AN ANALYSIS OF CACHE PARTITIONING TECHNIQUES FOR CHIP MULTIPROCESSOR SYSTEMS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2008

By
Konstantinos Nikas
School of Computer Science

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Currently, there is a trend to increase the number of processors on a single chip leading to the development of chip multiprocessor (CMP), and eventually manycore, architectures. Cache design has been extensively studied in the context of uniprocessor systems and computer architects have transfered existing policies and cache design techniques from uniprocessors to the new architectures. A typical example of such a migration is the employment of the Least Recently Used (LRU) replacement policy, which is widely accepted as the best line replacement policy for uniprocessor caches. However the parameters are different in CMP systems, as the sharing of the cache hierarchy amongst several concurrent threads imposes new constraints and creates new challenges. It is important, therefore, to reevaluate the effectiveness of these policies in CMP architectures.

This thesis investigates the interference between threads that run simultaneously on CMPs sharing different levels of the cache hierarchy and evaluates cache partitioning as a means of alleviating its consequences. Several schemes are studied and their advantages and drawbacks are used as a guide for the development of a novel, low-cost cache partitioning scheme that achieves better performance than LRU and shows increasing promise over alternative schemes as the number of on-chip processors increases.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

This research was completed under the supervision of Dr. Jim Garside to whom I am obliged. I would like to thank him for all the help, encouragement, support and constructive feedback he has given me over the last four years. I would also like to thank all the members of the APT group. Special thanks go to Dr. Jeremy Singer for efficiently proofreading this thesis in record time.

I would also like to thank all my friends, new and old ones. Especially Teta, Dimitris and Nikos for making life here more enjoyable and Aris, Antreas and Ioanna in Greece for their 24/7 phone support.

Finally, and certainly not least, I would like to thank my parents for encouraging me to keep on chasing my dreams, for their continued help and support all these years. I have come this far because of you.

Chapter 1

Introduction

As is well known, the feature size of silicon fabrication technology is continuously shrinking. In fact, in the last decade only, it has been scaled down from 350nm to 45nm [26]. This has enabled the improvement of system performance in two ways. Firstly, designers have been increasing system operating frequency at a rapid rate by exploiting the smaller and faster transistors and by building deeper pipelines, thus reducing the number of levels of logic per cycle. Secondly, the increasing transistor budgets have been utilised by several microarchitectural techniques, like superscalar issue, out-of-order issue, on-chip caching and deep pipelines supported by sophisticated branch predictors [57].

Unfortunately, as has been recently noted [19, 7], the future effectiveness of these approaches is limited due to the emergence of two main constraints. Firstly, the increase in the number of transistors and their switching frequency leads to an overall growth in power consumption and energy. Additionally, it is becoming extremely difficult to remove the dissipated heat from the chip. Secondly, as the feature size is decreased, wire delays do not scale efficiently and have become a major design limitation for large integrated circuits.

These problems have caused a change in the design paradigm of the microprocessor industry. Nowadays Chip Multi-Processors (CMPs) are being developed by all the main vendors such as AMD [3], IBM [29], Intel [40] and Sun [36]. However, the sharing of the on-chip resources amongst the cores imposes new constraints and creates new challenges for designers.

This thesis focuses on the sharing of cache between concurrently running applications and evaluates cache partitioning schemes as a mean of optimising the overall system performance. The remainder of this chapter presents the case for CMP architectures, outlines the cache hierarchies of some of the developed systems and concludes with an overview of the rest of the thesis.

1.1 Chip Multiprocessor Architectures

As mentioned previously, CMP architectures have been developed to provide solutions to the problems facing microprocessor designers. This section presents a short overview of some of these solutions.

1.1.1 Wire Delays

The signal delay for a wire increases in proportion to the product of its resistance and capacitance. As feature size decreases, the wires' section gets smaller and the resistance and capacitance per unit length increase. In the past few years increasingly larger fractions of the clock cycles have been consumed by the propagation delay of signals on wires, reducing the number of logic gates reachable within a cycle.

On the other hand, CMPs incorporate multiple processors on the same chip, with each core occupying a relatively small area. Therefore, the critical paths within each processor circuit are minimised and the effects of wire delays reduced. In these systems only the wires of the communication network between processors and caches need to be long. These are less frequently used compared to wires inside the cores and, therefore, less latency critical.

1.1.2 Limited Parallelism

Many processors are designed to exploit Instruction-Level Parallelism (ILP). More specifically, independent instructions found within an instruction stream are sent to multiple functional units to be executed concurrently. By allowing more than one instruction to be executed per clock cycle, the performance of the system is improved. However, this approach is limited by instruction dependencies and long-latency operations [62].

Thread-Level Parallelism (TLP) offers an attractive alternative to ILP. A thread is defined as a separate process with its own instructions and data. It could either represent an independent program or a process that is part of a parallel program consisting of multiple processes. In contrast to ILP, which exploits parallel operations within a code segment, TLP extracts parallelism by executing multiple threads which are inherently parallel.

Single processor systems have two options to exploit TLP. In "fine-grained multithreading" the execution of multiple threads is interleaved by switching between threads on each instruction. The second option, "coarse-grained multithreading", switches threads on costly stalls, for example L2 cache misses, or at specific time quantums.

CMPs are specifically designed to exploit TLP. In fact, by providing many processors, they can achieve even finer-grained parallelism than single processor systems that exploit TLP, as multiple threads can execute simultaneously.

1.1.3 Design Complexity

Recently Intel presented the Montecito processor [39]. Its 1.7 billion-transistors have signalled the start of the era of billion transistor architectures. However, as the number of transistor rises, the design complexity increases as well. The International Technology Roadmap for the Semiconductor Industry (ITRS) 2005 [1] has recognised that in order to maintain design quality, design implementation productivity must be improved to the same degree as design complexity is scaled.

Design replication offers an attractive solution to the complexity problem and is exploited in CMP designs. For example, Sun's Niagara architecture [36] employs eight identical SPARC processors. The modular nature of these systems allows the designers to reuse components, thus cutting down design and verification time.

1.2 Memory Hierarchy Design Challenges

CMPs have recently emerged as an attractive alternative to uniprocessor systems. However this shift in the design paradigm has created new challenges. One of these is the design of the memory hierarchy. The high number of threads running simultaneously increases the demand on the memory bandwidth. Cacheing helps to alleviate this, but the problems of cache design for CMPs are not yet fully understood.

Firstly, these are shared memory systems and all the processors are assumed to see the same memory space. In practice this is not directly feasible as different levels of the cache hierarchy could be shared amongst different numbers of processors. CMPs could exploit their multiple cores to schedule parallel threads that execute the same instruction stream but on independent data sets, as is the case in parallel programs such as raytracer [56] and several server applications. In a different scenario, the working sets of the parallel threads could overlap, as is the case for databases. Sharing the cache hierarchy is a straightforward solution; however shared caches exhibit a higher access time and the bandwidth requirements of the system are increased. On the other hand, private caches have lower hit latencies but the effective cache capacity of the system is reduced as cache entries are often replicated in multiple private caches.

Clearly there is a trade-off and the designers need to consider each system's requirements and specifications to make wise decisions. The majority of the proposed CMPs today keep the L1 caches private for each processor to avoid increasing their access time and share the next levels of the cache hierarchy. Table 1.1 outlines the memory hierarchies of Niagara [36], Power 5 [29] and Montecito [39] and reveals the different decisions made by the designers regarding cache sizes and associativity and the sharing or not of specific levels of the cache hierarchy.

Another important challenge that the designers of CMPs face is the selection of an efficient cache replacement algorithm. The architectures described in Table 1.1 employ the random, the Least Recently Used (LRU) and the Not Recently Used (NRU) policies. These policies have migrated from the single processor

	Sun Niagara	**IBM Power 5**	**Intel Montecito**	
Processors/chip	8	2	2	
Threads/processor	4	2	2	
Instruction Cache per processor	16KB 4-way	64KB 2-way	16KB 4-way	
Data Cache per processor	8KB 4-way	32KB 4-way	16KB 4-way	
Write Policy	write-through no allocation on writes	write-through	write-through no allocation on writes	
L1 replacement policy	Random	LRU	LRU	
On-chip L2 Cache	shared	shared	private	
	3MB 12-way	1.875MB 10-way	I$ 1MB 8-way	D$ 256KB 8-way
Write Policy	write-back	write-back	write-back	
L2 replacement policy	Random	LRU	NRU	
L3 Cache	-	off-chip 36MB 10-way	on-chip (private) 12MB 6-way	

Table 1.1: Memory hierarchy of developed CMPs

design domain and they are "*thread-blind*", as they select which cache entry should be rejected regardless of which thread brought it into the cache or which thread suffered the cache miss. This, however, can cause problems in a CMP architecture and have a significant impact on the performance of the system.

More specifically, when each core is used to execute a different application, multiple working sets compete for the cache space. The employment of a thread-blind replacement policy allows interference, as data belonging to one thread may be evicted by data blocks belonging to other threads. A typical example is the execution of streaming applications. These applications exhibit a small amount of temporal locality and pollute the caches with data that will not be accessed again in the near future. A thread-blind replacement policy is not able to identify and deal with these pathological cases and severe performance problems such as cache thrashing and thread starvation could be caused.

Designers have tried to deal with this problem by increasing the cache's size

and associativity. However, the effectiveness of this approach is limited due to area and power constraints and other alternatives are needed, highlighting the importance of research in this field.

1.3 Research Aims

In the previous section it became apparent that different factors need to be taken into consideration in order to propose a cache hierarchy that will satisfy all the requirements. As it seems likely that the number of processors integrated on a silicon chip will continue to increase in the near future, policies should scale to accommodate an increasing number of competing threads whilst maximising overall instruction throughput.

This work studies the interference between threads that run simultaneously on CMPs sharing different levels of the cache hierarchy and evaluates cache partitioning as a mean of alleviating its consequences. Different schemes are studied and the results of the analysis are used as a guide to develop a novel scheme that can optimise the overall performance of the system at a low cost in terms of hardware overhead and complexity.

1.4 Terminology

In general this work adopts the terminology that Smith used in his detailed survey of cache design [55]. The following list of definitions clarify some of the terms used throughout this thesis. For convenience, Figure 1.1 shows an example of a 6-way set associative cache illustrating the usage of these terms.

- **Entry/Block** : An entry is a set of words with a common tag, distinguished only by the least significant bits of the memory address. There are typically 4 or 8 words long.
- **Line/Set** : A line or set is a collection of entries, the tags for which are checked in parallel during a cache access.
- **Way** : The number of ways is defined by the “degree of associativity” of the cache and is equal to the number of entries contained in a line.

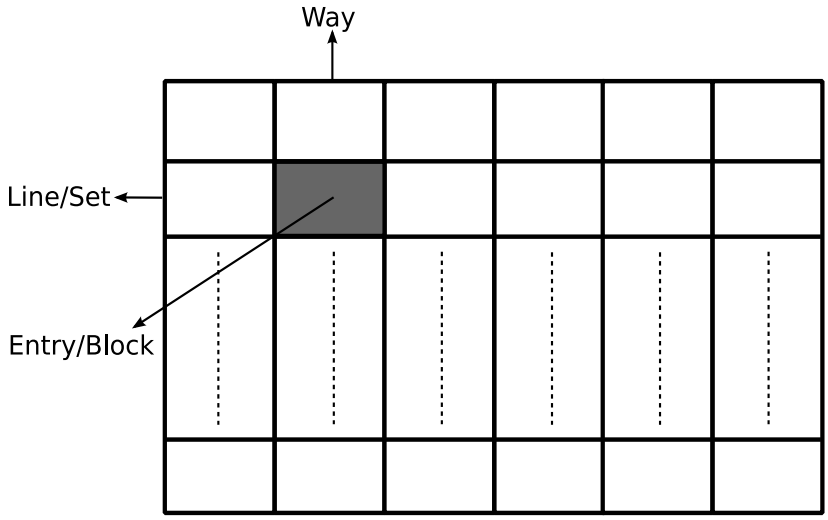


Figure 1.1: Example of a 6-way set associative cache

1.5 Thesis Overview

This chapter has highlighted the advantages of CMP architectures and the problems that this research attempts to investigate. The remainder of the thesis is arranged as follows.

Chapter 2 reviews several schemes that attempt to improve the efficiency of cache hierarchies both in the single processor and the CMP domain.

Chapter 3 provides a short overview of the Jamaica CMP that provided the test bench for the evaluation of the different cache partitioning schemes. It also presents the simulation platform and the benchmarks used during these studies.

Chapter 4 evaluates the employment of the LRU replacement policy in CMPs and presents evidence that a cache partitioning scheme can improve the performance of the system.

Chapter 5 introduces cache partitioning schemes that have been proposed by other researchers. They are evaluated together with other schemes that were

developed during this work but failed to improve the overall performance of the system. The advantages and drawbacks of each scheme are identified and then used to guide the development of a novel partitioning scheme.

Chapter 6 describes this novel scheme, evaluates its performance and analyses its complexity and hardware overhead.

Finally, Chapter 7 concludes this thesis by summarising the contributions made in addition to suggesting future expectations and possibilities.

Chapter 2

Cache Systems

Designers are responsible for selecting the appropriate cache according to the requirements of the system. The cache design space is big, as there are many variables that can affect the system's behaviour and performance. These include the total size of the cache, its associativity, the size of each cache line, the policy according to which lines are placed or replaced inside the cache array and the actual placement of the cache in the architecture and its distance from the processing cores. The increasing number of transistors per chip widens this range of options, as it is now possible to bring bigger caches closer to the processor or introduce multi-level cache hierarchies on chip.

This chapter presents several proposals aimed at improving the performance of the cache hierarchy for both single processor and CMP systems.

2.1 Cache Design

A straightforward solution to increase the cache's effectiveness and thus improve the overall performance, would be to increase the cache's size and associativity. Consequently, the cache would be able to hold more data blocks and reduce conflicts between data lines that map on the same cache line. This approach is primarily limited by the available area on the chip. Additionally, it makes the cache slower and more power hungry, which could ultimately have a negative effect on the system. Clearly, the effectiveness of this solution is limited and designers have been searching for other alternatives.

2.1.1 Single Processor Systems

Cache design has been extensively studied in the context of traditional uniprocessor systems. The schemes that are presented here attempt to adapt dynamically to the characteristics of the running programs and improve the performance of single processor systems.

Selective Caching

One group of these schemes has been looking at what needs to be put into the cache. Tyson *et al.* tried to reduce the average data access time by dynamically deciding whether to cache a particular data item based on the address of the instruction generating the request [63]. Their experiments showed that less than 5% of the total load instructions are responsible for causing over 99% of all cache misses. Based on that, they built "miss prediction tables", associating a 2-bit counter with each load instruction. The counter is incremented each time a load causes a miss and decremented in case of a hit. While the counter is at its highest value, the line is not stored in the cache.

González *et al.* proposed the "Dual Data Cache" which consists of two independent subcaches [20]. One is called "temporal cache" and is designed to exploit just temporal locality, while the other, named "spatial cache", is targetted on spatial locality as well. When the processor issues a memory request, both subcaches are looked up in parallel but, in case of a miss, the data that is brought from the next level of the memory hierarchy may be placed in one of the two subcaches or nowhere. The decision is based on a prediction of the type of the locality the memory reference exhibits. To make the prediction, the system uses a "Locality Prediction Table" which associates a memory reference with the instruction that issued the memory request and its past behaviour.

Johnson suggested the use of hardware monitors to track run-time memory access patterns [28]. More specifically, she used counters to monitor the access frequency of regions of memory. These were then used to guide caching decisions, such as cache bypassing, spatial locality optimisations and cache remapping.

A history table was also used by Collins *et al.* [13]. In their scheme one entry per cache set is used to store the last evicted tag. On a miss, the tag of the requested line is compared to the stored tag of the most recently evicted line from that set. If they are the same, a conflict miss has been identified, otherwise it is a capacity miss. This classification of misses is used as a prediction of future miss classification and incorporated into various decisions, like victim cache design, cache prefetching or cache exclusion mechanisms. For example, only lines that were brought into or forced out of the cache due to a conflict miss are stored into the victim cache.

Hashing Functions

Another approach to reducing the cache miss ratio is the employment of different hashing functions. Bodin *et al.* proposed "skewed associativity" to improve the system's performance [6]. In an X-way associative cache, similarly to other traditional cache structures, the memory address is hashed with the number of lines to deduce which line to access. An X-way skewed associative cache, however, employs different mapping functions for each way. On a cache access, these functions are used in parallel causing the system to access a different line in each way. Thus, the possibility of aliasing is reduced.

González *et al.* evaluated XOR-based mapping functions and concluded that they constitute a powerful technique for reducing conflict misses [21]. Later, Kharbutli proposed "prime displacement" [31]. This hashing function obtains the cache index by adding an offset, equal to a prime number multiplied by the tag bits, to the index bits of the memory address.

Cache Replacement Policy

Other studies target the cache replacement policy. Peir *et al.* developed a design called "adaptive group-associative" cache [47]. This scheme extends a direct-mapped cache by using a history table to track the lines that have been referenced recently. When a miss occurs, the line being replaced is checked against the entries of the table. If it has been referenced recently, it is placed into another location in the cache that has not been accessed in the recent past instead of being evicted. A small directory is used to keep track of the lines that have been so displaced.

Wong *et al.* suggested that implementing effective replacement algorithms might become more important than reducing conflict misses [66]. They modified the LRU policy for the L2 cache, so that lines which exhibit temporal locality are not replaced in preference to lines that have a low probability of being reused in the near future.

Kampe *et al.* proposed a "self-correcting" LRU replacement policy [30]. They identified three types of mistakes made by LRU. The first is created by blocks that are accessed only once, at the time of the miss. These blocks should have bypassed the cache. The second mistake is caused by blocks that exhibit some spatial locality or short term temporal locality. These are referenced more than once while being in the MRU position and are not accessed again before they are evicted. An ideal policy would remove them as soon as they leave the MRU position. The third mistake occurs when a block, which was evicted because it was in the LRU position, is accessed immediately after it was replaced. These blocks should be kept for a longer period. Detected mistakes are recorded into a "Mistake History Table", which is then used to refine the replacement decisions.

Kharbutli *et al.* suggested also that one of the main drawbacks of the LRU policy is that a line which is no longer needed occupies useful space for a long time until it becomes the LRU and is evicted from the cache [32]. This dead time is increased with larger degrees of associativity. To deal with this problem they proposed a counter-based cache replacement scheme, where each block is associated with a counter that records the number of accesses to that block. When the value of this counter reaches a threshold, the line becomes eligible for eviction.

Qureshi *et al.* detected a drawback of the set-associative cache in that it cannot adapt its associativity, as lines in the data array are statically mapped to entries in the tag array [48]. This means that when a miss occurs, a victim has to be identified within the target set, which prevents the cache from exploiting underutilised sets. To solve this problem, the tag array is expanded to hold more tags than the associativity dictates. Pointers are then used to map a valid tag entry to a location in the data array. When a miss occurs and an

invalid entry exists in the accessed line of the tag array, the replacement victim is supplied by the data array, which implements a global replacement policy. Thus, each line could contain a different number of valid tags, implementing a "demand-based" associativity that dynamically adapts to the access patterns of the running program.

Data Placement

A different set of schemes try to deal with the increasingly significant wire delays. Rises in on-chip communication delays affect the latency of large on-chip caches, making their hit time a function of a line's physical position within the cache. Therefore, Kim *et al.* suggested that traditional, monolithic cache designs with a single, uniform access time will be ineffective in future designs and proposed breaking the cache into banks that can be accessed at different latencies [34]. When a hit occurs to a cache line, it is swapped with a line in the bank that is the next closer to the cache controller. Thus, heavily used lines will migrate towards banks that are quickly accessed, while infrequently accessed lines will be demoted into farther banks with higher access time.

Chishti *et al.* exploited sequential tag-data access to decouple data from tag placement and define "Distance Associativity" [11], where a data block is placed at a certain distance from the core and thus suffers a certain latency. The data array is broken into different groups, each one with a different latency. The tag-data decoupling enables data blocks from the same cache set to be stored either in the same group or in a different one. Initially all cache blocks are placed in a fast group and then the rarely accessed blocks are demoted to slower groups. Thus, the majority of accesses happen to the fastest group, while the swaps between different groups are minimised, in contrast to the proposal by Kim *et al.* [34]. Thus this system results in higher performance and lower energy consumption.

Cache partitioning

Another set of studies target the sharing of the cache between different access streams. Stone *et al.* studied the optimal allocation of cache memory among multiple access streams [58]. They showed that LRU typically comes close to

achieving optimal performance and they focused on partitioning a cache between the instruction and data access streams of a single workload.

Chiou *et al.* were motivated by the presence of an increasing amount of streaming data and the development of multi-threaded parallel processing applications [10]. They recognised that these result in memory access pattern that exhibit less temporal locality and pollute the caches with data that will not be accessed again in the near future. They proposed a novel scheme, where a set-associative cache is partitioned at the granularity of a way and specific data is restricted to be only placed into a particular subset of the cache's ways. The difference from older static partitioning schemes is that the partitioning is dynamic, as the restrictions can be modified or even be completely removed changing the partitioned cache to a normal one.

Suh *et al.* developed an analytical cache model that depends on offline profiling of applications [59]. Using these offline miss rates the model can estimate the overall miss rate. This estimation is then used by a partitioning unit to determine an optimal partition of the caches that minimises the miss rates.

Summary

A 'simple' cache hierarchy is not always adequate and a number of more elaborate schemes for improving the performance of the system have been proposed. These range from selective caching depending on whether a memory reference exhibits temporal or spatial locality to modified versions of the LRU replacement policy and cache partitioning. All these proposals have been evaluated in uniprocessors but it is likely that the parameters are different in CMPs.

2.1.2 Chip Multithreaded Systems

Most previous studies of caches have been related to uniprocessors. There is now a trend to increase the number of processors on a single chip and, in the CMP model, these often share the memory hierarchy. However this sharing imposes new constraints and affects the cache utilisation. Therefore, it is necessary to reevaluate cache design for CMPs.

One of the major challenges is that the sharing of any level of the cache hierarchy allows interference, as data belonging to one thread may be evicted by operations by other threads. This interference impacts the running processes non-uniformly and could cause performance problems, like cache thrashing and thread starvation. Therefore different proposals have been made to improve the sharing of the cache and thus the overall performance of the system.

Thread scheduling

A group of proposals tries to tackle the cache contention problem by making appropriate scheduling decisions. Kihm *et al.* provided a set of counters for each thread, each of which is associated with a group of cache sets and records the accesses to these cache lines [33]. This monitoring was used to predict future access patterns for each thread, which were then exposed to the operating system scheduler. The scheduler then attempted to select threads that exploit different regions of the cache and thus can operate efficiently in parallel.

Chandra *et al.* studied the impact of the inter-thread cache contention on CMP architectures and proposed an analytical model to predict the impact of cache sharing on parallel running threads [8]. The input of the model is the isolated L2 cache access profile for each thread and the output is the number of extra L2 cache misses for each thread that shares the cache. Thus this model can be used by the scheduler to make the appropriate decisions.

A similar approach was employed by Fedorova [16]. An analytical model was proposed to estimate the L2 cache miss rate that a thread would have if the cache were shared equally among all the threads. The output was then used by the scheduler to adjusts the thread's share of CPU cycles in proportion to its deviation from its fair miss rate. Thus performance variability due to cache sharing was reduced.

Non-Uniform Caches

A different set of proposals tries to explore the range of sharing policies of large banked caches, mainly L2, between different on chip processors. More precisely, the L2 banks can be managed in two basic ways, as shown in Figure 2.1. They either form a 'single', large cache shared by all the processors or each

one is treated as a private L2 cache. The private scheme has low hit latency, providing good performance when the working set fits into the cache; however the effective cache capacity is reduced as each processor is allowed to use only its private L2 slice. Therefore, it is possible for a core to exhibit a high miss rate due to its working set not fitting into its slice while there is empty space in the other banks.

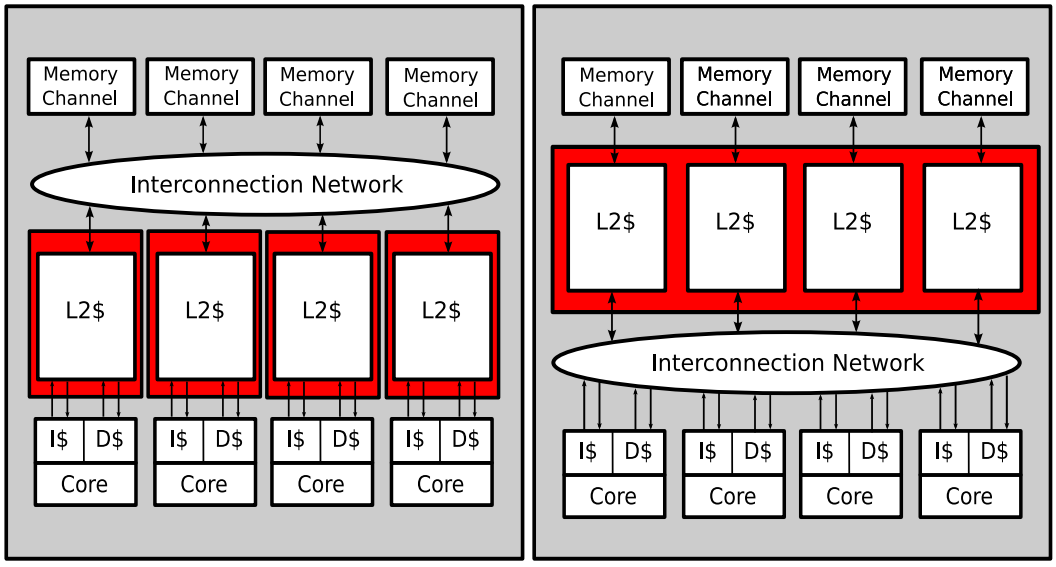


Figure 2.1: L2 Cache Designs

On the other hand, the shared scheme reduces the off-chip miss rates by increasing the effective cache capacity, as a core could potentially use all the available L2 cache space. However each bank could have a different access latency, depending on the interconnection network and the distance of the bank from the accessing core. The higher latency of a hit in a remote L2 slice combined with the increased bandwidth requirements and the coherence overhead can harm the performance of the system.

Huh *et al.* tried to bridge this gap by modifying the shared scheme [25]. Instead of mapping a data block statically to a specific L2 bank, they proposed dynamic mapping by allowing a block to be stored in one of multiple banks. Using proper placement and migration policies their scheme enables the cache to place frequently accessed blocks in the banks closest to requesting CPU.

Zhang *et al.* created a hybrid of the private and the shared scheme [68]. In their system the L2 cache is shared, but when a line is evicted from the L1 cache, they attempt to keep a copy of the victim line in the local L2 slice. Thus, a subsequent miss can be serviced by the local L2 using this replica instead of bringing the original line from the remote L2 slice.

The same problem was tackled by Chishti *et al.* [12]. They expanded their previous work on distance associativity [11] to CMP architectures and proposed three different mechanisms. The first one, called "controlled replication", places copies of data lines close to the requesters while trying not to reduce the effective on-chip capacity too much. The second mechanism provides fast access to read-write shared data without making copies or incurring coherence misses. The third suggestion, called "capacity stealing", enables a core to migrate its less frequently accessed data to a cache slice with less capacity demand.

The above proposals start out with a banked shared cache and attempt to incorporate some of the advantages of using private L2 caches by using data migration and replication. Recently, Chang *et al.* proposed "Cooperative caching" [9], which employs private L2 caches and uses a collection of mechanisms to create a globally managed, aggregate on-chip cache via the cooperation of the different on-chip caches. These mechanisms include a replication-aware replacement policy and global replacement for inactive data.

Cache Partitioning

Suh *et al.* extended their previous work [59] by using a set of hardware counters to gather information about the running processes [60]. This information was then used to determine the cache partitioning. The novelty of this scheme is that the optimal partition of the cache is calculated dynamically to capture the time changing characteristics of each thread; then every thread is assigned a space allocation limit. In the event of a cache miss, an extra cache block will be allocated to the thread if and only if its current allocation is below this limit.

Kim *et al.* noted that an operating system enforces thread priorities assuming

that the progress of each thread is uniformly affected by the sharing of the resources [35]. When this assumption of fair hardware is not met, then problems like thread starvation or priority inversion arise. Therefore they introduced different metrics to evaluate cache fairness and proposed the substitution of the 'thread-blind' LRU replacement policy by static and dynamic partitioning algorithms that lead to a partition which optimises fairness.

Following a similar approach, Iyer proposed a framework for enabling Quality of Service (QoS) in CMP systems [27]. This scheme tries to improve the shared cache efficiency by providing prioritised service to different threads. The priority can be enforced by three different mechanisms: set partitioning, selective cache allocation and heterogeneous cache regions. The first mechanism allows threads with a higher priority to occupy more ways in a line than lower priority threads. The second one associates a cache space allocation limit with each priority level. The number of lines occupied by each thread is monitored and used to determine whether the allocation of a new line is allowed or not. The final mechanism maps heterogeneous cache structures, such as set-associative caches, stream buffers and victim caches, that implement different replacement policies to different memory access streams based on their priority levels.

Settle *et al.* proposed using global cache reuse behaviour to control the cache space available to each thread [54]. In one of their schemes, when the thread ID of the cache request differs from that of the normal LRU candidate, the cache controller checks the line's reuse counter to determine the impact this eviction could have on the system performance. If the reuse counter is higher than a threshold value, the line is not suitable for eviction. Instead the next least recently used line is evaluated and the process is repeated until a suitable candidate is found. If none is found, the normal LRU candidate is used.

Wang *et al.* developed a scheme called "thread-associative memory" [65], where thread-specific information is explicitly incorporated into the on-chip cache. More specifically, it employs a set-associative cache, where each way within a set is grouped into N "rails", where N is the number of concurrent threads sharing the cache. An example of a two-rail thread associative cache is shown in Figure 2.2. The cache address is extended to include the thread ID, which is

then used to determine which rail to access. By not allowing a thread to access a rail owned by another thread, cache replacements for concurrent threads are decoupled, thereby avoiding inter-thread conflicts. The system also uses an "enable" bit, so that if the performance degrades or only one thread is using the cache, the thread-associative mapping is disabled and the whole cache is used.

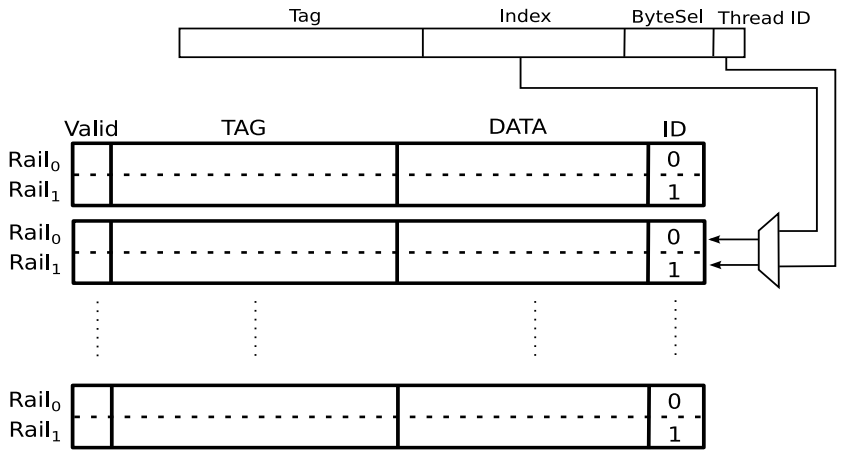


Figure 2.2: A two-rail thread associative cache [65]

Qureshi *et al.* developed a low hardware overhead monitoring circuit to estimate the utility of the cache resources for each running application [49]. Based on the observation that a reduction in misses correlates with a reduction in CPI, these estimations are used by an algorithm to divide the cache amongst the competing applications in an optimal way that will reduce the number of misses. A similar scheme was developed by Dybdahl *et al.* [15]. The two proposals differ on the amount of information gathered about each application and the way the partition sizes are modified each time the partitioning algorithm is executed.

Summary

The aforementioned studies have shown that a 'simple' cache hierarchy does not perform efficiently under the extra strain imposed by the sharing of on-chip resources amongst the concurrently running threads of a CMP architecture. A group of these advocate the partitioning of the cache to improve the overall

performance of the system.

The work presented in this thesis was primarily influenced by these cache partitioning schemes and focused mainly on the last two proposed by Qureshi *et al.* [49] and Dybdahl *et al.* [15]. The two systems were evaluated and their advantages and drawbacks were identified and used as a guide for the development of a different cache partitioning scheme that aims to improve the system's performance at a low cost.

2.2 Bloom Filters

In the process of developing a new cache partitioning scheme different ways of monitoring the cache contents were examined. One of these is the Bloom filter [5], which is briefly introduced here.

2.2.1 Description

A Bloom filter is a probabilistic algorithm to quickly test membership in a large set using multiple hash functions into an array of bits [5]. It consists of an array A of m bits and k hash functions $\{h_1, h_2, ..., h_k\}$. In an empty filter, all bits in A are zero. To add item s to the filter, $h_1(s)$ is computed and the value is used to index into A, where $A[h_1(s)]$ is set to one. This process is repeated for all the hash functions, as shown in Figure 2.3.

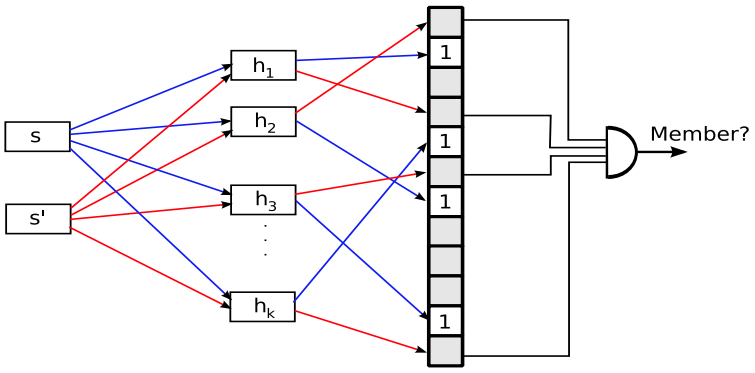


Figure 2.3: Example of a Bloom filter

To test for the presence of s', the bits $A[h_1(s')]$, $A[h_2(s')]$, ..., $A[h_k(s')]$ are looked up. If all these bits are one, the set membership test returns true, but if any are zero the test returns false. For these membership tests, false negatives are impossible. False positives, however, are possible, as any or all of the bits indexed by applying the hashing functions to s' could have been set due to the insertion of other elements into the filter. This problem is also known as "*aliasing*". The probability of such filter errors depends on the number n of items inserted into the set, the number k of hash functions, their effectiveness and the size m of array A. Assuming "good" hash functions, this possibility is given approximately by :

$$prob = \left(1 - e^{-kn/m}\right)^k \tag{2.1}$$

2.2.2 Applications of Bloom filters

Bloom filters have been successfully applied to a variety of fields. They have been used for text analysis [4] and the development of spell checking programs [42, 45], in databases to estimate the size of joins [43, 44] and access the files that cache the modifications of the database's records [22, 50] and in network applications [17, 53, 64].

Recently, Bloom filters have also been introduced into the computer architecture domain. Peir *et al.* used them to implement a hit/miss predictor that identifies cache misses early in the pipeline [46]. This early identification of cache misses is used by the processor to schedule instructions that are dependent on loads more accurately and to prefetch data into the cache.

Ricci *et al.* applied them on a CMP system where the L2 cache was split into several banks, with each bank having a different access latency depending on its proximity to the core [51]. In their system, each processor maintains as many Bloom filters as the number of the L2 cache banks and each filter stores information about the contents of one of the banks. On a L1 cache miss, the processor checks the filters and for those that return a hit, their associated L2 cache bank is looked up. Therefore the number of banks that need to be searched for a line is reduced. Moore *et al.* described a "Thread-Level Transactional Memory" system, where Bloom filters are used to detect potential conflicts between different coherence transactions [41].

In summary, Bloom filters provide a cost effective way to store information. Therefore, they have been added to systems to track the contents of caches or to identify conflicts of coherence transactions. The work presented in this thesis exploited them in a similar way to monitor cache misses, as it is described in Chapter 6.

2.3 Summary

Cache design has been extensively studied for uniprocessor systems. However, nowadays, CMP architectures have emerged, where the available on-chip cache is shared between several threads. This sharing imposes new constraints and creates new challenges that need to be tackled in order to ensure that the performance of the system is improved.

This chapter presented a variety of previous proposals, like the use of non-uniform caches, efficient thread scheduling and cache partitioning. The work presented in the next chapters belongs to the last category. Several partitioning systems were evaluated and the results of the analysis were used to develop a novel scheme that aims to improve the overall system's performance at a low cost.

The next chapter outlines the Jamaica CMP architecture and the *jamsim* simulation platform that was used to evaluate the different cache partitioning schemes. Finally, the software environment used to run applications on the Jamaica architecture is described.

Chapter 3

Jamaica and Simulation Framework

The work presented in this thesis is intended to be as generic as possible and, therefore, applicable to different CMP systems. Nevertheless, it was based on the Jamaica architecture, which provided a test bench for the evaluation of the different cache partitioning schemes that were studied.

This chapter provides a short overview of the Jamaica architecture as well as a description of the simulator and the benchmarks that were used.

3.1 The Jamaica Architecture

Jamaica (JAva Machine And Integrated Circuit Architecture) is a simulated CMP proposed by Wright [67]. As shown in Figure 3.1, the architecture originally consisted of N cores sharing a L2 cache. The system was later extended by Horsnell [24] to include multi-level cache hierarchies. However, this work will focus on the part of the system illustrated in Figure 3.1.

Each processor has its own private instruction and data L1 caches, which are connected through a shared bus to an on chip L2 cache. The L2 cache is unified and *inclusive*, i.e. it includes all the lines contained in the L1 caches. The requests that miss in the second level of the memory hierarchy are forwarded to the next level, which for this work is considered to be the main memory. To

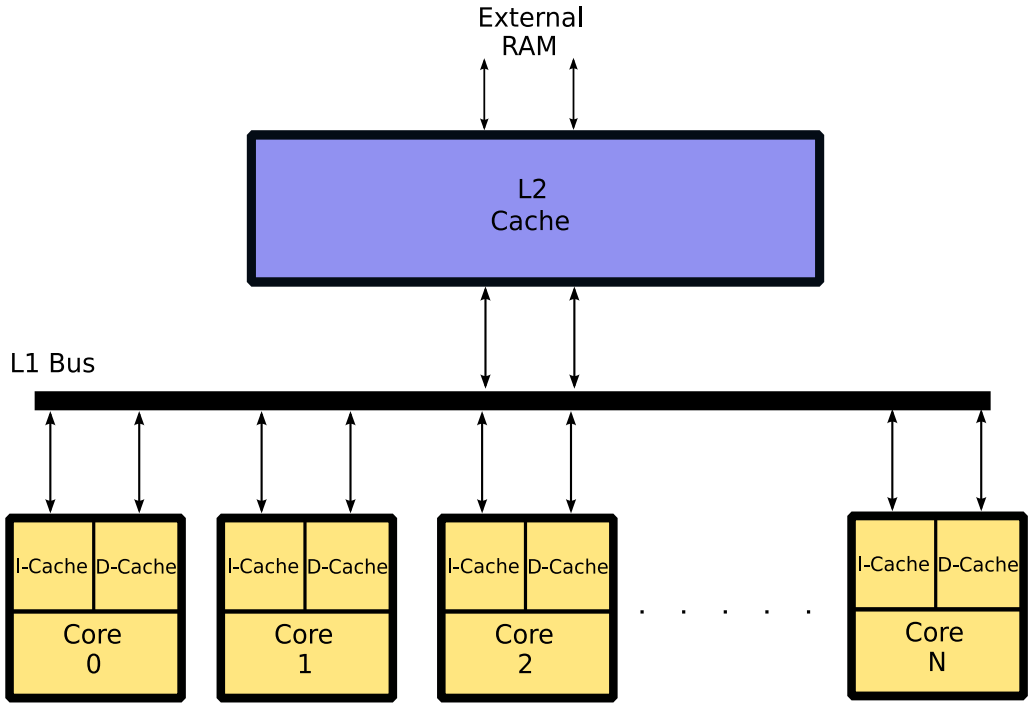


Figure 3.1: The Jamaica architecture

limit the amount of traffic on the interconnect, the caches implement a write-back policy.

Additionally, a lazy allocation policy is implemented within the multi-level hierarchy. On a cache miss, a request is generated and propagated into the network without allocating a line to hold the response. This occurs at each successive level in the hierarchy and no line is allocated until a response transaction is received. When the response is received by the cache, the LRU policy is used to determine which block should be rejected to create the necessary space for the new entry.

The system maintains sequential consistency to allow for standard shared memory programming. To be coherent, the caches snoop the L1 bus and implement the PIMMS coherence protocol [24]. PIMMS is an extension of the MOESI protocol [61] with additional states to support multi-level cache hierarchies.

In general, Jamaica is an architecture similar to other proposed CMP systems. Niagara [36], developed by Sun, has 8 cores and each one has its own private L1 instruction and data caches while they all share an on-chip L2 cache through a crossbar. Power 5 [29], developed by IBM, has 2 cores. Each processor has private instruction and data caches and they share an on-chip L2 cache through a bus. Intel developed Core Duo CMP [40], where 2 cores share a L2 cache through a bus. Therefore, it is justified to assume that the novel cache partitioning scheme developed in this work and evaluated on the Jamaica architecture will be applicable to other CMP systems.

3.2 Simulation Environment

A Java simulation platform, called *jamsim* [23], has been developed to execute binaries created for the Jamaica ISA. Jamsim supports different models of simulation, namely functional and cycle-level simulations. The former are fast as they employ an abstract model of the architecture and can be used for system software development. The latter use components implemented in sufficient detail to account for pipeline stalls, interconnection bus and queue contention, cache access contention and memory channel queueing[1]. Therefore, cycle-level simulations are slow but essential for quantitative evaluation of the system.

Additionally, the simulator provides the option of using components at different levels of modelling. As this work was focused on the caches, this option was exploited. More specifically, functional models of the processors were used to generate memory access traces that were used to exercise the memory hierarchy, which was modelled in full detail.

Jamsim is a structural simulation platform, where each hardware component is mapped to a different simulator component implemented by a Java class. Exploiting object oriented programming techniques, the cache models were easily extended to include the different cache partitioning schemes that constitute the target of this work.

[1] Jamsim does not simulate TLBs, as software execution is performed by a virtual machine, presented in Section 3.3, which is responsible for the handling of the address space between the running processes.

Finally, the simulation platform is parameterisable, as the user can select the number of cores, the size and associativity of the caches, the speed of the bus as well as which cache partitioning scheme to use.

3.3 Software Environment

The Jamaica instruction set is different from other common sets. Therefore, it is supported by a number of tools that enable the compilation and execution of applications. Software execution is performed by the Jamaica Virtual Machine (JaVM) [14], which is a port of the Jikes Research Virtual Machine (RVM) [2] to the Jamaica architecture. The JaVM compiles and optimises Java bytecode to native machine code allowing the execution of unmodified Java applications on top of the Jamaica architecture.

The Jikes RVM, and thus the JaVM, is an $M : N$ thread model, scheduling execution of an arbitrarily large number (M) of Java threads over a finite number (N) of virtual processors. To serve the purposes of this work, the virtual machine was configured to create one virtual processor for each simulated core. Then, to ensure that each application of the simulated workload would execute on the same processor for the duration of the simulation, the virtual machine was modified to allow the user to force a Java thread to execute on a specific core.

Therefore, to evaluate the performance of the studied cache partitioning schemes, a set of Java applications was developed. Each one creates as many Java threads as the simulated processors, assigns a different benchmark to each thread and ships it to a specific core to be executed. The benchmarks used to create these workloads are presented in Section 3.4.

3.4 Benchmark Descriptions

To evaluate the different cache partitioning schemes, a set of applications was selected from the "Java Grande Forum" benchmark suite [56] and the "NAS Parallel Benchmarks" [18]. This section provides an overview of each application.

3.4.1 Java Grande Benchmarks

The Java Grande Forum suite includes both sequential and multithreaded benchmarks. As the major scope of this work was to study how the cache is shared amongst different competing applications, the workloads were created by mixing different sequential applications. The sequential suite consists of three types of applications; low-level operations, kernels and large scale applications. From these benchmarks, six kernels were selected.

Crypt

The *crypt* benchmark performs IDEA (International Data Encryption Algorithm) encryption and decryption of an array of N bytes. It creates 3 byte arrays which hold the initial, the encrypted and the decrypted data.

HeapSort

The *HeapSort* benchmark sorts an array of N integers using a heap sort algorithm. It is a memory and integer intensive application. From this point it will be referred to as *heap*.

LUFact

The *LUFact* benchmark solves an $N \times N$ linear system using LU factorisation followed by a triangular solve. This is a Java version of the well known Linpack benchmark. It is memory and floating point intensive, as the algorithm creates several sub-matrices and then calculates the LU decomposition inside each matrix. Hereafter, it will be referred to as *lu*.

Series

The *series* benchmark computes the first N Fourier coefficients of the function

$$f(x) = (x + 1)^x \tag{3.1}$$

over the interval [0,2]. This benchmark heavily exercises transcendental and trigonometric functions. It creates a two dimensional array of doubles, which holds the computed Fourier coefficients.

SOR

The *sor* benchmark performs 100 iterations of successive over-relaxation on a $N \times N$ grid. The benchmark contains three loops, the outer iteration loop and two inner loops over the row elements to process the relaxation.

Sparse

The *sparse* benchmark performs a sparse matrix multiplication. It uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirect addressing and non-regular memory references. A $N \times N$ sparse matrix is used for 200 iterations.

3.4.2 NAS Parallel Benchmarks

The NAS Parallel benchmarks were developed by NASA to help evaluate the performance of supercomputers and are recognised as a standard indicator of computer performance. They are derived from computational fluid dynamics (CFD) applications and they consist of five kernels and three pseudo-applications. Their implementations include both serial and parallel versions. From this suite three applications were selected.

CG

The *cg* benchmark estimates the largest eigenvalue of a symmetric positive definite sparse matrix by the inverse power method. The core of *cg* is a solution of a sparse system of linear equations by iterations of the conjugate gradient method. This kernel tests unstructured computations and communications.

FT

The *ft* benchmark contains the computational kernel of a 3-D Fast Fourier Transform (FFT). Each ft iteration performs three sets of one-dimensional FFTs, one set for each dimension. The benchmark creates 8 double arrays, which are used to to hold the computed values.

MG

The *mg* benchmark uses a V-cycle Multi Grid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works iteratively on a set of grids that are made between the coarsest and the finest grids. It tests both short and long data movement.

3.4.3 Benchmark Configuration

The benchmarks described in Section 3.4 were selected in an attempt to create a representative mix of applications with different cache space demands. As presented in Chapter 4, some benefit as the cache size increases, some have a small working set and others suffer a large number of compulsory misses.

The selected benchmarks are required to exercise the cache hierarchy of the system. At the same time, as the simulations are a lot slower compared to real system execution, it is essential that their execution time is limited. Therefore they were appropriately configured and the parameters that were used are presented in Table 3.1.

Benchmark	**Parameters**
crypt	500000 bytes
cg	configuration 0
ft	configuration 0
heap	2000000 integers
lu	matrix = 500× 500
mg	configuration 2
series	800 Fourier coefficients
sor	grid = 600× 600
sparse	matrix = 30000× 30000, iterations = 250

Table 3.1: Benchmarks' configuration

The benchmarks were executed without any garbage collection. However, this should not have any effects on the validity of the simulation results, as these benchmarks create a small amount of objects. For example, as it was described in Section 3.4.1, *heapsort* creates only one array of N integers and performs $N \times logN$ comparisons between its elements.

3.5 Summary

This chapter outlined the Jamaica CMP architecture. Each processor has its own private L1 instruction and data caches, while they all are connected to an on-chip, unified L2 cache through a shared bus. The caches are kept coherent by snooping the bus and implementing an appropriate coherence protocol. This system provided the test bench for the evaluation of the various cache partitioning schemes that were the scope of this work.

Jamsim, the simulation platform, was extended to implement all these partitioning techniques and used to run several workloads. Finally, this chapter presented the different benchmarks that were mixed to create these workloads as well as their configuration.

Chapter 4

Cache Replacement Policy

In case of a miss, an entry of the accessed line needs to be rejected from the cache to create space for the new entry that has been requested by the running application. System designers typically employ the LRU policy to identify the replacement victim, as it is considered to be the best available replacement policy. The CMP domain has inherited this policy from uniprocessors, by default.

However, several people [60, 35, 27, 33, 49] have recently questioned the validity of that decision. More specifically, evidence has been presented that LRU could degrade the performance of a system by not allocating the cache resources optimally amongst the competing processes.

To study the efficiency and applicability of the LRU policy in CMP systems, several combinations of benchmarks were executed. This chapter presents the results of these simulations and identifies the drawbacks of LRU and their impact on the overall performance of the system.

4.1 Evaluation of LRU

4.1.1 Single Core Systems

First, the simulator was used to run the selected benchmarks on a single processor system to create a profile for each application. This profile provides information about the benchmark's working set size, shows the maximum performance each process can achieve and can be used to estimate roughly the

Processor Core	Functional Model
L1 Caches	ICache and DCache : 32KB, 32B line-size 4-way, LRU
L1-L2 Bus Speed	1/2 of processor speed
L2 Cache	4MB, 32B line-size, 32-way, LRU, 16 cycles hit latency
Memory	100 cycles access latency

Table 4.1: System's configuration

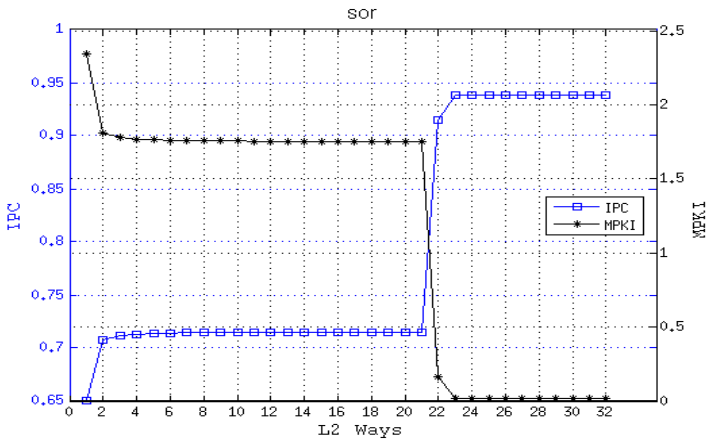
relative performance of the system when the cache is shared by these applications.

Table 4.1 shows the parameters of the baseline configuration. The simulator uses a functional model of the processor, which has L1 instruction and data caches, each 32 KB and 4-way associative. The on-chip L2 cache is 4096 KB, 32-way associative and uses the LRU replacement policy. A L1 cache miss that hits in the L2 cache takes 16 cycles to be satisfied; however, if a miss occurs in the L2 cache as well, an additional delay of 100 cycles is suffered. To vary the available cache space, the processor is restricted to using a different number of ways each time. At the end of each run the Instructions Per Cycle (IPC) was recorded, together with the total number of misses in the L2 cache as Misses Per Thousand Instructions (MPKI) executed.

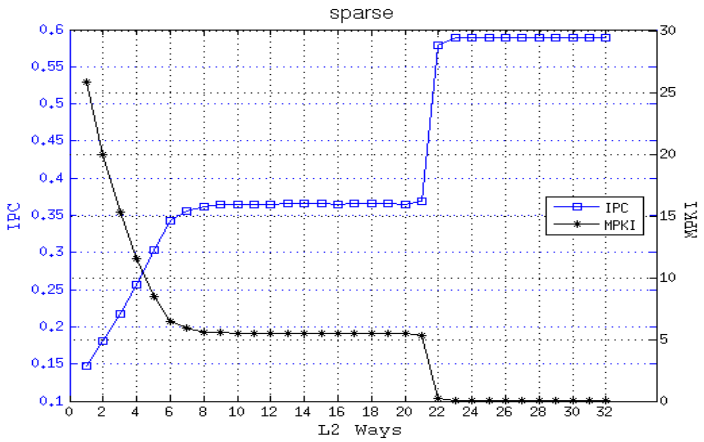
Following the classification of Qureshi *et al.* [49], and based on the simulation results, the benchmarks can be organised in the following categories.

1. **Low Utility :** Applications that do not benefit significantly as the available cache space is gradually increased. The reason for that could be a very big working set, larger than any available cache, or that the application suffers a large number of compulsory misses. This category contains *sor* and *sparse*, which are examples of the former, and *series*, an example of the latter.

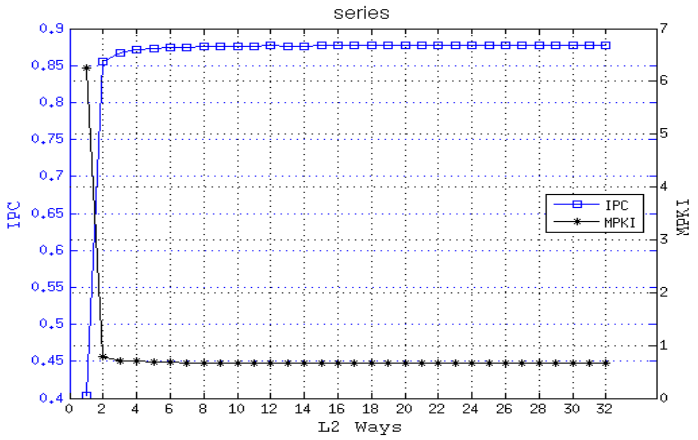
   In Figure 4.1(a) *sor*'s performance improves going from 1 to 2 ways and then remains stable while the cache is increased up to 21 ways, which corresponds to a cache of 2688 KB. However, when the cache is increased to 22 ways (or 2816 KB), the number of misses drops almost to zero and



(a) *sor* IPC - Misses



(b) *sparse* IPC - Misses



(c) *series* IPC - Misses

Figure 4.1: Low utility applications

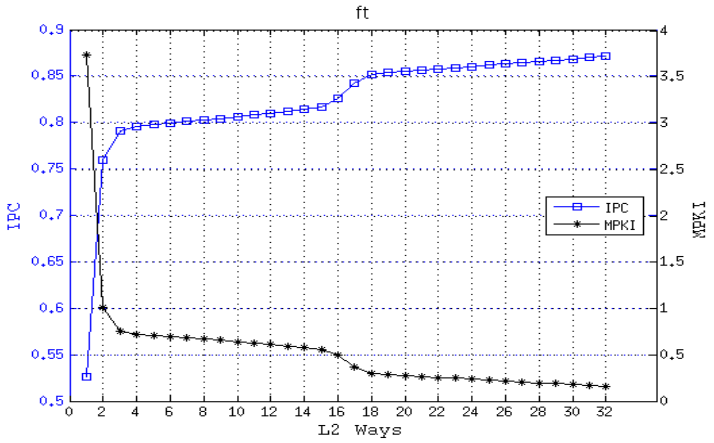
the performance is increased significantly. A similar behaviour can be observed for *sparse* in Figure 4.1(b). On the other hand, Figure 4.1(c) shows that *series*'s performance reaches a maximum for 2 ways and remains stable thereafter. At the same time, its misses remain stable as well, indicating that this benchmark suffers a number of compulsory misses, a behaviour demonstrated by streaming applications.

2. **High Utility :** Applications that continue to benefit as the L2 cache size is increased. These include *ft* [Figure 4.2(a)], *heap* [Figure 4.2(b)] and *mg* [Figure 4.2(c)]. For example *heap*'s IPC increases from 0.4 to about 0.7 as the cache is increased from 1 way to 32. At the same time, its L2 cache MPKI drops from about 6 to below 1.

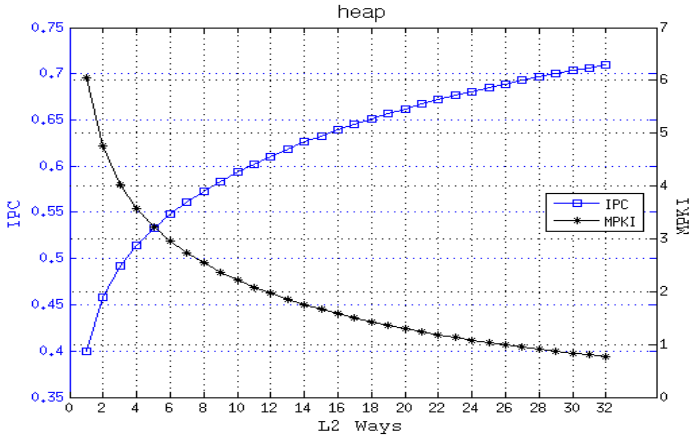
3. **Saturating Utility :** Applications that initially benefit significantly from increasing the available cache space, as more of their data set can fit into the cache. Then a point is reached, where the whole data set can be stored into the cache, after which, using a bigger cache has no effect on the performance of the application. This category includes *cg* [Figure 4.3(a)], *crypt* [Figure 4.3(b)] and *lu* [Figure 4.3(c)].

   For example, *lu*'s IPC starts from 0.5 for 1 way and increases to almost 0.9 for 16 ways. At that point, L2 cache misses drop to zero, an indication that its data set fits into the L2 cache. From this point on, providing more cache space to the application makes no difference and its performance remains stable.

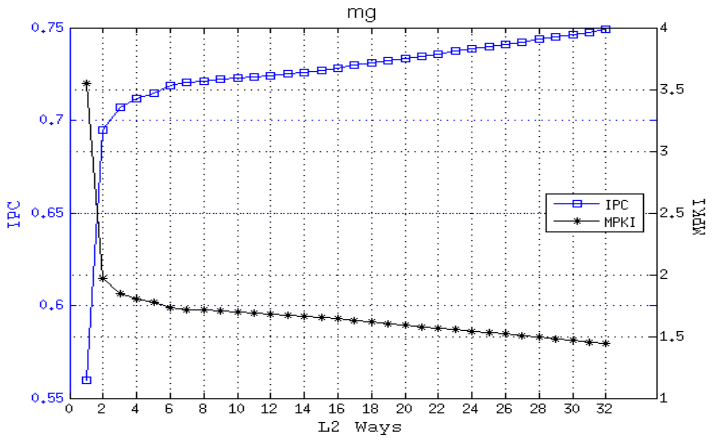
Another conclusion that can be drawn from all these profiles is that the performance correlates strongly with the number of L2 cache misses that each application suffers. These misses are expensive, as they need to be satisfied by the next level of the memory hierarchy, which is off chip. This means that a L1 cache miss that hits in the L2 cache will only take 16 cycles, according to Table 4.1, whereas if it also misses on the L2 cache, it will need 100 processor cycles more to be satisfied. So, by reducing the number of L2 cache misses, the performance of the application is expected to improve.



(a) *ft* IPC - Misses

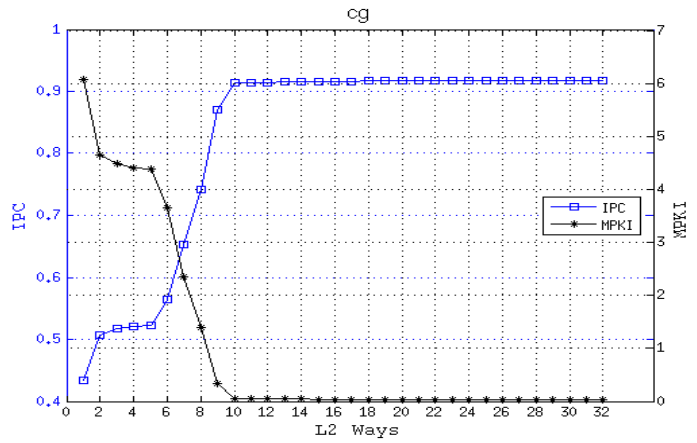


(b) *heap* IPC - Misses

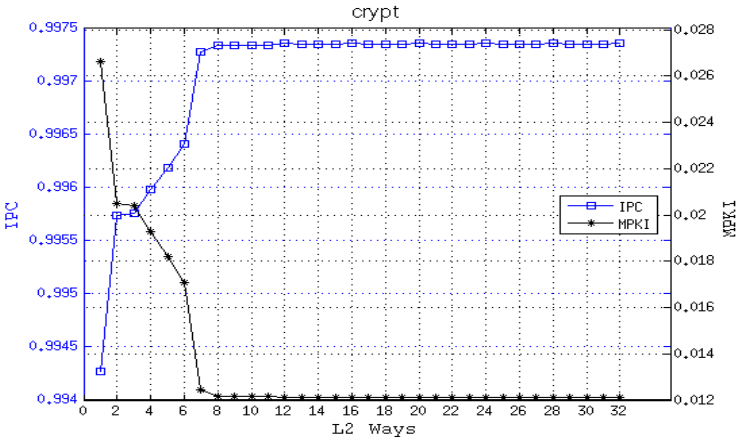


(c) *mg* IPC - Misses

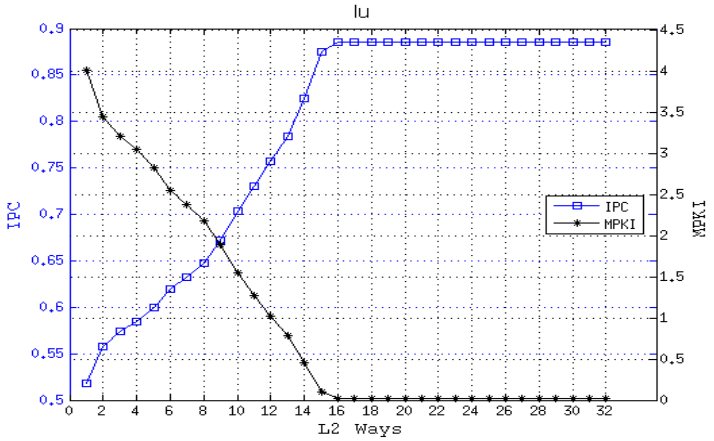
Figure 4.2: High utility applications



(a) *cg* IPC - Misses



(b) *crypt* IPC - Misses



(c) *lu* IPC - Misses

Figure 4.3: Saturating utility applications

Category		Benchmark	Cache Ways
1	Low Utility	*sor*	22
		sparse	22
		series	2-3
2	High Utility	*ft*	32+
		heap	32+
		mg	32+
3	Saturating Utility	*cg*	9-10
		crypt	1
		lu	15-16

Table 4.2: Benchmarks' cache space requirements

Each application has different cache space requirements. Based on the results of the previous simulations, Table 4.2 holds the space, expressed in cache ways, that each benchmark needs in order to achieve maximum performance. As these results are obtained on a system where the application is running alone, they tend to overestimate the size of the working set. Thus they can only be used as a rough estimate of the upper limit of each benchmark's requirements.

4.1.2 Dual Core System

A major objective of this research is to determine an efficient way of using the memory hierarchy of a CMP system. The next step was therefore to study the performance of a dual core system. More specifically, the system shown in Figure 4.4 was simulated running two of the selected benchmarks in parallel.

Each processor has its own private L1 instruction and data caches, 32 KB and 4-way associative each. Both cores share a 4096 KB, 32-way associative L2 cache, which employs the LRU replacement policy. The simulation is stopped when one of the benchmarks finishes. At first, saturating utility applications were selected.

Figure 4.5 shows the IPC achieved for each benchmark in every scheduled pair. The comparison with the profiles in Figure 4.3 reveals that each benchmark achieves maximum performance, which is defined as the performance when the application is running alone using all the available system resources. This is an indication that the L2 cache can support both applications at the same time.

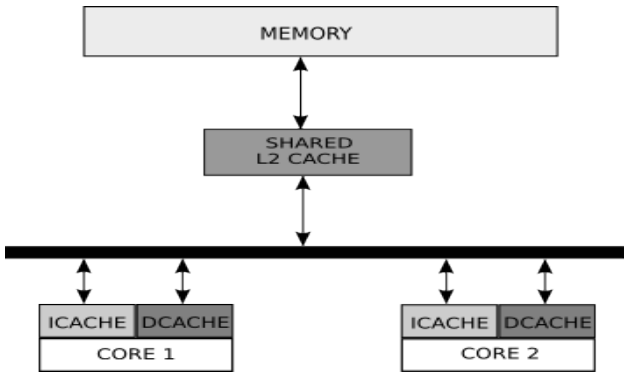


Figure 4.4: Dual core system

Indeed, according to Table 4.2, none of these benchmarks should require more than 16 ways, which means that the 32 ways of the L2 cache can accommodate the working sets of both applications for all the pairs and that interference between applications is minimal.

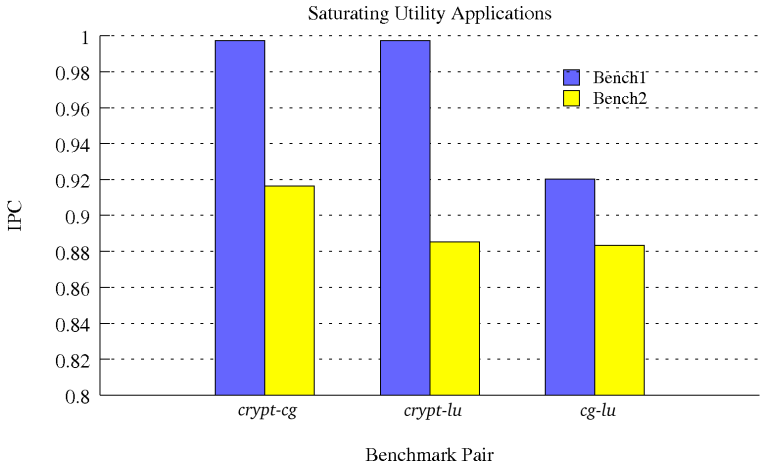


Figure 4.5: Performance of saturating utility benchmarks

Next, pairs created from benchmarks either of low or saturating utility were used and their performance is shown in Figure 4.6. Applications of saturating utility, namely *crypt*, *cg* and *lu*, achieve the same performance as when running alone, regardless of their co-runner. The same is not true though for the low utility applications. For example, *sparse* achieves an IPC of 0.57 when paired with *series* and only 0.36 when running together with *sor*, as it is shown in the last two columns of Figure 4.6. Low utility applications appear to be sensitive

to which benchmark they are scheduled with. This can be attributed to the different amount of cache space available to each process for every pair. To better illustrate this, the average cache occupancy in terms of cache ways is calculated for each case.

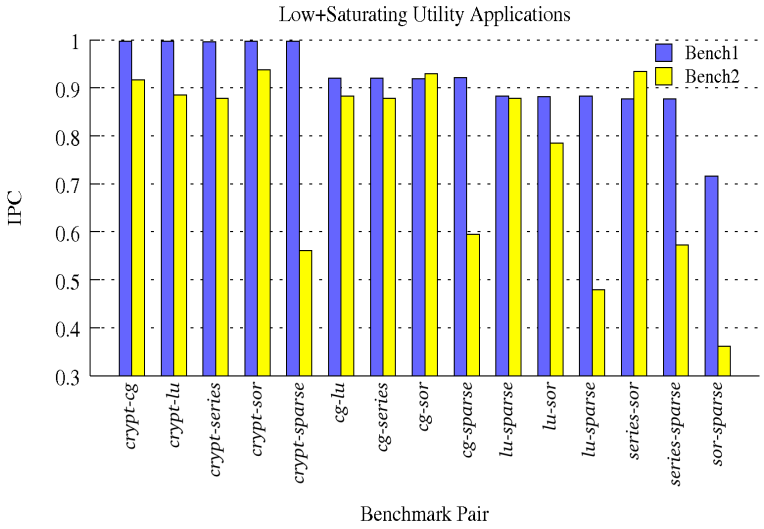


Figure 4.6: Performance of low and saturating utility benchmarks

Every 10 million cycles the L2 cache is examined to find out how many ways each context is occupying in each line. This information is used to calculate the average number of cache ways that have been assigned to a context at a time point t using the following formula :

$$ways_{context,t} = \frac{\sum_{w=1}^{CacheWays} w \times lines_w}{\sum_{w=1}^{CacheWays} lines_w} \tag{4.1}$$

where $lines_w$ is the number of lines in which the context occupies w ways. This number is then used to calculate the average number of ways the context occupies over T total points of the simulation.

$$ways_{context} = \frac{\sum_{t=1}^{T} ways_{context,t} \times entries_{context,t}}{\sum_{t=1}^{T} entries_{context,t}} \tag{4.2}$$

where $entries_{context,t}$ are the cache entries of the context at time point t. By using Equations 4.1 and 4.2 the average number of ways for each context is calculated and presented in Figure 4.7.

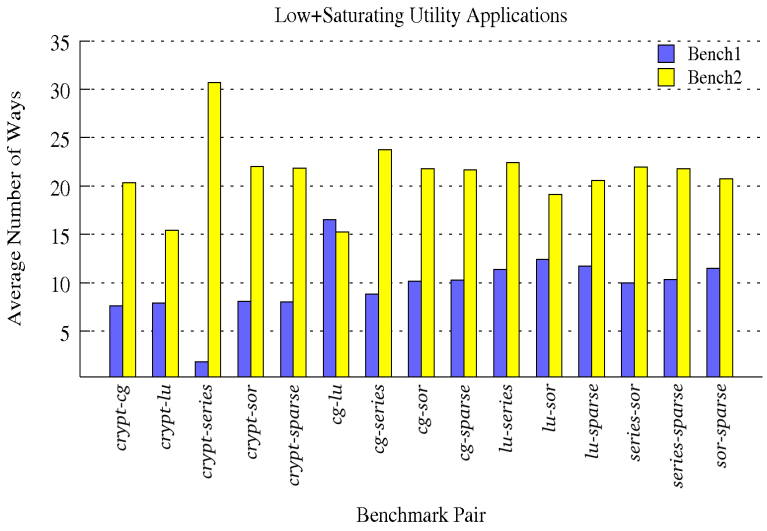


Figure 4.7: Average L2 cache ways occupancy

The number of cache ways that *cg*, *crypt* and *lu* occupy is equal to or greater than the estimates that were deduced from the single processor system simulations and included in Table 4.2. The only exception is when *lu* is paired with a low utility application. In these cases, *lu* gets around 11 ways on average and still achieves almost maximum performance which, according to Figure 4.3(c), would require around 15 ways. However, as it has been noted before, these are rough estimates and they tend to overestimate the space requirements of the application. On the other hand, when *sor* is executed together with *lu* or *sparse*, it occupies fewer than 20 ways. In every other case it gets around 22 ways, which is equal to the estimate in Table 4.2. This explains the differences in the performance of *sor* that were previously observed in Figure 4.6.

Similar observations can be made when a high utility application is scheduled together with another application. *Heap* is executed together with every other benchmark and the results are presented in Figure 4.8. Again, it is obvious that *heap*'s performance is sensitive to the application selected as its co-runner. Its IPC ranges from 0.59 to 0.68, while the maximum according to its profile in Figure 4.2(b) is 0.71.

To explain the difference in the achieved performance, the amount of cache space available to *heap* in each case needs to be examined. Using Equation

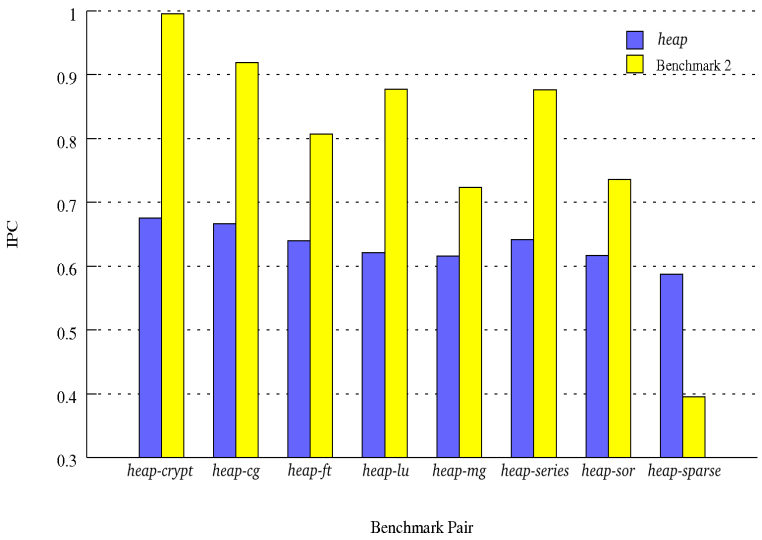


Figure 4.8: Performance of *heap* and co-runners

4.2 again, the average number of ways occupied by each of the two parallel contexts is calculated, presented in Table 4.3 and plotted in Figure 4.9.

Pair	**Average Ways**	
	Heap	**Benchmark 2**
heap-crypt	30	3
heap-cg	24	9
heap-ft	22	12
heap-lu	22	11
heap-mg	18	16
heap-series	22	13
heap-sor	18	15
heap-sparse	15	17

Table 4.3: Average number of ways occupied by *heap* and co-runners

The results reveal that when *heap* is scheduled together with *series*, it occupies around 22 ways out of the available 32, while its co-runner takes around 13[1]. However, according to its profile in Figure 4.1(c), *series* should not need more than 2 or 3 cache ways to achieve the same performance as when running

[1]In some cases the sum of the ways estimated to be occupied by the 2 contexts can be different from the actual number of L2 cache ways. This can be attributed firstly to the accuracy employed for the necessary divisions during the calculation of the average and secondly to the fact that this is an average over a period of time, during which the application can go through different phases characterised by different space requirements.

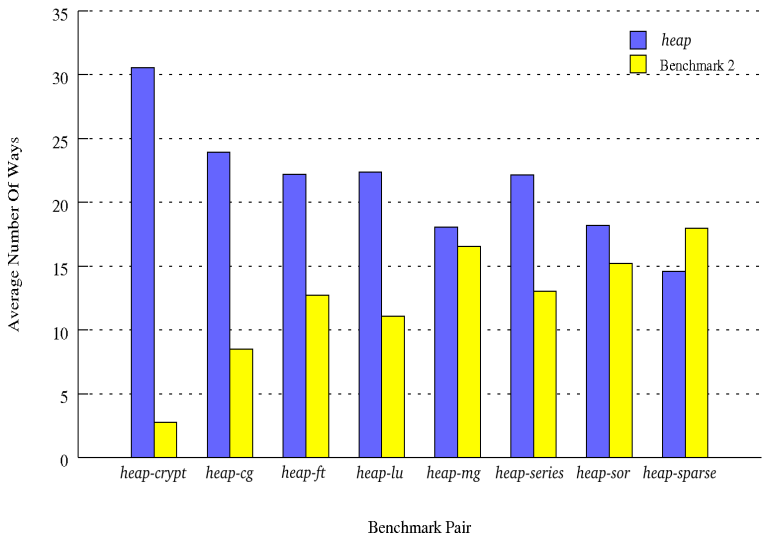


Figure 4.9: L2 cache ways occupancy for *heap* and co-runners

alone. This overbooking of resources by *series* is tolerable if the cache is still able to accommodate the working set of the other application. Otherwise it can be costly, as shown in this case, where *heap* could have used the extra cache space to improve its performance and thus the overall system performance. Similar observations can be made when *heap* is executed together with *crypt*.

At the same time, *sor* occupies only 15 cache ways instead of the desired 22. Thus it does not achieve maximum performance, managing an IPC of around 0.72. According to its profile in Figure 4.1(a) though, the same level of performance could be achieved by using no more than 3 or 4 ways. So, if *sor* was forced to occupy only 4 ways, then *heap* would be able to exploit 28 instead of 18 ways, causing its performance, and thus the total throughput of the system, to increase.

LRU treats all the requests equally, regardless of which process is issuing them. Fortunately, in some cases the cache is big enough to accommodate the working set of both applications and the system's performance is not degraded. However, as the previous observations illustrate, there are cases where that replacement policy results in suboptimal sharing of the L2 cache between the parallel running contexts, hurting the overall performance of the system.

4.1.3 Quad Core System

In the previous subsection, evidence was presented that it is possible for the LRU policy to result in suboptimal sharing of the cache in a dual core system. This possibility becomes more significant when the number of processors increases. As the cache needs to satisfy the requirements of more applications, cache space becomes a valuable commodity and its allocation between the competing processes has an increasingly significant impact on the system performance. To illustrate this better, a selection of cases was simulated in a quad core system.

Case	App.	Performance		Cache Ways	
		IPC	Single IPC	Occupied	Preferred
1	*crypt*	0.995	0.997	2	1
	heap	0.58	0.71	16	32+
	lu	0.85	0.89	11	15
	series	0.87	0.87	7	2-3
2	*crypt*	0.995	0.997	2	1
	heap	0.55	0.71	13	32+
	series	0.87	0.87	5	2-3
	sparse	0.37	0.59	15	22
3	*heap*	0.54	0.71	10	32+
	lu	0.67	0.89	8	15
	series	0.87	0.87	4	2-3
	sparse	0.37	0.59	13	22

Table 4.4: Results of quad core system simulations

The parallel running applications are sharing a 4MB, 32-way L2 cache. Table 4.4 presents the achieved IPC and the average number of cache ways occupied by each benchmark. It also contains the performance of each application when it executes in isolation, which is derived from the appropriate profiles in Figures 4.1, 4.2 and 4.3, as well as its estimated cache space requirements, which were presented in Table 4.2.

In the first case, *crypt* and *series* achieve the same performance as if they were running alone. However, they appear to occupy more ways than actually needed. For example, *crypt* exhibits such a frequency of accesses to the L2 cache, that it manages to occupy 2 ways in average, while it really needs 1.

The extra ways that *crypt* and *series* occupy could have been used to improve the performance of *heap* and *lu*. The same problem occurs in the second case. Moreover, *sparse*'s profile in Figure 4.1(b) reveals that it would have accomplished the same level of performance by using only 7 or 8 ways. Again, these extra ways could have been exploited by *heap* to boost its performance and thus the overall performance of the system. Similar observations can also be made for the third case, where *series* and *sparse* grab more ways than they need, hurting the performance of *heap* and *lu*.

4.1.4 Conclusion

Running different combinations of benchmarks on a dual and a quad core system showed that the LRU policy does not guarantee optimal sharing of the cache. In several cases, a process was allowed to occupy more cache space than it needed, causing the performance of the other processes to degrade. This can be attributed to the fact that LRU is a 'thread blind' policy that treats all the requests equally, irrespective of which application is issuing them.

Another scheme is needed, that will be able to identify the space requirements of each application and partition the cache in such a way, that will increase the overall performance of the system.

4.2 Static Cache Partitioning

In Section 4.1 it was suggested that the LRU replacement policy could fail to provide optimal sharing of the L2 cache between parallel running contexts. Instead it was proposed that the total throughput of the system, defined as the sum of instructions executed by all the competing applications per cycle, could be improved by enforcing limits on the amount of cache space each process is allowed to use. To further illustrate this proposal, a few examples will be provided.

The same dual core system shown in Figure 4.4 is used to run *heap* paired with a set of other benchmarks. The cores are still sharing a 4MB, 32-way associative L2 cache but the replacement policy is changed to implement way partitioning [60]. One bit is added to the tag of each cache entry to identify

which of the 2 processors brought it into the cache. Each core is also assigned a maximum number of ways. When a cache miss occurs, the entries that belong to the application in the accessed cache line are counted. If they are fewer than the number of ways allocated to the application, then the LRU entry that *does not belong to the application* is selected for replacement. Otherwise, the LRU entry of the application that caused the miss is evicted.

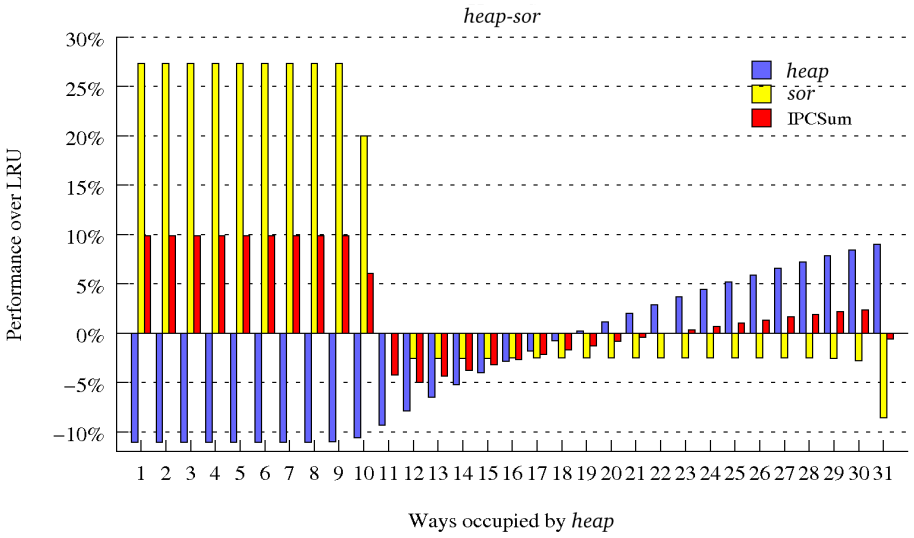


Figure 4.10: Performance gains/losses for *heap* and *sor* for different partitions

First *heap* is executed together with *sor* and in Figure 4.10 the achieved performance is compared to the performance of a system where the LRU policy is used. IPCSum is the overall system performance given by :

$$IPCSum = \sum_{i=1}^{N} IPC_i \tag{4.3}$$

where IPC_i is the IPC of the ith process and N the number of processors. If *heap* is allowed to use only up to 9 ways, then its performance drops by around 10%. However, *sor*'s performance is increased by almost 28%, causing the total throughput of the system to increase by 10%. If *heap* is given more than 11 ways, its performance starts to improve. At the same time though, *sor*'s performance degrades, causing the total performance of the system to be worse than when using the LRU policy. When *heap* is given 23 or more ways, the

improvement of its performance is greater than the losses suffered by *sor*, resulting in increased total throughput. Finally, when *sor* is limited to only 1 way, its losses are so great, that the total performance of the system is worse. In this case, sharing the cache roughly equally between the processes is the *worst* strategy.

Similar observations can be made when *heap* is scheduled together with another high utility application, as shown in Figure 4.11. Unfortunately though, for some pairs it is not possible to perform better than when using the LRU policy. This is the case when *heap* is executed with an application of saturating utility and it is illustrated in Figure 4.12, where the results of running *heap* and *lu* are presented.

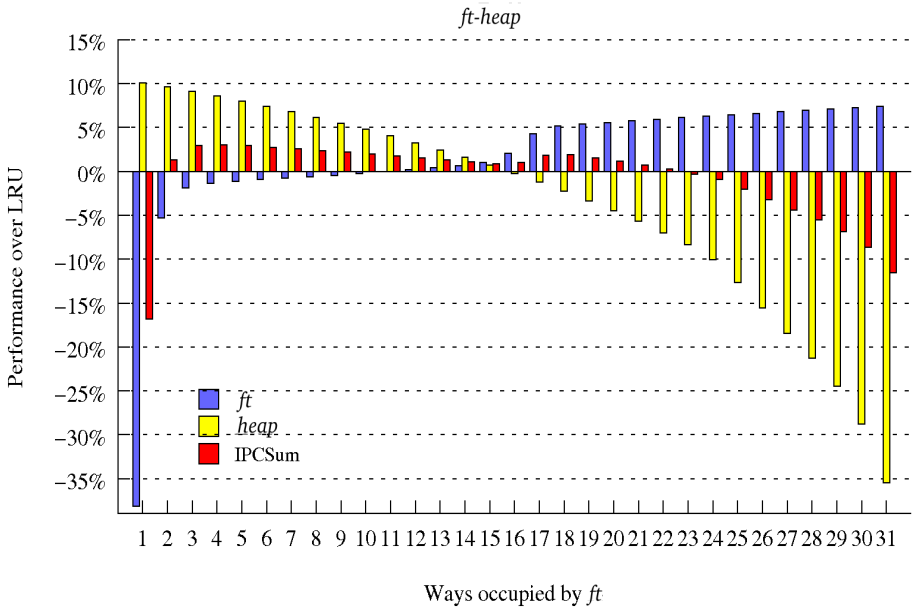


Figure 4.11: Performance gains/losses for *ft* and *heap* for different partitions

It has been shown that by enforcing appropriate partitions, it is sometimes possible to improve the total throughput of the system compared to using the traditional LRU replacement policy. However static partition schemes have several drawbacks. First of all, they rely on prior knowledge of the characteristics of the applications that are going to be executed. Unfortunately this knowledge can only be acquired by profiling the applications offline. Moreover, it is not

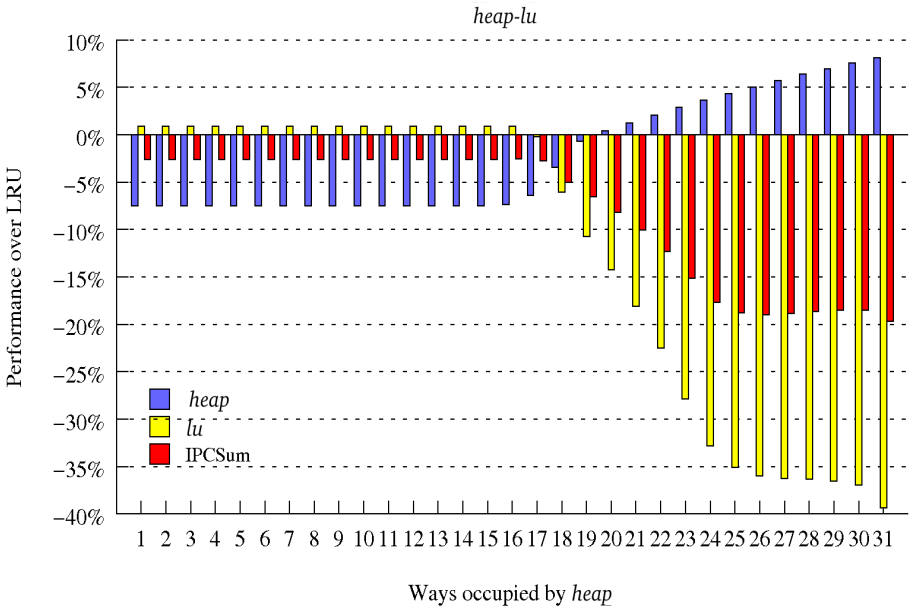


Figure 4.12: Performance gains/losses for *heap* and *lu* for different partitions

trivial to deduce the optimal partition from these profiles and as the number of processors and competing applications increases, the process of discovering the best allocation becomes more and more complex.

Secondly, the partition is enforced from the start until the end of the execution. A program though could go through different phases and each one may well have different cache space requirements. At the same time, if the wrong partition has been selected, there is no way of discovering and amending that mistake.

Another drawback is that the same partition is enforced for all the lines of the cache, a decision based on the assumption that each application places the same load on every line. However, this may not be true, as an application could have a small working set that fits in a subset of the cache lines. By enforcing in this case the same partition for every line, the application is allocated space that it does not need and could have been exploited by another process. An example is illustrated in Figure 4.13, where Application A has been allocated 5 out of the available 8 ways, even though it does not use the last 3 lines.

In summary, the results from this static cache partitioning scheme illustrate

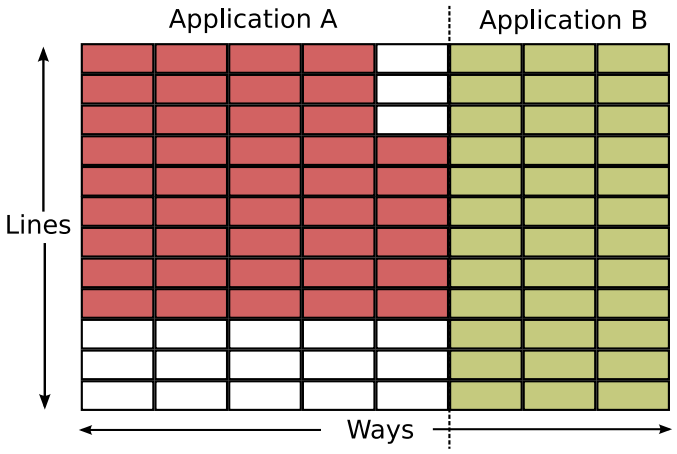


Figure 4.13: Example of non optimal static partitioning

that it is possible to achieve a better performance compared to using the LRU policy. Unfortunately, there are several drawbacks that if any gains are to be made, there needs to be some form of dynamic adaptability.

4.3 Conclusions

This chapter presented several results from simulations of single, dual and quad core systems, in an attempt to evaluate the usage of the LRU replacement policy in CMPs. These show that the allocation of cache space between the competing applications performed by the LRU is not always optimal. This suboptimal sharing results in degradation of the overall system's performance.

Based on these observations, a static cache partitioning scheme was implemented and simulated. The presented results suggest that enforcing restrictions on the space that each application occupies in the cache could improve the performance. However, this scheme has several drawbacks.

To solve these, another scheme needs to be developed, that is capable of identifying, on the fly, the characteristics of the running applications. These will be used to partition the cache dynamically, adapting to the different execution phases of the processes. Of course the complexity and hardware overhead over the LRU policy must also be taken into consideration.

Chapter 5

Dynamic Cache Partitioning

LRU is a 'thread blind', demand driven, replacement policy. When a cache miss occurs, the replacement mechanism selects a victim by looking only at the order in which the entries in the accessed line have been used. There is no information about the 'ownership' of each entry or the relative occupancy of the cache by each application. Therefore, applications with a high demand, i.e. many accesses to different entries, are allocated more cache space than applications that have a low demand.

However, there is no guarantee that the new entries, brought into the cache because of a miss, will be reused. So by replacing entries owned by other processes, an application could be acquiring space that it does not need, resulting in suboptimal sharing of the cache. Consequently the performance of the system degrades. A representative example of such cases is when one of the competing processes is a streaming application.

Chapter 4 presented evidence of problems caused by LRU and proposed cache partitioning as a solution. This chapter evaluates different schemes that attempt to partition the cache dynamically.

5.1 LRU Variation

5.1.1 Overview

To avoid suboptimal sharing of the cache a variation of LRU was investigated. On a cache miss, the LRU entry in the accessed line is looked up. If it belongs to the application that caused the miss, then it is selected for replacement. Otherwise, the LRU entry of the miss-causing application is looked up and its age, i.e. the amount of time since it was last used, is multiplied by a weight and then compared with the age of the global LRU entry. The older of the two is rejected from the cache.

The reasoning behind this scheme was to make it harder for an application that has high demand to reject the least recently used entries owned by the other processes. Thus, it will be more difficult for such applications to claim the majority of cache resources. At the same time, the implementation of such a scheme should be reasonably simple.

5.1.2 Evaluation

This LRU variation was tested on a dual core system. To implement it, the simulator records which processor brings each entry into the cache. To model age, the cycle at which each entry is accessed is recorded as well. On a cache miss, the stored cycle values are subtracted from the current cycle count.

The two processors share a 4MB, 32-way associative L2 cache. Every benchmark pair was simulated using a set of different values for the weight used to scale the age of a cache entry. The total throughput over the case where the traditional LRU policy is employed is shown in Figure 5.1.

It is obvious that this scheme fails to improve the performance of the system for the majority of the simulated cases. Moreover, many times it seems to partition the cache worse than the traditional LRU policy, causing the system's performance to degrade. To understand the reasons for that failure, a few cases will be studied in more detail.

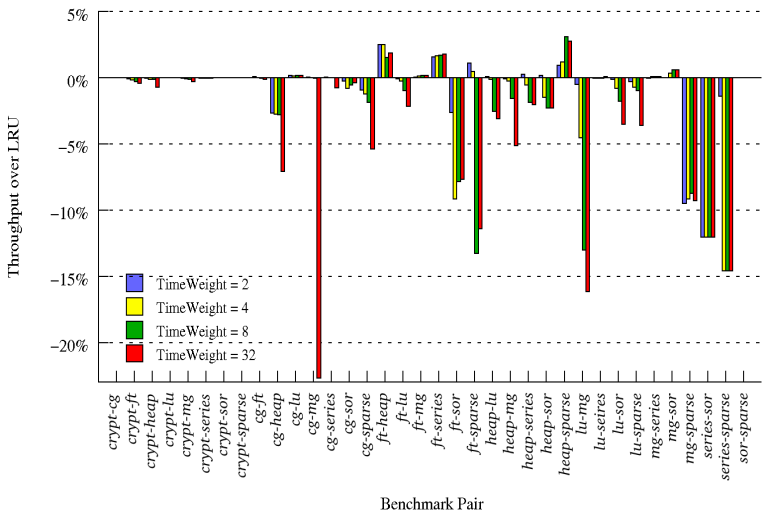


Figure 5.1: Throughput over LRU for different time weights

One of the few combinations where the performance is improved is "*heap-sparse*". The average number of cache ways allocated to each application is shown in Figure 5.2. This LRU variation favours *heap* and allocates it more cache resources than the normal LRU. To evaluate the success of this decision, an estimate of potential gains that can be achieved by partitioning the cache is needed. This is provided by the simulations performed for the static partition scheme described in Section 4.2 and presented in Figure 5.3. It appears, that, although the total throughput of the system has been improved, the sharing

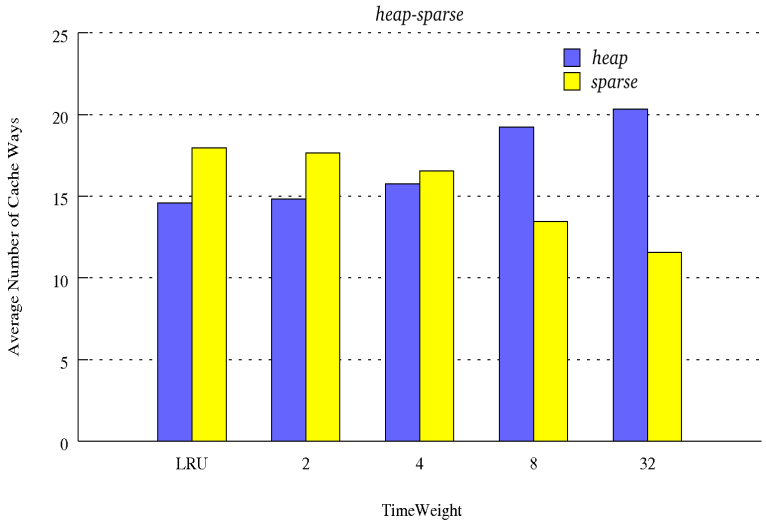


Figure 5.2: Average number of cache ways occupied by *heap* and *sparse*

of the cache is still suboptimal. In fact, if more space had been allocated to *sparse* instead of *heap*, the improvement of system's throughput would have been much greater.

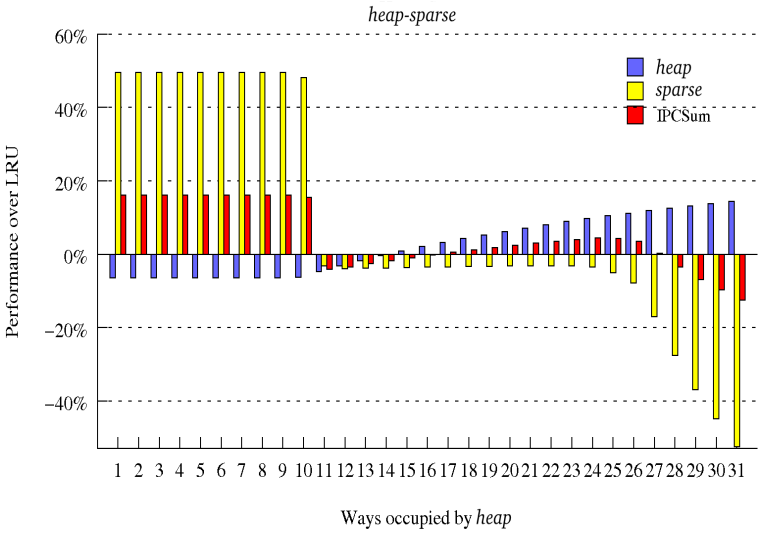


Figure 5.3: Performance over LRU for static cache partitioning scheme

On the other hand, this scheme performs worse for the "*series-sparse*" combination. The average number of ways assigned to each process is plotted in Figure 5.4 for LRU and the different weights. The replacement policy allocates more resources to *series* as the weight increases. However, as it was described in Section 4.1.1 and presented in Table 4.2, *series* can achieve the same performance as when running alone by using only 2-3 cache ways. Consequently, in order to maximise the performance, *sparse*, and not *series*, should have been the recipient of the majority of cache space.

Summarising, scaling the age of the cache entries when looking for a replacement victim does not appear to solve the problem of allocating more space to the applications with high demand. Even when the performance of the system is improved, the sharing of the cache is still not optimal. In total, this LRU variation fails to identify the characteristics of each application. Therefore, it cannot make the decision that will result in optimal sharing of the cache and will maximise the performance of the system.

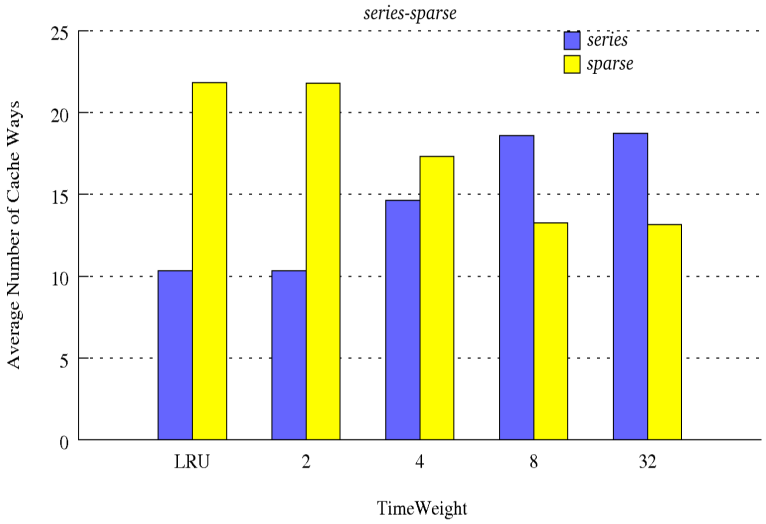


Figure 5.4: Average number of cache ways occupied by series and sparse

5.2 Monitoring Schemes

5.2.1 Overview

A major drawback of the LRU policy and its variation presented in Section 5.1 is that there is no feedback on how the cache space allocation is affecting the running applications. Based on that observation, different schemes have been proposed that monitor the processes and then use this information to determine the optimal sharing of the cache. This work focuses on two of the latest schemes developed for cache partitioning. The first one is the "*cache partitioning aware replacement policy*" implemented by Dybdahl *et al.* [15] and the second is the "*utility-based cache partitioning*" proposed by Qureshi *et al.* [49].

The LRU policy exhibits what is known as the *stack property* [38]. More specifically, an access that hits in a N-way associative cache using this replacement policy is guaranteed to hit also if the cache had more than N ways, provided that the number of lines remains the same.

Figure 5.5 presents an example of how the hits of an application using a N-way associative cache may be distributed. According to the stack property, the accesses that hit on the MRU entries will still be hits even if the application was using a direct mapped cache. Furthermore, if the cache ways were reduced

from N to 4, then the application's hits would be equal to the sum of the first 4 bars. At the same time, the number of misses would increase by the sum of bars 5 to N.

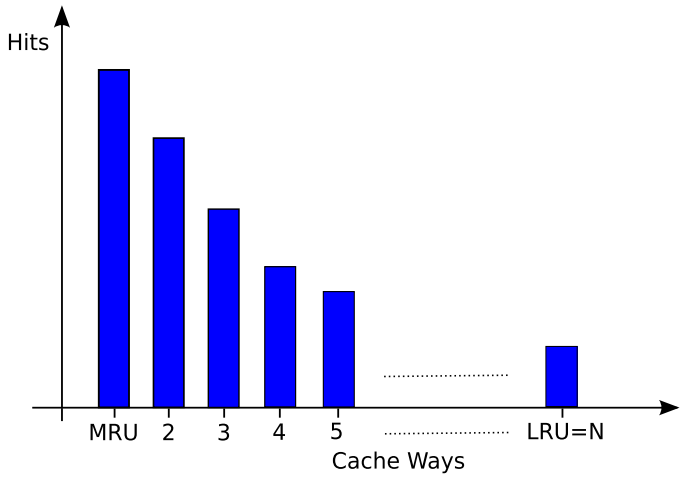


Figure 5.5: Distribution of an application's hits

This property was first exploited for cache partitioning in the scheme proposed by Suh *et al.* [60], on which the two previously mentioned proposals are based. Suh *et al.* suggested keeping a set of N counters for each running process sharing a N-way associative L2 cache. On a cache hit, the appropriate counter of the hit-causing application is increased. If the hit is on the LRU entry of the line, then $counter_{N-1}$ is updated. If the hit is on the MRU entry, $counter_0$ is increased.

Exploiting the stack property and assuming that the past behaviour is an accurate estimate of the future, the counters are used to estimate marginal gains and losses. If an application occupies m out of the N ways of the cache, then the marginal gain of obtaining one more way can be approximated by the number of hits on the $(m + 1)$-th most recently used block, which is the value of $counter_m$. Respectively, the marginal loss of losing one way, is approximated by the value of $counter_{m-1}$.

Every T cycles the cache is repartitioned. The algorithm first allocates the cache ways randomly to the competing applications. Then, using the appropriate counters, ways are taken from processes that will lose the least by giving

up one way, i.e. processes with the smaller marginal losses, and they are given to the applications that will benefit most by having one more way, i.e. applications with the greater marginal gains.

Eventually, a partition estimated to maximise the total hits of the system, and thus the overall performance, is determined and normal execution is resumed. However, if a process has a non-convex hits distibution curve, then a local maximum could have been identified instead of a global maximum. The ability to discover the global maximum depends on how wide the local maxima are.

Two options are suggested for implementing the deduced partition. The first one is on a per way basis and is implemented by column caching [10]. Each process is restricted to replacing a set of ways as the partition dictates.

The second option is to modify the LRU policy. The partitioning algorithm still estimates the number of ways each process is allowed to occupy. That number is then multiplied by the number of cache lines to produce the allowed total number of entries per application. During execution, the number of each processor's entries in the whole cache is monitored. On a cache miss, if that number is smaller than the limit imposed by the partition, the LRU entry of an over-allocated application is selected for replacement. Otherwise, the LRU entry of the miss-causing process is rejected.

5.2.2 Cache-Partitioning Aware Replacement Policy

Dybdahl *et al.* proposed a cache-partitioning aware replacement policy [15]. Their scheme differs from the proposal by Suh *et al.* described in Section 5.2.1 mainly on two points. First of all, they suggested adding an extra register, called "*shadow tag*", in every cache line for each processor. When an entry is rejected, its tag is stored in the appropriate shadow tag register, depending on which processor had brought it into the cache.

In addition, two counters per processor are used for each cache line. If a processor's access hits on the LRU entry owned by it, then the first counter, "*LRU hits*", is increased. If the access is resolved as a miss, the requested tag is compared to the contents of the processor's shadow tag register. In case of

a tag match the second counter, "*shadow tag hits*", is increased.

According to the stack property of the LRU policy, the *LRU hits* counter monitors the hits that would become misses if the application was using one less way. On the contrary, the *shadow tag hits* counter records the portion of the application's misses that would have been hits, if the application had been allocated one more way.

The second main difference is that this scheme partitions the cache on a cache line granularity. Every 2000 cache misses the repartitioning algorithm is executed. For every line, the shadow tag hits counter with the highest value, i.e. the process with the highest gain when allocated more cache space, is compared to the LRU hits counter with the lowest value, i.e. the process with the lowest loss when deprived of a cache way. If the gain is higher than the loss, then a way is removed from the latter process and allocated to the former.

The scheme proposed by Dybdahl *et al.* was implemented and a dual core system sharing a 4MB 32-way associative cache was simulated. The performance over a system using the normal LRU policy is presented in Figure 5.6. The simulated policy was able to improve the overall performance of the system only for one out of the 36 benchmark combinations. In fact, for almost half of the combinations the system's performance was degraded by an average of 5% while the maximum degradation was almost 15%.

Dybdahl *et al.* reported an average improvement of around 7% [15] for a quad core system. In an attempt to reproduce these results, a system with four cores sharing a 4MB L2 cache was simulated. However, as Figure 5.7 shows, it was not possible. Again for the majority of the 126 simulated benchmark combinations the overall system's performance was degraded.

Dybdahl *et al.* reported their results for a system where 4 processors shared a 4MB L3 cache and each one had its own private L2 cache, while in these experiments the 4 cores share a 4MB L2 cache. However, this cannot account for the difference observed in the results, as the system used by Dybdahl is the equivalent of the simulated system, but with a bigger L1 cache.

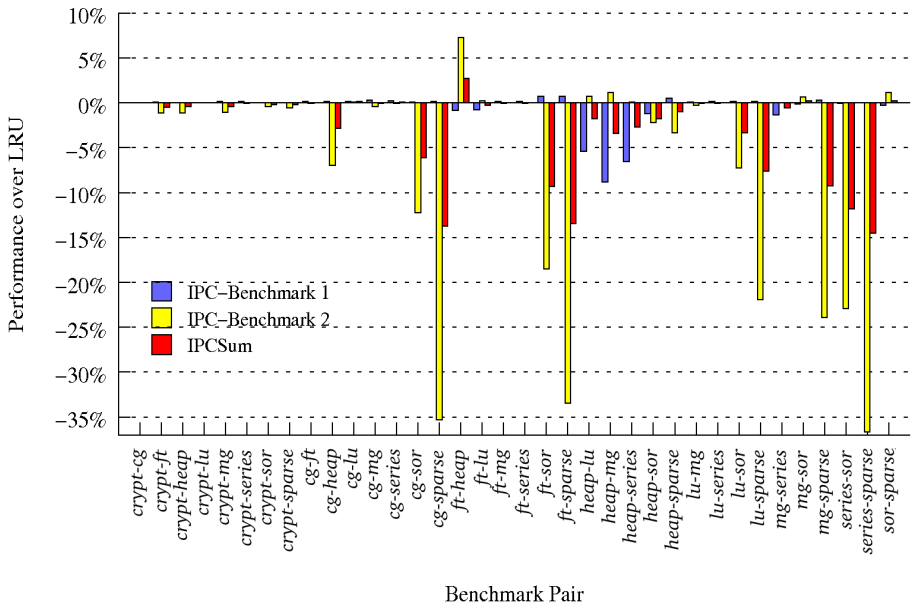


Figure 5.6: Cache partitioning aware replacement performance over LRU for a dual core system

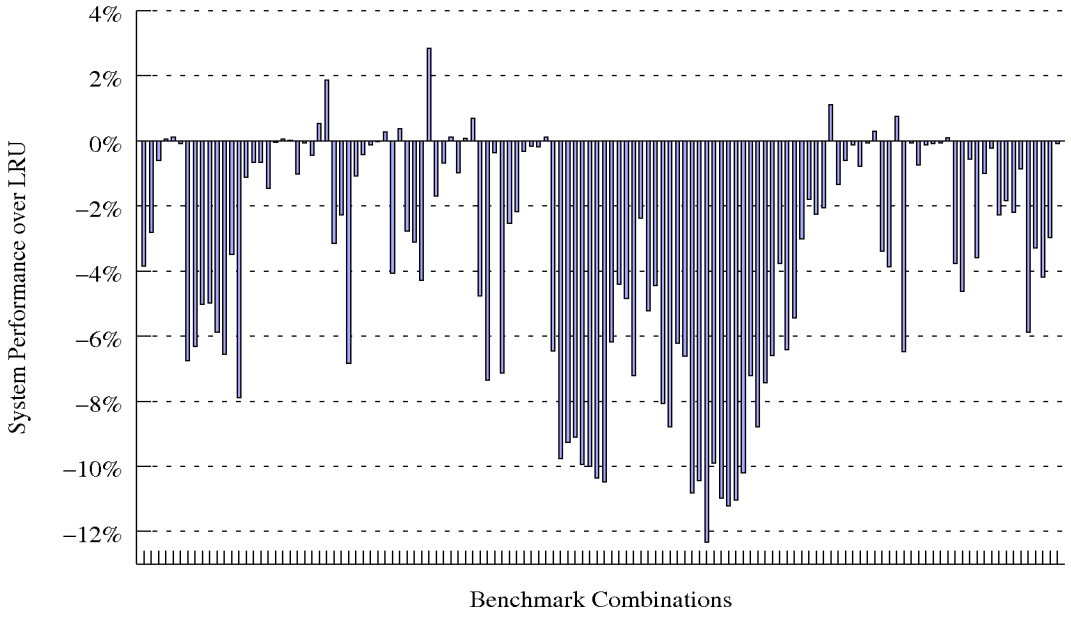


Figure 5.7: Cache-partitioning aware replacement performance over LRU for a quad core system

On the other hand, the discrepancies could be attributed to the usage of different benchmarks. Even so, the results presented in Figures 5.6 and 5.7 raise considerable doubts about the efficiency of the proposed scheme.

5.2.3 Utility-Based Cache Partitioning

Utility-based cache partitioning was developed by Qureshi *et al.* [49]. They suggested assigning a "*utility monitoring*" (UMON) circuit to each processor in order to identify the characteristics of the applications executing on them. A UMON circuit has its own tag array that has the same associativity as the tag array of the shared L2 cache and uses the LRU replacement policy. By tracking the assigned processor's accesses to the L2 cache, the UMON circuit effectively emulates a L2 cache that is private to that core.

Similarly to the previously described schemes, in order to exploit the stack property of the LRU policy a set of N counters is included in each UMON circuit, where N is the associativity of the shared cache. These counters record the hit counts for each of the N recency positions ranging from MRU to LRU. As it was described in Section 5.2.1, Suh *et al.* record similar information by attaching counters to the shared cache. However, as Qureshi argues, this has some drawbacks.

First, the number of entries in each line that the counters can monitor for a given application depends on the other applications. Secondly, when a process gets a hit in the shared cache, the recency position of the accessed entry has also been affected by accesses from the competing applications. On the other hand, a processor's UMON circuit emulates the state of the cache, had it been used only by that specific processor. Therefore the information deduced in this case is not polluted by the concurrently executing applications and can be used with greater confidence to determine the cache partitioning.

Every five million cycles the partitioning algorithm is executed. As the partitioning is done on a way-based granularity, the algorithm evaluates all the possible allocations of the cache ways amongst the competing applications by reading the hit counters from all the UMON circuits and calculating the total number of hits for each case. The partition that maximises the total hits of the system is selected.

To implement the partition, the LRU policy is modified. On a cache miss, the entries belonging to the miss-causing application are counted. If that number

is less than the limit imposed by the partition, then the LRU entry that does not belong to the application is evicted. Otherwise, the LRU entry owned by that application is selected for replacement.

Finally, in an attempt to implement a scheme with low hardware overhead, Qureshi *et al.* proposed *Dynamic Set Sampling*. They showed that the similar improvements of the overall system's performance can be achieved when each UMON circuit monitors only 32 out of the 1024 cache lines.

The utility-based cache partitioning scheme was implemented and evaluated by simulating a dual core system sharing a 4MB 32-way associative L2 cache. The results, presented in Figure 5.8, show that this scheme manages, for a subset of the simulated combinations, to improve the performance of the system compared to a system using the traditional LRU policy. In addition, degradation of the performance is only observed for a few cases and is relatively small.

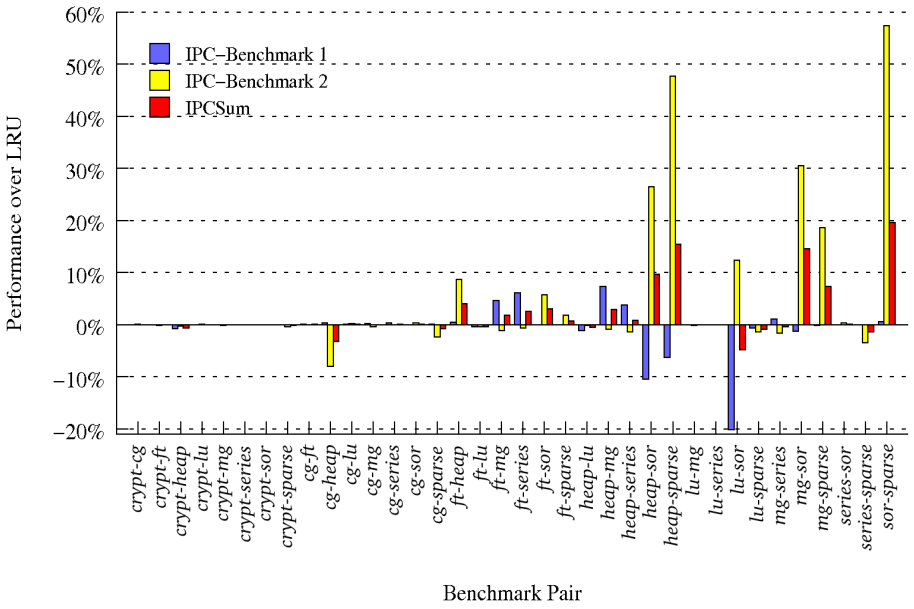


Figure 5.8: Utility-based partitioning scheme's performance over LRU for a dual core system

However, for many benchmark combinations the simulated scheme does not provide any improvements. This could be attributed though to the cache being large enough, allowing the LRU policy to achieve optimal performance. To test

this assumption, the scheme was evaluated in a quad core system and the results are presented in Figure 5.9.

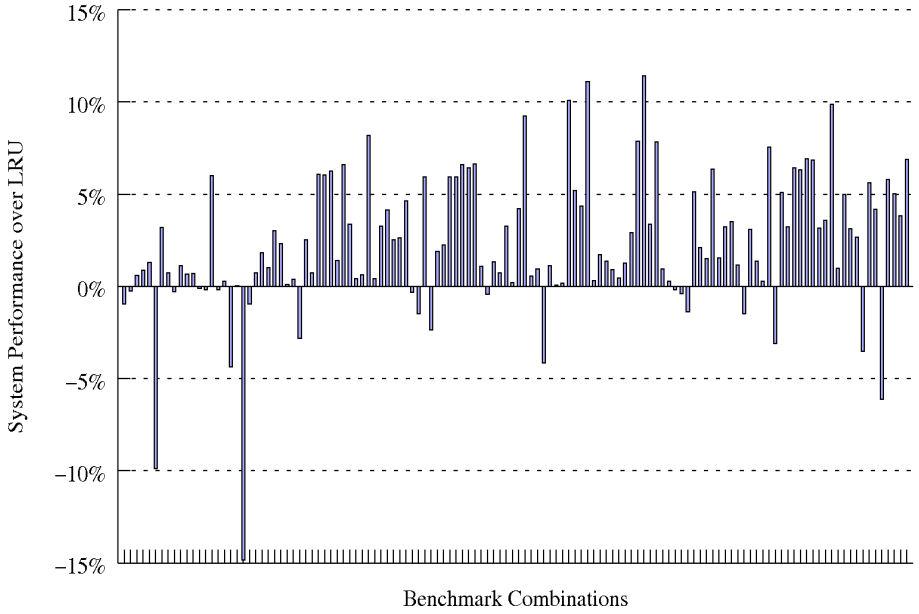


Figure 5.9: Utility-based partitioning scheme's performance over LRU for a quad core system

In this case, where more applications are competing for the cache space and its allocation has a more significant impact on the overall performance, the utility-based cache partitioning scheme performs better than the LRU for the majority of the simulated combinations. Still, there are a few cases where the achieved performance is worse than when LRU is used.

To investigate these, the utility-based cache partitioning scheme was reimplemented and the compromises made to reduce the hardware overhead were removed. More specifically, the UMON circuits were built to monitor all the cache lines and the partitioning was performed on a line instead of a way granularity, allowing each line to have a different partition. This "ideal" scheme was evaluated again for a quad core system and the performance over LRU is shown in Figure 5.10.

Comparison with Figure 5.9 reveals that the previously mentioned cases, where degradation of performance was observed, have now disappeared. In fact, the

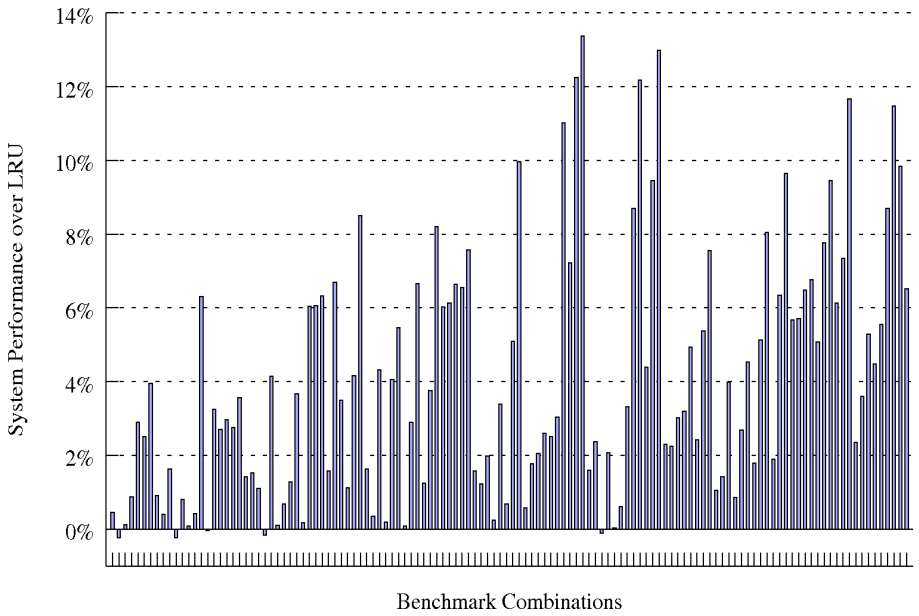


Figure 5.10: Ideal scheme’s performance over LRU for a quad core system

achieved performance gains for the ideal scheme are greater for the majority of the simulated benchmark combinations. Therefore, the assumption that the afore observed performance degradations are due to the compromises made to lower the hardware overhead, is correct. However, as the scheme still manages to achieve better results than the LRU for the majority of combinations, the required sacrifices, which render its practical implementation worthwhile, may be considered justifiable.

5.3 Partitioning ‘on the fly’

5.3.1 Description

A common property of all the cache partitioning schemes described in Section 5.2 is the monitoring of the parallel executing applications for a specific period of cycles. The ‘optimal’ partition is then decided based on information gathered in that period. The length of this monitoring period needs to be carefully selected to be able to track the dynamic changes in the behaviour of the applications.

A different approach would be to determine the appropriate partition on the

fly, every time an application misses. This could make it easier for the system to adapt to the characteristics of the running processes. However it is not practically possible to determine if every miss would have been a hit, had the application been allocated more space. Therefore, the decision was made to limit the scope of such a scheme to *"near misses"*, i.e. misses that would have been hits if the process had been allocated one more cache way.

To implement this approach, the scheme proposed by Dybdahl *et al.* and described in Section 5.2.2 was modified. The "shadow tag" registers and the "LRU hits" counters were retained. However, as there is no monitoring period, the "shadow tag hits" counters were discarded, thus lowering the hardware overhead.

On a cache miss, the requested tag is compared to the contents of the processor's shadow tag register. If there is no match, the LRU entry amongst the application's entries is selected for replacement. A tag match detects a *near miss* and signals the application's desire for one more cache way. This is obtained from the process with the lowest estimated performance loss when deprived of a way. The repartitioning is implemented by rejecting the LRU entry that belongs to the process with the minimum LRU hits counter value. After every repartition, the LRU hits counters associated with the accessed line are reset to zero.

5.3.2 Evaluation

The proposed scheme was evaluated for a dual core system sharing a 4MB, 32-way associative L2 cache. Figure 5.11 presents the performance over a system using the normal LRU policy. For half of the simulated cases there was no difference between this scheme and LRU. For a few cases the performance was slightly improved, with the maximum improvement of 4% observed for *heap-sparse*. However, for all the other cases the overall performance was degraded by an average of 9% while the maximum degradation was almost 21%.

To gain a better insight into the behaviour of the scheme, the average number of ways each application is allocated was examined. Every 10 million cycles the average cache occupancy for each benchmark of the *heap-sparse* combination

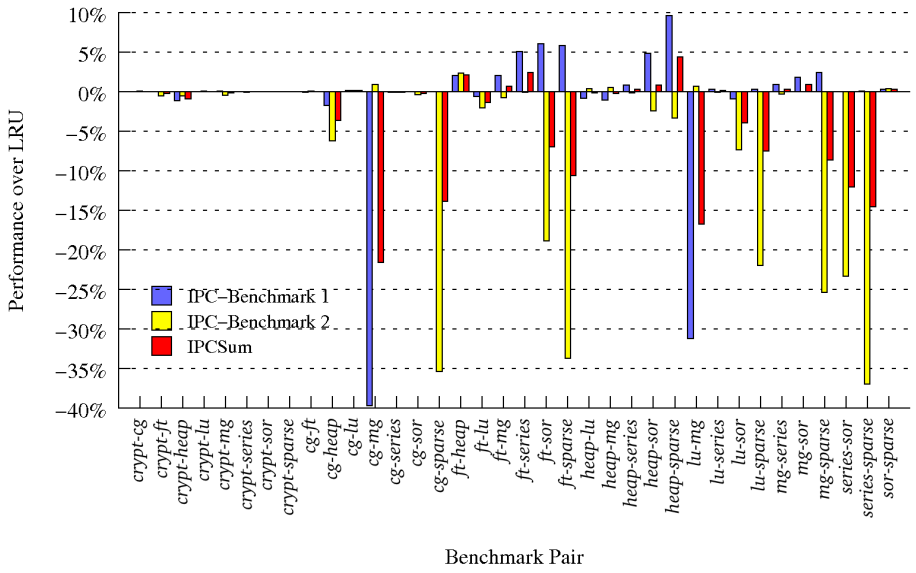


Figure 5.11: Performance over LRU when repartitioning every near miss

was calculated and the results are shown in Figure 5.12 for two systems, one using the normal LRU replacement policy and one using the developed cache partitioning scheme. By allocating more cache space to *heap* the performance of the system is improved. However, as it was shown in Figure 5.3, the improvement would have been much greater, if *sparse* had been the recipient of the majority of cache space.

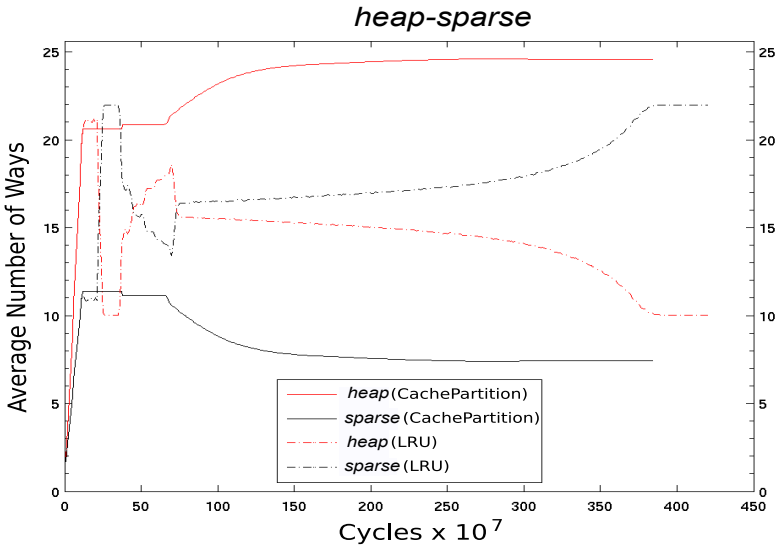


Figure 5.12: L2 cache ways occupancy for heap and sparse

The same analysis was performed for *series-sparse*, where the partitioning scheme degrades the system's performance, and the results are presented in Figure 5.13. In this case, *series* receives almost two times the cache space it is allocated when LRU is used. However, as it was described in Section 4.1.1 and presented in Table 4.2, *series* can achieve the same performance as when running alone by using only 2-3 cache ways. Consequently, in order to maximise the overall system's performance, *series* should have received fewer cache ways than when LRU is employed.

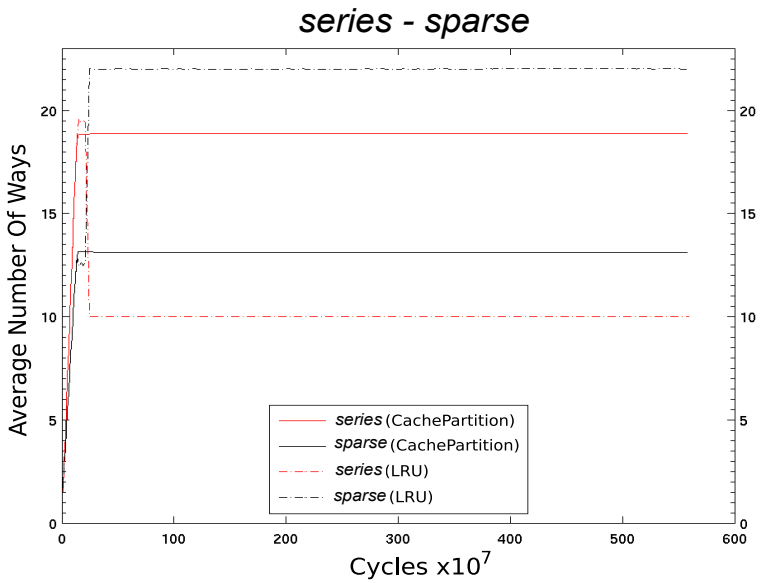


Figure 5.13: L2 cache ways occupancy for series and sparse

It appears that this cache partitioning scheme is unable to detect the requirements of the applications. At first it was assumed that this was due to the resetting of the LRU hits counters every time a line was repartitioned, as this does not guarantee that if an application is allocated one more cache way, it will be able to retain the extra space long enough to exploit it. Therefore the following strategies were considered :

1. **ZERO:** Reset the counters on a *near miss*.
2. **DIV2:** Divide the counters by 2 on a *near miss*.
3. **TIME:** Reset the counters every *T cycles*.

4. **AVG:** After a repartition, the average value of the counters associated with the two affected processes is calculated. The two counters are set to that average while the others remain unaffected.

These variations were implemented and evaluated for a dual core system. However there was no improvement and the results were almost identical to the results presented in Figure 5.11.

5.3.3 Conclusions

The proposed scheme appears not to improve the overall system's performance, as it is not able to identify the characteristics and requirements of the applications. To speculate on the reasons of that failure, the scheme is compared to the "*utility-based cache partitioning*" scheme that was developed by Qureshi *et al.* and presented in Section 5.2.3.

Qureshi *et al.* emulate for each application the state of the cache, had it been used only by that specific process. Thus their scheme obtains an accurate profile of the application's access distribution across the N recency positions, where N is the associativity of the cache. This allows the system to modify the space allocated to the process by x ways, where x is a number between 0 and $N - 1$. On the other hand, the scheme described in this section monitors only *near misses* and hits to the *LRU position*. This limits it to adjusting the partition only by 1 way. At the same time, it fails to recognise cases where an application's allocation should be increased by more than 1 way in order to improve the overall performance.

To illustrate this better, an example is shown in Figure 5.14. Assume that the applications are sharing the cache equally, each one occupying 4 out of the 8 cache ways. At this point the total sum of hits is $75 + 60 = 135$. By obtaining the profiles of these applications, the utility-based cache partitioning scheme is able to detect that if *application 2* is allocated 6 ways, then the total hits will be increased to $55 + 85 = 145$ and the performance of the system will be improved. However, the proposed scheme monitors only the hits in the LRU positions, obtaining only the information inside the dashed box. If *application 2* suffers a near miss, the LRU hit counters are looked up and since the counter's

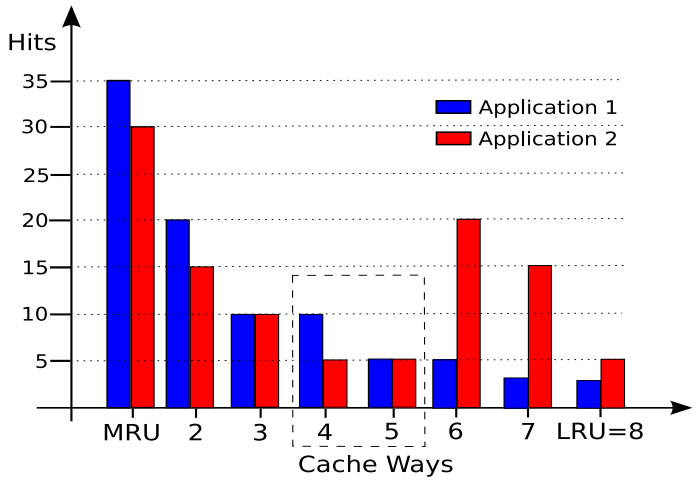


Figure 5.14: Example of 2 applications' profiles that share a L2 cache

value for *application 1* is higher than that associated with *application 2*, the latter will not be allocated an extra cache way. Consequently, optimal sharing cannot be achieved.

Another major difference is that the "utility-based cache partitioning" scheme determines the optimal partition based on information gathered in a specific monitoring period. The simulations' results presented in Sections 5.2.3 and 5.3.2 reveal that this approach is superior to repartitioning on the fly, i.e. on every "near miss". It appears that in order to be able to predict the future accurately based on past behaviour, a sufficient amount of history needs to be gathered.

5.4 Summary

This chapter presented several cache partitioning techniques. First, a variation of the LRU replacement policy was evaluated. This scheme failed to improve the overall performance of the system. Analysis of the results identified as a major drawback the lack of feedback on how the cache space allocation is affecting the running applications.

Therefore, schemes that monitor the processes and then use this information

to determine the optimal partition were introduced. More specifically, two of the latest proposals, namely "*cache-partitioning aware replacement policy*" and "*utility-based cache partitioning*", were evaluated. The former failed to improve the system's performance, while the latter performed better than the normal LRU policy for both a dual and a quad core system.

Inspired by these, another approach that repartitioned the cache on every "near miss" was proposed. The reasoning behind this scheme was to enable a quicker adaptation of the cache space allocation to the dynamic changes in the programs' behaviour while lowering the hardware overhead. Unfortunately, that scheme failed to achieve optimal sharing of the cache and in many cases caused a significant degradation of the system's performance. Further analysis of the results, showed that a monitoring period is essential in order to estimate the future behaviour of the applications based on their history. At the same time, if any gains are to be made, the cache partitioning mechanism needs to be able to increase or decrease the allocation of a process by more than one cache way each time.

Chapter 6

A New Cache Partitioning Scheme

Several schemes that attempt to partition the cache dynamically were presented in Chapter 5. All these were evaluated and some were found to be effective while others not. The evaluation process highlighted a set of properties that a successful cache partitioning scheme must possess.

More specifically, it was suggested that the partitioning module could estimate the appropriate cache space allocation based on the past behaviour of the running processes. However, for that estimation to be accurate, history needs to be recorded for a sufficient amount of time. Additionally, this monitoring period should not be too long, as the repartitioning needs to be frequent enough to ensure a better adaptation of the system to the changing characteristics of the applications.

The performed simulations also indicated that the partitioning mechanism should be able to modify the allocation of a process by more than one cache way each time. It was shown that pathological cases exist, where limiting the allocation's changes to only one cache way could prevent the system from achieving optimal sharing of the cache.

Based on these observations, another cache partitioning scheme was designed. Its goal was to improve the overall performance of the system at a minimal cost. This chapter provides the description and evaluation of this scheme as well as a cost analysis of its implementation.

6.1 Adaptive Bloom Filter Cache Partitioning

6.1.1 Motivation

Dybdahl *et al.* [15] proposed a scheme that records only the "near-misses" of each application, i.e. the misses that would have been hits, had the application occupied one more cache way. On the contrary, Quershi *et al.* [49] developed a system that monitors each running process separately and records the distribution of each application's hits across the recency positions ranging from the MRU to LRU. Both schemes were described in detail in Sections 5.2.2 and 5.2.3 respectively.

The performed simulations showed that the second approach is superior, as the first scheme fails to identify the requirements of each process correctly. However, the cost of the scheme proposed by Qureshi *et al.* is significantly higher. For each monitored cache line, a private tag array that has the same associativity as the shared cache is added to every processor. On the contrary, Dybdahl *et al.* employ only a register for each processor.

Therefore another approach was sought that could provide the same accuracy as the scheme developed by Qureshi *et al.* but without having to track all the tags of the cache, thus, decreasing the hardware overhead. A solution was found using Bloom filters.

As it was described in Section 2.2 and illustrated in Figure 2.3, a *"Bloom Filter"* (BF) is a probabilistic algorithm employed to test membership in a large set using multiple hash functions into an array of bits [5]. Its main advantage is the ability to identify quickly non-members without having to query the entire set.

Peir et al. developed the *"Partial-Address Bloom Filter"* [46]. The system used a BF array of 2^k bits to predict whether a load would hit in the L1 cache. This prediction was then used by the processor's pipeline to schedule the dependent instructions accordingly. To index the array, the k least significant bits of the load address were used. A similar technique was employed in the proposed cache partitioning scheme.

6.1.2 Description

First, the partitioning module needs to track the actual cache occupancy of each application. Therefore, a *processor ID* field is added to each tag. Whenever a processor brings a new entry into the cache, its ID is stored in the appropriate field. The length of each field is $log_2 P$, where P is the number of processors.

Next, a record is needed of the misses by each application that could have been hits had the process been allowed to use more cache ways. The misses can be divided into two main categories :

- **Near-misses :** Misses that *would* have been hits, had the process been allocated *one* more cache way.
- **Far-misses :** Further misses that *may* have been hits, had the process been allocated *more than one* extra cache way.

To monitor the first kind of misses, an extra tag register, called '*near-miss register*', is added to each line for every processor. When an entry is rejected from the cache its tag is stored into the appropriate near-miss register, based on the processor ID that brought it into the cache in the first place. On a cache miss, the requested tag is compared to the tag stored into the near-miss register of the accessed line associated with the processor running the miss-causing application. A tag match signals a near-miss.

An example is illustrated in Figure 6.1. The application is only allowed to occupy 3 cache ways and has accessed the entries with tags a, b and c. When the entry with tag d is requested, a will be rejected from the cache and stored into the near-miss register. If the next miss occurs for a, then a near-miss will be identified.

To track far-misses, a BF array with 2^k bits is added to each line for each processor. When a tag is rejected from the near-miss register, its k least significant bits are used to index a bit of the BF array that is set to *true*. On a cache miss that is not identified as a near-miss, the appropriate entry of the BF array is looked up using the k least significant bits of the requested tag. If the array bit is *set*, then a far-miss has been detected.

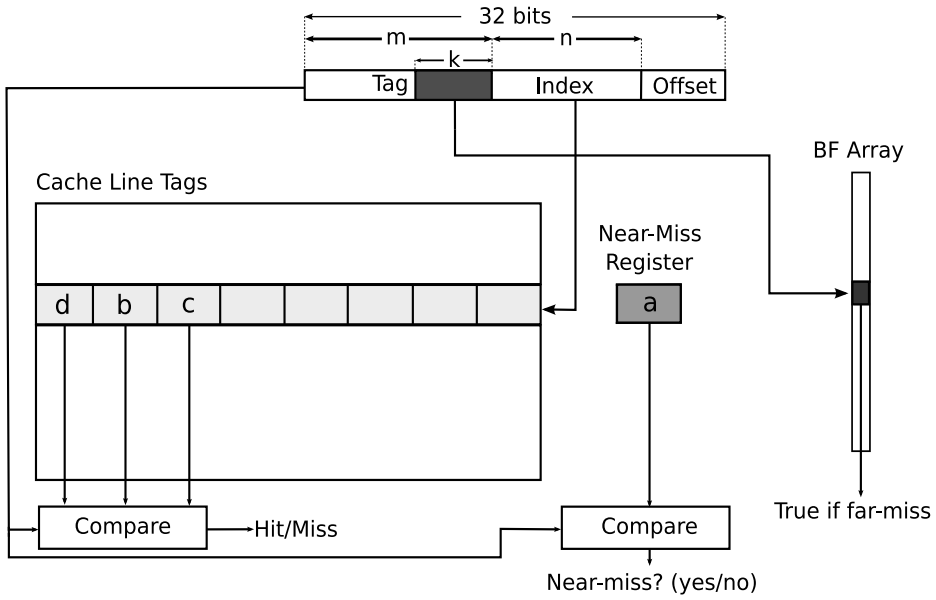


Figure 6.1: Monitoring components of the adaptive Bloom filter cache partitioning scheme

In the example shown in Figure 6.1, an access for tag e is resolved as a miss. Comparison with the near-miss register reveals that it is not a near-miss. Therefore, the BF array is accessed using the k least significant bits of e. If tag e was previously stored into the cache, then the array entry will be set and a far-miss will be identified.

When the BF array entry is found to be *false*, it is certain that the requested tag was not stored in the cache in the past. On the contrary, if the entry is found to be *true* there is a possibility of a *false positive*, due to the problem known as "*aliasing*". As only k bits of the tag are used to index the BF array, it is possible for more than one tag to map to the same array bit. This means, that the system may detect a far-miss for an address that was never brought into the cache before.

Moreover, in the system described in Figure 6.1, the order in which the BF array bits are set is not recorded. So, there is no information on the order in which lines were rejected from the cache. At the same time, the number

of true entries in the BF array could be higher than the associativity of the cache. Therefore, when a far-miss is detected, it is not possible to deduce how many more ways the applications should have been allocated, for that miss to become a hit.

That information could have been acquired, if the Bloom filter was extended with a set of counters and registers. However, as these extensions would increase the hardware overhead and the complexity of the system, they were rejected. Due to the possibility of filter errors though, far-misses are defined as misses that *may* have become hits, had the application been allocated more than one extra cache way in general.

In summary, by monitoring the far and near-misses of each application, the partitioning module acquires an estimate of each process' need for more cache space. However, as the cache is limited, this extra space has to be created by reducing the allocation of other processes. To identify these processes, an estimate of potential losses is required. According to the stack property of the LRU policy, the hits to the x last recency positions, i.e. the positions ranging from $LRU - x$ to LRU, will become misses if the process' allocation is decreased by x cache ways. Therefore, this number of hits can provide the required estimate of losses.

The cache partitioning mechanism tracks the hits to the LRU and $LRU - 1$ recency positions. In total, a set of four counters per processor is used for each cache line. The counters are the following :

1. $\mathbf{C_{NearMiss}}$: Incremented every time a near-miss is detected.
2. $\mathbf{C_{FarMiss}}$: Incremented when the BF array identifies a far-miss.
3. $\mathbf{C_{LRU}}$: Incremented when a cache access hits on the LRU entry owned by the application.
4. $\mathbf{C_{LRU-1}}$: Incremented when a cache access hits on the $LRU - 1$ entry owned by the application.

6.1.3 Partitioning the Cache

The cache partitioning mechanism can modify the space allocation of a process by *two* cache ways at most. As the total number of cache ways does not change, the possible partitions can be deduced from the following formula:

$$\sum_{i=1}^{P} \Delta x_i = 0, \qquad \Delta x_i = \{-2, -1, 0, 1, 2\} \tag{6.1}$$

where Δx_i is the change in the allocation of process i and P is the number of processors. Depending on the number of ways n given to or taken from an application, its performance gain or loss $g(n)$ is estimated using the hit and miss counters described in Section 6.1.2.

$$g(n) = \begin{cases} -(C_{LRU-1} + C_{LRU}) & , \quad n = -2 \\ -C_{LRU} & , \quad n = -1 \\ 0 & , \quad n = 0 \\ C_{NearMiss} & , \quad n = 1 \\ C_{NearMiss} + a \times C_{FarMiss} & , \quad n = 2 \end{cases} \tag{6.2}$$

As it was explained in Section 6.1.2, the counter $C_{FarMiss}$ records the misses that may have been hits, had the application been allowed to occupy more than one extra cache way. There is no information though on how many ways are actually required. In the partitioning algorithm, the value of this counter is used to estimate the gain when the application is allocated two more cache ways. Therefore, it needs to be scaled down and for that a is used :

$$a = 1 - \frac{ways_i}{ways_{cache}} \tag{6.3}$$

where $ways_i$ is the number of ways that application i was allowed to occupy before the repartitioning and $ways_{cache}$ is the associativity of the shared cache.

Using Equation 6.2, the total gain of each possible partition deduced from Equation 6.1 can be calculated :

$$Gain = \sum_{i=1}^{P} g(\Delta x_i) \tag{6.4}$$

where Δx_i is the change in the allocation of process i and P is the number of processors.

Initially the cache is divided equally amongst the running applications. Every T cycles the partitioning algorithm reevaluates the partition sizes per processor for each cache line, allocating at least one cache way to each process. The total gain of each possible partition is calculated using the hit and miss counters of each processor. The partition with the maximum gain is selected and all the counters and filters are reset.

To enforce the selected partition, the LRU replacement policy is modified to take into account the number of each application's entries in a cache line. On a miss, the LRU entry of the application is chosen for replacement if its actual allocation is larger or equal to the limit imposed by the partition. Otherwise, the LRU entry of an over-allocated process is rejected.

6.1.4 Simulation Results

First, the proposed scheme was evaluated for a dual core system sharing a 4MB, 32-way associative L2 cache. The monitoring period T, at the end of which the partitioning algorithm is executed, was selected to be 1 million cycles. Also, to eliminate the Bloom filter's aliasing, all the tag bits were used to index the BF array. Figure 6.2 presents the performance compared to a system using the traditional LRU policy.

On average the proposed scheme improved the performance by 3% over the LRU policy, increasing the geometric mean of the throughput from 1.58 to 1.61. More specifically, for 11 out of the 36 simulated combinations the overall performance was improved, where the maximum achieved improvement was 21%. In Section 5.3.2, where a failed cache partitioning scheme was presented, the *heap-sparse* and *series-sparse* combinations were examined closely to identify the drawbacks of the employed scheme. These drawbacks were used as a guide to design the system evaluated in this section. Therefore, these two combinations were examined in more detail to confirm that the partitioning module behaves as intented.

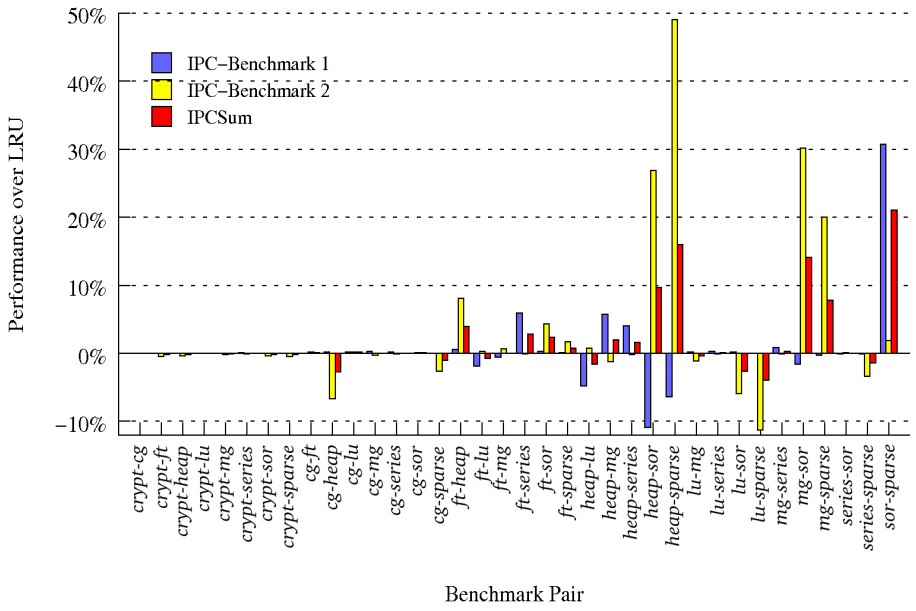


Figure 6.2: Performance over LRU for a dual core system

Figure 6.3 presents the average number of ways each application is allocated during the execution of the *heap-sparse* combination in two systems, one using the normal LRU policy and one using the proposed cache partitioning scheme. As it was shown in Figure 5.12, the rejected scheme was allocating more space

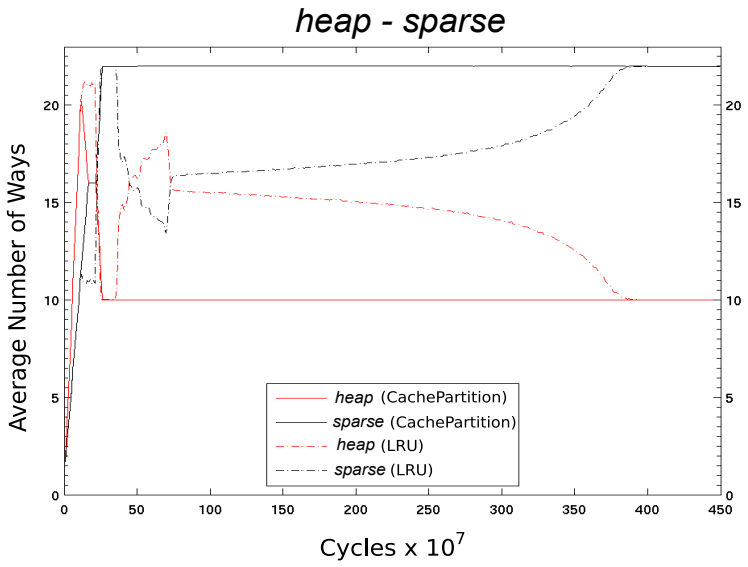


Figure 6.3: L2 cache ways occupancy for *heap* and *sparse*

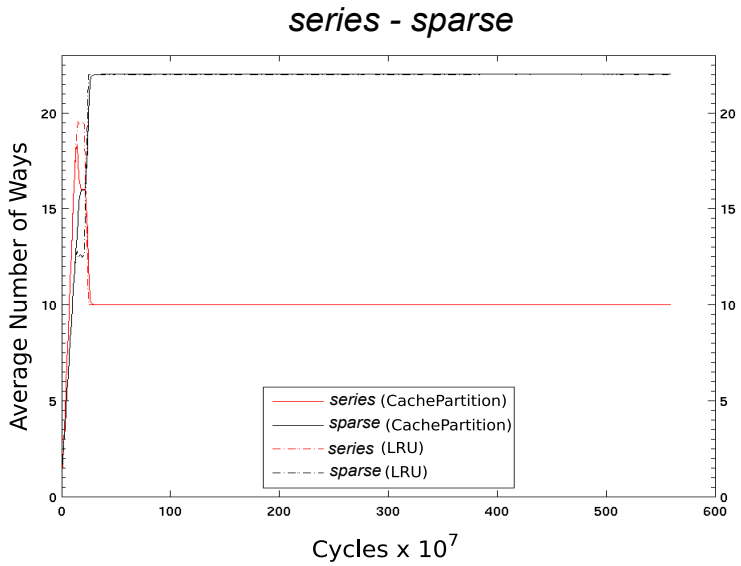


Figure 6.4: L2 cache ways occupancy for *series* and *sparse*

to *heap* than the LRU policy and was improving the overall system's performance by almost 5%. In this system however, *sparse* is the recipient of the majority of the cache space and the overall performance is improved by 16%.

The results for the *series*-*sparse* combination are presented in Figure 6.4. Figure 5.13 revealed that the rejected scheme was allocating the majority of cache space to *series*, which resulted in the overall performance degrading by 15%. On the contrary, the proposed system manages to allocate almost the same space to the running processes as when the normal LRU policy is used. Therefore, although it fails to improve compared to a system employing the LRU policy, the degradation of the performance is only 1%, as it is shown in Figure 6.2.

For almost half of the cases presented in Figure 6.2 the achieved performance was similar to that achieved when LRU is used. As it was explained in Section 5.2.3 this can be attributed to the cache being large enough to accommodate the working sets of the competing applications, thus allowing the LRU policy to achieve optimal sharing of the cache. Therefore, the scheme was reevaluated for a quad core system and the results are presented in Figure 6.5.

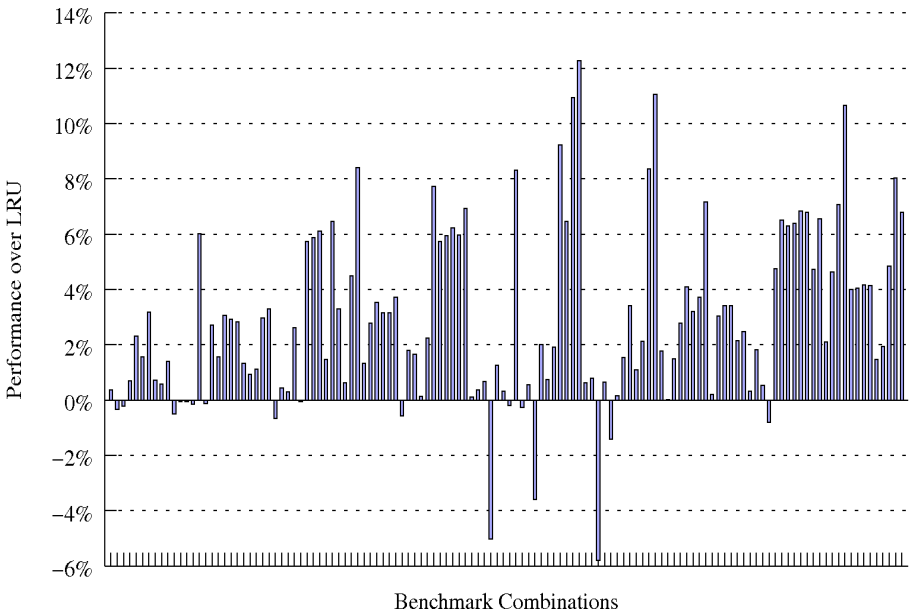


Figure 6.5: Performance over LRU for a quad core system

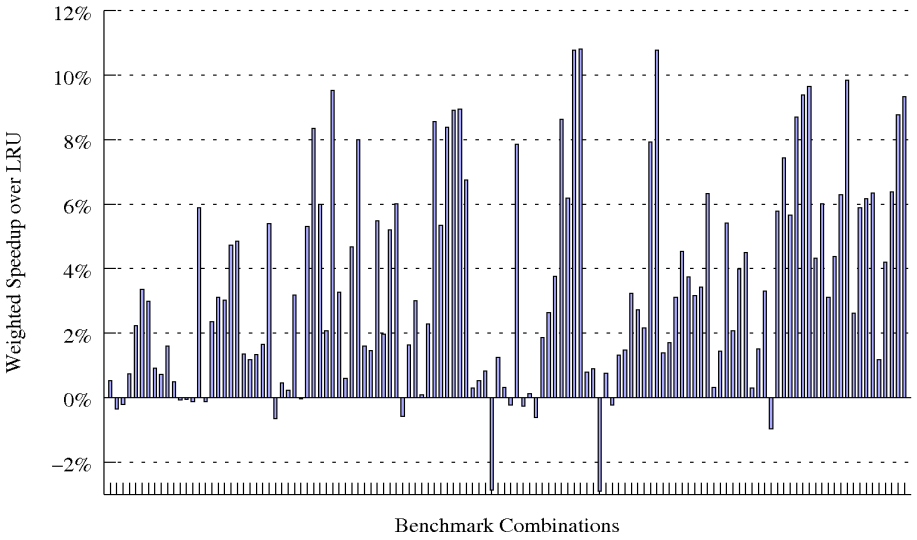
In this case, where more applications are competing for the cache space and its allocation has a more significant impact on the overall performance, the proposed cache partitioning scheme performs better than the LRU for the majority of the simulated combinations. There are a few cases where the achieved performance is worse than when LRU is used. While the maximum degradation is almost 6%, the performance degrades by more than 1% only for four cases.

Up to this point, the different schemes have been evaluated using the total throughput of the system. However *IPCSum* could be unfair to a low IPC application. Therefore, more metrics to quantify the performance of a system have been proposed in the literature [37]. Two of them are the *Weighted Speedup* and the *harmonic mean of normalised IPCs*. If IPC_i is the the IPC of the ith application and $SingleIPC_i$ is the IPC of the same application when it executes in isolation, then for a system where N processes execute concurrently, the two metrics are given by the following formulas :

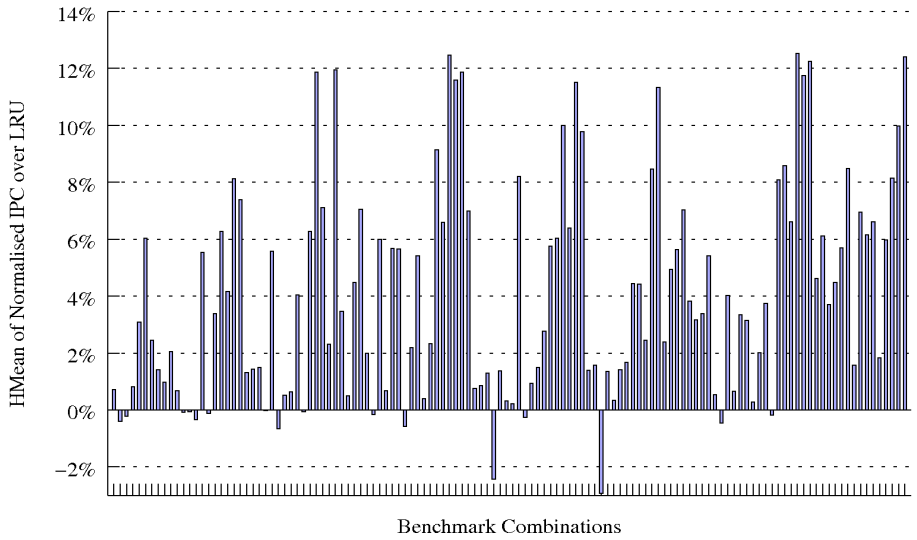
$$Weighted\ Speedup = \sum_{i=1}^{N} \frac{IPC_i}{SingleIPC_i} \tag{6.5}$$

$$IPC_{norm\_hmean} = \frac{N}{\sum_{i=1}^{N} \frac{SingleIPC_i}{IPC_i}} \tag{6.6}$$

The *Weighted Speedup* metric indicates reduction in execution time, while the IPC_{norm_hmean} metric balances both fairness and performance.



(a) Weighted Speedup over LRU for a quad core system



(b) Fairness over LRU for a quad core system

Figure 6.6: Evaluation on weighted speedup and fairness metrics

Figure 6.6 shows the results of the comparison between the developed cache partitioning scheme and a system using the LRU policy for the afore mentioned metrics. For the majority of the 126 simulated cases the weighted speedup was increased, which means that the execution time was reduced. On average the performance was improved by almost 4%, as the geometric mean weighted

speedup was increased from 3.48 to 3.61. Similarly, the fairness metric reveals that the proposed scheme performs better than the LRU for most of the simulated benchmark combinations. The average improvement was almost 5% as the geometric mean was increased from 0.85 to 0.89.

These two metrics reveal, similarly to IPCSum, that the developed cache partitioning scheme benefits the system. For the remainder of the thesis though, only the IPCSum metric will be used.

6.2 Implementation Considerations

The proposed scheme could probably be implemented as a small processor which monitors the running processes, keeps the appropriate information and then runs the partitioning algorithm. However there are several problems that need to be addressed. These are presented in this section.

6.2.1 Bloom Filter Arrays

The proposed scheme employs a Bloom filter array per processor for each cache line. In the evaluation presented in Section 6.1.4, 'ideal' filters were assumed in order to minimise the possibility of filter errors. For that reason, each BF array was as big as possible and all the bits of the tag were used to index it.

For a 4MB, 32-way associative cache that holds 8 words per entry, the length of the tag is 15 bits assuming a 32-bit physical address space. Therefore the size of each array was $2^{15} = 32Kb$. This means that for the whole cache, the hardware overhead for using the bloom filters was :

$$4096\ lines \times 32\ Kb/(lines \times processor) = 16MB/processor \tag{6.7}$$

Of course this overhead renders the implementation of the scheme impossible. By making the BF arrays smaller though, the possibility of filter errors is increased as more than one address maps to the same array bit. However, as it was explained in Section 6.1.2, due to their simplicity, the filters' information on far-misses is not expected to be strictly accurate and a factor a is used to scale down the value of the $C_{FarMiss}$ counters. Therefore, the system should be

able to tolerate a few extra filter errors.

The sensitivity of the scheme to the size of the BF arrays was evaluated for a dual core system. Figure 6.7 compares the performance when 3, 5, 7 and 10 bits are used to index the BF arrays to the case where 'ideal' filters are used.

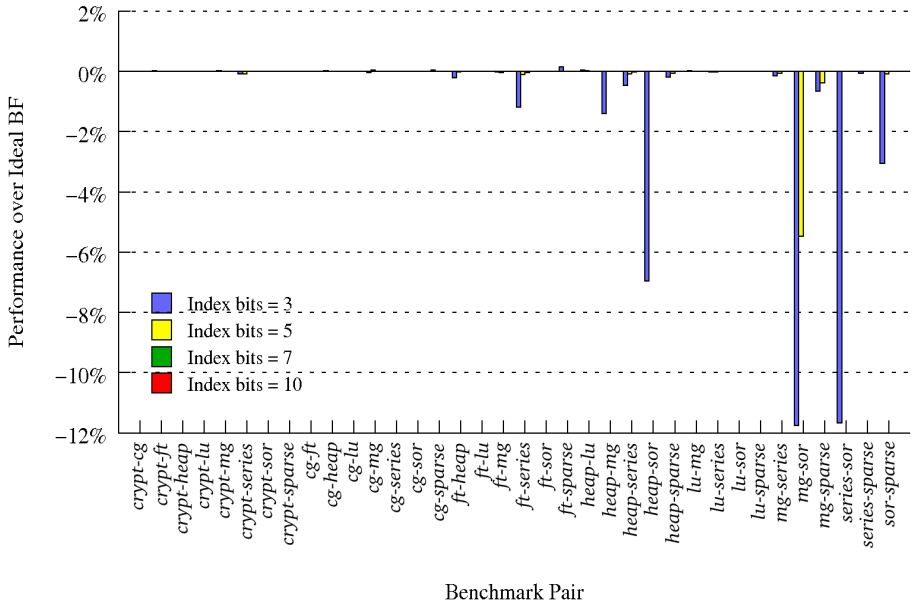


Figure 6.7: Effect of different Bloom filter arrays' sizes for a dual core system

Reducing the length of the index from 15 to 10 or 7 bits had no effect on the performance of the system, while a 3-bit index seems to be too small, for a few combinations at least. Based on the simulations' results the 5-bit index was selected. This choice only caused degradation in 5 out of the 36 benchmark combinations and only 'noticeable', i.e. greater than 1%, degradation in one case. The size of each BF array was therefore reduced to $2^5 = 32 bits$. Consequently, the hardware overhead for the whole cache can be reduced to :

$$4096\ lines \times 32\ b/(lines \times processor) = 16KB/processor \tag{6.8}$$

This decision was evaluated for a quad core system as well. The comparison between using 32-bit and 32Kb BF arrays is presented in Figure 6.8. The geometric mean is 0.995, which means that the performance was degraded on average by less than 1%. At the same time, according to formulas (6.7) and

(6.8) the hardware overhead was reduced from 64*MB* to 64*KB*.

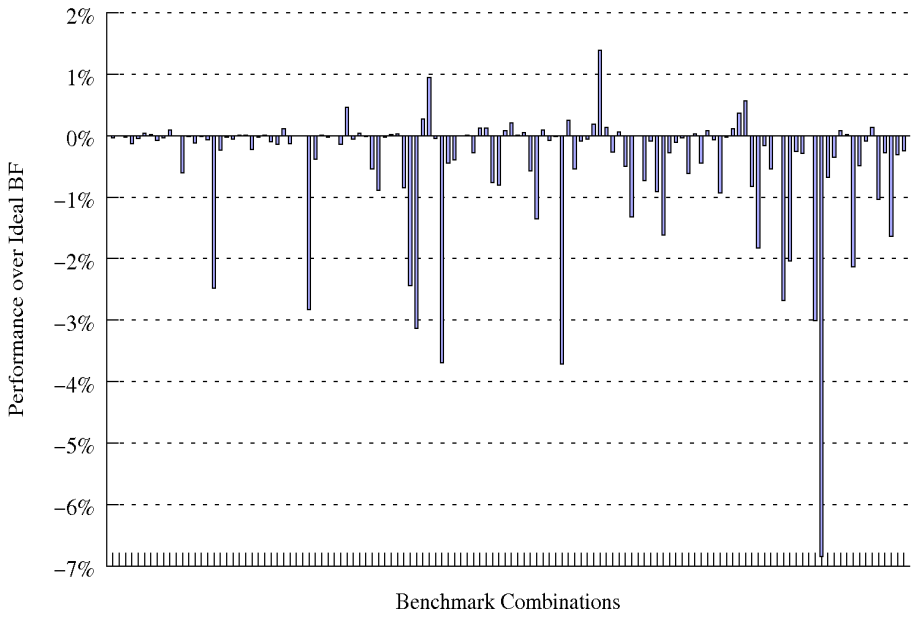


Figure 6.8: Effect of using 32-bit instead of 32-Kbit Bloom filter arrays for a quad core system

6.2.2 Monitoring Period

Until this point, all the simulations have been performed setting the monitoring period T to one million cycles. In this subsection, this decision is evaluated. Figure 6.9 compares the overall performance of a dual core system for four different monitoring periods to the case where $T = 1,000,000$ cycles. The partitioning modules use 5-bit indexes for the BF arrays and the two cores share a 4MB, 32-way associative L2 cache.

For almost half of the simulated cases, the fifty million cycles period is slightly worse. This can be attributed to the monitoring period being too long, not allowing the cache partition to adapt to the dynamic changes in the applications' behaviour quickly enough.

On the other hand, the short monitoring periods were expected to hurt the performance as well, as they were thought to be too short for the sufficient

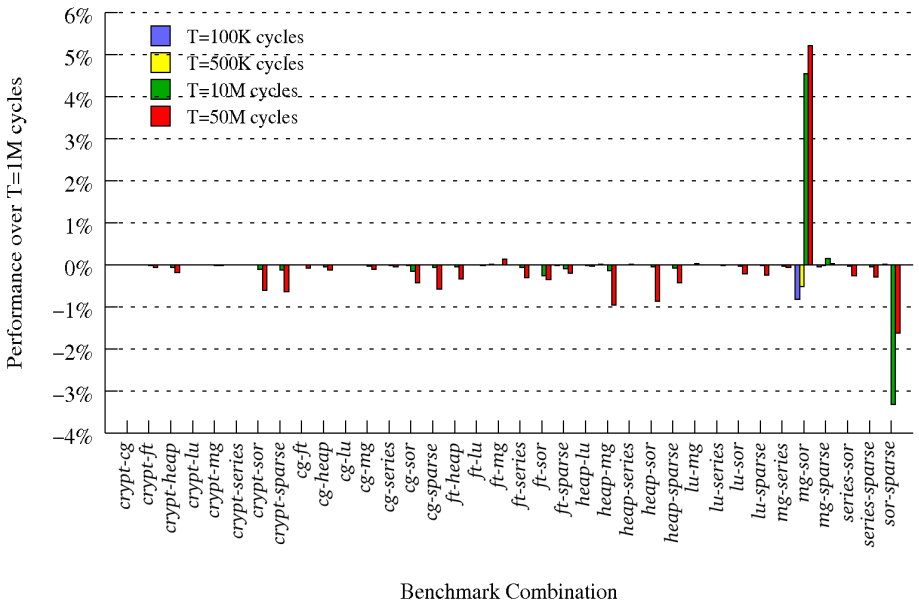


Figure 6.9: Effects of different monitoring periods for a dual core system

amount of history to be gathered. However, the results indicate that for these benchmark combinations, a period between 100,000 and 10,000,000 cycles achieves the same performance. At the same time though, in a real system there will be an overhead for computing the new partitions at the end of each monitoring period. Therefore, short periods should be avoided and setting T to one million cycles appears to be a reasonable decision.

6.3 Cost Analysis

6.3.1 Partitioning Algorithm

As was described in Section 6.1.3, the cache partitioning algorithm evaluates all the possible partitions and selects the one that is estimated to maximise the total number of hits. These partitions are deduced from formula (6.1), which is equivalent to :

$$\sum_{i=1}^{N} \Delta x_i = 2 \times N, \qquad 0 \leqq \Delta x_i \leqq 4 \tag{6.9}$$

where Δx_i is the change in the allocation of process i and N is the number of processors. The number of integer solutions of equations is a known problem

[52]. The number of solutions for Equation 6.9 is given by :

$$\binom{3 \times N - 1}{N - 1} + \sum_{r=1}^{N} (-1)^r \times \binom{N}{r} \times \binom{3 \times N - 5 \times r - 1}{N - 1} \tag{6.10}$$

Table 6.1 shows the number of possible partitions for different number of cores. It is obvious that the algorithm does not scale efficiently with the number of cores, as for 16 processors the partitioning module needs to evaluate over 10^{10} different partitions. Even for 8 cores, the comparison of 38165 partitions is going to be too expensive, as it needs to be done for every cache line. Therefore, a different way of evaluating the possible partitions and selecting the best one or a good approximation thereof needs to be developed.

Processor Cores	**Possible Partitions**
2	5
4	85
8	38165
16	1.065×10^{10}

Table 6.1: Number of possible partitions

The pseudo code for the new algorithm is shown in Algorithm 1 in page 97, where min2 is the value next to min and max2 is the value next to max. The algorithm begins by reading the four counters of each processor and inserting them into four lists that are ordered from minimum to maximum. The first and second lists show how strongly the processes want another 2 and 1 ways, respectively. The third and fourth lists show how much harm is suffered by losing 2 and 1 ways, respectively. There are four possibilities that need to be examined.

1. process i wins 2 ways and process j loses 2 ways.
2. process i wins 2 ways and processes j and k lose 1 way.
3. processes i and j win 1 way and process k loses 2 ways.
4. process i wins 1 way and process j loses 1 way.

First, the algorithm determines if a processor should be assigned 2 extra ways (case 1) by comparing the maximum number of the first list against the minimum number of the third list. If it is greater, the allocation of the two processors

Algorithm 1 Partitioning Algorithm

```
processes = N
for application i = 0 to N do
  C_2[i] = C_NearMisses + a × C_FarMisses, C_1[i] = C_NearMisses
  C_-2[i] = C_LRU + C_LRU-1, C_-1[i] = C_LRU
end for
order C_2, C_1, C_-1, C_-2 from min to max
while max C_2 > min C_-2 do
  increase allocation[i] by 2
  decrease allocation[j] by 2
  remove i and j from C_2, C_1, C_-1, C_-2
  processes -= 2
  if processes ≤ 1 then
    return allocation;
  end if
end while
while max C_2 > (min C_-1 + min2 C_-1) do
  increase allocation[i] by 2
  decrease allocation[j] and allocation[k] by 1
  remove i, j and k from C_2, C_1, C_-1, C_-2
  processes -= 3
  if processes ≤ 1 then
    return allocation;
  end if
end while
while (max C_1 + max2 C_1) > min C_-2 do
  increase allocation[i] and allocation[j] by 1
  decrease allocation[k] by 2
  remove i, j and k from C_2, C_1, C_-1, C_-2
  processes -= 3
  if processes ≤ 1 then
    return allocation;
  end if
end while
while max C_1 > min C_-1 do
  increase allocation[i] by 1
  decrease allocation[j] by 1
  remove i and j from C_2, C_1, C_-1, C_-2
  processes -= 2
  if processes ≤ 1 then
    return allocation;
  end if
end while
return allocation;
```

is modified, the processors are deleted from the lists as they need no longer to be considered and the process is repeated. When the comparison reveals that the gain is less than the loss, the algorithm moves to the next case. The algorithm finishes either when there are no processors left to be considered or all the four possibilities have been examined.

An example of how the algorithm works is presented in Figure 6.10. Each table shows the gain and loss estimates for four processors. More specifically, $C_2 = C_{NearMisses} + a \times C_{FarMisses}$ estimates the gain for obtaining 2 more ways, $C_1 = C_{NearMisses}$ the gain for obtaining 1 more way, $C_{-2} = C_{LRU} + C_{LRU-1}$ the loss for losing 2 ways and $C_{-1} = C_{LRU}$ the loss for losing 1 way. The algorithm first looks for processes that strongly desire two more cache ways. Therefore, the maximum C_2 is compared to the minimum C_{-2} and, if it is found to be greater, the process is allocated 2 more ways. This is shown in case A, where applications 1 and 3 are given 2 more ways, which are taken from processes 2 and 0 respectively. As each process' allocation can be altered only once, there is no need for more comparisons and modifications. According to Equation 6.4, the total gain of the deduced partition is $-10 + 60 - 6 + 50 = 94$. The algorithm that evaluates all the possible partitions would have increased the allocations of applications 1 and 3 by 2 and 1 ways respectively, as the estimated gain for this selection is $-6 + 60 - 6 + 50 = 98$.

A.

App	C_{-2}	C_{-1}	C_1	C_2
0	10	6	1	1
1	8	6	30	60
2	6	6	0	0
3	12	9	50	50

B.

App	C_{-2}	C_{-1}	C_1	C_2
0	20	12	5	5
1	30	2	0	0
2	10	10	7	7
3	8	4	1	2

C.

App	C_{-2}	C_{-1}	C_1	C_2
0	10	6	1	1
1	8	6	4	5
2	6	7	2	2
3	12	9	5	5

D.

App	C_{-2}	C_{-1}	C_1	C_2
0	20	12	5	5
1	30	4	0	0
2	10	10	7	7
3	15	4	1	2

Figure 6.10: Four examples of the partitioning algorithm for a quad core system

If $max\ C_2 \leq min\ C_{-2}$, the algorithm checks if it is possible to allocate a process 2 more ways that will be taken from 2 different processes. This is illustrated in case B of Figure 6.10, where the allocation of applications 1 and 3 are reduced by 1 way and application 2 receives 2 extra ways. Again, the algorithm that evaluates all the possible partitions would have made a different decision. More specifically, it would have selected to increase the allocation of applications 0 and 2 by 1 way.

The next step is to try and increase the allocation of 2 processes by 1 way each at the expense of a single victim. This is shown in case C. The algorithm compares the sum of the 2 greatest C_1 values to the minimum value of C_{-2}. In that example the sum is greater, so 2 ways are taken from process 2 and are allocated to processes 1 and 3. Finally, the algorithm checks the possibility of modifying the partition sizes by only one way. This is done by comparing C_1 to C_{-1}. As shown in case D, applications 0 and 2 receive one more way each, which are taken from applications 3 and 1 respectively.

To evaluate the complexity of the new algorithm, assume that N is the number of running processes and x, y, z and w the number of comparisons executed in each of the four steps, i.e. the four while loops of Algorithm 1. Then the following is true:

$$
\begin{aligned}
1 &\leq x \leq \frac{N}{2} \\
0 &\leq y \leq \frac{N - 2 \times (x-1)}{3} \\
0 &\leq z \leq \frac{N - 2 \times (x-1) - 3 \times (y-1)}{3} \quad , y \geq 1 \\
0 &\leq w \leq \frac{N - 2 \times (x-1) - 3 \times (y-1) - 3 \times (z-1)}{2} \quad y, z \geq 1
\end{aligned}
$$

Therefore, for the total number of possible comparisons the following is true:

$$Comparisons_{Total} < \frac{5 \times N}{3} < 2 \times N \tag{6.11}$$

which means that the new algorithm's complexity is O(N) and is linear in regard to the number of running process.

In the previously presented example, the old algorithm that evaluates all the possible partitions would have made the same decisions for the last two cases. However, there are cases where the decisions would be different. The chance of the two algorithms reaching different decisions becomes higher as the number of processes sharing the cache is increased. Therefore, the system's performance is expected to be worse compared to when using the 'ideal', theoretical algorithm. It is, though, a justified sacrifice that renders the practical implementation of the scheme possible.

The effects of the new algorithm on the performance of the cache partitioning mechanism was initially evaluated for a dual core system sharing a 4MB, 32-way L2 cache. The size of each BF array was 32 bits and the monitoring period was set to 1M cycles. The results are presented in Figure 6.11. Only for one case the new algorithm degrades the performance of the partitioning scheme. For the other 35 simulated cases the two algorithms achieve similar results.

Next the proposed scheme was evaluated for a quad core system. Figure 6.12(a) reveals that for the majority of the simulated cases the new algorithm improves the performance of the system. Figure 6.12(b) presents the performance over

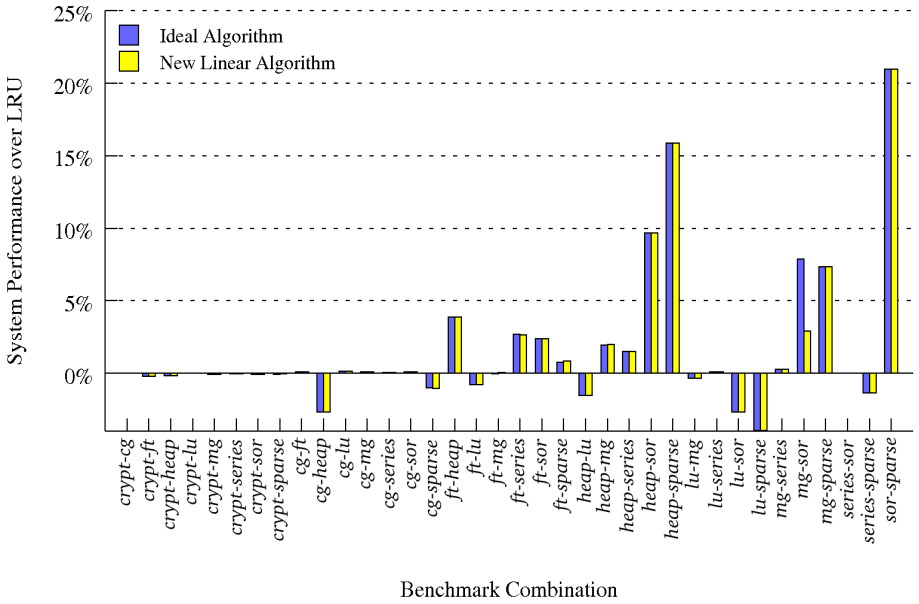
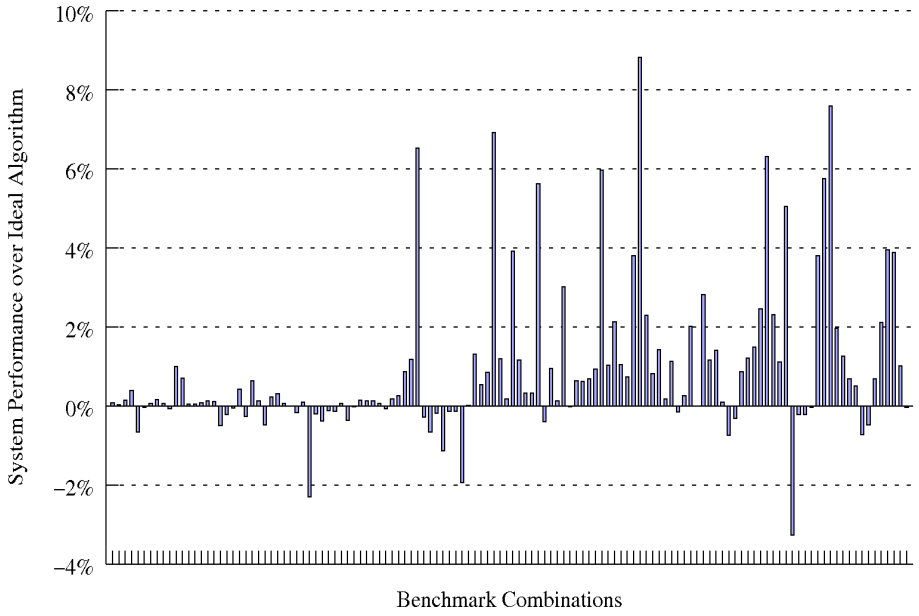
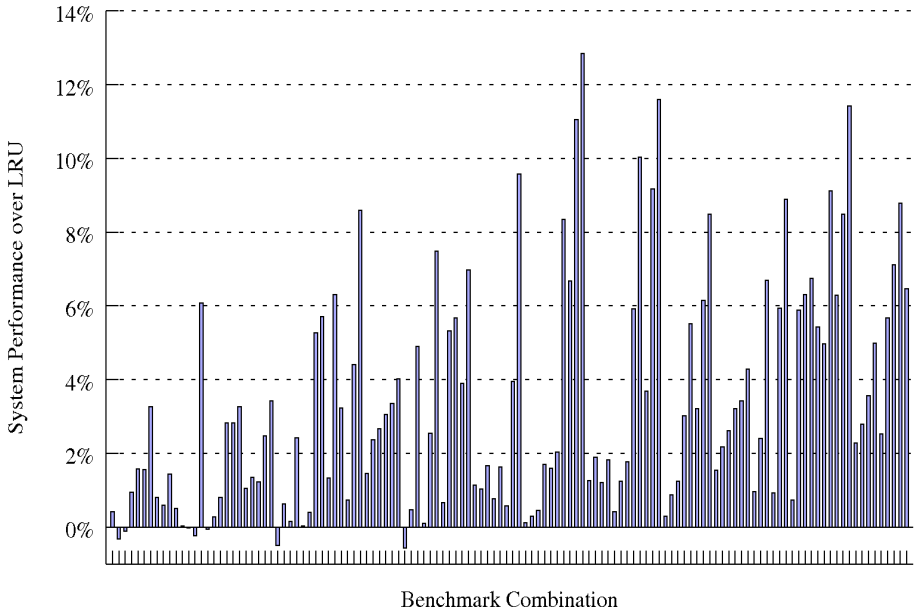


Figure 6.11: Comparison of new and old partitioning algorithm for a dual core system

the normal LRU policy. A comparison with Figure 6.5, where the partitioning module was using the 'ideal' algorithm and the 'ideal', large BF arrays, reveals that this scheme provides gains for cases where the overall system's performance was previously degraded.



(a) Comparison of new and old partitioning algorithm for a quad core system



(b) Performance over LRU for a quad core system using the new partitioning algorithm

Figure 6.12: Effects of the new partitioning algorithm for a quad core system

As it was previously noted, the new algorithm was expected to perform slightly worse than the old 'ideal' algorithm, as it is not guaranteed to select the partition that will maximise the gain of the system. However, the simulations' results for the quad core system appear to refute that. To investigate this further, the proposed scheme was also evaluated for a system where 8 processors share a 4MB, 32-way L2 cache. The performance over a system where the normal LRU policy is used are shown in Figure 6.13.

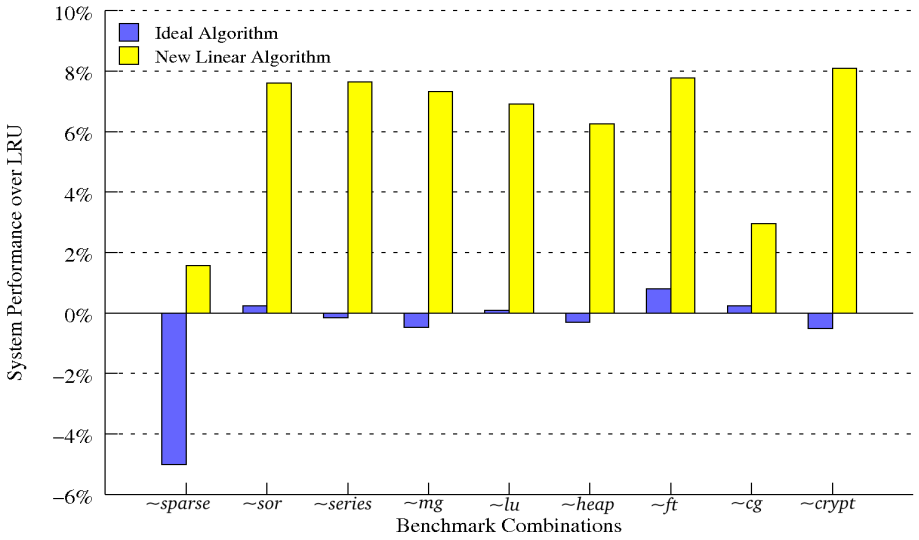


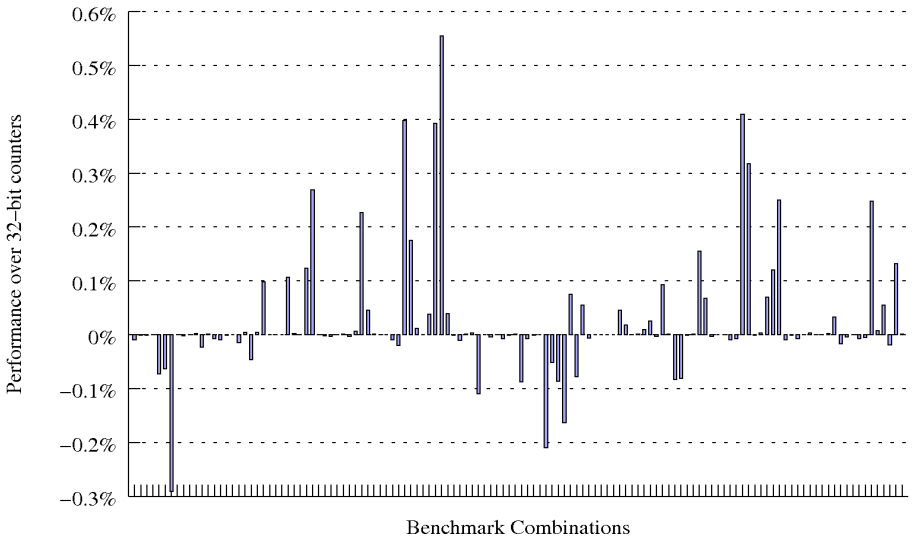
Figure 6.13: Comparison of new and old partitioning algorithm for a system with 8 cores

Again the results indicate that the proposed algorithm performs significantly better than the old 'ideal' one. In fact, while the 'ideal' algorithm achieves on average the same performance as the normal LRU policy, the proposed scheme improves the performance on average by 6% over LRU.

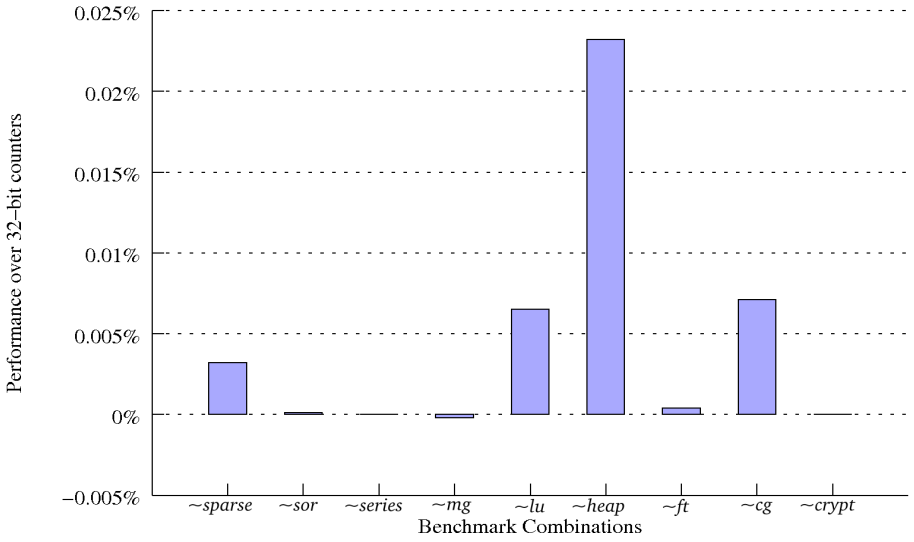
The difference between the two partitioning algorithms is that the 'ideal' one evaluates all the possible partitions and selects the one that will maximise the total gain of the system, while treating all the processes equally. On the other hand, the new linear algorithm, in its attempt to limit its complexity, is not fair and prioritises the applications that demand two extra cache ways. The results indicate that this approach provides, at least for the employed benchmarks, a cache space allocation closer to the actual system requirements improving the overall performance of the system.

6.3.2 Hardware Overhead

Each processor uses one BF array, one NearMiss register and four counters for each cache line. Until this point the counters were assumed to be 32 bits long. However, as they are reset every one million cycles, their length can be reduced to 20 bits. In practice, it can be reduced further to 8 bits. Figure 6.14(a) shows the performance of the system using 8-bit counters compared to using 32-bit counters.



(a) Comparison of 8 and 32-bit counters in a quad-core system



(b) Comparison of 8 and 32-bit counters in an 8-core system

Figure 6.14: Performance using 8-bit counters over a system that uses 32-bit counters

The geometric mean is equal to 1.0002, which means that the reduction of the counters' length has no significant effect on the performance of the cache partitioning module. Similar conclusions can be drawn for an 8-core system, as shown in Figure 6.14(b).

In Section 6.2.1 the hardware overhead of each Bloom filter array was analysed and reduced from $32KB$ to $32bits$. The storage overhead per processor for a 4MB, 32-way associative cache, assuming a 32-bit physical address space, is presented in Table 6.2. The storage overhead per processor could be further reduced by using partial tags for the NearMiss registers.

BF arrays (4096 lines * 32bits)	16KB
NearMiss registers (4096 lines * 15 bits)	7.5KB
Counters (4096 lines * 4 counters * 8 bits)	16KB
Total overhead	39.5 KB
Area of L2 cache (240KB tags + 4MB data)	4336KB
% increase in area	0.9%

Table 6.2: Storage overhead per processor

At the same time, the proposed scheme requires a processor ID for each cache entry. The length of each ID is $log_2 N$ bits, where N is the number of processors. For an eight-core system sharing the 4MB, 32-way associative cache, the overhead of the IDs will be $4096 \times 32 \times 3b = 48KB$, an increase of 1.1%. Therefore, the total storage overhead for an eight-core system is $8 \times 39.5 + 48 = 364KB$, an increase of 8.3% over the L2 cache area.

In addition to the storage bits, adders are needed to increase the counters of each processor and the partitioning algorithm requires a comparator circuit. So the true overhead will be slightly greater than indicated here. However this is still proportionately small.

To evaluate the effectiveness of cache partitioning as a solution to increasing the overall performance of the system, it needs to be compared against increasing the cache size, which is the usual approach taken by system designers. Therefore, an eight-core system sharing a 4MB, 32-way associative L2

cache and using the proposed cache partitioning module was compared against other 8-core systems sharing bigger L2 caches employing the LRU replacement policy. The results are presented in Figure 6.15.

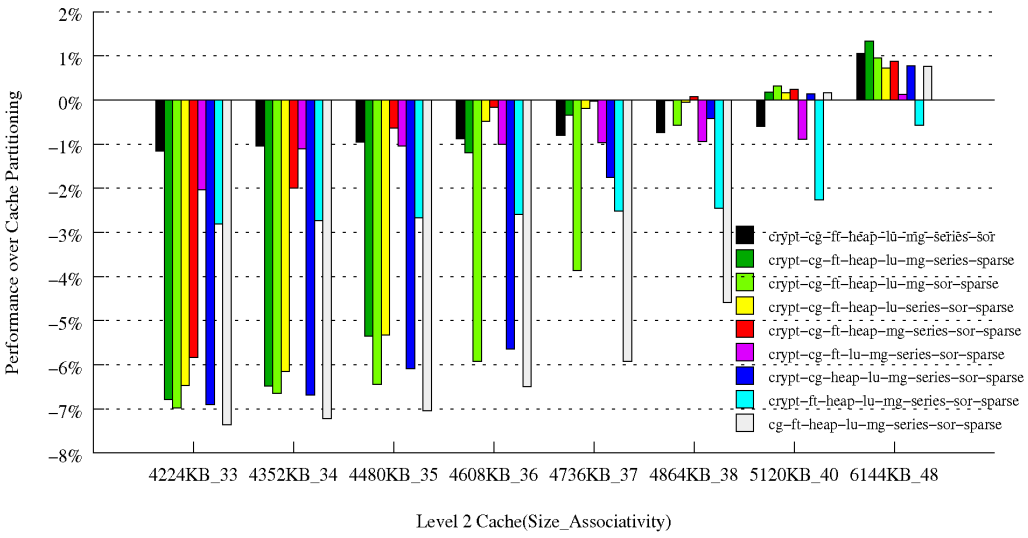


Figure 6.15: Performance of a 4MB, 32-way associative L2 cache using cache partitioning against bigger caches employing the LRU policy

The L2 cache was increased by adding extra ways. Each way adds 128*KB* of data and 7.5*KB* of tags, an overhead of 135.5*KB*. The storage overhead of the cache partitioning module was previously calculated for an eight-core system and found to be 364*KB*, slightly less than adding three ways to the baseline cache. Figure 6.15 shows that a 4480KB, 35-way associative cache using LRU performs worse than the baseline 4MB, 32-way associative cache combined with cache partitioning. The geometric mean for that case is 0.96, which means that the smaller cache with cache partitioning performs better on average by 4%.

Moreover, Figure 6.15 reveals that for LRU to perform better than cache partitioning for the majority of the simulated benchmark combinations, the baseline cache's size needs to be increased to 6MB, an increase of 50%. And even in that case, the average improvement of LRU over cache partitioning is only 0.7%. Therefore, it appears that the proposed cache partitioning module offers a cost effective solution to improving the overall performance of a CMP system.

6.4 Comparison with Other Schemes

Several cache partitioning schemes [15, 35, 49, 54, 60] have been proposed for CMP systems. The most effective of these, the *utility-based cache partitioning scheme* proposed by Qureshi *et al.* [49], was presented and evaluated in Section 5.2.3. In this subsection it is compared against the *adaptive Bloom filter cache partitioning* scheme presented in this chapter.

Figures 6.16, 6.17 and 6.18 compare the two cache partitioning schemes for a dual, a quad and an eight-core system respectively. For the dual-core system the utility-based scheme performs better than the adaptive Bloom filter scheme for 2 out of the 36 simulated benchmark combinations. For all the other cases the two schemes achieve almost the same performance. Comparison with Figure 6.11 reveals also a discrepancy, as for the *lu-sparse* combination ABCP degrades the performance by around 6%, while in Figure 6.11 the degradation was around 4%. This difference is due to ABCP employing here 8-bit adders, while before it was using 32-bit adders.

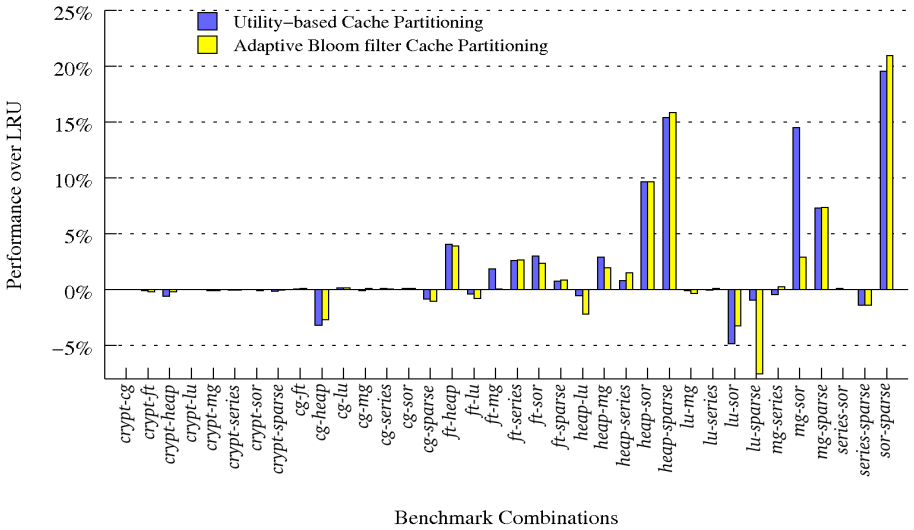


Figure 6.16: Comparison of the utility-based and adaptive Bloom filter cache partitioning schemes for a dual-core system

However, as shown in Figure 6.17, the adaptive Bloom filter partitioning scheme is more effective for a quad-core system, as it outperforms the utility-based partitioning scheme for 76 out of the 126 simulated cases. Additionally, its

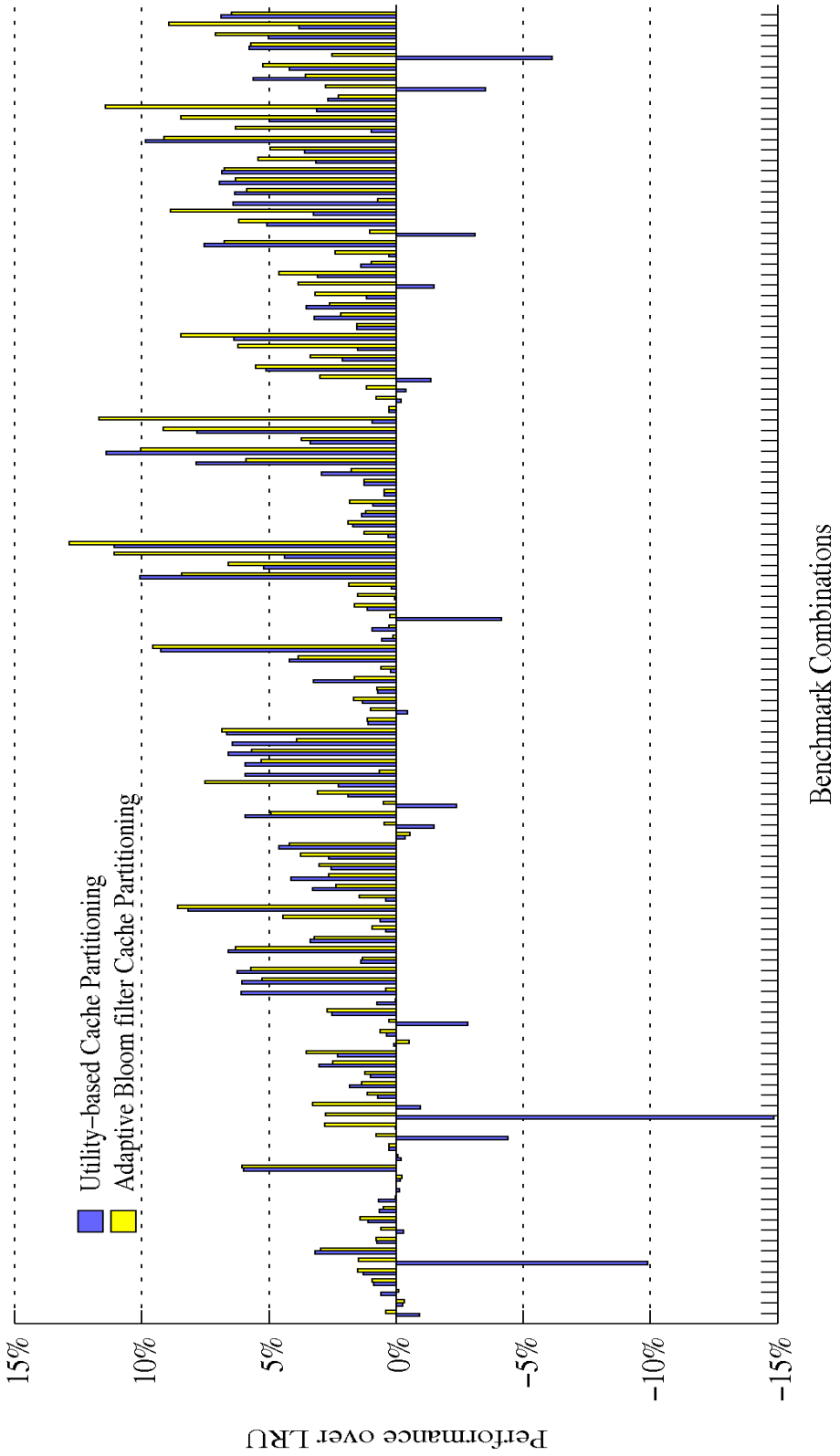


Figure 6.17: Comparison of the utility-based and adaptive Bloom filter cache partitioning schemes for a quad-core system

performance is always greater or at least equal to a system employing the LRU replacement policy. On the other hand, there are a few cases where the utility-based scheme performs worse than the LRU, so it appears that the proposed scheme may be more robust.

Similar conclusions can be drawn for an eight-core system. As shown in Figure 6.18, the adaptive Bloom filter scheme outperforms the utility-based scheme for every simulated case, while always achieving higher overall system performance compared to LRU.

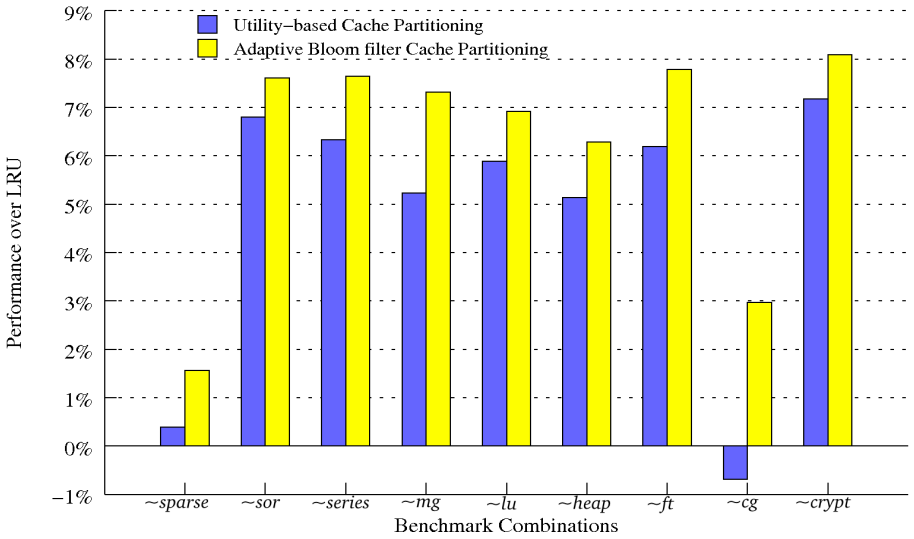


Figure 6.18: Comparison of the utility-based and adaptive Bloom filter cache partitioning schemes for an eight-core system

The scheme developed by Qureshi *et al.* has a small hardware overhead. As described in Section 5.2.3, each processor uses a monitoring circuit to acquire the information needed by the partitioning algorithm. For an eight-core system sharing a 4MB, 32-way associative L2 cache each monitoring circuit requires only 2.75*KB*. At the same time, 3 bits are required for each cache entry to identify which processor brought it into the cache. Therefore the total overhead is 70*KB*, an increase of 1.6% over the L2 cache area. For a similar system, the adaptive Bloom filter cache partitioning scheme requires 364*KB*, an increase of 8.3% over the L2, as it was calculated in Section 6.3.2.

To achieve such a low hardware overhead, the utility-based cache partitioning scheme monitors only 32 out of the 4096 lines of the cache. However, as only a subset of the lines is monitored, the partitioning module has to enforce the *same* partition for all the cache lines. To be able to partition on a line-basis, each monitoring circuit would have required $11MB$, which is practically impossible. On the other hand, the adaptive Bloom filter cache partitioning scheme is able to deduce a different partition for each line at a low cost. As the previously presented results indicate, this finer-grained partitioning allows the adaptive Bloom filter scheme to be more robust and outperform Qureshi's scheme. Since its overhead is still small compared to the area of the L2 cache, the decision of partitioning on a cache line basis is justifiable.

6.5 Summary

This chapter presented a new cache partitioning scheme. This mechanism uses a register to track the near-misses of each application and Bloom filters to record misses that could have been hits if the application had been allocated more than one extra cache way. At the same time, the system counts the hits to the last two recency positions, i.e. $LRU - 1$ and LRU.

All these are monitored over a period of T cycles and then they are used to predict the future behaviour of the system. Based on this prediction, the algorithm reevaluates the partition sizes per line for each process. The scheme was evaluated for a dual, a quad and an eight-core system and was shown to achieve better performance than a system using the familiar LRU policy.

Several design issues were addressed. More specifically, the size of the Bloom filters, the length of the monitoring period and the complexity of the partitioning algorithm were studied. Finally, the hardware overhead was analysed and an eight-core system was used to demonstrate that the proposed scheme constitutes a cost effective solution to the challenge of improving the overall performance of CMP systems.

Chapter 7

Conclusions

The continuously increasing number of transistors integrated into a chip has given rise to chip multiprocessor (CMP) architectures. These are shared memory systems, where the available cores typically share some levels of the on-chip memory hierarchy. In the majority of the currently developed systems, each processor has private L1 instruction and data caches while they all share the L2 cache.

Cache design has been extensively studied in the context of traditional single processor systems and many design choices have migrated to the CMP design domain. The parameters may be different for these systems though, raising doubts over the effectiveness of this migration. A typical example of such a design choice is the employment of LRU, which is widely accepted as the best available replacement policy for uniprocessor systems.

In CMPs the processing cores are used to execute different applications or several threads of the same application in parallel. This introduces a new variable in the cache design process. However, 'pure' LRU is a "thread-blind" policy as it selects the cache entry to be rejected regardless of which thread brought it into the cache or which thread suffered the cache miss. This policy, combined with the sharing of the cache, allows interference, as multiple working sets compete for the cache space and data belonging to one thread may be evicted by data blocks belonging to other threads. Therefore, new alternatives are needed that will tackle the challenges that have arisen and improve the overall performance of the system.

7.1 Cache Partitioning

The work presented in this thesis evaluated cache partitioning as a solution to achieving optimal sharing of the cache amongst the concurrently executing applications and improving the overall system's performance. Initially the effects of LRU on the performance of multiprogrammed workloads were studied; in Chapter 4 the results indicated that this replacement policy can result in suboptimal sharing of the L2 cache. In contrast to LRU, statically partitioned caches were found to improve the overall performance of the system. However, these are not practical as they rely on prior knowledge of the characteristics of the applications that are executed.

Therefore several dynamic cache partitioning schemes which attempt to adapt the cache space allocation to the dynamically changing characteristics of the applications were analysed in Chapter 5. This analysis was necessary as the identified advantages and drawbacks were then used to guide the development of an improved cache partitioning scheme.

7.1.1 Adaptive Bloom Filter Cache Partitioning

Adaptive Bloom filter Cache Partitioning (ABCP) is a new scheme that was described and evaluated in Chapter 6. Its distinguishing feature is the employment of Bloom filters to create a low-cost mechanism which tracks misses that may have been hits had each process been allocated more cache ways. A linear algorithm is executed every one million cycles which deduces the partition sizes for each processor in each cache line.

The scheme was evaluated for a dual, a quad and an eight-core system and was shown to achieve better overall performance than the LRU policy. For an eight-core system sharing the L2 cache it was found that, for LRU to achieve similar results to cache partitioning, the cache size had to be increased from $4MB$ to $6MB$, an area increase of 50%. At the same time, the storage overhead of the partitioning mechanism was $364KB$, an increase of only 8.3% over the L2 cache area.

The scheme was also compared with *Utility-based Cache Partitioning* (UCP) - probably the best pre-existing scheme - and was shown to produce similar, but slightly better, overall performance at a somewhat increased hardware cost. Significantly, ABCP appears more effective than UCP as the number of cores increases. More specifically, whilst it is comparable for 2 cores, it is better for 4 and the trend increases at 8 cores. Thus the relatively small extra hardware overhead can be justified as the number of cores increases.

7.1.2 Cache Partitioning Evaluation

LRU and ABCP were evaluated for a different number of processors sharing a 4MB, 32-way associative L2 cache. Table 7.1 presents the geometric mean of the overall performance for each case and shows that the gains of cache partitioning over LRU increase with the number of processors. This is also evident in Figure 7.1, where the overall performance normalised with the number of processing cores is shown for LRU and ABCP.

Processor Cores	ABCP IPC	LRU IPC	% Performance Improvement
1	0.8276	0.8276	0%
2	1.6044	1.5815	1.45%
3	2.3280	2.2737	2.39%
4	3.0449	2.9467	3.33%
5	3.7365	3.6081	3.56%
6	4.4014	4.2409	3.79%
7	5.0203	4.8166	4.23%
8	5.6756	5.3436	6.21%
9	6.3061	5.9310	6.32%

Table 7.1: Comparison of LRU and cache partitioning

The effectiveness of cache utilisation degrades as the number of cores, and therefore the demands on the cache, increases. This is true of all the schemes that were examined in this work. However, the cache partition scheme is able to allocate cache space to competing processes efficiently by monitoring each running process and adapting dynamically to the characteristics of the muiltiprogrammed workload. Therefore, the utilisation of an appropriately partitioned cache can be seen to degrade more slowly than one using the pure,

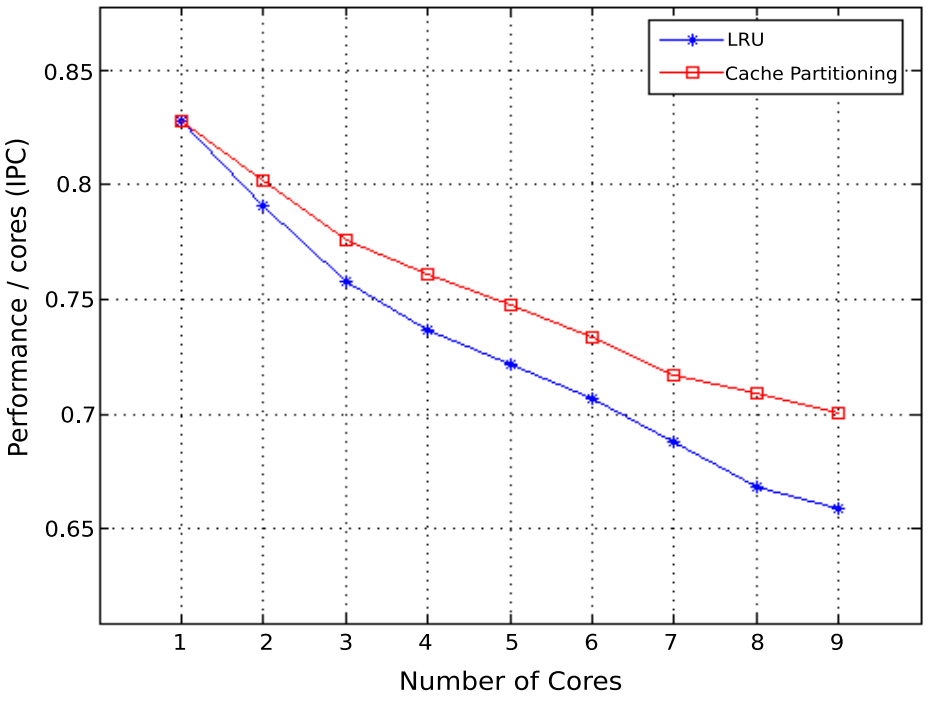


Figure 7.1: Performance normalised with number of cores for LRU and cache partitioning

thread-blind LRU replacement policy. This is important as the current trend is to increase the number of cores integrated on chip, so improving the cache utilisation is essential.

7.2 Future Work

The research presented in this thesis evaluated cache partitioning as a method for improving the overall performance of CMP architectures. This work can be expanded further. This section outlines possible directions for future research.

7.2.1 Many-core Architectures

As the feature size of fabrication technology continues to increase, computer systems are moving towards many-core architectures, where tens of processors will be integrated on the same silicon chip. The benefits of the proposed cache partitioning scheme over LRU were found to increase with the number of

processors. However, the evaluation was performed for a maximum of 9 cores sharing the L2 cache. It is hoped that this trend continues as the number of cores rises higher, however this needs to be confirmed by future study. More importantly, as the hardware overhead of the partitioning module scales with the number of cores, it needs to be evaluated whether cache partitioning can improve the utilisation of the cache on economic cost.

7.2.2 Multithreaded Workloads

In this work, cache partitioning was evaluated for multiprogrammed workloads. Each core executed a different application that had its own private working set. However, CMPs can be used to execute parallel applications, where several threads could share either instructions or portions of the data set. Therefore, it is important to evaluate the proposed cache partitioning scheme for multithreaded workloads as well. The scheme could be extended to identify cache entries that are shared amongst concurrent threads and partition the cache in such a way that entries which exhibit a high degree of sharing are not rejected from the cache.

7.2.3 Power Consumption

The hardware overhead of the proposed cache partitioning scheme was analysed and found to be relatively small compared to the area of the cache. Additionally, a linear partitioning algorithm was presented in an attempt to reduce the complexity of deducing the partition sizes for each cache line.

However, power is nowadays becoming an increasingly significant constraint in computer design. Therefore, it is necessary to evaluate the effect of the cache partitioning mechanism on the power consumption of the system.

7.2.4 Exposure to Higher System Levels

This work proposed a new cache partitioning scheme in an attempt to improve the overall performance of the system. The partitioning module was designed as an extension to the existing hardware infrastructure of CMPs to remain invisible from higher levels of the system. However it could be modified in order

to be exposed to the operating system or the compiler.

More specifically, the partitioning mechanism acquires an estimate of the cache usage for each running process. This information could be used by the operating system to determine which processes should be scheduled together in order to minimise the impact of inter-thread cache contention and improve the overall performance as it was proposed by Fedorova [16] and Chandra *et al.* [8].

In a different scenario, the partitioning scheme could be exploited to provide Quality of Service (QoS), similarly to the framework proposed by Iyer [27]. The operating system could assign different priorities to the running processes. The partitioning algorithm could then be extended to take into account the priority level of each process and regulate their cache space allocation accordingly in order to guarantee the desired performance.

7.3 Final Remarks

Chip multi- and many-core architectures are posing new challenges for system designers. This work has shown that it is possible to increase cache efficiency and alleviate some of the new problems by partitioning the cache appropriately. It is hoped that the results of this research will illustrate the importance of reevaluating 'accepted wisdom' in cache design.

Bibliography

[1] The International Technology Roadmap for Semiconductors 2005 Edition: Executive Summary, 2005.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[3] AMD. Multi-core processors: The next evolution in computing. *AMD White Paper*, 2005.

[4] M. Bernstein, J. D. Bolter, M. Joyce, and E. Mylonas. Architecture for volatile hypertext. In *Hypertext '91 Proceedings*, pages 243–260, December 1991.

[5] B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] F. Bodin and A. Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE Transactions on Computers*, 46(5):530 – 544, May 1997.

[7] D. Burger and J. R. Goodman. Billion-transistor architectures: there and back again. *Computer*, 37(3):22–28, March 2004.

[8] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, pages 340–351, 2005.

[9] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 264 – 276, 2006.

[10] D. Chiou, L. Rudolph, and S. Devadas. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference*, June 2000.

[11] Z. Chishti, M. D. Powell, and T. N. Vijayjumar. Distance associativity for high performance energy efficient non-uniform cache architectures. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 55–66, 2003.

[12] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication and capacity allocation in CMPs. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 357 – 368, 2005.

[13] J. D. Collins and D. M. Tullsen. Hardware identification of cache conflict misses. In *Proceedings of International Symposium on Microarchitecture*, pages 126–135, 1999.

[14] A. Dinn, I. Watson, C. Kirkham, and A. El-Mahdy. The Jamaica virtual machine: A chip multiprocessor parallel execution environment. Technical report, University of Manchester, August 2005.

[15] H. Dybdahl, P. Stenström, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. In *Proceedings of the 13th International Conference on High Performance Computing*, pages 22–34, 2006.

[16] A. Fedorova. *Operating system scheduling for chip multiprocessor architectures*. PhD thesis, Harvard University, 2006.

[17] W. C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin. Stochastic Fair Blue: A queue management algorithm for enforcing fairness. In *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1520–1529, 2001.

[18] M. A. Frumkin, M. Schultz, H. Jin, and J. Yan. Implementation of the NAS Parallel Benchmarks in Java. Technical report, NASA Advanced Supercomputing Division, 2002.

[19] P. P. Gelsinger. Microprocessors for the new millennium: Challenges, opportunities and new frontiers. In *Proceedings of the IEEE International Solid-State Circuits Conference*, pages 22–25, 2001.

[20] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of ACM International Conference on Supercomputing*, pages 338–347, 1995.

[21] A. González, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 11th International Conference on Supercomputing*, pages 76 – 83, 1997.

[22] L. L. Gremilion. Designing a Bloom filter for differential file access. *Communications of the ACM*, 25(9):600–604, September 1982.

[23] M. J. Horsnell. Cycle-accurate, distributed chip multiprocessor simulation. In *Proceedings of the EPSRC Postgraduate Research in Engineering and Physical Sciences (PREP)*, 2004.

[24] M. J. Horsnell. *A chip multi-cluster architecture with locality aware task distribution*. PhD thesis, School of Computer Science, The University of Manchester, 2007.

[25] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proceedings of the 19th International Conference on Supercomputing*, pages 31–40, June 2005.

[26] Intel. Introducing the 45nm Next-Generation Intel® Core™ Microarchitecture. *Intel White Paper*, 2007.

[27] R. Iyer. CQoS: A framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of 18th International Conference on Supercomputing*, pages 257–266, June. 2004.

[28] T. L. Johnson. *Run-time adaptive cache management*. PhD thesis, University of Illinois, 1998.

[29] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power 5 chip: A dual-core multithreaded processor. *IEEE Micro*, pages 40–47, Mar 2004.

[30] M. Kampe, P. Stenström, and M. Dubois. Self-correcting LRU replacement policies. In *Proceedings of the 1st Conference on Computing Frontiers*, pages 181–191, 2004.

[31] M. Kharbutli, K. Irvin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 288–299, Feb. 2004.

[32] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. In *Proceedings of the International Conference on Computer Design*, pages 61 – 68, Oct. 2005.

[33] J. Kihm, A. Settle, A. Janiszewski, and D. Connors. Understanding the impact of inter-thread cache interference on ILP in modern SMT processors. *Journal of Instruction Level Parallelism*, 7, 2005.

[34] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of International conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002.

[35] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 111–122, Sept. 2004.

[36] P. Kongerita, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, Mar/Apr 2005.

[37] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *Proceedings of International Symposium on Performance Analysis of Systems and Software*, pages 164–171, 2001.

[38] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[39] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, March-April 2005.

[40] A. Mendelson, J. Mandelblat, S. Gochman, A. Shemer, R. Chabukswar, E. Niemeyer, and A. Kumar. CMP Implementation in Systems based on the Intel® Core™ Duo Processor. *Intel® Technology Journal*, 10(2), May 2006.

[41] K. E. Moore, M. D. Hill, and D. A. Wood. Thread-level transactional memory. Technical report, Computer Science Dept., UW-Madison, March 2005.

[42] J. K. Mullin. A second look at Bloom filters. *Communications of the ACM*, 26(8):570–571, August 1983.

[43] J. K Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, May 1990.

[44] J. K. Mullin. Estimating the size of a relational join. *Information Systems*, 18(3):189–196, 1993.

[45] J. K. Mullin and D. J. Margoliash. A tale of three spelling checkers. *Software - Practice and Experience*, 20(6):625–630, June 1990.

[46] J. K. Peir, S. C Lai, S. L. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of International Conference on Supercomputing*, pages 189–198, 2002.

[47] J. K. Peir, Y. Lee, and W. W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 240–250, 10 1998.

[48] M. Qureshi, D. Thompson, and Y. N. Patt. The v-way cache: Demand-based associativity via global replacement. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 544 - 555, 2005.

[49] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high performance runtime mechanism to partition shared caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 423–432, 2006.

[50] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237–1239, October 1989.

[51] R. Ricci, S. Barrus, D. Gebhardt, and R. Balasubramonian. Leveraging bloom filters for smart search within NUCA caches. In *Proceedings of Workshop on Complexity-Effective Design*, June 2006.

[52] S. Ross. *A first course in probability*. Pearson Prentice Hall, seventh edition, 2006.

[53] A. Rousskov and D. Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22-23):2155–2168, April 1998.

[54] A. Settle, D. Connors, E. Gilbert, and A. González. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, pages 221–233, Dec. 2005.

[55] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[56] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.

[57] L. Spracklen and S. G. Abraham. Chip multiprocessing: Opportunities and challenges. In *Proceedings of International Symposium on High Performance Computer Architecture*, pages 248 - 252, 2005.

[58] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, Sept. 1992.

[59] G. E. Suh, S. Devadas, and L. Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of International Conference on Supercomputing*, pages 1–12, 2001.

[60] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28:7–26, 2004.

[61] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 414–423, 1986.

[62] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, 1995.

[63] G. Tyson, M. Farrens, J. Mathews, and A. R. Pleszkun. A modified approach to data cache management. In *Proceedings of International Symposium on Microarchitecture*, pages 93–103, 1995.

[64] J. Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, 29(5):36–39, October 1999.

[65] S. Wang and L. Wang. Thread-associative memory for multicore and multithreaded computing. In *Proceedings of the International Symposium on Low Power Electronics and Design*, Oct. 2006.

[66] W. A. Wong and J. Baer. Modified LRU policies for improving second-level cache behavior. In *Proceedings of International Symposium on High Performance Computer Architecture*, pages 49–60, 2000.

[67] G. Wright. *A single-chip multiprocessor architecture with hardware thread support*. PhD thesis, Department of Computer Science, The University of Manchester, 2001.

[68] M. Zhang and K. Asanović. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.