual workloads presented in the paper, the performance of UCP and PIPP is as good as PriSM, and at times better. Taking the average of these graphs reveals that PIPP is the best scheme for quad core. UCP has some poor performance in a few of the workloads presented, which is hard to explain without access to the raw data they used.

PriSM [16] presents performance measurements of a 32 core system, where they have significant improvements over LRU, UCP and PIPP. A 64-way cache is used, and this is perhaps the most limiting factor for UCP. With 64 ways and 32 cores, half the available ways are already assigned before each partition, as each application needs a minimum of 1 line. This almost reduces UCP to a static way-partitioning. Similarly for PIPP, with 32 cores sharing 64 ways, the average insertion position would be just 1 spot above LRU. Two misses or a promotion followed by a miss would then be sufficient to evict a newly inserted line. This is an admitted flaw with PIPP at high core counts. Last, LRU does not perform well at this high amount of cache contention, as we can see from our own 8 core simulations. As we have only simulated up to 8 cores in this work, we have not been able to test PriSMs claims. However, the competition is not very strong for these high core counts.

Regardless, our results unfortunately show that PriSM does not manage to keep its target cache occupation, and that it slowly slips further away over time. This leads to poor performance as the cache space is not used efficiently.

Chapter 7

Conclusion

7.1 Conclusion

In this work we have studied the relative performance of several suggested shared cache management schemes. We have compared them to LRU and static way-based partitioning using a well-defined simulation methodology.

In our research questions, we asked how much LRU degrades performance when used in a multicore setting. From our results we observe that destructive interference is an issue, making other management schemes much better in many workload scenarios. The average improvement on LRU for the best algorithm (UCP) ranges between 0 to 5 %, increasing with the number of cores. The interference is highest when high-demand and low-utility applications are paired with high-utility applications, causing contention over the available cache space. However, many workloads perform well when using LRU, as there are no limitations on cache space available or lag from a partitioning scheme.

We also asked how well each of the schemes performed in comparison to LRU and static partitioning, and what their limitations were. UCP performs well in our experiments, outperforming all the other cache schemes as well as LRU. We have looked at its strengths and weaknesses, and identified the types of workloads where it works poorly. UCP struggles with workloads that have rapidly shifting access patterns, but performs very well with stable cache demands and applications with different cache utilization.

PIPP performs decently, but is not able to reach the performance of UCP. Its pseudo-partitioning of the cache helps it with shifting access patterns, but prevents it from getting the peak performance. Our implementation of PIPP does not get as close to the target allocation as it should, which reduces its performance. And as the number of cores grow, the effectiveness of PIPP is reduced as insertions are being performed closer to the least recently used position.

PriSM did not manage to compete with the other schemes, falling behind in performance. It is not able to reach the target occupation of the cache, and is unable to recover once it starts slipping away from the optimal occupancy. This reduces its performance significantly compared to the other schemes, and gives it the poorest performance in our experiments.

Finally we were tasked with assessing the simulation methodology used, a topic we discussed in Chapter 6. Our methodology has some flaws, and suffers from a tight schedule and imperfections along the way. Problems such as a small set of benchmarks, inaccurate component specifications and some incomplete simulations does influence the results. We attempt to maintain the credibility of the results by exposing these flaws to the reader. Overall we are mostly happy with the results and the methodology used, and believe this reflects the true performance of the various cache management schemes.

7.2 Future work

It would be desirable to have looked at Vantage in this work as well. Unfortunately, mainly due to time constraints, a proper implementation could not be completed. Vantage takes a whole new look at the problems around caches and uses a vastly different type of cache from all the other schemes proposed. This makes it an interesting research target but also more time consuming to implement. PriSM was published after Vantage, and claims better performance than it [16]. If a continuation of this work is to be attempted, Vantage should be high on the list of target cache schemes for testing.

An energy assessment was originally planned for this topic, but was dropped after an earlier project did not manage to get useful estimates of power consumption [10]. This could be interesting to pursue, to evaluate which scheme is the most energy efficient. The overhead of each cache scheme would then perhaps come more into play, as would the energy effects from performance and slowdowns.

Improved simulation for 8 cores and simulation of higher number of cores would also be beneficial. In particular for PriSM, which claims significantly better performance than the other schemes at very high core counts.

Bibliography

[1] The gem5 simulator system. http://www.m5sim.org/.

[2] Notur - the norwegian metacenter for computational science. http://www.notur.no/.

[3] Qualcomm snapdragon 600. http://www.qualcomm.com/snapdragon/processors/600, 2012.

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

[5] Intel Corp. Intel hyper-threading technology. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[6] Intel Corporation. Intel core i7-2600k processor. http://ark.intel.com/products/52214/Intel-Core-i7-2600K-Processor-8M-Cache-up-to-3_80-GHz.

[7] Standard Performance Evaluation Corporation. Spec cpu2006. http://www.spec.org/cpu2006/, 2011.

[8] Haakon Dybdahl, Per Stenstrom, and Lasse Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *Proceedings of the 13th international conference on High Performance Computing*, HiPC'06, pages 22–34, Berlin, Heidelberg, 2006. Springer-Verlag.

[9] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008.

[10] Christian Vik Groevdal. Towards an infrastructure for evaluating energy efficiency in cache systems. 2012.

[11] Magnus Jahre. *Managing Shared Resources in Chip Multiprocessor Memory Systems*. PhD thesis, Norwegian University of Science and Technology, 2010.

[12] David A. Patterson John L. Hennessy. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 2011.

[13] David A. Patterson John L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. 2011.

[14] HP Labs. Cacti, an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. `http://www.hpl.hp.com/research/cacti/`.

[15] ARM Ltd. Arm annual report 2011. `http://ir.arm.com/phoenix.zhtml?c=197211&p=irol-reportsannual`.

[16] R Manikantan, Kaushik Rajan, and R Govindarajan. Probabilistic shared cache management (prism). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 428–439, Washington, DC, USA, 2012. IEEE Computer Society.

[17] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. A case for mlp-aware cache replacement. *SIGARCH Comput. Archit. News*, 34(2):167–178, May 2006.

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

[19] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 298–, Washington, DC, USA, 1997. IEEE Computer Society.

[20] Daniel Sanchez and Christos Kozyrakis. The zcache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 187–198, Washington, DC, USA, 2010. IEEE Computer Society.

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

[22] Norway University of Tromsoe. Stallo compute cluster. `http://www.notur.no/hardware/stallo/`.

[23] Yuejian Xie and Gabriel H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM.

Appendix A

Workloads

This appendix lists the workloads used in simulation for dual core, quad core and 8 core experiments.

A.1 Dual core

Table A.1: Dual core workloads from the top 10 benchmarks.

Workload ID	Benchmark 1	Benchmark 2
2H-1	gobmk	soplex
2H-2	libquantum	soplex
2H-3	bzip2	soplex
2H-4	omnetpp	soplex
2H-5	mcf	soplex
2H-6	soplex	xalancbmk
2H-7	perlbench	soplex
2H-8	leslie3d	soplex
2H-9	astar	soplex
2H-10	libquantum	gobmk
2H-11	gobmk	bzip2
2H-12	omnetpp	gobmk
2H-13	mcf	gobmk

2H-14	xalancbmk	gobmk
2H-15	perlbench	gobmk
2H-16	leslie3d	gobmk
2H-17	astar	gobmk
2H-18	bzip2	libquantum
2H-19	omnetpp	libquantum
2H-20	mcf	libquantum
2H-21	xalancbmk	libquantum
2H-22	perlbench	libquantum
2H-23	leslie3d	libquantum
2H-24	astar	libquantum
2H-25	omnetpp	bzip2
2H-26	mcf	bzip2
2H-27	xalancbmk	bzip2
2H-28	perlbench	bzip2
2H-29	leslie3d	bzip2
2H-30	astar	bzip2
2H-31	omnetpp	mcf
2H-32	xalancbmk	omnetpp
2H-33	perlbench	omnetpp
2H-34	leslie3d	omnetpp
2H-35	astar	omnetpp
2H-36	xalancbmk	mcf
2H-37	perlbench	mcf
2H-38	leslie3d	mcf
2H-39	mcf	astar
2H-40	xalancbmk	perlbench
2H-41	leslie3d	xalancbmk
2H-42	astar	xalancbmk
2H-43	leslie3d	perlbench
2H-44	astar	perlbench
2H-45	astar	leslie3d

Table A.2: Dual core workloads from all benchmarks.

Workload ID	Benchmark 1	Benchmark 2
2A-1	mcf	gromacs
2A-2	wrf	dealII
2A-3	astar	h264ref
2A-4	gobmk	namd
2A-5	omnetpp	namd
2A-6	bzip2	wrf
2A-7	gromacs	milc
2A-8	namd	dealII
2A-9	astar	hmmer
2A-10	leslie3d	h264ref
2A-11	gobmk	gromacs
2A-12	mcf	hmmer
2A-13	mcf	h264ref
2A-14	gobmk	milc
2A-15	namd	milc
2A-16	bzip2	gromacs
2A-17	xalancbmk	wrf
2A-18	omnetpp	dealII
2A-19	gromacs	sjeng
2A-20	hmmer	sjeng
2A-21	povray	dealII
2A-22	gromacs	namd
2A-23	leslie3d	milc
2A-24	leslie3d	wrf
2A-25	libquantum	h264ref
2A-26	libquantum	sjeng
2A-27	xalancbmk	hmmer

2A-28	perlbench	milc
2A-29	wrf	namd
2A-30	astar	povray
2A-31	sjeng	povray
2A-32	povray	wrf
2A-33	libquantum	bwaves
2A-34	omnetpp	sjeng
2A-35	libquantum	dealII
2A-36	namd	bwaves
2A-37	perlbench	sjeng
2A-38	xalancbmk	dealII
2A-39	leslie3d	bwaves
2A-40	bzip2	hmmer
2A-41	hmmer	h264ref
2A-42	hmmer	milc
2A-43	mcf	wrf
2A-44	wrf	bwaves
2A-45	gobmk	bwaves
2A-46	libquantum	wrf
2A-47	dealII	milc
2A-48	hmmer	namd
2A-49	perlbench	h264ref
2A-50	gromacs	omnetpp

A.2 Quad core

Table A.3: 4 core workloads based on the top 10 benchmarks in terms of L2 accesses. These are generated from the top 10 benchmarks in terms of number of L2 accesses, by creating all possible combinations and randomly selecting 50.

Workload ID	Benchmark 1	Benchmark 2	Benchmark 3	Benchmark 4
4H-1	leslie3d	perlbench	xalancbmk	omnetpp
4H-2	astar	leslie3d	bzip2	soplex
4H-3	astar	leslie3d	xalancbmk	omnetpp
4H-4	perlbench	xalancbmk	bzip2	soplex
4H-5	astar	mcf	bzip2	soplex
4H-6	leslie3d	perlbench	xalancbmk	gobmk
4H-7	astar	leslie3d	bzip2	libquantum
4H-8	perlbench	xalancbmk	omnetpp	gobmk
4H-9	astar	omnetpp	bzip2	libquantum
4H-10	xalancbmk	omnetpp	libquantum	soplex
4H-11	leslie3d	mcf	omnetpp	gobmk
4H-12	astar	perlbench	libquantum	gobmk
4H-13	astar	leslie3d	omnetpp	gobmk
4H-14	astar	leslie3d	omnetpp	libquantum
4H-15	astar	perlbench	xalancbmk	soplex
4H-16	astar	xalancbmk	omnetpp	libquantum
4H-17	astar	leslie3d	mcf	bzip2
4H-18	mcf	omnetpp	bzip2	libquantum
4H-19	xalancbmk	omnetpp	libquantum	gobmk
4H-20	perlbench	mcf	omnetpp	libquantum
4H-21	astar	xalancbmk	libquantum	soplex
4H-22	perlbench	xalancbmk	libquantum	gobmk
4H-23	astar	perlbench	omnetpp	gobmk
4H-24	perlbench	xalancbmk	gobmk	soplex
4H-25	mcf	omnetpp	bzip2	soplex

4H-26	astar	perlbench	mcf	libquantum
4H-27	leslie3d	omnetpp	gobmk	soplex
4H-28	astar	perlbench	gobmk	soplex
4H-29	perlbench	mcf	bzip2	gobmk
4H-30	leslie3d	perlbench	omnetpp	bzip2
4H-31	leslie3d	bzip2	libquantum	soplex
4H-32	leslie3d	xalancbmk	mcf	omnetpp
4H-33	leslie3d	mcf	bzip2	libquantum
4H-34	astar	mcf	gobmk	soplex
4H-35	mcf	bzip2	gobmk	soplex
4H-36	perlbench	mcf	bzip2	libquantum
4H-37	leslie3d	perlbench	omnetpp	gobmk
4H-38	leslie3d	omnetpp	bzip2	libquantum
4H-39	leslie3d	xalancbmk	mcf	libquantum
4H-40	perlbench	xalancbmk	bzip2	gobmk
4H-41	astar	perlbench	mcf	bzip2
4H-42	astar	omnetpp	bzip2	gobmk
4H-43	leslie3d	mcf	omnetpp	bzip2
4H-44	astar	mcf	bzip2	libquantum
4H-45	astar	leslie3d	perlbench	bzip2
4H-46	astar	xalancbmk	gobmk	soplex
4H-47	xalancbmk	omnetpp	gobmk	soplex
4H-48	perlbench	omnetpp	bzip2	soplex
4H-49	perlbench	mcf	libquantum	soplex
4H-50	astar	omnetpp	libquantum	soplex

Table A.4: 4 core workloads, selected from all benchmarks.

Workload ID	Benchmark 1	Benchmark 2	Benchmark 3	Benchmark 4
4A-1	dealII	namd	sjeng	gromacs
4A-2	milc	dealII	h264ref	libquantum
4A-3	sjeng	h264ref	xalancbmk	leslie3d

4A-4	povray	libquantum	omnetpp	perlbench
4A-5	h264ref	gobmk	mcf	xalancbmk
4A-6	sjeng	gromacs	soplex	bzip2
4A-7	bwaves	hmmer	gobmk	bzip2
4A-8	milc	povray	mcf	perlbench
4A-9	bwaves	milc	sjeng	perlbench
4A-10	dealII	povray	omnetpp	astar
4A-11	dealII	namd	gromacs	astar
4A-12	sjeng	gobmk	xalancbmk	leslie3d
4A-13	povray	libquantum	perlbench	astar
4A-14	milc	wrf	sjeng	leslie3d
4A-15	bwaves	gromacs	libquantum	astar
4A-16	sjeng	hmmer	soplex	xalancbmk
4A-17	dealII	gromacs	soplex	xalancbmk
4A-18	namd	soplex	bzip2	xalancbmk
4A-19	milc	dealII	gromacs	soplex
4A-20	wrf	gromacs	libquantum	perlbench
4A-21	hmmer	libquantum	omnetpp	leslie3d
4A-22	namd	h264ref	mcf	perlbench
4A-23	bwaves	namd	wrf	bzip2
4A-24	milc	sjeng	h264ref	astar
4A-25	milc	dealII	soplex	astar
4A-26	bwaves	sjeng	bzip2	xalancbmk
4A-27	milc	dealII	libquantum	xalancbmk
4A-28	wrf	gromacs	soplex	leslie3d
4A-29	bwaves	hmmer	leslie3d	astar
4A-30	h264ref	mcf	perlbench	leslie3d
4A-31	bwaves	xalancbmk	perlbench	leslie3d
4A-32	dealII	povray	h264ref	gobmk
4A-33	gobmk	bzip2	omnetpp	mcf
4A-34	milc	dealII	namd	gromacs

4A-35	milc	gromacs	gobmk	xalancbmk
4A-36	wrf	gromacs	libquantum	leslie3d
4A-37	dealII	namd	xalancbmk	perlbench
4A-38	namd	hmmer	bzip2	omnetpp
4A-39	bwaves	wrf	gromacs	xalancbmk
4A-40	dealII	wrf	perlbench	leslie3d
4A-41	dealII	sjeng	hmmer	mcf
4A-42	gromacs	gobmk	xalancbmk	astar
4A-43	wrf	povray	h264ref	bzip2
4A-44	gromacs	libquantum	bzip2	astar
4A-45	hmmer	gobmk	bzip2	astar
4A-46	povray	gromacs	gobmk	leslie3d
4A-47	povray	soplex	omnetpp	leslie3d
4A-48	bwaves	hmmer	libquantum	xalancbmk
4A-49	dealII	hmmer	gromacs	libquantum
4A-50	bwaves	dealII	namd	astar

A.3 8 core

Table A.5: 8 core workloads.

Workload ID	Benchmark 1	Benchmark 2	Benchmark 3	Benchmark 4
	Benchmark 5	Benchmark 6	Benchmark 7	Benchmark 8
8A-1	bwaves	dealII	wrf	povray
	hmmer	libquantum	mcf	astar
8A-2	milc	dealII	namd	wrf
	soplex	gobmk	libquantum	perlbench
8A-3	milc	wrf	hmmer	bzip2
	omnetpp	xalancbmk	perlbench	leslie3d
8A-4	milc	namd	povray	hmmer
	gromacs	mcf	xalancbmk	leslie3d
8A-5	dealII	gromacs	soplex	gobmk
	omnetpp	mcf	perlbench	astar

8A-6	bwaves	milc	dealII	wrf
	gromacs	libquantum	xalancbmk	astar
8A-7	bwaves	wrf	sjeng	h264ref
	mcf	xalancbmk	leslie3d	astar
8A-8	dealII	namd	wrf	gromacs
	soplex	mcf	perlbench	leslie3d
8A-9	bwaves	namd	povray	h264ref
	gromacs	soplex	omnetpp	perlbench
8A-10	bwaves	namd	povray	hmmer
	soplex	libquantum	bzip2	perlbench
8A-11	bwaves	sjeng	h264ref	gromacs
	gobmk	libquantum	perlbench	leslie3d
8A-12	milc	hmmer	gromacs	libquantum
	bzip2	omnetpp	mcf	leslie3d
8A-13	bwaves	milc	dealII	povray
	hmmer	gromacs	xalancbmk	perlbench
8A-14	bwaves	dealII	namd	hmmer
	libquantum	bzip2	omnetpp	perlbench
8A-15	povray	hmmer	gromacs	libquantum
	mcf	xalancbmk	perlbench	leslie3d
8A-16	bwaves	sjeng	bzip2	omnetpp
	mcf	xalancbmk	leslie3d	astar
8A-17	wrf	sjeng	hmmer	gromacs
	libquantum	mcf	xalancbmk	leslie3d
8A-18	bwaves	milc	povray	hmmer
	soplex	libquantum	mcf	astar
8A-19	dealII	sjeng	gobmk	libquantum
	omnetpp	mcf	xalancbmk	perlbench
8A-20	dealII	namd	wrf	h264ref
	libquantum	mcf	perlbench	leslie3d
8A-21	dealII	namd	gromacs	gobmk

	omnetpp	mcf	xalancbmk	perlbench
8A-22	bwaves	milc	dealII	wrf
	sjeng	soplex	xalancbmk	leslie3d
8A-23	bwaves	namd	wrf	povray
	gromacs	soplex	leslie3d	astar
8A-24	milc	namd	wrf	hmmer
	gromacs	gobmk	mcf	leslie3d
8A-25	bwaves	milc	dealII	hmmer
	gromacs	gobmk	libquantum	mcf

Appendix B

Simulation results

This appendix presents the detailed simulation results for all workloads. We show the WS of each workload in each cache scheme. Some workloads did unfortunately not complete simulation, this is indicated by missing bars in the figures.

B.1 Dual Core

Here we present the WS of all dual core workloads, ordered by LRU performance.

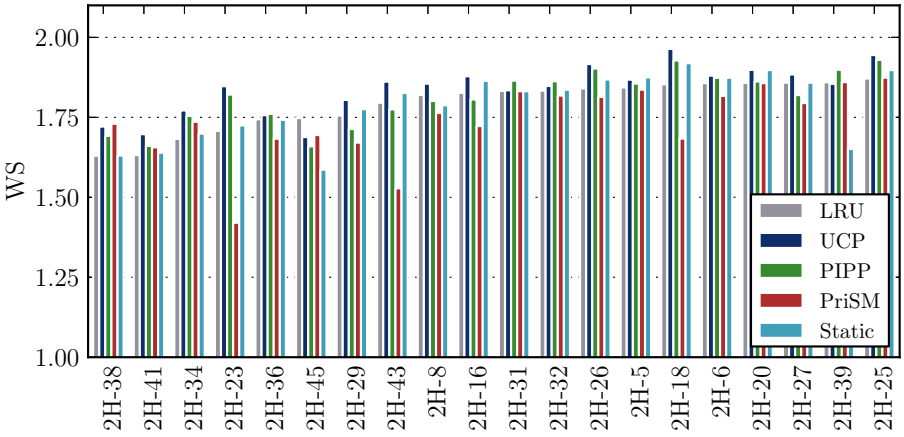

Figure B.1: Dual core workloads, 1-20 of 95.

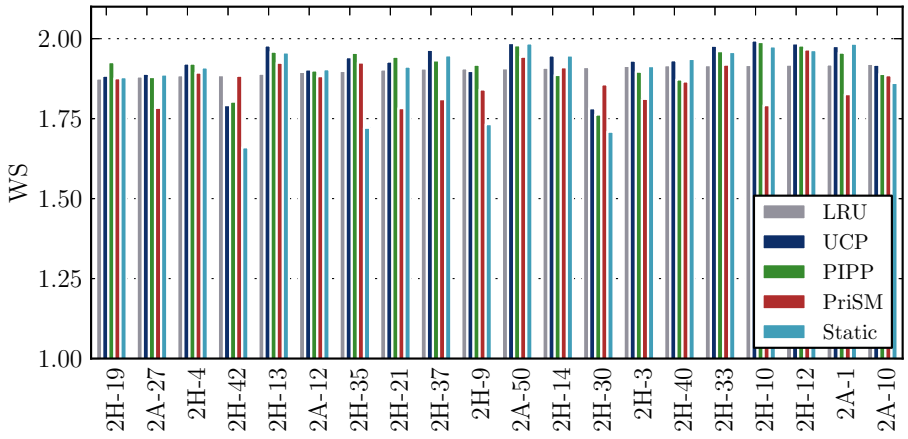

Figure B.2: Dual core workloads, 21-40 of 95.

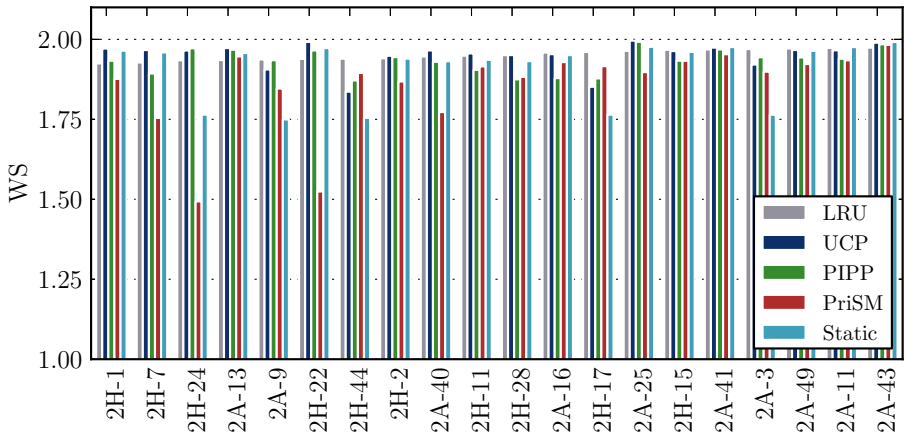

Figure B.3: Dual core workloads, 41-60 of 95.

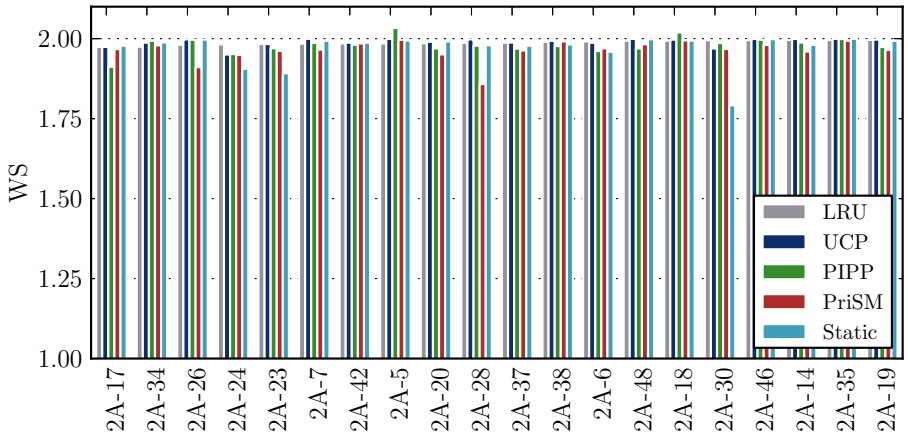

Figure B.4: Dual core workloads, 61-80 of 95.

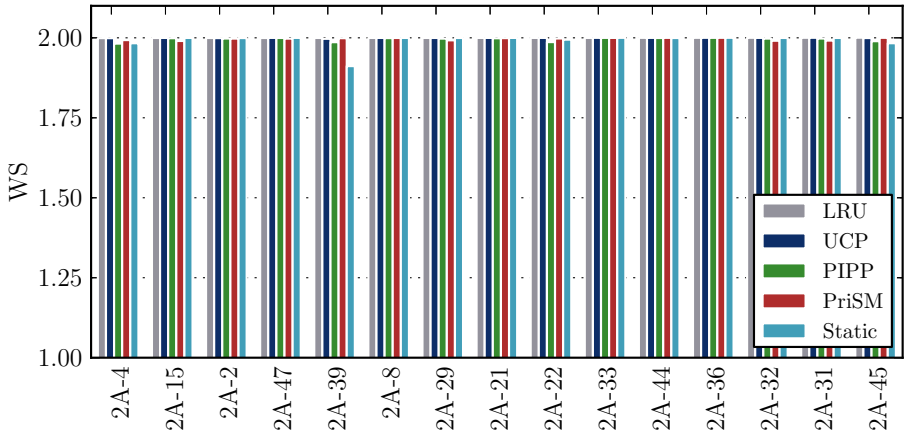

Figure B.5: Dual core workloads, 81-95 of 95.

B.2 Quad Core

Here we present the WS of all quad core workloads, ordered by LRU performance.

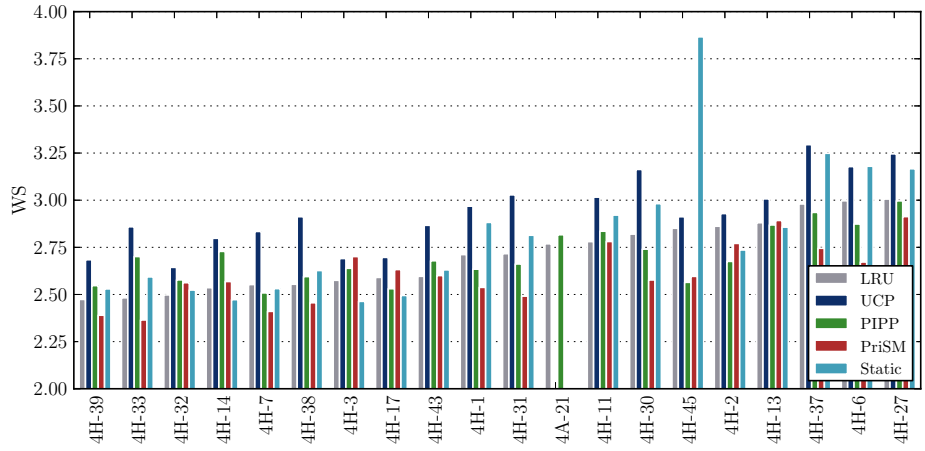

Figure B.6: Quad core workloads, 1-20 of 100.

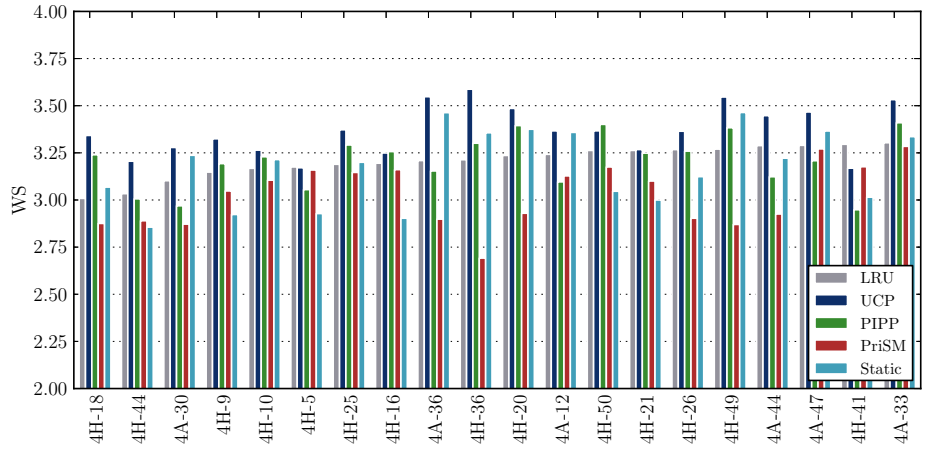

Figure B.7: Quad core workloads, 21-40 of 100.

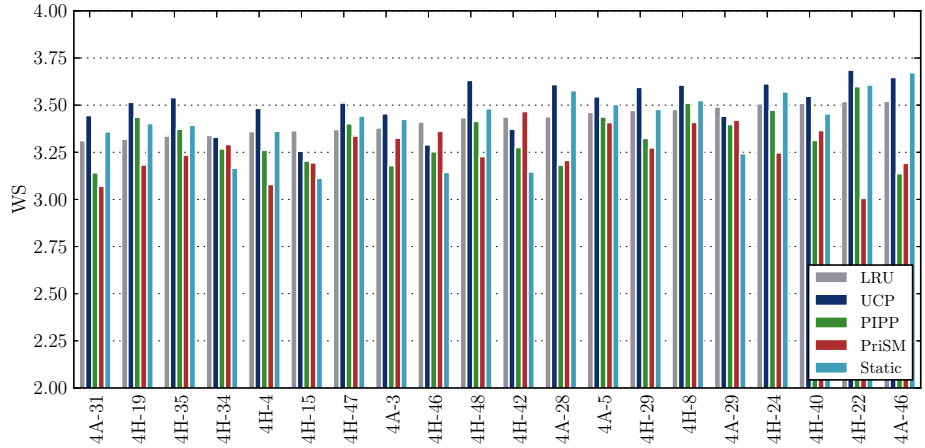

Figure B.8: Quad core workloads, 41-60 of 100.

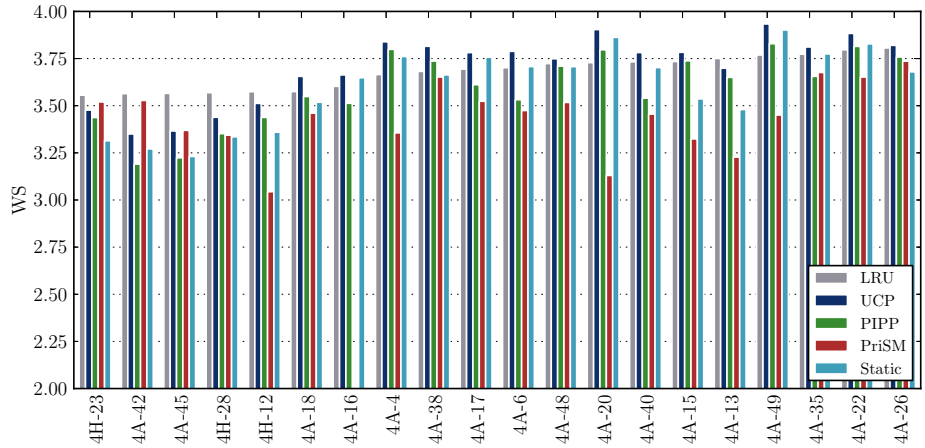

Figure B.9: Quad core workloads, 61-80 of 100.

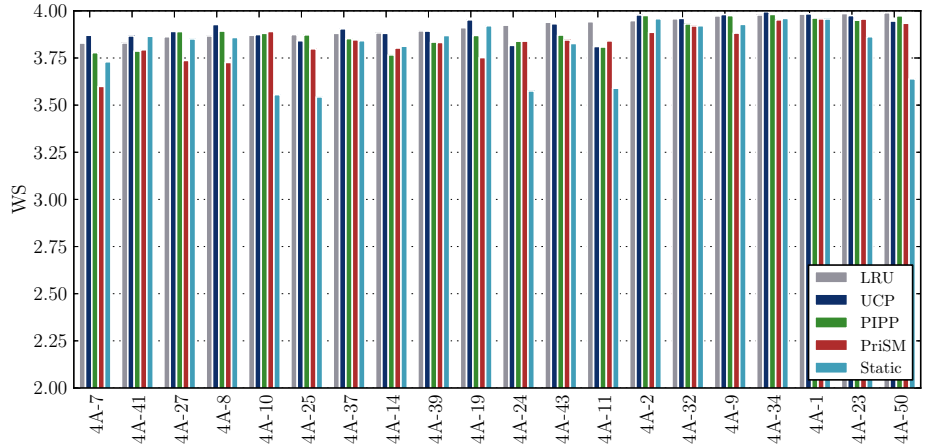

Figure B.10: Quad core workloads, 81-100 of 100.

B.3 8 Core

We present the WS of all 8 core workloads, ordered by LRU performance. Bars are missing in the figures when a workload did not complete for a given scheme.

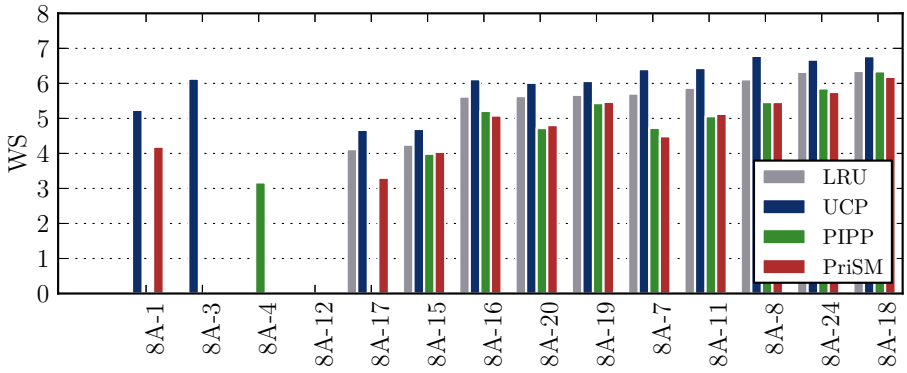

Figure B.11: WS of 8 core workloads, 1-15 of 25.

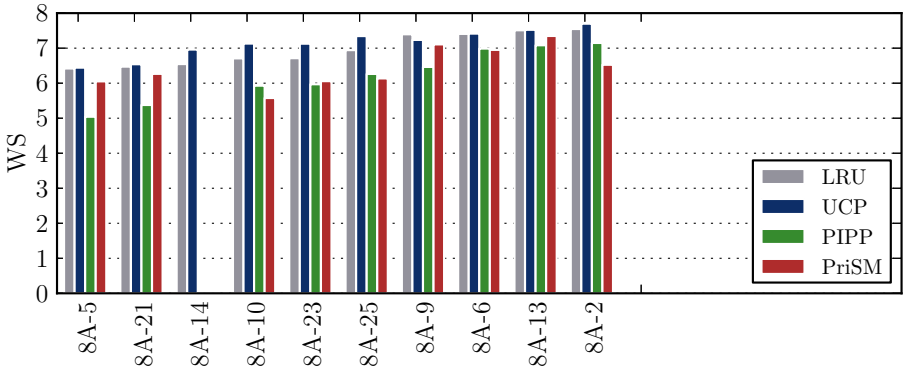

Figure B.12: WS of 8 core workloads, 16-25 of 25.