Dynamic Cache Partitioning and Adaptive Cache Replacement Schemes for Chip Multiprocessors

Norfadila Mahrom

B.Eng.(Computer Engineering)
M.Sc.(Intelligent System)

A THESIS SUBMITTED IN FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING
THE UNIVERSITY OF ADELAIDE
AUSTRALIA

2014

CONTENTS

List of Figures ix

List of Tables xiii

List of Algorithms xv

Abstract xvii

Declaration of Originality xix

Acknowledgments xxi

List of Abbreviations xxiii

1 Introduction 1

1.1 Shared Cache of CMP Architectures and Design Challenges 2

1.2 Research Goal 6

1.3 Contributions of this Thesis 7

1.4 Thesis Overview. 9

2 Background 13

2.1 Multiprocessor System Design Space Exploration. 13

2.1.1 Multiprocessor architectures. 16

2.2 Cache Management Schemes 21

2.3 Cache Replacement Policy 28

2.4 Summary 35

3 Simulation Framework **37**

3.1 The Xtensa LX4.0 Processor 37

3.2 Simulation Framework and Environment 43

3.3 SPEC2000 Benchmarks 49

3.4 Methodology 52

3.5 Summary 55

4 Cache Partitioning Schemes **57**

4.1 Simple Static Allocation Scheme 58

4.1.1 Introduction 58

4.1.2 Evaluation 59

4.2 Utility-based Cache Partitioning Scheme 66

4.2.1 Overview 66

4.2.2 Evaluation 67

4.3 CPI-based Cache Partitioning Scheme 72

4.3.1 Overview 72

4.3.2 Evaluation 73

4.4 Summary 77

5 Cache Replacement Policy **81**

5.1 Introduction 82

5.2 Motivation 83

5.3 Adaptive Insertion and Promotion Policy 85

5.3.1 Description 85

5.3.2 Comparing Operation of MI2PP policy to PIPP 88

5.3.3 Simulation Results 92

5.3.4 Hardware Overhead 107

5.3.5 Performance Analysis 110

5.3.5.1 Comparison with LRU, UCP, and PIPP 110

5.3.5.2 Comparison with another cache partitioning scheme. 115

5.3.5.3 Cache size sensitivity study 117

5.3.5.4 Implementation in the unpartitioned shared cache. 121

5.3.6 Conclusion 123

5.4 Partition-based Replacement Policy 124
5.4.1 Description 124
5.4.2 Evaluation 126
5.4.3 Conclusion 129
5.5 Summary 133

6 Shared Cache Partitioning Based On Performance Gain Estimations 135
6.1 Introduction 136
6.2 Motivation 138
6.3 Structure of The New Scheme 141
6.3.1 Monitoring Environment and Scheduler 143
6.3.2 Identifying the Partitioning Dependence. 145
6.4 Adaptive CPI-based Cache Partitioning Scheme 147
6.5 Experimental Evaluation. 150
6.5.1 Simulation Methodology 151
6.5.2 Simulation Results 151
6.6 Performance Analysis 155
6.6.1 Cache Scaling and Sensitivity Analysis 155
6.6.2 Effect of Dynamic Set Sampling 159
6.6.3 Comparison with Different Cache Replacement Policies 161
6.7 Complexity Management and Hardware Implementation 164
6.8 Summary 166

7 Conclusion 169
7.1 Cache Replacement Policy 170
7.2 Cache Partitioning Scheme 171
7.3 List of Contributions 173
7.4 Limitations 174
7.5 Future Work 175
7.6 Final Remarks 177

Bibliography 179

LIST OF FIGURES

2.1 Multithreading in a multiprocessor system. 14

2.2 Basic structure of a CMP. Multiple processors with a private level 1 cache share the same physical level 2 cache [Pal et al. 2012]. 17

2.3 Basic structure of a SMP. Multiple processors interconnected to their private level 1 and level 2 caches [Pal et al. 2012]. 18

2.4 The basic structure of CLMP consists of multiple processors grouped into clusters (cluster size = 2) [Pal et al. 2012]. 19

3.1 Xtensa LX4.0 processor architectural block diagram [Tensilica Inc. 2011]. 39

3.2 Xtensa processor interfaces. 41

3.3 The baseline architecture. 44

3.4 The interconnection of the cores and cache hierarchy using the PIF in a dual-core system. 46

4.1 Total L2 cache misses improvement over standard LRU. 61

4.2 Distribution of total misses caused by each individual core in the quad-core system. 63

4.3 The speed-up of the partitioned cache with the augmented LRU policy over the standard LRU replacement policy. 65

4.4 Comparison of UCP, LRU, and EQ cache misses in a quad-core system. 68

4.5 Total L2 misses incurred by each core when the system using UCP, LRU and EQ. 70

4.6 Performance improvement of UCP over LRU and EQ for a quad-core system. . . 71

4.7 Reduction of total cache misses of CPI-based partitioning scheme over LRU and EQ on in a quad-core system. 74

4.8 Distribution of total L2 misses incurred by each core in the system. 75

4.9 Performance of CPI-based partitioning scheme over LRU and EQ for a quad-core system. 76

5.1 Example of insertion positions in PIPP. Consider Core 1 is allocated with 5 ways and Core 2 receives 3 ways from UCP scheme. 84

5.2 Example operation of (a) PIPP from Xie and Loh [2009], and (b) MI2PP policy for a variety of insertions and promotions activities. Assume evictions for both policies always choose the lowest-priority cache line in the set. 89

5.3 Performance result for IPC throughput of UCP, PIPP, and MI2PP policy normalised to an LRU-managed cache. 94

5.4 Performance as measured by the weighted speedups of IPC for UCP, PIPP and MI2PP policy compared to baseline LRU. 96

5.5 Comparison of UCP, PIPP and MI2PP policy over traditional LRU for the harmonic mean fairness metric. 97

5.6 L2 miss rate in MPKI of UCP, PIPP and MI2PP compared to baseline LRU policy. 98

5.7 (a) IPC throughput of the 8-core system, and (b) L2 miss rate of the 4-core and 8-core systems, using PIPP and MI2PP, relative to the baseline LRU policy. 100

5.8 (a) IPC throughput, and (b) L2 miss rate of the 16-core system, using PIPP and MI2PP, relative to the baseline LRU policy. 103

5.9 Distribution of L2 miss rate of each core in the 16-core system. 105

5.10 Distribution of L2 miss rate between MI2PP and LRU. 111

5.11 Number of cores using lower priority positions as the insertion location. 113

5.12 Distribution of L2 miss rate among UCP, PIPP and MI2PP. 114

5.13 Performance gain of MI2PP over a system using CPI-based cache partitioning scheme. 116

5.14 Total miss rate reduction normalised to the original LRU replacement policy. . . 117

5.15 Miss rate relative to the baseline LRU policy of a system with shared cache sizes of (a) 512KB, and (b) 1MB using different cache partitioning schemes and MI2PP policy. 118

5.16 Miss rate, relative to UCP, of MI2PP/UCP policy implemented in different cache sizes. 120

5.17 Miss rate reduction of MI2PP policy implemented in different sizes of unmanaged shared cache. 122

5.18 Performance improvement of PRP normalised to baseline LRU policy and UCP. 126

5.19 Performance results for the weighted speedup and fair speedup metrics. 127

5.20 L2 miss rate of PRP normalised to LRU and UCP. 129

5.21 Performance improvement of UCP, PRP and EPRP in a 512KB cache, normalized to LRU policy. 131

5.22 L2 miss rate of PRP and EPRP, relative to LRU policy. 132

6.1 Each cache line is extended to include core identification. 141

6.2 The shadow tag array and four miss counters associated with each recency position in the monitoring hardware structure of each core. 144

6.3 Miss rate improvements relative to unmanaged traditional LRU cache. 152

6.4 Performance comparison of UCP, CPI and ACCP over unmanaged LRU L2 shared cache. 153

6.5 Performance comparison using the (a) weighted speedup, and (b) harmonic mean of weighted IPC metrics. 154

6.6 Performance comparison of ACCP, UCP and CPI-based schemes on different sizes of L2 shared cache, relative to baseline LRU. 156

6.7 Miss rate reduction of ACCP over UCP and CPI-based schemes. 157

6.8 Distribution of L2 miss rate among UCP, CPI and ACCP. 158

6.9 IPC performance improvement of ACCP using (1) DSS (labelled as ACCP_S), and (2) UMON-global (labelled as ACCP) for different cache sizes, relative to baseline LRU. 160

6.10 Performance comparison of MI2PP replacement policy employed on ACCP, UCP, and CPI-based schemes, relative to baseline LRU. 162

6.11 Miss rate of MI2PP replacement policy employed on ACCP, UCP and CPI-based schemes, relative to baseline LRU. 162

LIST OF TABLES

3.1 Xtensa LX4.0 processor configuration. 44
3.2 Baseline system's configuration. 46
3.3 Classification of SPEC CPU2000 applications. 50
3.4 Workload summary. 50
3.5 List of stoppers/first applications to finish its execution in each workload mix. . . 54

5.1 Workload mixes for the 8-core system. 102
5.2 A hardware cost of an UMON circuit associated to a single processor. 108
5.3 Replacement logic complexity of the LRU, PIPP and MI2PP policies. 109

6.1 A snapshot of the information recorded during one of the execution intervals. . 146
6.2 Hardware overhead per processor. 165

LIST OF ALGORITHMS

6.1 Pseudo code for finding a line for eviction. The function returns the position of the line to be replaced. 142
6.2 Adaptive CPI-based Cache Partitioning scheme. 149

ABSTRACT

One of the dominant approaches towards implementing fast and high performance computer architectures is the Chip Multi Processor (CMP), in which the design of the memory hierarchy has a critical effect on performance. Performance can be improved by the use of a shared cache on the chip, but it is a matter of ongoing research as to how each processor can gain the greatest advantage from the cache without affecting the performance of other processors. Moreover, power is a critical issue in CMP design.

Cache replacement policies and cache partitioning schemes have been investigated and proven able to enhance shared cache management. However, it is still desirable to have an optimal replacement policy that can retain useful data as long as possible to minimise miss rate and not degrade performance in a partitioned shared cache. Many of the metrics that have led to innovations in various partitioning schemes have increased the complexity of the partitioning strategies and the hardware overhead. There is scope for more work in achieving the right balance between power consumption and performance improvement in the CMP.

This thesis investigates the effects of the cache replacement policy in a partitioned shared cache. The goal is to quantify whether a better power/performance trade-off can be achieved by using less complex replacement strategies. A Middle Insertion 2 Positions Promotion (MI2PP) policy is proposed to eliminate cache misses that could adversely affect

the access patterns and the throughput of the processors in the system. The insertion, promotion and eviction strategies of the replacement policy are investigated and modified to improve shared cache utilisation by the competing processors. The MI2PP policy employs a static predefined insertion point, near distance promotion and the concept of ownership in the eviction policy to avoid resource stealing among the processors. With these strategies, the performance of the shared cache and the overall system were enhanced and the miss rate showed significant improvement over the Least Recently Used (LRU) policy.

While existing cache partitioning schemes use a variety of performance metrics to allocate the cache for each competing processor, most of the schemes focus only on one metric in their partitioning algorithm. An Adaptive Cycles per Instruction (CPI)-based Cache Partitioning (ACCP) scheme is introduced to investigate the efficiency of using two metrics to optimise partitioning decisions and to study the trade-offs between the complexity of using more performance metrics in partition decision-making and additional hardware cost. The analysis performed on ACCP showed that the performance of the processors was improved compared to the existing CPI-based partitioning scheme introduced by Muralidhara et al. [2010], which uses only one of the performance metrics employed in ACCP. Evaluation on a more complex scheme, namely the Utility Cache Partitioning (UCP) scheme demonstrated that the ACCP on average achieved similar performance although the ACCP is simpler to implement. The low hardware overhead incurred by ACCP showed that it is superior to UCP. ACCP demonstrates that the complexity of the partitioning mechanism and hardware cost could be reduced without degrading the overall system performance.

DECLARATION OF ORIGINALITY

This work contains no material that has been accepted for the award of any other degree or diploma in any university or other tertiary institution to Norfadila Mahrom and, to the best of my knowledge and belief, contains no material previously published written by another person, except where due reference has been made in the text.

I give consent to this copy of the thesis, when deposited in the University Library, being available for loan, photocopying, and dissemination through the library digital thesis collection, subject to the provisions of the Copyright Act 1968.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library catalogue, the Australasian Digital Thesis Program (ADTP) and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

_____ _____

Signed Date

ACKNOWLEDGEMENTS

In the Name of Allah, the Most Gracious, the Most Merciful.

All praise and thanks to Allah.

My very great appreciation and deep gratitude to my principal supervisor, Associate Professor Michael J. Liebelt for his patient guidance, assistance, encouragement and useful criticism of my research work. His willingness to allocate his time so generously to support me in many aspects and his trust in me has been very much appreciated. I also would like to express my sincere thanks to Dr. Braden J. Phillips for his valuable and constructive suggestions during the planning and development of this research work.

The generosity of the School of Electrical and Electronic Engineering, The University of Adelaide in funding extensive access to the Tensilica tools under Tensilica Inc. University Program is highly appreciated. This work would not have been possible without the support of David Bowler in all issues related to technical resources. My research candidature was sponsored by the Ministry of Higher Education, Malaysia and Universiti Malaysia Perlis, and supported by Prof. Dr. R. Badlishah Ahmad, Prof. Dr. Ali Yeon Md Shakaff and Prof. Dato' Dr. Zul Azhar Zahid Jamal.

My grateful thanks are extended for the members of CHiPTec, particularly for everyone in the Journal Club for the valuable meetings that had introduced me to the best platform in expending my knowledge and research work. Not to forget, million thanks to Noorfazila

Kamal, Dr. Puteh Saad, Deborah Coleman-George, Ivana Rebellato and Chris Andrew Haerberli for their motivation and continuous moral support.

Finally, to my family. Thank you all for being you. You know how much you have done to make me come this far.

Norfadila Mahrom

LIST OF ABBREVIATIONS

ACCP	Adaptive CPI-based Cache Partitioning
ACCP_S	ACCP with Sampled sets
API	Application Programming Interface
ART 2	Adaptive Resonance Theory 2
BIP	Bimodal Insertion Policy
CLMP	Clustered Multi Processor
CMP	Chip Multi Processor
CPI	Cycles per Instruction
CPU	Central Processing Unit
DIP	Dynamic Insertion Policy
DPP	Dynamic Promotion Policy
DSS	Dynamic Set Sampling
EQ	Equally Partitioned
FIFO	First-In First-Out
FPGA	Field-Programmable Gate Array
HU	High Utility
I/O	Input/Output
ID	Identification
IPC	Instructions per Cycle
ISA	Instruction Set Architecture

ISS	Instruction Set Simulator
ISSCC	IEEE international Solid-State Circuits Conference
L1	Level one
L2	Level two
L3	Level three
LFU	Least Frequently Used
LIP	LRU Insertion Policy
LLC	Last Level Cache
LRU	Least Recently Used
LU	Low Utility
MI2PP	Middle Insertion 2 Positions Promotion
MMU	Memory Management Unit
MPKI	Misses per 1000 Instructions
MPP	MRU Promotion Policy
MRU	Most Recently Used
MU	Marginal Utility
PIF	Processor Interface
PIPP	Promotion/Insertion Pseudo-Partitioning
PRP	Partition-based Replacement Policy
PSEL	Policy Selector
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
SIPP	Single-step Incremental Promotion Policy
SMP	Symmetric Multi Processor

SPEC	Standard Performance Evaluation Corporation
SU	Saturating Utility
TADIP	Thread-Aware Dynamic Insertion Policy
TADPP	Thread-Aware Dynamic Promotion Policy
TIE	Tensilica Instruction Extension
UCP	Utility-Cache Partitioning
UMON	Utility Monitor
UMON_DSS	Utility Monitor with Dynamic Set Sampling
VPR	Versatile Place and Route
XCC	Xtensa C/C++ Compiler
XPG	Xtensa Processor Generator
XTMP	Xtensa Modeling Protocol

CHAPTER 1

INTRODUCTION

In today's microprocessor industry, a trend to increase the number of processors to execute multiple tasks simultaneously on a single die has lead to the development of multi-core architectures with high performance memory hierarchies. This trend is targeted to meet desired energy, power and performance metrics of high performance computer architecture design by focusing on how to utilize the numerous on-chip transistors. However, this alternative design paradigm to uniprocessor systems has created new challenges in on-chip memory hierarchy design. As a result, the Chip Multi Processor (CMP) has become a key component in this quest. In a CMP, multiple independent processors interact and share common resources such as caches and main memory to achieve higher computational throughput. The key feature of a CMP is that it allows applications/threads to be executed concurrently on a single chip and to exploit task/thread-level parallelism. However, by increasing the number of program threads running in parallel, processor architects face design issues in organising and managing the on-chip shared resources. These issues relate to memory sharing mechanism problems such as memory contention, cache coherence and increasing hardware cost. This has occurred because all the elements in the CMP are sharing the same main memory resource.

The growth in transistor densities has dramatically increased the number of transistors

integrated on a single chip. Over the past two decades, the trend has followed an exponential curve, reaching 4 billion transistors in 2013 [ITRS Organization 2014a]. Today, the increasing demand for high performance processors, the growth in dimensional and functional scaling and memory size has triggered many popular multiprocessor designs including the latest development from IBM Corporation which contains more than 2 billion transistors on a die [Bursky 2013]. However, these developments have caused power consumption and design complexity to become critical issues in CMP architecture. The growth in the number of processors contributes to increasing demands on the shared resources. As a result, the performance and power optimisation of a CMP has not increased in proportion to the growth in the number of transistors per chip [Leverich et al. 2007; Kędzierski et al. 2010a].

It is known that the abovementioned memory sharing issues are caused by the interconnection of elements in the CMP. It is crucial to investigate and optimise this interconnection by focusing on the memory sharing mechanisms. Problems with shared memory have motivated the work presented in this thesis and will be discussed further. This will be followed by a summary of the research goal and finally, for easy reference, the structure of the thesis.

1.1 Shared Cache of CMP Architectures and Design Challenges

The CMP is an important design innovation in the high performance and embedded processors domain. According to Jing et al. [2007], Leverich et al. [2008], Mandke et al. [2010], Tam et al. [2011], Huang et al. [2012] and Nanehkaran and Ahmadi [2013], CMP is the standard processor for most manufacturers due to its improved parallel processing

techniques (program threads or multiple applications are executed by the many processors simultaneously) and its advances in nanometer technologies. Interconnection plays a major role, enabling the multiple processors or cores to interact and share programs and data concurrently. However, the number and length of the interconnections in CMP significantly impact cost and power consumption. Consequently, there has been a great deal of research in on-chip communication architectures to help overcome this limitation [Cheng et al. 2006; Flores et al. 2007; Flores et al. 2010; Mandke et al. 2010].

With this increase in the number of processors, the interconnection framework for the shared resources has introduced other issues in CMP design [Penry et al. 2006; Hennessy and Patterson 2007; Mutlu and Moscibroda 2007; Poletti et al. 2007; Theelen and Verschueren 2002; Shiue and Chakrabarti 1999 and Chandra et al. 2005]. Competition among multiple processors executing different threads that require simultaneous access to the shared memory may cause memory coherence problems or inter-thread interference. Inter-thread interference occurs when two continuous accesses to a memory location are from different threads [Becchi et al. 2007]. However, multiple processors may need to access the same location concurrently. Uncontrolled inter-thread interference among multiple cores might lead to memory contention problems and increase the cache miss rate [Sato et al. 2012]. This will eventually unfairly prioritise some threads in the system while others have to wait for a long period to get access to the shared resources.

In multiprocessor architecture, a cache is a small high-speed device used for staging the movement of data between main memory and processors. Its purpose in CMPs is to reduce the external bus activities that would consume high power, as well as to control the competition among the many processors. Multilevel cache design is employed in CMP to improve the overall system performance by resolving the issues of memory contention and

cache coherence among processors. This is possible because the multilevel cache tries to contain the whole working sets of each processor in the caches, thereby eliminating those conflicts that could increase the cache miss rate. Moreover, the different size and access time of each cache level, i.e. the proximity of the first cache levels to the processors, can provide an advantage for the system by reducing the access penalty and speeding up the system. For that reason, cache architecture has become one of the critical factors in the overall performance of a CMP.

Previous research has proposed several techniques to optimise the performance of CMP by designing a faster CMP with a large memory size and a large number of processors. Continuous growth in the transistor count, i.e. doubling in every technology generation, has increased the static power consumption of the memory. In recent technologies, increasing power consumption has become a primary issue. According to the ITRS roadmap [ITRS Organization 2014a, 2014b], power management will continue to be one of the critical challenges for the next 14 years. The scale of the memory/cache size in multiprocessor design may be limited by power consumption issues [Leverich et al. 2007].

Note that, power consumption in a memory/cache is due to dynamic power consumption and static power consumption. Currently, dynamic power consumption is the biggest contributor to the total power consumption in the memory/cache system [Juan and Shuai 2012]. However, as the size of a transistor becomes smaller and the threshold voltage continues to decrease, the static power consumption in the memory/cache also increases. These facts therefore have diverted most CMP experts' focus from adding more individual memory/cache in a CMP, which may significantly increase power usage, to optimize the utilization of the available shared memory/cache. Consequently, shared cache management remains of critical interest to CMP architecture researchers in their attempts to provide

continuous improvements to system performance.

Even though system performance improvement is usually accompanied by energy consumption in a system, additional energy consumption due to on-chip memory/cache traffic, snoop requests or directory lookups has become another crucial issue for CMP designers [Leverich et al. 2007]. This means that the energy usage due to the interconnect network and memory will constrain the improvement in individual processor performance, hence in the overall CMP performance. Therefore, CMP researchers also committed to addressing issues of energy efficiency in CMP [Juan and Shuai 2012]. In this thesis, the energy issues are not addressed in detail but the hardware complexity analysis of the work presented in this thesis is used as an indicator of energy consumption. Energy models in memory/cache systems are challenging to address, and are highly dependent on the technology of implementation. It is beyond the scope of this thesis to implement a detailed model, so we use an analysis of the complexity of the logic as a proxy for a measure of increase in power consumption. We acknowledge that this is only a coarse indicator and that this is a limitation of this work.

There are at least two main elements in managing the shared cache resources: the cache replacement policy and the cache partitioning scheme. The main principle is to maximise the shared resource utilisation among the competing cores. Research in these areas has resulted in many replacement policies and partitioning schemes using different mechanisms and techniques to manipulate the information in the shared cache.

Some researchers have focused on optimising the cache replacement policy to improve the utilisation of cache resources [Chen et al. 2009; Jaleel et al. 2008; Li et al. 2011; Qureshi et al. 2007] and reduce the interference among multiple cores in the system [Jiang and

Zhang 2002; Sato et al. 2012; Wu et al. 2010], while others preferred to improve the distribution of cache resources among the cores so that each core obtains ownership in the shared cache, thereby reducing competition in the CMP [Chaturvedi et al. 2010; Dybdahl et al. 2006; Sharifi et al. 2012]. Many measures were used to evaluate the cores' performance and to decide the partition sizes of each core in the system [Manikantan et al. 2012; Qureshi and Patt 2006; Reddy and Petrov 2010; Xie and Loh 2008]. An approach that combines both a replacement policy and a partitioning scheme in one single technique targeted to enhance shared cache management has also been investigated by system designers [Duong et al. 2012; Muralidhara et al. 2010; Sui et al. 2010; Xie and Loh 2009]. Even so, various techniques used to optimise the shared cache utilisation in CMP would introduce a significant hardware overhead, an issue of major concern to CMP architects [Becchi et al. 2007; Chen et al. 2009; Kędzierski et al. 2010a; Qureshi and Patt 2006].

1.2 Research Goal

In the previous section, we have discussed how shared cache management is important to optimise overall system performance. While a cache replacement policy is used to organise and manage the occupancy of data lines in the cache, a cache partitioning scheme must also be taken into consideration. Its ability to improve the cache throughput, guarantee fairness among cores and eliminate inter-thread shared cache conflicts and design complexity are beneficial to both system performance and cost.

Therefore, the goal of the work in this thesis is made up of three parts:

- To investigate the effects of a cache replacement policy in a partitioned shared

cache and the impacts of various performance metrics used in a cache partitioning scheme,

- To introduce an improved cache replacement policy that focuses on a partitioned shared cache and

- To optimise a cache partitioning scheme that could provide new insight by using multiple performance metrics in partition decision-making without incurring significant hardware overhead.

1.3 CONTRIBUTIONS OF THIS THESIS

This thesis addresses the distribution of the shared cache resources among the cores to minimise the adverse effects on performance of competition for the cache. This work has included the development and evaluation of new eviction, insertion and promotion policies in the cache. The major contribution of this thesis is derived from the goal to improve cache resource sharing among competing applications and is summarised in the following points:

- **Middle Insertion 2 Positions Promotion (MI2PP) replacement policy – Chapter 5:** Some drawbacks of the commonly used Least Recently Used (LRU) replacement policy employed in multiprocessor systems are due to the cache lines residing in the cache longer than required. In this thesis, the predetermined middle position insertion and two priority positions promotion policies of MI2PP help retain cache lines long enough in the cache to manifest more hits and to effectively improve cache thrashing in a partitioned shared cache. MI2PP has also shown that strictly enforcing the shared cache partitioning remains a better partitioning strategy than the

pseudo-partitioning of Pseudo Insertion/Promotion Policy (PIPP) proposed by Xie and Loh [2009].

- **Partition-based Replacement Policy (PRP) – Chapter 5:**

  This thesis proposes the PRP to investigate the impact of an adaptive eviction policy while introducing the consideration of whether an application is over-allocated as another metric to determine the victim in the cache replacement strategy. The PRP uses the allocation status of applications in the system to avoid the possibility of resource stealing from an under-allocated application, causing the application to suffer from inadequate resources and cache thrashing that may further hurt the overall performance of whole system. The PRP has shown that it is able to improve upon the baseline LRU policy, but it did not demonstrate any overall performance improvement compared to Utility-based Cache Partitioning (UCP) by Qureshi and Patt [2006].

- **An Extended Partition-based Replacement Policy (EPRP) – Chapter 5:**

  To improve the performance of PRP over UCP, the original PRP is modified to use the middle position insertion policy, instead of the LRU position insertion policy. The goal of EPRP is to reduce the number of cache misses in PRP that could be due to multiple-reuse lines in the cache. This is because these lines are quickly evicted in PRP before their next reuses, as they are installed at the LRU position. Overall, the EPRP has demonstrated that it has improved the overall performance of both UCP and the original PRP.

- **A new Adaptive CPI-based Cache Partitioning (ACCP) scheme – Chapter 6:**

  In this thesis, the goal of an ACCP scheme is to dynamically determine the cache

requirements of each individual application based on its latest performance and to enhance the overall throughput of multiple applications sharing a L2 cache in a system. Applications in an ACCP scheme are run at approximately the same speed by accelerating the slowest without significantly decelerating the others. The ACCP has achieved at least similar performance improvements with UCP and better performance compared to a CPI-based partitioning scheme by Muralidhara et al. [2010]. In addition, ACCP shows better performance in managing the hardware overhead compared to the UCP scheme.

1.4 THESIS OVERVIEW

The work presented in this thesis investigates different cache replacement schemes to assess overall system performance when a particular component of the cache replacement policy is changed. The findings are then used as a guide to determine resource distributions among multiple cores in the system. A cache replacement policy is optimised to enhance the performance of a partitioned shared cache. Also, several existing cache partitioning schemes are analysed and a new scheme is proposed to improve the reallocation strategy of the shared cache resources with a low hardware overhead.

In this first chapter, several issues related to the design of high performance shared caches have been highlighted, leading to the formulation of the research aim of this thesis. The process of achieving this goal will be discussed in the following chapters. A summary of the remaining chapters follows:

- Chapter 2 reviews several cache schemes and policies that attempt to improve the

efficiency of a shared cache in multilevel cache hierarchies. A detailed discussion of the relevant background is also presented, which includes cache partitioning schemes and cache replacement policies.

- A review of the Tensilica modeling and simulation platform used for this research work is presented in Chapter 3. The modeling tools, benchmarks and methodology that form the basis of this investigation are discussed, along with the modeling protocol and performance metrics used in the experimental work.

- The employment of cache partitioning schemes proposed by other researchers on a dual-core and quad-core system is evaluated in Chapter 4. The performance of a utility-based partitioning scheme is compared to a CPI-based partitioning scheme, to identify the advantages and drawbacks of the different performance metrics used in both schemes. The results from the evaluation are then used to direct the development of a new partitioning scheme.

- In Chapter 5, some modifications are performed on the traditional LRU replacement policy. The effects of each component in a cache replacement policy are discussed and investigated. A new replacement policy targeted for a partitioned shared cache is proposed.

- Investigations presented in Chapter 4 have motivated the design of a new cache partitioning scheme that aims to improve the cache distributions among multiple cores in the system. In Chapter 6, the drawbacks of the schemes discussed in Chapter 4 are addressed, while a new scheme, which tries to reduce additional hardware and design complexity, is introduced. This contribution is advantageous because the partitioning algorithm of the new scheme uses multiple performance

metrics to determine the partition sizes of many cores.

- Finally, the conclusion of this thesis and suggestions for further work are presented in Chapter 7.

CHAPTER 2

BACKGROUND

This chapter discusses the cache performance issues and related prior research that have motivated the work in this thesis. The discussion starts in Section 2.1 with an introduction to memory design space exploration in multiprocessor systems and the several limitations that are encountered in system design. Section 2.2 reviews existing implementations of efficient and smart cache management schemes to manage the issues identified in Section 2.1. Section 2.3 presents new and adaptive cache replacement rules that are aimed at improving system performance.

2.1 MULTIPROCESSOR SYSTEM DESIGN SPACE EXPLORATION

With the demand for increased computer capability and advances in integrated circuit technology, multiprocessor systems have become the default architecture for achieving high performance processing. Multiprocessor system architectures are designed with multiple independent processors sharing some common resources, particularly memory structures and I/O buses, on a single die. Ideally, all the components on the die can be configured by designers to meet the specific requirements of the workloads running on it. As described by Olukotun et al. [1996], Nayfeh and Olukotun [1997], Lee and Lam [1998], and Mutlu and

Moscibroda [2007], workloads will be compiled into multiple threads of control that will be executed simultaneously by multiple processors. A compiler in a multi-threaded system will identify and distribute workload instructions and data into a number of threads (Figure 2.1).

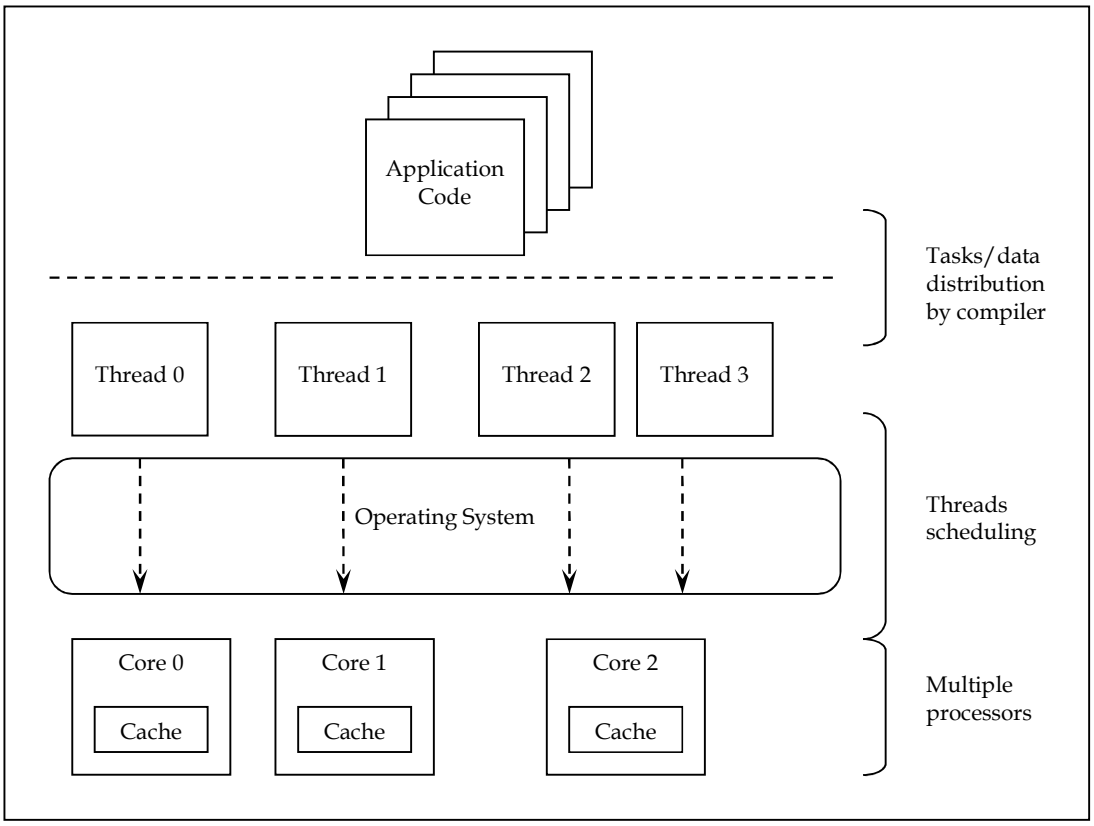


FIGURE 2.1: Multithreading in a multiprocessor system**.**

A thread is a unit of sequential tasks within a parallel program that may consist of multiple tasks or just one independent program. The threads are interruptible during execution, allowing the processor to execute another thread at any instant. Under the control of an operating system, multiple threads can be scheduled to be executed by different available processors in the system. Therefore, the execution path of the threads may be

interleaved, with the scheduling managed by the operating system [Stallings 2006; Hennessy and Patterson 2007].

Multiple processors in a multi-threaded system share a common memory, thereby allowing different threads access to common programs and data in the memory to complete their tasks and to communicate with each other. The need to make use of the common memory is due to the phenomenon of locality of reference among the multiple threads [Stallings 2006]. An efficient interaction and sharing mechanism among different processors is very important in achieving better multiprocessor system performance as the workloads can vary from highly sequential to highly parallel applications. However, these depend on the environment of the applications' code with respect to either explicit or implicit parallelism. As a result, various architectural configurations have been proposed and modelled to deal with a diverse range of application characteristics.

In multiprocessor systems, memories are designed in a hierarchy with different speeds and sizes to provide a large size of less expensive primary memory, while providing the fastest speed available to the processors. Every level of the memories employs similar access strategies, while the principle of locality in the memories gives the processors a chance to overcome the long latency of the primary memory accesses. The size and the access time of the memories in the hierarchy increase according to their distance from the processors. A cache is defined by Stallings [2006] and Jing et al. [2007] as a small high-speed device, usually invisible to the programmer or to the processor, used for staging the movement of data between main memory and processor. Most of the contemporary designs of multiprocessor systems have multiple levels of caches configured as level 1 (L1) and level 2 (L2) [Jing et al. 2007], as well as level 3 (L3) in some architectures. Typically, the size of an L1 cache is 16-64KB with the access time of 3-10 clock cycles. The typical size

of an L2 cache is around 256KB to 1MB while the access time is about 15-100 clock cycles. An L3 cache is typically larger than 1MB and up to tens of megabytes [Stallings 2006; Hennessy and Patterson 2007]. For example, the Intel Core i7 920 processor offers a L3 cache shared by four cores in the chip with the following sizes and access times for each level of caches: (L1) 32KB/ 4 clock cycles, (L2) 256KB/ 10 clock cycles, and (L3) 8MB/ 35 clock cycles [Intel Corp. 2014; Patterson 2014]. Due to the differences in access time and cost of each level, L1 cache is configured to be private to the individual processor. It is fast, small in size and located close to the processor so that the frequently accessed data can be rapidly and efficiently accessed by the processor. Meanwhile, the L2 cache or sometimes the L3 cache, which is often a slower and less expensive memory type, acts as the last level cache (LLC) in the system. The LLC may be shared by multiple processors containing copies of information of the lower level memory such as L1 cache. An on-chip cache is able to reduce the memory latency, since the desired information may already exist in the shared cache, which is faster to access [Larus 1993; Srinivasan et al. 2004; Chandra et al. 2005].

2.1.1 Multiprocessor architectures

There are three different architectural configurations in which multiprocessors are commonly implemented: chip multiprocessor, symmetric multiprocessor and clustered multiprocessor.

Chip Multi Processor (CMP)

Figure 2.2 shows the configuration of a CMP. In a CMP, it is possible for the processors to share a single centralised cache and memory structure. Each processor in the CMP has a private level 1 data and instruction cache, as well as a shared level 2 (or level 3) cache. They are interconnected via buses, while the L2 cache is connected to the main memory via a

memory bus. The CMP configuration is also known as centralised shared-cache design and is generally beneficial for workloads sharing large amounts of data, as the key architectural property is to provide fast data accesses to all the processors [Chang 2007; Hennessy and Patterson 2012].

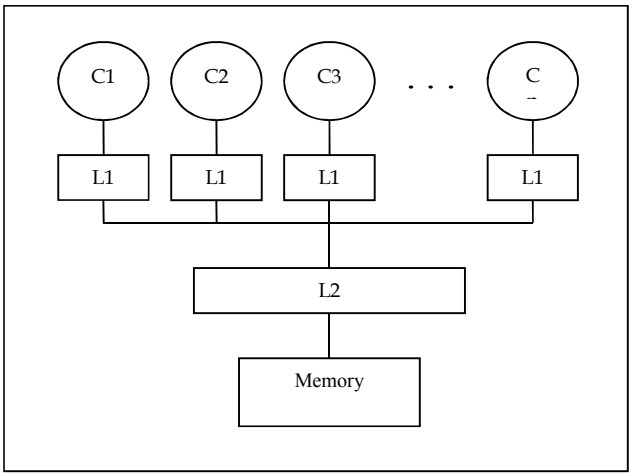


FIGURE 2.2: Basic structure of a CMP. Multiple processors with a private level 1 cache share the same physical level 2 cache [Pal et al. 2012].

Symmetric Multi Processor (SMP)

In the SMP, the caches are physically distributed among the processors to support the processors' bandwidth demands without incurring excessively long access latency. In addition to the private data and instruction level 1 cache, each processor has a private level 2 cache. The L2 caches are privately connected to their respective L1 caches but share an interconnection to the main memory. Figure 2.3 depicts the SMP configuration, where the L2 caches are connected to the main memory by a shared memory bus. Pal et al. [2012] found workloads with minimal data sharing are suitable to be executed in the SMP.

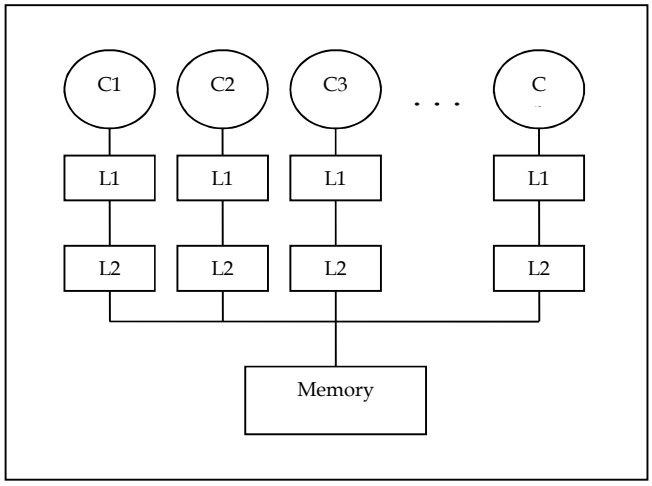


FIGURE 2.3: Basic structure of a SMP. Multiple processors interconnected to their private level 1 and level 2 caches [Pal et al. 2012].

Clustered Multi Processor (CLMP)

In CLMP, multiple processors are grouped into clusters that consist of various numbers of processors, such as 2, 4, 8, and 16. Figure 2.4 shows a CLMP with clusters of 2 processors. Generally, each processor in the cluster has a private level 1 data and instruction cache connected to a shared L2 cache by a bus. Processors typically have a private L1 cache but share an L2 cache. Workloads with a high volume of shared data tend to perform better in a large cluster as they can take advantage of the bigger shared space and high speed L2 [Pal et al. 2012].

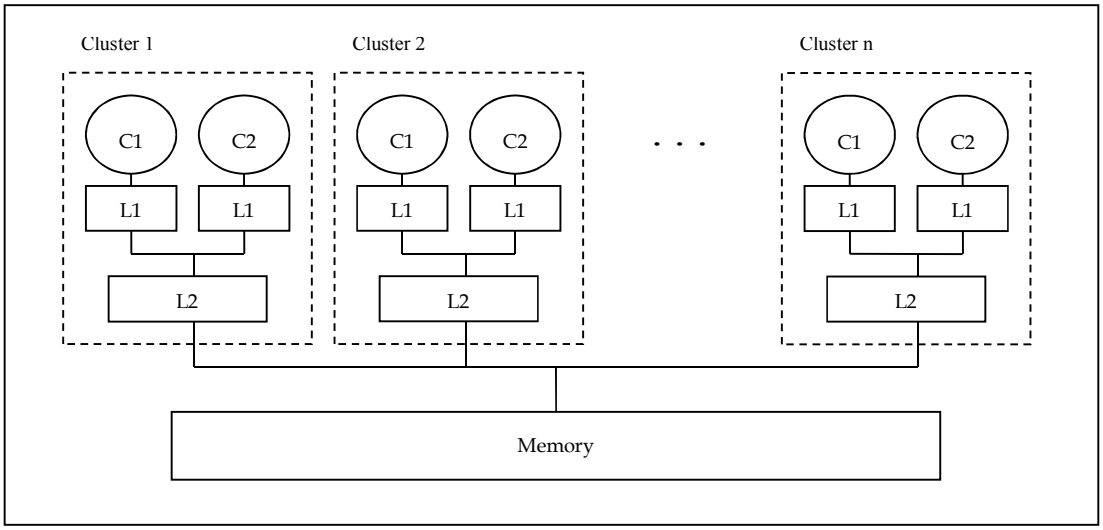


FIGURE 2.4: The basic structure of CLMP consists of multiple processors grouped into clusters (cluster size = 2) [Pal et al. 2012].

In a CMP, there are many architecture parameters that can be varied to optimise the performance of the system. This includes the core count, the cache size and the multilevel cache hierarchy. Unfortunately, depending upon the characteristics and the behaviours of the application workload running on the system, problems such as memory contention and cache coherence may slow down the system and degrade the overall performance [Chandra et al. 2005]. Memory contention may happen if multiple processors in CMP that execute different threads share and need to retrieve instructions or data at specific locations in the memory at the same time [Chandra et al. 2005; Patterson 2014]. Meanwhile, cache incoherence might occur whenever multiple caches in the CMP have copies of shared data. Changes to the data will cause the associated caches to have inconsistent or invalid data because none of the associated caches will notify the others of changes that have been made to one of the data copies.

Lee and Lam [1989], Olukotun et al. [1996], and Mutlu and Moscibroda [2007] claim

that uncontrolled inter-thread interference may cause cache thrashing and cache starvation. Cache thrashing is caused by heavy data sharing among multiple processors. Different data mapped into a same location may be repeatedly referenced by different processors. The continuously swapping data may eventually evict each other from the cache [Patterson and Hennessy 2005; Stalling 2006; Seshadri et al. 2012].

Cache starvation happens when a thread consumes a lot of shared cache space thereby reducing the amount that can be used by other threads. This problem can become worse if multiple threads starve each other because one or more threads will make no forward progress. In other words, each thread will end up waiting for the thread that just starved for the shared cache to complete its tasks and progress [Schauer 2008; Chisnall 2014]. As a result, some threads may be unfairly prioritised due to some threads receiving a disproportionate share of the cache, or some threads may have to wait for a long time for the execution unit or memory to become available during runtime, while others have to wait for longer periods to access the available shared resources. Priority failures in the multiprocessor system can slow down the execution of the multiple threads, thus reducing the overall multiprocessor system performance.

To try to address the issues related to the CMP architectural configurations in optimizing the utilization of the shared memory, several studies have been done to compare the efficiency of different multiprocessor architecture parameters and shared memory models. Different cache size, number of cores, and cache sharing needs of multiprocessor systems have been proven to have different impacts on the shared cache usage in a multiprocessor system. By using a variety of applications, Pal et al. [2012] found that the number of cores that remain highly utilized in a system for a given time period, the amount of active cache usage (or number of cache block that are being used) over a specific time

interval, and the number of common cache blocks shared by multiple cores for a given time quantum, are all important parameters to evaluate in order to determine the best architectural configuration for a multiprocessor system. However, as the execution of applications moves to different phases/patterns over a specific time interval, the selection of memory model or memory organization may also impact in different ways on the system performance, energy consumption, bandwidth requirement and latency tolerance. For example, for some applications without significant data reuse, a streaming memory model has been found to be beneficial but with the use of locality-aware task scheduling, a cache-based system may achieve equal performance and be as efficient as the streaming memory system [Leverich et al. 2007]. On top of that, Mai et al. [2000] have observed that the memory structure, the interconnection and the mapping strategy should be altered to match the variety of applications' needs in order to achieve high performance and efficiency. This is supported by Chang and Sohi [2006] as they combined the strength of private and shared cache organizations to fulfil various capacity sharing points of multiple applications. Overall, a careful decision should be made in defining the best architectural configuration for a multiprocessor system.

2.2 CACHE MANAGEMENT SCHEMES

One of the key design decisions to improve overall multiprocessor system performance is to organise and manage the flexible memory hierarchy configurations efficiently. Shiue and Chakrabarti [1999] conducted research on a memory design space exploration strategy to determine an efficient on-chip data memory architecture based on performance metrics such as cache size and the number of processor cycles. In general, many studies on cache management schemes have been proposed by cache architects to cater for the cache

demands of the threads running on the system and to improve system power management. Kin et al. [1997] and Becchi et al. [2007] have implemented the use of an additional small cache in their approach, while several other researchers have reconfigured the architecture of the cache. Albonesi [1999], Powell et al. [2000], Yang et al. [2002], Zhang et al. [2003], and Kim et al. [2006] implemented a technique of disabling several segments of the cache, while Suh et al. [2004], Qureshi and Patt [2006], and Muralidhara et al. [2010] have developed methods to fairly allocate the available cache space among processors depending on certain metrics.

Su and Despain [1995] have organised the cache into several segments/banks to investigate low power cache design, naming their method "horizontal cache partitioning". Kim et al. [2006] also have structured an instruction cache into small sub-caches, which are configured as direct-mapped caches. In this kind of technique, each segment can be accessed independently and is capable of being powered up individually. On a cache request, a segment with a high probability of containing the requested information (e.g. the recently accessed line) will be accessed, while the remaining sub-caches are put into inactive mode. Thereby power is selectively consumed by the cache segments that contain the requested data during cache accesses. The amount of power saving depends on the number of cache sub-banks and the cache size. Reduction of unnecessary accesses in this technique therefore contributes to the reduction of dynamic energy consumption, as well as the size of the active cache.

Prior to the implementation of multi-level caches in multiprocessor systems, an additional very small cache had been introduced and brought in to the system architecture as a trade-off between performance, energy consumption and area efficiency. This small cache is located at the bottom of the memory hierarchy. Kin et al. [1997] introduced the use of an

extremely small “filter cache” of between 128 and 256 bytes to filter frequent cache accesses that are power costly. Meanwhile, “micro-caches” proposed by Becchi et al. [2007], which vary in size from tens to hundreds of bytes, are used to reduce the area footprint of individual cores, so that additional cores can be placed in the same given area. Relative to a traditional cache organisation, micro-caches improve the efficiency of the multiprocessor system. Filter caches and micro-caches were evaluated on a 0.8µm and 0.13µm process technology respectively. Overall, the use of extra small caches in the system has extended the hardware structure of the multilevel cache design and is targeted to eliminate potential problems that may be harmful in the higher level of the cache hierarchy.

Another approach that has been explored by cache architects is in the area of reconfigurable cache architectures. Configurable caches are beneficial in providing adequate cache space among threads to avoid under utilisation of cache resources during runtime. Unlike conventional caches, the cache can be fine-tuned at a way or set granularly depending on the threads’ or workloads’ behaviour. Albonesi [1999] introduced a selective-way cache that was implemented by Zhang et al. [2003] to selectively disable a set of associativity ways. This technique is similar to cache sub-banks, but at a finer level of granularity. Powell et al. [2000] and Yang et al. [2002] on the other hand chose to reconfigure the number of sets that can be accessed by the cache based on miss rates. The disabled ways in selective-way caches are not precharged during cache access as they have been gated beforehand. This leads to a reduction of dynamic power. During subsequent cache-intensive accesses, selective ways do not harm the cache by limiting the operation and utilisation of the cache. Zhang et al. [2003] have combined the idea of disabling cache ways with the implementation of concatenating ways in making the accessible cache sets larger.

To maximise cache performance, it is preferable to keep every portion of the cache active at runtime with none of the cache portions put into inactive mode. However, if power consumption is considered, it might be desirable to shut down some of the cache portions with a small performance loss. Even so, for a shared LLC in a multilevel cache system, each of the competing applications/threads may not need the entire shared cache space. An optimal cache management scheme will find an optimal cache partition, which is a set of cache ways to be allocated to the applications in the system that minimises the total number of misses for the simultaneous applications over a specific period (e.g. a number of clock cycles or instructions count) [Suh et al. 2004]. In addition, researchers have also tried to find the optimal partition that maximises system throughput [Qureshi and Patt 2006; Xie and Loh 2010] or the optimal partition that provides fairness between applications [Kim et al. 2004]. As a result, some effort has been made to manage the cache in a thread-wise manner with regard to fully utilising the capacity of LLC and to reduce the average access latency. Each running thread in the system would be statically allocated with a same size of cache space prior to runtime, thereby minimising the potential of wasting any cache resources at runtime. In a scenario when the cache access pattern is changing, the thread may suffer from insufficient allocated cache space. Therefore, it is beneficial to continually monitor the cache usage and make cache allocation decisions dynamically based on the execution behaviour of each concurrently running thread.

In an attempt to optimally partition the cache, Suh et al. [2004] used the number of misses to estimate the performance gain or loss of competing threads in different cache allocations. Multiple marginal gain counters recorded the number of cache hits in the previous time period and were used to determine the cache allocation of each thread for the next time period. The partition sizes depend on the total number of misses that can be saved whenever one or more cache blocks is dedicated or taken away over a time interval. In

contrast, Qureshi and Patt [2006] developed a low overhead monitoring circuit to track the cache utility information. Utility is defined as the benefit or the number of additional hits that can be obtained in response to the increasing number of cache ways allocated to an application. Since the cache resource is partitioned on a utility basis, the use of Stack Distance Counters is very important to record the cache hits of the competing applications according to how recently they were accessed. Stack distance consists of addresses belonging to a sequence of cache accesses, with the most recently accessed address stored at the top position. The number of counters are kept equal to the number of cache ways that are associated with the stack distance positions. If an access results in a cache hit, the counter that corresponds to the accessed address is incremented. Hence, the counters are used to predict the number of additional hits and the reduction of misses when the number of ways allocated to a thread is increased. This is as opposed to the least recently used (LRU) scheme, in which cache space is distributed among threads in proportion to the cache utility rather than to cache demand.

For distribution on a cache demand basis, the number of unique cache blocks accessed in a given execution interval is used to determine the cache allocation [Qureshi and Patt 2006; Moreto et al. 2007; Jaleel et al. 2008]. Consider two applications A and B sharing N cache ways (assuming a fully associative cache). Suppose it were observed that A and B have accessed N_A unique blocks and N_B unique blocks, respectively, in the last N unique accesses to the cache. The LRU scheme will distribute the cache resource based on demand, which is by allocating N_A ways to application A and N_B ways to application B. However, for distribution on a cache utility basis, the number of misses an application incurs when it receives α and β cache ways will be used to determine the cache allocation for that application for the next execution intervals [Qureshi and Patt 2006; Moreto et al. 2007; Jaleel et al. 2008; Xie and Loh 2009; Kedzierski et al. 2010b]. This means, if $MISS_\alpha$ and

$MISS_{\beta}$ are the number of misses incurred by application A when it is allocated with α and β ways (α is less than β), the benefit or utility that A will receive due to increasing the number of cache ways from α to β is $MISS_{\alpha} - MISS_{\beta}$. The estimated utility of application A will be compared with the estimated utility of application B to determine the target partitions of both applications for the next execution interval. Note that, an application with a higher estimated utility will be allocated with more cache space compared to an application with a lower estimated utility. Overall, these proposals tried to partition the cache in a manner that minimises the total misses and maximises the total hits incurred by all applications.

The variation of execution behaviour of a single multithreaded application motivated Muralidhara et al. [2010] to distribute cache resources and allocate more cache ways to the slowest thread (e.g. the thread with the highest cycles per instruction (CPI)). They aimed to minimise the slack time among individual threads at every time interval and dedicate large portions of cache to the threads with the slowest performance. Their intra-application cache partitioning strategy used a dynamic curve fitting based cache partitioning scheme to distribute the cache space based on the CPI of the individual thread. By speeding up the slowest thread, the overall performance enhancement can be achieved in the application execution, ideally without slowing down the faster threads.

Although various proposed techniques have yielded several insights in the area of shared cache management, the cache partitioning approach has become the great interest of the work presented in this thesis. Some of the proposed techniques were found to be very complicated as well as having a significant hardware overhead [Kim et al. 2006; Qureshi and Patt 2006; Chang and Sohi 2007; Moreto et al. 2007; Kedzierski et al. 2010b; Sharifi et al. 2012]. Meanwhile, some others that are less complicated were not found to be good if evaluated in system configurations different from those in which they were originally

simulated and assessed (e.g. number of cache levels or different cache replacement policies) for assorted application benchmarks (e.g. single-threaded or multi-threaded applications) [Chang and Sohi 2007; Moreto et al. 2007; Jaleel et al. 2008; Xie and Loh 2009; Kedzierski et al. 2010b; Muralidhara et al. 2010; Sharifi et al. 2012]. To optimise the proposed techniques for producing better results, these limitations were taken into consideration and several partitioning approaches were revisited, with an aim to improve performance with more innovative strategies.

The work in this thesis was mainly influenced by the proposed cache partitioning techniques because the cache is guaranteed to be utilised at all times and each competing application can be made to perform better by allocating adequate cache resources. Most of the existing techniques employed various difficult performance metrics in partitioning decisions, such as demand-based or throughput-oriented. It is a challenge to determine the best metrics used in the cache partitioning algorithm. To guarantee fair speed-up among applications in multi-application workloads, the concept of a dynamic curve fitting based cache partitioning scheme introduced by Muralidhara et al. [2010] is used as a reference scheme in this thesis. As compared to the use of miss rate for evaluating the application's performance, which does not account for the number of memory accesses made by an application, this scheme uses CPI values to identify and speed up the slowest application, and to compute the target partitions. Meanwhile, the utility cache partitioning (UCP) scheme proposed by Qureshi and Patt [2007] is used for comparison and further discussed and demonstrated in Chapter 4 because their technique is widely accepted and used as a reference scheme for the evaluation of cache partitioning schemes [Xie and Loh 2008; Xie and Loh 2009; Kędzierski et al. 2010b; Li et al. 2011]. Therefore, these two techniques were evaluated and used as a guide for a new cache partitioning scheme.

2.3 Cache Replacement Policy

As described earlier, the allocation of cache resources among threads may be ineffective at runtime, causing consistency for significant speed-up to the system cannot be assured [Xie and Loh 2009; Muralidhara et al. 2010]. To address this problem, a new block/line replacement strategy has been investigated to better utilise the dedicated cache resources. There have been several research efforts that modify the cache replacement policy in order to retain some fractions of the working set in the cache longer, so that various types of cache references can be exploited appropriately and contribute more cache hits [Khan and Jiménez 2010].

A cache replacement scheme consists of an eviction/victim, insertion and promotion policy. In the *eviction/victim policy*, each cache line is given an eviction priority that is used to determine the line to be replaced in the selected set whenever a miss occurs and a new line has to be brought in. It is not necessary to install the new cache line at the position occupied previously. The *insertion policy* specifies the initial position of the requested line in the eviction priority. On every cache hit, the eviction priority is updated according to the *promotion policy* [Xie and Loh 2009].

Several widely used cache replacement schemes such as the least recently used (LRU) policy, first-in first-out (FIFO) policy, least frequently used (LFU) policy and most recently used (MRU) policy have been evaluated and refined over time to improve the overall cache utilisation and data locality in the multiprocessor system. The LRU policy is the main focus of this thesis and while it has several limitations, it remains the de-facto standard replacement policy in the shared cache. On a cache miss, the LRU policy inserts a new line in the highest priority position (usually referred to as the MRU position). One of the common drawbacks of the LRU policy is that lines accessed only once and never accessed

again are kept unused in the cache for the longest time [Qureshi et al. 2007; Xie and Loh 2009]. Furthermore, LRU replacement is unable to avoid early eviction because the lines that are located in the lowest priority position (known as the LRU position) are always selected as victims to be replaced. This is inefficient for the lines, which are re-referenced immediately after eviction. In the case of cache lines that are accessed more frequently while in the MRU position but not receiving any hits when traversing to the lower priority positions, the LRU policy results in poor cache space utilisation. The lines are kept longer in the cache than they are needed, but if they are not going to be referenced after leaving the MRU position they should be replaced [Khan and Jiménez 2010]. Such a scenario is ideal for the lines that exhibit poor locality.

There were several modifications made to the replacement policies to minimise the amount of time that zero-reuse lines spend in the cache [Kampe et al. 2004; Qureshi et al. 2007; Xie et al. 2009]. The lines that are re-referenced at a greater distance than the available cache size are preserved long enough before they are evicted. In other words, these lines are allowed to reside in the cache as long as possible until their next reference occurs to incur additional cache hits. This can be done by managing the movements of the lines in the cache recency stack. The Dynamic Insertion Policy (DIP) proposed by Qureshi et al. [2007] is focused on protecting the cache from thrashing by controlling the insertion rule of a new incoming line in the cache. Their observation was that the traditional LRU policy wasted cache resources because more than 60% of the cache lines fetched are never re-accessed before they are evicted. The DIP allowed a choice between two policies, namely the LRU Insertion Policy (LIP) and the Bimodal Policy (BIP). The LIP inserts the new cache line in the LRU position increasing the probability for zero-reuse lines to be evicted immediately after they are accessed. For multiple-reuse lines, LIP maintains the lines in the cache long enough for their next reuse. This is done by promoting the hit lines in the LRU

position to a higher priority position, allowing the lines to be re-referenced while traversing from MRU to LRU position. However, LIP was found to be inadequate for lines with greater distance of reference. This usually happens when the working set is larger than the available cache space. Extra misses are very likely to occur on their next accesses. BIP on the other hand was introduced to install a small percentage of new cache lines in the MRU position at a certain regularity (with a low probability). It is not always true that an application will receive a performance benefit through BIP, especially for applications with a LRU friendly access pattern. Thus, DIP dynamically chooses between LRU and BIP, selecting the policy that performs better and incurs fewer misses to benefit and manifest more hits in the cache.

As DIP was originally designed and evaluated for a single-threaded application in a uniprocessor system, Jaleel et al. [2008] have extended it for a multi-threaded system with a shared cache. Their Thread-Aware Dynamic Insertion Policy (TADIP) aimed to dynamically select the best policy (LRU or BIP) for an application. Due to diversity in the cache behaviour and working set size, extra attention is given to the cache demand of the multiple concurrently running applications. For that reason, some applications may show performance gain when using LRU policy, while others may get benefit from the BIP policy.

The high frequency of subsequent accesses by an application force cache lines of other applications to occupy lower priority positions that shortens their lifetime. This is because on every cache hit, both the traditional LRU policy and the proposed BIP promote all hit lines in the cache to the MRU position. However, the difference between the two policies is in making the promotion decision that is based on the location at which the cache hits occurred. For the BIP policy, the hit lines are promoted to the MRU position only if the

cache hits have occurred in the LRU position. Meanwhile, the LRU policy always promotes the hit lines to the MRU position, even though the cache hits happened in non-LRU positions. Therefore, it appears that the promotion distance for BIP is greater than in the LRU policy.

Kron et al. [2008] proposed a dynamic promotion policy (DPP) implemented in their Double-DIP technique. They attempted to fairly treat the cache lines of all cores by leaving them in the sets long enough to demonstrate cache hits in their next references before being evicted. On a cache hit, DPP incrementally moves a line towards the MRU by a specific number of priority positions to give a cache line with a low access frequency extra time to spend in the set but not so long that the line becomes an evictee. Comparing with DIP and TADIP, two potential promotion policies in Double-DIP are traditional MRU promotion policy (MPP) and the proposed Single-Step Incremental Promotion Policy (SIPP). Together, the implementation of both promotion policies is defined as DPP. The SIPP was introduced to promote the hit lines by only a single priority position so that the amount of time the dead lines spend in the cache can be minimised. However, in some cases, SIPP is inefficient when direct promotion to MRU is selected and happens frequently, thereby causing unsolved cache contention. As a result, DPP was improved with variable promotion incrementation selections, i.e. up to $+n$ positions (where n is the set-associativity). In particular, Double-DIP augmented the original DIP to exploit DPP, which is invoked on a cache hit of an already cached line.

A similar approach was implemented by Xie and Loh [2009] in their proposed Promotion/Insertion Pseudo-Partitioning (PIPP). They combined both dynamic insertion and promotion policies, but with an intention to implicitly partition the shared cache resources among processors. The objective of PIPP was to overcome underutilisation of allocated

cache resources among the processors. Nevertheless, PIPP attempted to locate a heap of cache lines at lower priority positions in order to reduce their residency period in the set, likely for zero/less reuse lines. Initially, PIPP used the Utility Monitor proposed by Qureshi and Patt [2006] to compute the target partitions of the cores. For n cores, the number of ways allocated to each core is $\sum_{i=1}^{n} \pi_i = \omega$, where ω is the total set-associativity. The installation position of a new line from a core is based on the partitioning allocation, i.e. core_i always installs a new line in position π_i. PIPP promotes a hit line by a single priority position towards the MRU position and, as described earlier, this approach, as opposed to Double-DIP, tends to keep lines at a lower priority position, close to the eviction position. For the lines with single-reuse, occupying priority positions near to the end of ordering leads to faster eviction. In PIPP, the LRU position is the final pit-stop for any cache line before it is evicted.

Sui et al. [2010] and Wu et al. [2010] augmented TADIP with their Thread-Aware Dynamic Promotion Policy (TADPP). TADIP is used to handle insertion policy selection, while TADPP promotes a hit line either by a single priority position, or directly to the MRU position. The appropriate promotion policy is chosen based on the best policies currently exploited by the remaining applications. Since the shared cache is partitioned using UCP's Utility Monitor, victim selection is made according to the number of lines dedicated to the miss-causing application. Therefore, TADPP is different from Double-DIP in the way it attempts to select the evictee.

With the deployment of partitioning schemes on the shared cache, the rules of the replacement policy require extra attention. Researchers have revised and refined traditional replacement policies such as the LRU policy by performing simple to more complex modifications. They attempted to enhance components of the replacement policy to

guarantee full utilisation of the shared cache. Suh et al. [2004] and Qureshi et al. [2006] extended the generic LRU policy to work appropriately in their partitioned shared cache. Considering that different numbers of ways are allocated among processors, an extra bit is added to the tag of each line representing the ID of the processor that installed the line in the cache. Each processor owns a counter that keeps track of the number of lines in the set that belongs to the processor and updates on every cache miss. Whenever a new line is brought into the cache, the counter is incremented; when a line is taken away from the cache, the counter is decremented. The decision about which line is to be a victim is based on the number of lines allocated to the miss-causing application. If the number of lines belonging to the application is smaller than or equal to the number of allocated lines, the victim to be replaced is the LRU line belonging to the application. Otherwise, Qureshi et al. [2006], Sui et al. [2010] and Wu et al. [2010] in their TADPP chose the LRU line among all lines in the set that do not belong to the miss-causing application to be the evictee. Suh et al. [2004] on the other hand selected the LRU line of another application with over-allocated lines to be replaced.

The use of the LRU replacement policy in a shared cache could result in low overall performance since an application may over-utilise the cache space with its own data, while other applications may underutilise the shared cache and generate higher miss rates [Muralidhara et al. 2010; Sharifi et al. 2012]. This scenario could happen in both an unpartitioned and a partitioned shared cache, so managing the shared cache space among multiple applications is important to guarantee better cache utilisation. Generally, the goal of partitioning a shared cache is to provide large enough cache space to each application so that it could hold all the application's working set. The goal is that the allocated cache space could minimise the total number of misses of the simultaneously executing applications. The allocated space may be adequate for each of the applications, but fail to achieve better

overall performance, due to cache misses that are incurred because of the changes in the cache access patterns and changes in the applications' execution behavior at runtime [Al-Zoubi et al. 2004; Moreto et al. 2007]. Even though various methods proposed by many researchers have allocated cache space for each application by monitoring their runtime performance, there is still more room to improve the number of cache misses. This is because different applications give different reactions to the allocated cache space along with the different applications' attributes used to partition the shared cache [Jaleel et al. 2008; Wu et al. 2010]. To address limitations of under-utilised cache allocation by the cache partitioning, insertion, promotion and eviction policies are investigated in this thesis.

Zero-reuse lines, single-reuse lines and subsequent reuse lines have been found to be the contributors to cache misses in partitioned cache systems. This problem needs to be resolved because the longer these lines occupy the cache, the more misses will be incurred. It is identified that the cache replacement policy is one of the approaches that can be used to overcome this issue. The insertion, promotion, and eviction strategies of a replacement policy have a great impact on cache line management. For the zero-reuse lines, the strategy should be to insert the lines at the lower (or lowest) priority position so that they can be evicted right after they are used [Qureshi et al. 2007; Khan and Jiménez 2010]. This ensures that another line can be inserted into the cache to allow additional cache hits. For the single-reuse lines and the subsequent reuse lines, better insertion and promotion strategies are important to give the lines a longer time to occupy the cache before their final reuse and speedy eviction. This could provide the lines more chances to incur more cache hits before being evicted [Kron et al. 2008; Khan and Jiménez 2010]. Therefore, the insertion and promotion policy in PIPP proposed by Xie and Loh [2009] is the primary replacement strategy explored in the work of this thesis. The dynamic insertion policy of PIPP that is not too near the highest or lowest priority positions seems favorable for zero-reuse lines to

traverse to the lowest priority position and be quickly evicted. The promotion by a single priority position of PIPP benefits the single-reuse lines and the lines with a burst of references because they occupy the cache longer before they become dead and promptly become a potential victim when ready to be replaced. The strengths and weaknesses of the insertion and promotion policies of PIPP are studied and augmented to produce a better replacement policy that is effective in minimising the number of cache misses incurred by the zero-reuse lines, the single-reuse lines and the multiple-reuse lines in a partitioned shared cache.

To eliminate the cache misses incurred in the partitioned shared cache as much as possible, the eviction policy is adopted from the approach used by Suh et al. [2004] and Qureshi et al. [2006]. The eviction policy is founded on an application-ownership basis. The evictee to be replaced upon a cache miss is chosen among the lines that are belonging to the miss-causing application. This strategy could also guarantee that the shared cache will not be polluted with the data of one particular application because of uncontrolled cache allocation stealing among applications and degrading overall performance. The eviction policy is studied, revised and implemented with additional information gathered from the cache partitioning engine. A detailed description of the method is provided in Chapter 5.

2.4 SUMMARY

With the increasing number of processors in a system, memory design space exploration has been an important area of investigation in research efforts to improve system performance. This is due to the increasing competition among processors for the on-chip memory resources, which has become a challenging and crucial issue that must be resolved by

system designers. To ensure that the performance of the system is improved, several existing techniques that are proposed to reduce or overcome conflicts incurred in the system were reviewed in this chapter.

A variety of previous proposals that belong under two cache performance improvement categories have been presented: namely partitioning the on-chip shared cache in multiprocessor systems, and on improving the traditional cache replacement rules. Several cache partitioning strategies and adaptive cache replacement policies have been selected as the main references points for this thesis, which aims to improve multiprocessor system performance at low hardware cost. Thus, the selected references will be analysed and further discussed in later chapters.

CHAPTER 3

SIMULATION FRAMEWORK

This chapter describes the baseline system architecture, toolset and benchmarks that form the basis of the simulation techniques employed in the work presented in this thesis. The evaluation and observation of the system is described in the subsequent chapters.

3.1 THE XTENSA LX4.0 PROCESSOR

The Xtensa LX4.0 processor core and Xtensa Xplorer RD2011.2 development environment are the main system-simulation components in this work. They were developed by Tensilica Inc.[1] and provide a simulation environment with several development tools that enable designers to configure, customise and develop the Xtensa LX4.0 processor cores. The Tensilica Xtensa Xplorer RD2011.2 is an integrated development environment usable as an interface to the Tensilica development tools, which assists designers by giving more options to create and build a complicated system. The Tensilica Xtensa design tool provides a cycle accurate system simulator that allows designers to perform software emulation of systems containing a number of interconnected components. The components include Xtensa processors, memories and interconnecting devices. Under the Tensilica simulation

[1] Tensilica has been a part of Cadence Design Systems since April 2013 [Cadence Design Systems 2013].

environment, these components are configurable and extensible, thereby allowing the creation and activation of local processor memories, system memories and device-to-device connectors.

The Tensilica simulator is similar to the widely-used GEM5 simulator [Butko et al. 2012] in that both are cycle-accurate simulators, but Tensilica offers some advantages due to its reconfigurable and extensible processor model. Moreover, when compared with the Simics simulator [Magnusson et al. 2002], Tensilica is preferable as Simics is not claimed to be cycle-accurate simulator but functional-accurate simulator that requires a commercial license. Hence, the Tensilica Xtensa design tool can be an attractive option for those who would like to experiment a system with a cycle-accurate simulator and to potentially take advantage of the configurable and extensible processor model. This is to allow designers to precisely create and configure each processor to match its assigned tasks or target applications requirements. Additionally, the Tensilica Xtensa design tool permits designers to define custom devices according to the available application programming interface in order to meet their system requirements and design. Figure 3.1 illustrates the Xtensa configurable processor architecture.

The Tensilica processors are commercially used in several embedded CMP designs including the 188-core Cisco Metro router chip. A number of researchers have also used the Tensilica Xtensa design tool and the Xtensa LX core in their CMP designs. Unfortunately, not many of them used processor interface ports to interconnect the Xtensa processors and the memories in the system as they were focused on a streaming memory model, which uses message-passing communications. The streaming memory model allows a part of the on-chip storage to be organised as an independently addressable structure, while the on-chip storage in a cache-based memory model is used for private and shared caches that need to

always be coherent during runtime [Leverich et al. 2007; Leverich et al. 2008]. In addition to the configurable and extensible features of the Xtensa processor, it became our interest to make use of the processor interface ports in organising the cache-based memory model of our CMP system. Throughout this thesis, the Xtensa LX4.0 processor core is referred as the Xtensa core.

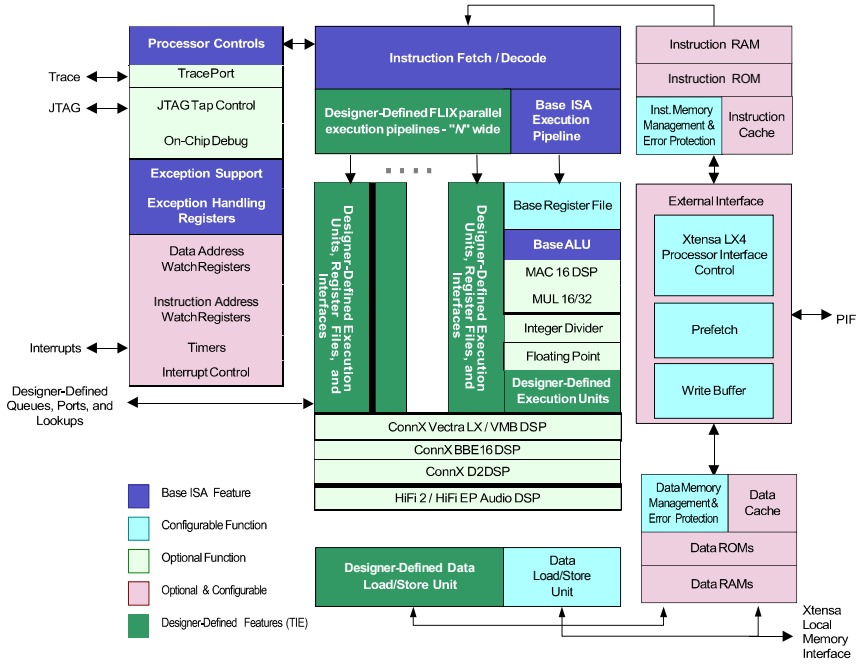


FIGURE 3.1: Xtensa LX4.0 processor architectural block diagram [Tensilica Inc. 2011].

The Xtensa core has special attributes that differentiate it from commercial processors developed by other companies. The core is configurable, extensible, low power, small in size and fully synthesizable. Tensilica provides a wide range of configuration options to instantiate the processor core and the Xtensa simulator during the modeling process. As reported by Ezer [2000], designers can extend and enhance the Xtensa base Instruction Set

Architecture (ISA) and its coprocessors with new instructions and registers, execution units, and I/O ports through the Tensilica Instruction Extension (TIE) language, a high level and easy language to program. Generally, the Xtensa core is configured with two types of interface, namely *queues* and *ports*. They are used to interconnect local memories with other peripherals in the system; these could be external devices, memories or cores. Figure 3.2 shows the Xtensa core interfaces and local memories, as well as system components that can be connected to each interface. TIE *queues* as described in Shee et al. [2006] are used for incoming and outgoing data with the implementation of *push* and *pop* functions. Patel et al. [2007, 2008], Shee et al. [2006, 2007] and Becchi et al. [2007] have used TIE *queues* to connect cores and external devices in their multiprocessor system, while FIFO instructions were used for communication between the individual processors. In contrast, *ports* in the Xtensa toolset are wires used to send and retrieve external signals from and into the Xtensa core.

The *queue* interface width is independent of the width of any processor memory or peripheral interface attached to the *queue*. The width is limited to a maximum of 1024 bits. The *queue* interface has been designed so that only a synchronous FIFO type device can be connected to this interface. It is synchronous in the sense that the data transfer takes place at the rising edge of the processor clock. Note that the Xtensa processor and bus clocks are synchronous and at the same frequency. Hence, the TIE *queue* provides synchronous access to the external data and will stall the Xtensa processor if no data is available or can be written in response to the corresponding request it has received. Xtensa *ports* are the local processor interfaces designed with multiple width selections. While the width of all the Xtensa *ports* can be configured to 32-, 64- or 128-bits wide (including local cache *port* and processor interface), the width of some of the *ports* can also be configured to 256- or 512-bits wide (i.e. RAM and ROM *ports*). The *ports* base latency is one- or two-cycle access

latency, respectively, for five stage and seven stage pipelines. However, with the use of available Xtensa functions, system designers can add more delays to the base latency.

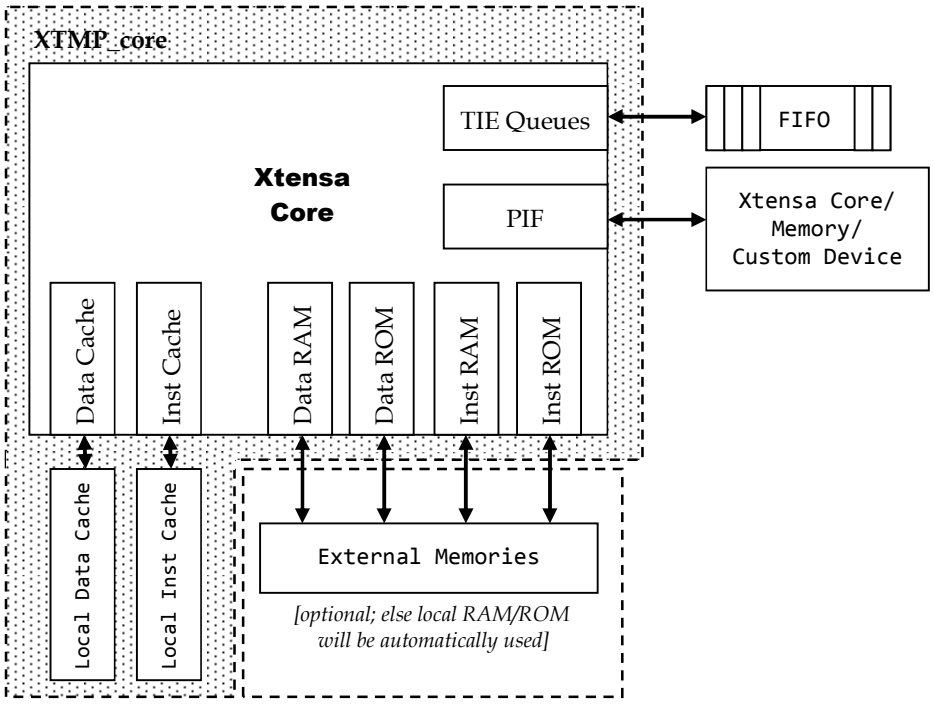


FIGURE 3.2: Xtensa processor interfaces.

The instruction set simulator (ISS) application programming interface (API) for multiple Xtensa core systems called the Xtensa Modeling Protocol (XTMP), is written in C. It is a cycle-accurate simulation environment that allows designers to use the TIE language to enhance the Xtensa core and to write a customised multiple Xtensa core or multithread simulator in the process of modeling a complicated system.

The architecture of the Xtensa core consists of a sequential pipeline of either five or seven stages. It is a high-performance 32-bit RISC processor and can be configured as a customised single core or a multiple core using a generator tool running remotely on

Tensilica's server. Named the Xtensa Processor Generator (XPG), XPG is used to select and implement many processor features such as memory hierarchy, processor interface size, write-buffer size, on-chip RAM and ROM size, as well as cache configurations and attributes to match the target application. Once the core is created and built with all the additional features, XPG will create a parameter file containing the description of the customised Xtensa core. This file will be used to create an XTMP_core object, which is the model of a single Xtensa core in a system simulator program. An XTMP_params object that describes the core configuration must first be created by using the available parameter file. Once the XTMP_params object is created, it will be used in instantiating the single Xtensa core as a XTMP_core object. These are the first steps that need to be performed in a XTMP simulation program before the creation of any system memory and interconnections among components in the system.

In the Xtensa LX4.0 databook, there are two versions of Xtensa LX4.0 core manufactured with a 65nm process [Tensilica Inc. 2011]. The cores are manufactured with either a generic process or a low power process. The area size for an Xtensa LX4.0 core manufactured for the 65nm generic process could be approximately between 0.044mm² and 0.078mm², while for the low power process the area is roughly 0.039 to 0.071mm². The power dissipated by the Xtensa LX4.0 core for both processes are comparable with the generic process version dissipating around 16 to 31µW/MHz dynamic power and 579.4 to 872.7µW leakage power, whereas low power processes dissipates dynamic power at approximately 21 to 38µW/MHz and 5.6 to 8.6µW leakage power[2]. The clock speed for the base configuration of the generic process is 816MHz and 448MHz for the low power process. Note that at the operating frequencies of interest, the dynamic power is much

[2] The core area size and power is obtained with the XPG set for synthesis prioritization to optimize area and frequency at 50MHz [Tensilica Inc. 2011].

greater than the leakage power.

3.2 Simulation Framework and Environment

The architecture of the multi-core system used in this project is illustrated in Figure 3.3 and is a typical CMP platform with private level 1 (L1) caches for each core and a shared level 2 (L2) cache [Al-Zoubi et al. 2004; Suh et al. 2004; Qureshi and Patt 2006; Chang and Sohi 2007; Qureshi et al. 2007; Jaleel et al. 2008; Kron et al. 2008; Xie and Loh 2009; Kędzierski et al. 2010b; Khan and Jiménez 2010; Wu et al. 2010; Wang et al. 2011; Jia et al. 2012]. A number of cores, *N* are configured and built using XPG, as discussed previously. The cores are configured without local/private data and instruction caches due to the following limitations. The local caches cannot be arranged in a hierarchy and they cannot be shared by more than one device. The maximum configurable size for the local caches is 256Kbytes. The use of local cache to interconnect external blocks in the system is not preferable as it constrains the address space cacheable for external memories to 256Kbytes and the configuration of the cache hierarchy. This implication has been discussed by Becchi et al. [2007].

The cores used in this work are configured with the Processor Interface (PIF). PIF is an interface with read and write busses of equal width, that supports external communication and is used for inter-core communications in an Xtensa multi-core system. In addition, PIF does not limit the size of external peripherals that can be connected to the core. The configurable features used on top of the base ISA of the Xtensa core are shown in Table 3.1. It is retrieved from the core description file created by XPG.

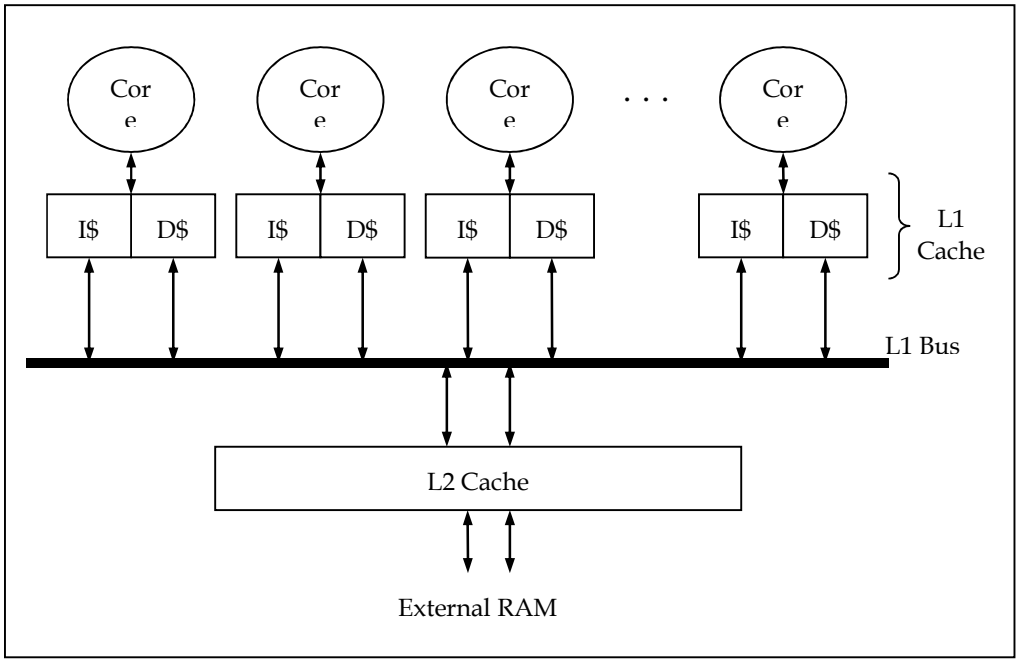


FIGURE 3.3: The baseline architecture.

TABLE 3.1: Xtensa LX4.0 processor configuration.

Speed	392 MHz
Process	45nm GS
Core size	0.556 mm²
Core power	26.72 mW
Pipeline length	5
Count of Load/Store units	1
Enable Density Instruction	√
Enable Boolean Registers	√
Enable Processor ID	√
Thread Pointer	√
Enable Processor Interface (PIF)	√
Write buffer entries	4
Width of PIF interface	32
Memory Protection/MMU	Region Protection
Maximum Instruction Width	3 bytes

The last level cache is the main focus in this work. The caches in the system are arranged in a hierarchy that consists of private level one caches belonging to each core, as well as a unified level two shared cache. All memories in the system are created as custom memory-mapped devices using XTMP_device objects. In XTMP, a memory-mapped device refers to any component in the system that is accessed by load or store transactions from a simulated Xtensa core. This means that the XTMP_device uses a transaction-oriented interface. When a load or store request is issued by an Xtensa core, the information about the transaction (i.e. a transfer record (XTMP_deviceXfer)) will be processed by the callback function that is associated with the XTMP_device so that a response can be sent back to the Xtensa core. A detailed explanation about the transaction and the callback function of the XTMP device is provided below.

Note that all load and store requests from the Xtensa cores in a multi-core system are issued to the XTMP_device objects that are attached to the PIF. The level one caches in the system are interconnected to the cores via the PIF and the level two cache bus and memory bus are implemented to model contention among multiple cores. The width of the shared level two cache and the main memory busses have been configured to be equal to the PIF width. This is because all the requests that are not responded to by the level one caches will be forwarded to the shared level two cache that is directly connected to the main memory. Figure 3.4 shows the interconnection of the Xtensa cores and the cache hierarchy in the XTMP system simulation, while Table 3.2 lists all of the parameters of the system baseline configuration. The size of level one caches used in this work is smaller than the size usually used in a CMP for general-purpose applications, which is typically a 32KB L1 cache. This is because the small L1 cache accentuates the dependency on L2 cache, hence the effects of the proposed cache policy changes can be easily observed. However, note that the benefits gained by the L2 cache in this case may be reduced in practice.

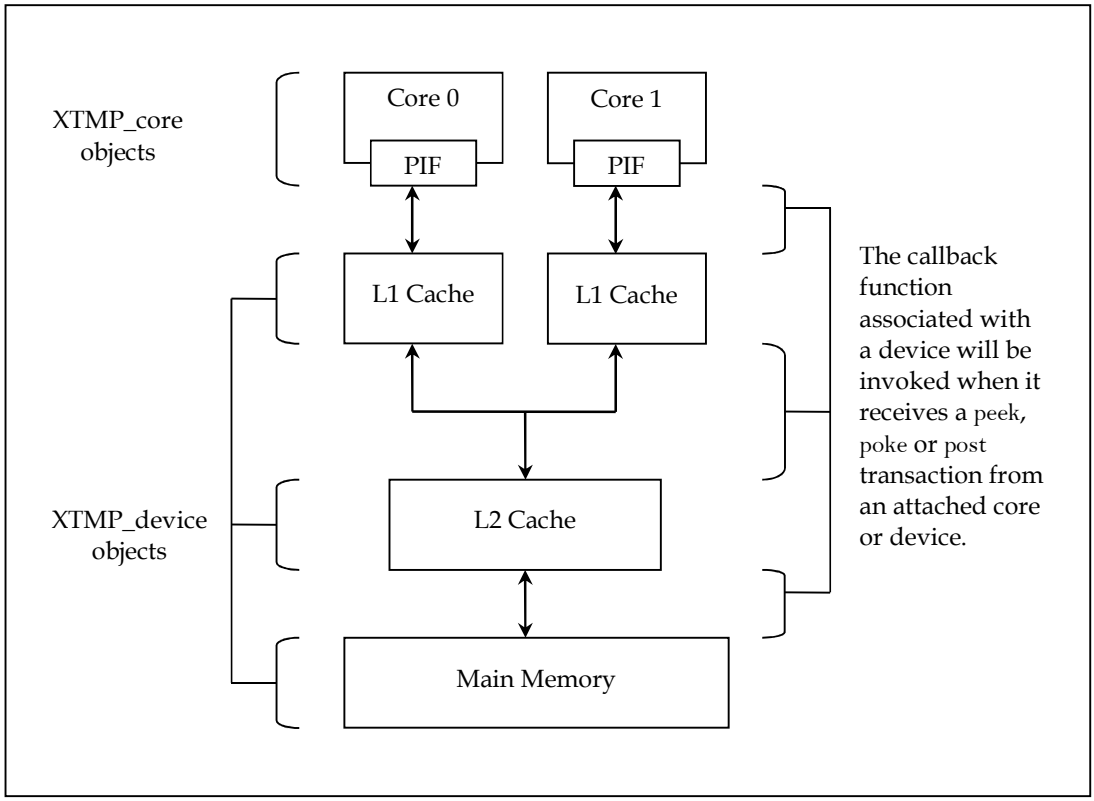


FIGURE 3.4: The interconnection of the cores and cache hierarchy using the PIF in a dual-core system.

TABLE 3.2: Baseline system's configuration.

Processor Core	Xtensa Tensilica LX4.0, 32-bit PIF width
L1 Caches	Private 4KB, 16B line-size, 4-way, LRU, write-back
L2 Cache	Shared 512KB, 16B line-size, 32-way, LRU, write-back, 15 cycles hit latency [Wu et al. 2010]
Memory	300 cycles access latency [Wu et al. 2010]

The XTMP environment and Xtensa simulator are used to create, debug, profile and verify the software model of the baseline architecture. In XTMP, system components including the core, its configuration and memories must be created before any interconnection can be made. By using several XTMP functions, the Xtensa simulator is

customised so that all the system components can be controlled. This means that the state of the system components during runtime can be examined and modified to meet the goal and requirements of the simulated system. For example, the state of the available Xtensa core registers can be read, written, saved and restored at runtime. System designers can access the system memories during core execution so that the read and write transactions can be performed and the delay parameters of the memories can be specified. Furthermore, the customised Xtensa simulator is able to ensure that the system simulations can be run appropriately by ensuring each thread executes on the assigned core during simulation. All of the threads are directly loaded into the simulated processors and memories by calling XTMP_loadProgram(). In this thesis, the tasks loaded into the Xtensa cores are grouped for comparison and evaluation of the overall system performance. Note that, the tasks here are all single-threaded applications.

Since all the memories are created as XTMP_device objects, the transaction-oriented interfaces and the peek, poke and post transactions are used to communicate among cores and memories in the system. The peek(), poke(), and post() callback functions associated with the device attached to the core PIF will be invoked to respond to all load or store requests issued by the corresponding core. Note that the peek() and poke() callback functions will accept peek and poke transactions respectively. They are initiated by the simulator in order to examine and modify memory mapped to the device. The post() callback function is used to model the hardware transactions caused by the load and store requests of the Xtensa core. This means that the post transaction contains the detailed information of the core transaction. The information includes whether the transaction is a request or a response, the type of memory access, as well as the transaction size, which later will be used in the peek and poke transactions so that appropriate actions or responses can be performed by the corresponding peek() and poke() callback functions.

In our cycle-accurate simulations, a ticker function that is invoked once every cycle is used to examine pending transfer records to custom devices/memories, to send responses for transactions that have matured, as well as to terminate the system when all cores/threads have exited. This ticker function performs all its work in a loop and calls XTMP_wait(1) at the end of each iteration. The ticker function is also used in this project to simplify the management of multiple requests or responses to the last level shared cache in which contention may happen. As all custom devices or memories in our simulated system are connected (directly or indirectly) to the cores via the PIF, requested data cannot be returned immediately because the PIF transaction delay must be at least one-cycle. Therefore, whenever a new request arrives at the custom devices including the shared L2 cache, the device firstly will check if there is a transaction already being processed. If that is the case, the request will be rejected by responding to the requester with the XTMP_NACC status. Otherwise, the new request will be accepted, and the time in which the response will be sent is computed based on the current time and the delays that we have set for the respective device. Note that the XTMP_NACC status models a busy signal that indicates the device is not ready to accept any transactions [Tensilica Inc. 2010, 2011].

In general, the architecture of the multi-core system used in this project is similar to other proposed systems in previous research which has used CMP configurations. The system architecture used in Muralidhara et al. [2010] was developed by Sun. It has 4 cores with an on-chip shared L2 cache and each core maintains local L1 instruction and data cache. According to Xie et al. [2009], the architecture they used consists of 2 and 4 cores. Each core has level one instruction and data caches, as well as L2 cache as the last level cache shared by all the cores. Qureshi et al. [2006] also has modeled a system with two processors. Each processor has private instruction and data caches and a shared L2 cache connected through a bus. We have configured our baseline architecture and enhancements

similarly to these systems to ensure that our results are comparable to this prior research.

3.3 SPEC2000 BENCHMARKS

We use the SPEC2000 Benchmark suite to evaluate the performance of our cache architecture enhancements. The SPEC CPU2000 suite is intended to evaluate compute intensive performance and it emphasises the performance of the CPU, memory architecture and the compilers. It has been used by many other researchers to evaluate the performance of cache systems and other aspects of computer architecture. Note that, we started the work presented in this thesis in 2008, when there was not much prior work reporting the current SPEC CPU2006 results. Thus, we decided to continue using SPEC CPU2000 for comparability until our work was completed. In our simulations, performance evaluation of different cache partitioning and replacement schemes has been undertaken using sets of applications that were selected from the SPEC CPU2000 benchmark suite. The chosen applications are the combinations of three different applications categories, which are classified as proposed by Qureshi et al. [2006]. The three categories are based on the demand for L2 cache space, and are named *high utility*, *low utility* and *saturating utility*.

Applications that do not benefit from the allocation of any additional cache space are classified as *low utility* (LU). The argument could be that the applications fit in the L1 cache and suffer frequent compulsory misses in the L2 cache. *High utility* (HU) applications improve their performance whenever more L2 cache space is allocated. If increasing the available cache space can improve the performance of applications up to a point at which their performance saturates, these applications were classified as *saturating utility* (SU).

Note that the Tensilica Xtensa development tool includes the Xtensa C/C++ compiler (XCC), which supports compiling C and C++ applications on Tensilica processors. Only applications from the SPEC CPU2000 benchmark suite that are written in C were selected to be used for evaluation purposes. All the applications were compiled with Tensilica Xtensa's optimising compiler xt-xcc at the -O3 optimisation level. The classification of each application is shown in Table 3.3 and applications from different categories were combined to represent eleven workload mixes, as depicted in Table 3.4. We selected application(s) from each category to represent workload mixes of 4HU, 3HU 1LU, 3HU 1SU, 2HU 2LU, and 2HU 1LU 1SU.

TABLE 3.3: Classification of SPEC CPU2000 applications.

High utility	twolf, vpr, parser, art, ammp
Low utility	mcf, bzip2, equake
Saturating utility	gzip, mgrid, mesa

TABLE 3.4: Workload summary.

Workload	**Applications**
Mix-1	art, bzip2, parser, vpr
Mix-2	vpr, parser, mcf, bzip2
Mix-3	ammp, twolf, vpr, art
Mix-4	equake, bzip2, twolf, vpr
Mix-5	bzip2, ammp, vpr, art
Mix-6	mcf, vpr, twolf, gzip
Mix-7	parser, mesa, vpr, art
Mix-8	twolf, art, bzip2, mgrid
Mix-9	art, vpr, twolf, equake
Mix-10	equake, bzip2, art, vpr
Mix-11	ammp, vpr, twolf, bzip2

The following are brief descriptions of the selected applications from the SPEC CPU2000 benchmark suite that we have used as simulation workloads.

VPR (Versatile Place and Route)

This integrated circuit computer-aided design program implements a technology-mapped circuit. It performs a placement and routing program within a Field-Programmable Gate Array (FPGA) chip. The input files named net.in and arch.in provide the netlist of the circuit and a description of the FPGA architecture, while place.in is used for routing. Al-Zoubi et al. [2004] found this program is very sensitive to the size of cache space, which could result in a large number of conflict misses. Henning [2000] reported that the memory resident size of 175.vpr is 50Mbytes, whereas the virtual size is 55.2Mbytes.

179.art

The Adaptive Resonance Theory 2 (ART 2) is a neural network benchmark used to recognise objects in the thermal image of a helicopter and an airplane. The input file contains a thermal image composed of several data-parallel vector operations. It is trained and further used to match with other thermal views of the helicopter and airplane. It performs a relatively small amount of computation on its input elements, hence this benchmark is sensitive to the memory latency. The memory resident size and the memory virtual size of this benchmark are 3.7Mbytes and 5.9Mbytes, respectively [Hanning 2000].

183.equake

This application simulates seismic wave propagation in large, highly heterogeneous valleys. Computations are performed on the input data to produce a case summary of the earthquake including the seismic source data and reports of the ground motion. This application uses 49Mbytes of memory resident size, while the virtual size is 51.1Mbytes.

181.mcf

This program requires approximately 100 to 190 megabytes, respectively, for a 32 and 64 bit architecture [Hanning 2000]. The memory resident size and the memory virtual size of this program is 190Mbytes and 192Mbytes. This program is designed to solve single-depot vehicle scheduling problems in public mass transportation. Log information will be provided and an optimal schedule will be computed by this program.

300.twolf

TimberWolfSC is another place and global route program in the SPEC CPU2000 suite, used in the production of microchips. It determines the placement and global interconnections of groups of transistors in a microchip design. Three different benchmark circuits will be input to this program and each circuit will produce two output files describing the placement and the global routing of the circuit. This program requires 1.9Mbytes of resident memory and a larger size of virtual memory which is 4.1Mbytes in size [Hanning 2000].

3.4 METHODOLOGY

The main objective of this work is to optimise the organisation of the last level cache (LLC) efficiently by gaining a new insight into cache partitioning schemes. Problems in the LLC specific to a shared cache are known to be due to cache contention and could be improved by optimising the sharing mechanism in the cache. Overall, multi-core system performance is highly dependent on the performance of each individual core running simultaneously. To practically investigate the behaviour of each core sharing the last level cache, finer-grain observation and analysis can be done in the system running different sets of workload. Each individual core would behave in a different manner and the access pattern of the shared

cache would change because of the differences in the working set and the cache space allocated to each core. Findings from the investigation could provide information about the effectiveness of the sharing mechanism in the system in meeting the needs of each core when running different application mixes. This is useful in improving and optimising the sharing mechanism among each individual core in the multi-core system. By implementing some mechanisms to partition the shared cache, the cache space can be better distributed to each core according to its needs and thus improve the overall performance of the multi-core system.

In this work, the utility-based cache partitioning and the CPI-based cache partitioning as discussed in the previous chapter, were used to monitor, evaluate and determine the best method for partitioning the shared cache. A quad-core system has been simulated over a range of workloads selected from those presented in Table 3.4. In particular, the algorithm that has demonstrated better performance in terms of speedup has been selected as the partitioning scheme throughout this work. A detailed discussion on the procedures and findings of the implemented cache partitioning algorithms can be found in the next chapter.

Apart from the cache partitioning algorithm, it is very important to guarantee that the shared cache is fully utilised. This is to make sure the data stored in the shared cache could be kept longer and reused as many times as possible. Therefore, the investigation also addressed cache replacement policies. Several modifications were made on the traditional LRU replacement policy. Performance of the traditional LRU and the policy proposed in this work were compared by implementing various types of cache architectures, cache partitioning schemes and replacement policies. Results from the analysis were then used to develop an improved replacement scheme, which was implemented in the system as a new dynamic cache partitioning scheme. While most of the experimentation in this work was

done on a 4-core system model, the scalability to 8-core and 16-core systems will be demonstrated later in Chapter 5, section 5.3.3.

For each multiprocessor system simulation that we have run for evaluation purposes, the entire system including all the cores was forced to shut down when any one of the cores has reached the end of its execution. We defined the first core that finished executing its application as the stopper. This concept is almost the same as if all the cores are restricted to perform detailed simulation for a specific number of instructions or cycles. Table 3.5 lists all the stoppers of all application mixes that were used in our studies.

TABLE 3.5: List of stoppers/first applications to finish its execution in each workload mix.

Workload	Stopper
Mix-1	parser
Mix-2	mcf
Mix-3	ammp
Mix-4	twolf
Mix-5	ammp
Mix-6	mcf
Mix-7	parser
Mix-8	twolf
Mix-9	vpr
Mix-10	vpr
Mix-11	ammp

3.5 SUMMARY

This chapter outlined the multi-core system architecture, the simulation framework and the methodology used in this work. In a quad-core system, each core has its own private L1 instruction and data caches connected to a unified L2 cache through a shared bus. The Tensilica Xtensa development tool is used to build the cores and to configure the system. Xtensa is also used to provide a test bench to implement the various cache partitioning and replacement schemes which are the scope of this work. The SPEC 2000 benchmark suite is used to evaluate the modified algorithms proposed. This chapter discussed the characteristics of the different application benchmarks in the SPEC CPU2000 suite which were selected and combined in sets of four to provide different workload sets that used to evaluate the multi-core system.

CHAPTER 4

CACHE PARTITIONING SCHEMES

In a CMP, the key design issue facing processor architects is organising and managing the on-chip last level cache which acts as a shared resource. The shared cache may become a performance bottleneck due to contention among parallel running applications or threads in the CMP system. The unmanaged shared on-chip cache may underutilise the shared resource and lead to severe performance degradation. To alleviate this shared resource management issue, cache resource partitioning approaches have been widely studied so that CMP performance and energy efficiency can be improved. The cache partitioning technique is about distributing the shared cache by allocating a particular size of cache space to each core in the system. This can be done statically or dynamically. Note that the cache resource distribution can use way granularity partitioning or set granularity partitioning which allocates each core cache resources in the units of cache ways or cache sets.

This chapter discusses two cache partitioning schemes, namely utility cache partitioning (UCP) and CPI-based cache partitioning. UCP is widely accepted and referred to as one of the cache partitioning schemes that aimed to improve throughput [Xie and Loh 2008; Xie and Loh 2009; Kędzierski et al. 2010b; Li et al. 2011] using the cache utility function which is defined as the improvement in miss rate when extra cache ways are allocated to an application. This is in contrast to prior partitioning schemes which improved miss rate based

on the number of cache lines an application uses. A CPI-based cache partitioning scheme has been found helpful in reflecting the benefit of using CPI values in making partition decisions [Muralidhara et al. 2010] because the use of CPI values for evaluating the application's performance is better than using miss rate, since the miss rate does not account for the number of memory accesses made by the application. Evaluations of both schemes are presented to justify the benefits of dynamically partitioning the shared cache among multiple application workloads.

4.1 Simple Static Allocation Scheme

4.1.1 Introduction

The need to deal with competition among multiple threads (or applications) for a shared cache has led to the introduction of cache partitioning policies in cache designs. The cache partitioning can be done either statically or dynamically. In a static cache partitioning scheme, each core will be assigned a fixed allocation of cache space. The assignment of the amount of cache space that each core is restricted to use is performed prior to the commencement of the thread's execution and the allocated cache space will remain unchanged throughout execution. To partition the cache dynamically, the characteristics and the cache requirements of the running threads (or applications) in each core at different execution phases will be taken into consideration. This information will be used to determine the cache partitioning and to assign an efficient amount of space that each core can occupy in the next execution phase. The cache volume each core is allowed to use may vary during different execution phases.

Taking advantage of the underutilisation of the shared cache by some of the threads (or

applications) towards the overall system performance, existing partitioning policies use a number of performance metrics to decide on a cache space distribution among the threads. Traditional partitioning policies tend to partition the cache resources based on cache demand or by using the simplest methodology whereby the cache is distributed equally among the threads. However, a thread with a high demand does not always benefit from a large amount of cache resources. Streaming applications for example, have an extremely large working set but poor cache reuse. Devoting the same amount of cache resources to all threads is not likely to be very effective in handling various execution behaviour and threads' requirements at runtime. The work in this thesis started with investigations on the effectiveness of partitioning the cache resources among the threads/applications using the aforementioned simple methodology.

4.1.2 Evaluation

Before implementing any complex cache policies or schemes, it is worth demonstrating the importance and benefits of partitioning the shared cache in a quad-core system. The attributes of the system were shown in Table 3.2. All of the private L1 caches are connected to the shared cache via a shared bus. A write-back policy has been implemented in the system to reduce traffic on the bus.

For implementation simplicity, the shared cache is initially statically partitioned and a simple allocation policy is used in the system. All individual cores were assigned the same amount of cache space equal to (*set-associativity*/*number_of_cores*) ways. The allocation policy is simple and was expected to reduce the shared cache miss rate, improve the IPC throughput and exhibit the necessity of partitioning the cache compared to when the cores share the complete unpartitioned cache using the traditional LRU replacement policy. Two extra bits were required in the quad-core system as an extension to the tag bits which

represents the ownership of each line in the cache. In addition, every core has a counter to keep track of the number of cache entries belonging to it at runtime. The counters are used to gather the actual cache occupancy of each application running in each core so that the amount of cache volume each application is allowed to use will remain unchanged.

To ensure the cache space allocated to each core is sufficient to reduce cache contention and safeguard that the data stored in the allocated space is kept long enough for future references of each core, the standard LRU policy has been modified as an enhancement towards the partitioned shared cache. In the traditional LRU policy, all new incoming lines will be marked as the most recently used, while on every cache miss, the least recently used cache line will always be selected as a victim to be replaced. As the victim is selected based on the order in which the cache entries in the same set have been accessed, there is no information about the ownership of each entry occupied by each application. The applications with many accesses (high demand) to different cache entries will own more cache space than applications with low demand.

Even though the demand driven LRU replacement policy can implicitly implement cache way partitioning, the augmented LRU policy is used so that each core can have ownership to each line that it occupies in the cache. The augmented LRU policy is employed to guarantee that each core is forced to utilise a limited amount of cache ways at a specific execution phase. When a cache miss occurs, the replacement policy will select the least recently used line to be evicted based on the recency value of each counter and the ownership of the cache entries. If the miss-causing application has fewer entries in the counter as compared to the amount it has been allocated from the beginning, the LRU line that belongs to another application will be selected for replacement of a new incoming line, or else the LRU entry belonging to the miss-causing application is selected. With this

strategy, the cache partitions can be maintained by controlling which thread can evict which cache line. Thus, the counter of the miss-causing core in this augmented LRU is only updated on a cache miss.

The simulations of the quad-core system are run until any one of the cores has reached the end of its execution. Once the core has finished running the whole application workload, the entire system will be forced to shut down. This concept is almost the same as if the cores perform detailed simulation for a specific number of instructions or cycles. As a result, the end point of the execution of other cores in the system can be different in every simulation, if the time taken for the first core to finish is prolonged or shortened. Figure 4.1 illustrates the improvements of the cache misses in the equally partitioned cache (EQ) over an un-partitioned cache with the standard LRU policy for each of the workload mixes listed in Table 3.4. The improvement is defined as the difference in miss rate between unpartitioned cache and partitioned cache, then normalised to the miss rate of the unpartitioned cache.

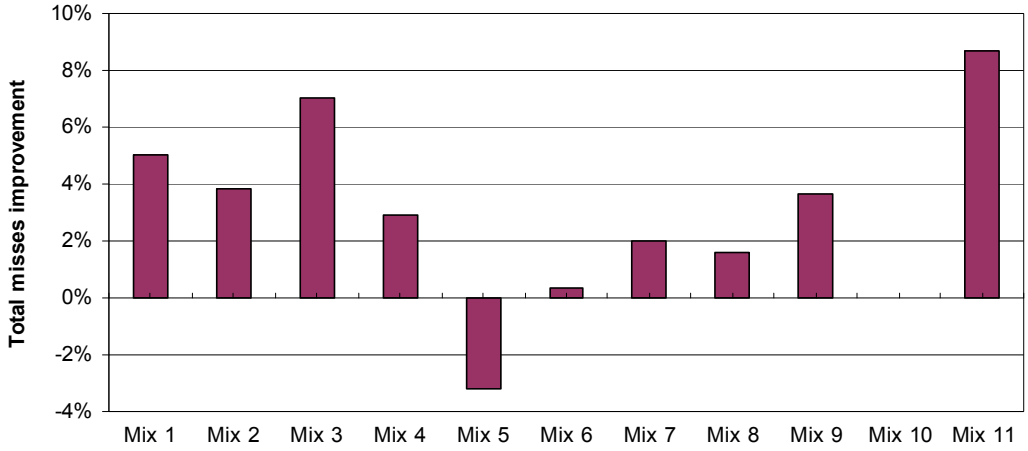


FIGURE 4.1: Total L2 cache misses improvement over standard LRU.

In most cases, partitioning the shared cache has reduced the total misses. In the best case, the result reveals that Mix11 achieved an 8.7% reduction in misses followed by Mix3 with a 7% reduction. Partitioning the cache has no effect on Mix10 and the worst case is Mix5 for which the number of misses is increased by 3.1%. The figure shows that by partitioning the L2 cache equally, it does not guarantee that overall performance can be improved as several cores in the system may not fully utilise the cache space that has been allocated to them. However, in some cases when the cache space allocated to several cores in the system is large enough for their working set, a large number of misses generated by the cores can be saved. As a result, the time taken for the cores to process their jobs can be reduced.

To explain in detail the achieved improvement of the total L2 misses, Figure 4.2 compares the miss rate percentage of both LRU and EQ strategies that contributes to a total miss rate experienced in each core. Note that the blue bar in the figure depicts the percentage of LRU miss rate normalised to the summation of both LRU miss rate and EQ miss rate in a particular core. To understand how the figure is plotted, let us first focus on Mix11. It is found that the core C4 has been allocated with sufficient cache space after EQ partitioning, it therefore has a large reduction in the miss rate. Next, core C2 has been identified as a stopper, which is the first core that will finish executing its application in the system and force other cores to stop executing as well. While waiting for C2 to finish, suppose C4 in LRU can execute 500 million instructions, but in EQ it can execute an extra 100 million instructions. Consequently, the L2 miss rate experienced by core C4 in EQ is less than the miss rate in LRU.

With Mix3, it can be seen that core C4 has the greater miss rate percentage in EQ compared in LRU. Assuming that the remaining three cores in the system have used the

allocated cache space efficiently (such as in Mix11), the total L2 cache miss rate in EQ was decreased and became less than the total miss rate incurred in LRU. The augmented LRU policy that has been implemented in EQ has kept data stored in the cache long enough for performance to benefit from frequent future references. Therefore, EQ can save more misses and more instructions can be executed on the system.

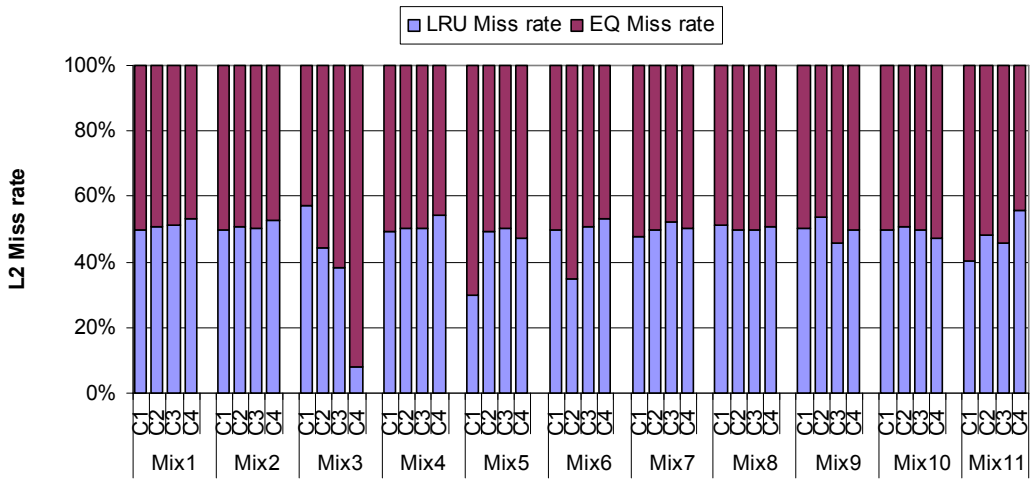


FIGURE 4.2: Distribution of total misses caused by each individual core in the quad-core system.

In contrast for Mix5, core C1 in EQ is expected to be the reason for the increasing total miss rate in L2 cache. Although the miss rates generated by the other cores are similar to the LRU case, at least one of the three cores might benefit from partitioning the cache. Let us assume that the cache space allocated to core C2 is large enough for its working set and core C3 is the stopper in the application mix. Even though the core C2 benefitted from the efficient cache space allocation and is able to execute more instructions than in LRU, more cache misses are generated prior to the execution of the additional instructions. The

percentage difference in the EQ miss rate and LRU miss rate in core C2 is insignificant. As a result, the total miss rate of Mix5 in EQ is greater than in LRU (from Figure 4.1).

Notice that there is an increase in the number of instructions executed in several cores in the system. Thus, the overall performance can be measured using the total instructions per cycle (IPC) count. In this study, the total IPC of each system is given by:

$$IPC_{tot} = \sum_{i=1}^{N} IPC_i \tag{4.1}$$

where N is the number of cores and IPC_i is the IPC of the ith application. Note that the IPC is defined as the number of instructions executed by an application per cycle averaged over the execution interval of interest.

Figure 4.3 reports the IPC speed-up of the quad-core system with the augmented LRU policy over the standard LRU replacement policy.

$$IPC_{Speed-up} = \frac{IPC_{AugmentedLRU} - IPC_{LRU}}{IPC_{LRU}} \tag{4.2}$$

The result demonstrates that Mix3 and Mix11 achieved comparable improvements in IPC, up to 4.4% and 2.7%, respectively, since the total number of misses improved in their L2 cache is quite significant. However, in the case of Mix5, the total number of L2 misses is significantly increased (from Figure 4.1) compared to other workloads, but there is a small IPC improvement which is not observed in several workloads that have notably reduced the total L2 misses. This is due to the considerable additional number of instructions executed by any of the cores in the system as a result of the cache being partitioned efficiently.

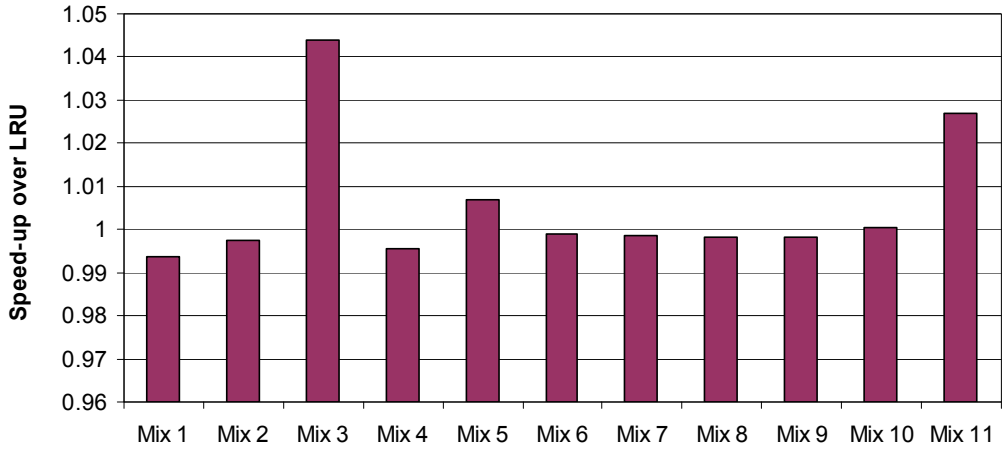


FIGURE 4.3: The speed-up of the partitioned cache with the augmented LRU policy over the standard LRU replacement policy.

On the other hand, Mix1 experienced different behaviour when the difference in the number of misses generated by all the individual cores is negligible (from Figure 4.2), despite the total number of L2 cache misses improving by 5% (from Figure 4.1). This figure shows that the IPC improvement is not relative to the reduction of the total number of L2 misses. The reason for this is that there is a core(s) which suffers insufficient cache space after the L2 cache is partitioned slowing down the processing of the specified core and therefore the whole system.

When the shared L2 cache is partitioned and the augmented LRU replacement policy is implemented, there are at least small improvements in either the miss count or the IPC count. This fact confirms that the size of the cache space allocated to the cores if sufficient, could efficiently decrease the number of misses generated by individual cores. However, it is not guaranteed that it would reduce the total number of L2 misses in the cache as well because there are cores that could experience a significant performance gain/loss due to the

allocated cache space. The changes would therefore later influence the overall system performance including the IPC.

With respect to the weaknesses of the traditional LRU policy discussed in Chapter 2, the results presented so far show that the negative aspects can be managed by implementing a cache partitioning scheme. With an adequate cache space allocated to each core, improvement on the individual performance (measured as IPC) of the thread is achieved, since the number of misses is reduced. Appropriate rules in the cache replacement policy to allow each core to effectively use its dedicated cache space were very beneficial. Having a well-tuned cache partitioning scheme, so that the overall system performance of the multi-core system can be enhanced, is established by the presented results.

4.2 UTILITY-BASED CACHE PARTITIONING SCHEME

4.2.1 Overview

Utility-based cache partitioning (UCP) was developed by Qureshi and Patt [2006] to efficiently manage the last level shared cache among multiple applications. Their goal was to obtain better throughput by giving more cache space to the application which could gain most benefit compared to the remaining competing applications.

The three main components of UCP are the Utility Monitor (UMON) circuit for each core, the partitioning algorithm and the cache replacement support that is responsible for enforcing partitioning decisions. Each UMON contains a set of hit counters and a tag directory of a few sampled cache sets that simulates LRU policy. The tag arrays sampled only 32 sets so that the hardware overhead can be reduced. For a shared cache with

associativity *n*, a set of *n* hit counters is assigned to each UMON. Each counter represents a recency position ranging from LRU to MRU and the hit counts for each recency position are recorded at runtime. These counters provide the utility information of an application and the number of misses saved by each recency position.

The partitioning algorithm, called the lookahead algorithm, is invoked every five million instructions. Partitioning decisions are calculated on a way basis using information recorded by the stack counters. The maximum marginal utility (*MU*) of all applications are computed in parallel by all UMONs and any application that gets maximum value of marginal utility among the competing applications will be given extra cache ways. The possible partition that could maximise the total hit counts of the entire system will be selected and used in the next execution interval. The marginal utility is defined as the utility per cache resource and it is written as:

$$MU_a^b = (miss_a - miss_b) / (b - a) \tag{4.3}$$

where $miss_a$ and $miss_b$ are the number of misses incurred when an application receives *a* ways and *b* ways respectively.

To cope with the cache way partition decision, the baseline LRU policy is modified in the context of eviction control. On a cache miss, the total number of lines used by the miss-causing application is calculated and if the number is less than the number of lines allocated by the partitioning algorithm, the LRU entry in a set that does not belong to the application is evicted or the LRU line of the miss-causing application is selected as the evictee.

4.2.2 Evaluation

The utility-based cache partitioning scheme of Qureshi and Patt was re-implemented and

evaluated on a quad-core system sharing the last L2 cache. The results illustrated in Figure 4.4 show the total number of misses saved by UCP compared to the previously discussed partitioning schemes, namely LRU and EQ. In this graph, the miss improvement is defined for LRU as:

$$\frac{(miss_{LRU} - miss_{UCP})}{miss_{LRU}}.100\% \qquad (4.4)$$

and for EQ as

$$\frac{(miss_{EQ} - miss_{UCP})}{miss_{EQ}}.100\% \qquad (4.5)$$

For the majority of the workload combinations, UCP improves the total number of misses by up to 47.29% over LRU and up to 15.06% over EQ. However, degradation of the cache miss rate is observed in a few cases, but is relatively very small.

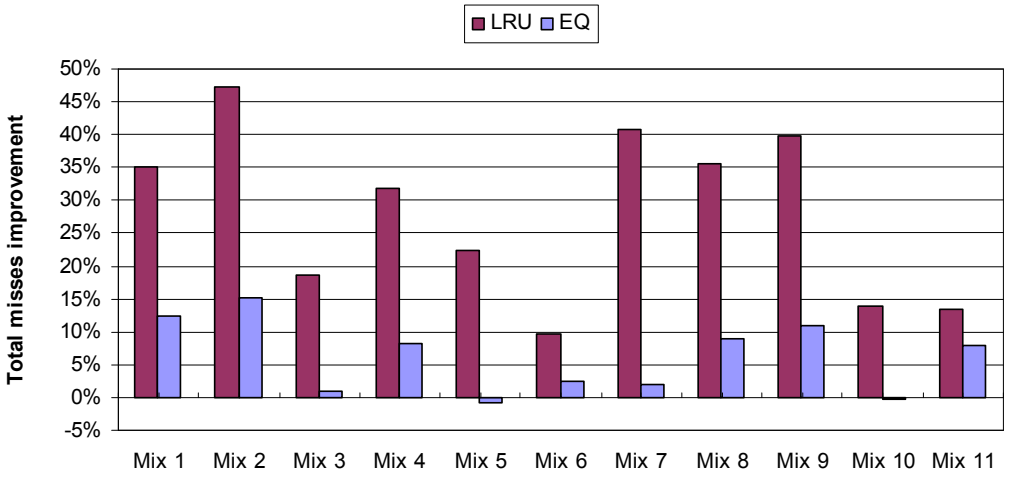


FIGURE 4.4: Comparison of UCP, LRU, and EQ cache misses in a quad-core system.

As observed in Figure 4.4, UCP outperforms LRU and EQ in terms of the total reduction in cache misses. This was expected and happens in response to dynamic and efficient cache management by UCP. UCP partitions the cache among the competing applications based on cache utility which has a more significant impact on the total number of cache misses in the system compared to LRU and the simple static partitioning scheme of EQ. When the applications in the workload receive nearly the ideal amount of cache resources, the benefits gained by these applications contribute to a significant improvement in the total number of L2 misses.

Although EQ achieves improvement in the total number of cache misses, dynamic changes in the behaviour of the applications at runtime require a better and practical scheme to further improve the cache miss rate. UCP tries to allocate more cache space to the highest utility application in response to the varying demand and cache utility of the applications executed in parallel. Unlike EQ, high utility applications or applications that experience high utility at certain execution intervals, may receive inadequate cache resources during some execution intervals. The individual performance of this application in UCP is improved and it does not degrade the overall system performance when it is executed concurrently with badly behaving applications.

There are few cases where UCP achieved a lower improvement in the number of L2 misses than EQ. They include Mix5 and Mix10. To help understand why this occurs, Figure 4.5 shows the total number of misses incurred by each core in the quad-core system. For Mix5, the number of misses produced by each core for both UCP and EQ are equal. This happens because the total working set size of all the applications is larger than the available cache space. Even though the cache ways have been distributed based on the cache utility, the applications receive no benefits from the well-managed number of ways allocated in

every cache set.

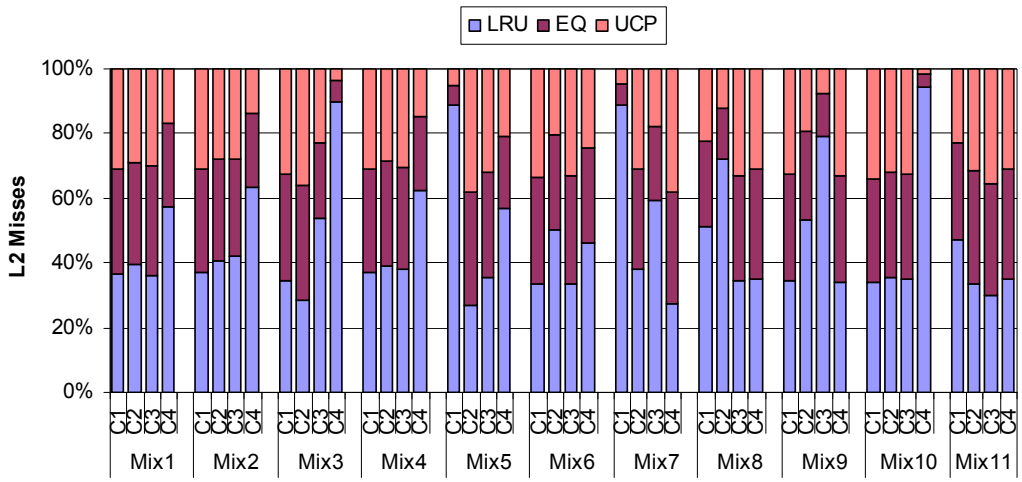


FIGURE 4.5: Total L2 misses incurred by each core when the system using UCP, LRU and EQ.

In Mix10, UCP results in a very small increase in the total number of misses compared to EQ. A significant reduction in cache misses incurred by core C4 could not prevent the remaining applications with bad behaviour from incurring more misses. Due to insufficient cache resources, the allocated cache ways in every cache set are not adequate to cater for the potential utility of all the competing applications. The number of ways allocated to core C4 is enough for the application running on it to fully utilise the cache resources, but the remaining ways in the set are still not sufficient for other competing cores to perform better and lead to an improvement in total cache misses.

In Figure 4.6, performance of UCP in terms of IPC shows an improvement of up to 6.12% over LRU and 1.62% over EQ. The performance improvement over LRU is defined

as:

$$\frac{(IPC_{UCP} - IPC_{LRU})}{IPC_{LRU}}.100\% \tag{4.6}$$

while for UCP, performance improvement over EQ is defined as:

$$\frac{(IPC_{UCP} - IPC_{EQ})}{IPC_{EQ}}.100\% \tag{4.7}$$

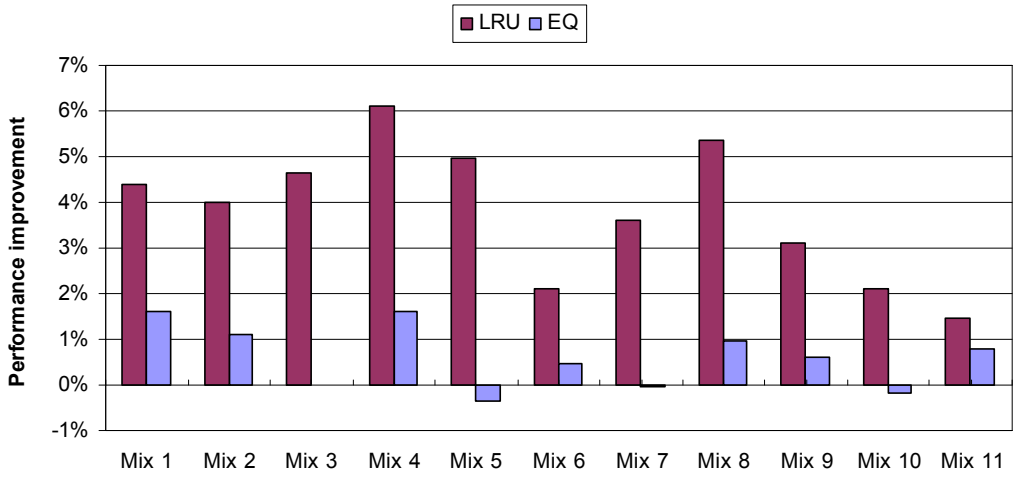


FIGURE 4.6: Performance improvement of UCP over LRU and EQ for a quad-core system.

The IPC throughput of the system as illustrated in Figure 4.6 shows that system performance reflects the total reduction of cache misses in Figure 4.4. In the case of Mix3, small improvements in cache misses compared to EQ lead to no performance gain for the system. One of the key reasons is that even if the cache misses are reduced significantly by one of the cores in the system (core C4 in Mix3), the total number of instructions executed

in the remaining cores and their individual total number of misses are still the same. Thus, the system experienced no performance improvement.

4.3 CPI-BASED CACHE PARTITIONING SCHEME

4.3.1 Overview

Muralidhara et al. [2010] proposed an intra-application cache partitioning strategy using a dynamic curve fitting based cache partitioning scheme that tries to improve performance and fairness of a multithreaded application. The execution progress of each individual thread is monitored and the shared cache space is partitioned, based on the CPI value of the threads. The scheme is referred to as the CPI-based cache partitioning scheme in this thesis. The information gathered at runtime is used to identify the slowest thread (the critical path thread) as well as the fastest thread (the lowest CPI thread). In particular, the CPI-based cache partitioning scheme attempts to minimise the speed difference (slack time) of the highest CPI thread at every execution interval.

At the end of each execution interval of 15 million instructions, the partitioning algorithm is invoked to collect the total instruction and cycle counts of each thread. The chosen execution interval is similar to the one applied by Muralidhara et al. [2010] in their work. Then, the CPI value of the threads is calculated and with the number of ways assigned to each thread at the current execution interval, a runtime CPI model of each thread is built. The partitioning algorithm first uses the CPI values calculated earlier to identify the slowest and the fastest threads and a cache way of the fastest thread is given to the slowest thread. Since the way partition is now changed, the CPI value of each thread is estimated using the CPI models built earlier. If the thread that just received an extra way remains the critical

path thread, another cache way is given to it and this process is repeated until some other thread is identified as becoming the new slowest thread. At this point, a cache way is returned to the fastest thread before the new way partition is implemented in the next execution interval. The best partition decision is aimed to allocate more ways to the slowest thread, thereby speeding up the execution of the entire application.

Similar to UCP scheme, the LRU policy is modified so that the cache can be implicitly partitioned among the threads. The modified LRU policy works in thread-wise manner, since the victim selection must be among the lines belonging to the miss-causing thread. In a scenario where the number of allocated lines is less than the number of lines belonging to the miss-causing thread, a cache line belonging to another thread is replaced.

4.3.2 Evaluation

Simulation of a quad-core system exploiting the CPI-based cache partitioning scheme was performed and evaluated using the previously detailed workloads. This means, multiple applications were executed on the system instead of a single multithreaded application as used by Muralidhara et al. [2010] in their work. Although different single-threaded applications were chose to be executed on the system may not extensively competing for a shared resources such as by different threads of a multi-threaded application, it was expected that the CPI-based cache partitioning scheme may able to improve the system performance and produce comparable results. The reason is because each thread of a multi-threaded application may experience different execution behavior and shared resource demand at runtime that could be as similar to the execution progress of each running single-threaded application. Thus, the CPI-based cache partitioning scheme was used in a system running multiple single-threaded applications so that the effectiveness of the scheme's concept and the reliability of the scheme to improve the sharing mechanism of a shared

resources among multiple single-threaded applications can be investigated, thereby can be enhanced accordingly later.

The results presented in Figure 4.7 show that even though this scheme was implemented on multi-application workloads, it manages to improve system performance and outperforms LRU by up to 48.45% (on average 26.89%) and outperforms EQ by up to 8.71% (on average 3%). The significant average performance improvement between the implementation of the scheme against LRU and EQ may be due to the varying shared cache demand among cores in the system during runtime. Note that, the shared cache is not explicitly partitioned/distributed among cores in the baseline system using the LRU policy, which is as opposed to the system using EQ. Thus, the cores in the system using EQ may have received adequate shared cache resources in which the cache demand by each cores could be satisfied, in which case the implementation of CPI-based cache partitioning scheme would not provide significant improvement against EQ.

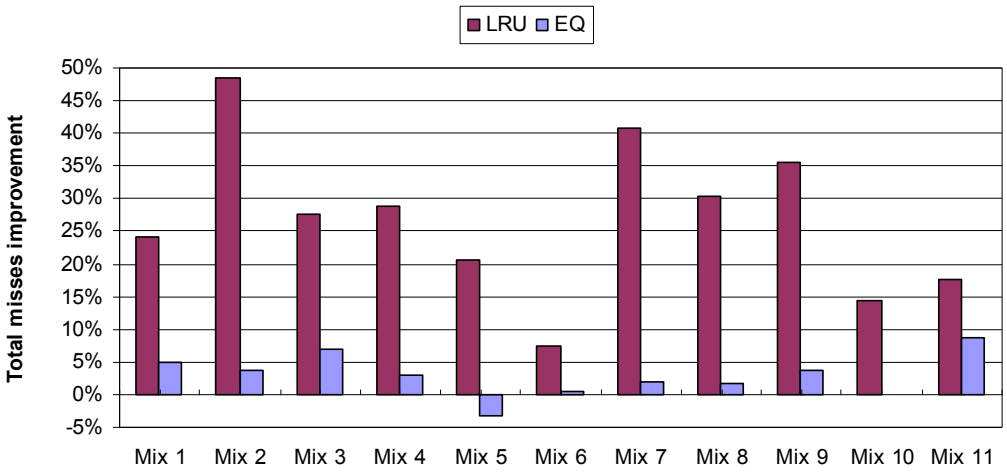


FIGURE 4.7: Reduction of total cache misses of CPI-based partitioning scheme over LRU and EQ on in a quad-core system.

Additionally, in the system using LRU, each core must compete for the cache resources along with the varying cache demand of the applications executed in parallel. Hence, some cores in the system may be starved because of inadequate cache resources, while some others may seize more cache resources than they really need. As a result, the comparison between the CPI-based cache partitioning scheme and LRU shows significant performance improvement compared to performance against EQ. However, the comparison with EQ shows that one of the eleven workload combinations yields an additional 4% of misses in the total cache miss count. This is experienced by Mix5 and Figure 4.8 illustrates that the extra misses were produced mainly by core C1.

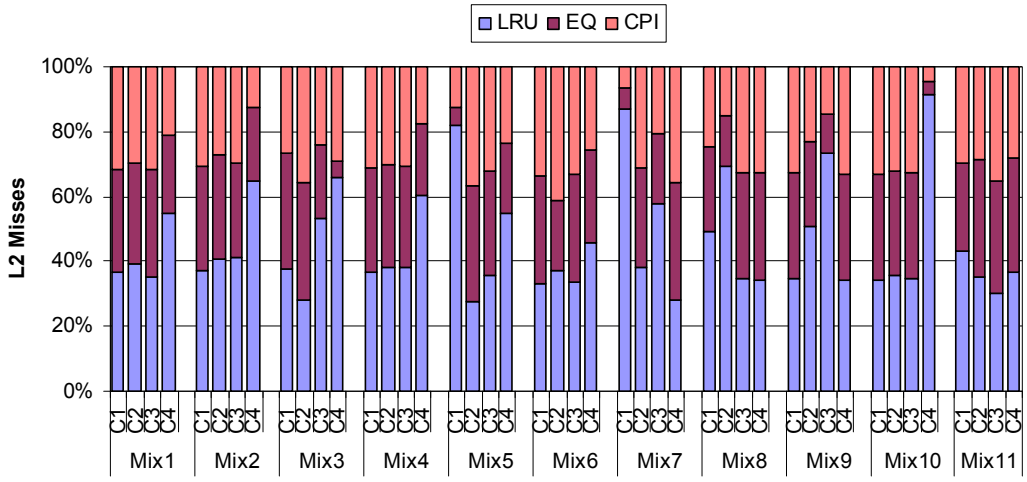


FIGURE 4 8: Distribution of total L2 misses incurred by each core in the system.

In this scenario, based on the total number of misses produced by core C1 in a system using LRU, the CPI value of the core at different execution phases/intervals may frequently become the highest compared to the CPI value of other cores in the system. Thus, the CPI-

based partitioning scheme assigns more cache ways to the core C1 to speed up its execution. However, the extra cache ways allocated are sufficient to improve the execution speed of the core so that extra instructions can be executed, but fail to prevent production of cache misses from the additional instructions. As a result, with the CPI-based partitioning scheme, the total cache misses of Mix5 is slightly increased when compared to the total cache misses produced by EQ. Similarly in Mix3, core C4 incurs significant cache misses after the cache is repartitioned based on CPI, compared to EQ. The number of misses produced by other cores in the system is observed to have a trivial reduction relative to EQ, thereby improving the total L2 cache misses created in the system. Such misses produced by C4 are caused by the extra instructions executed by the system which are not processed in EQ. Notice that CPI-based partitioning tries to improve the speed of the slowest application. Additional instructions may be executed when the application has achieved the improvements. To better understand this, Figure 4.9 shows the CPI improvement among all the workloads.

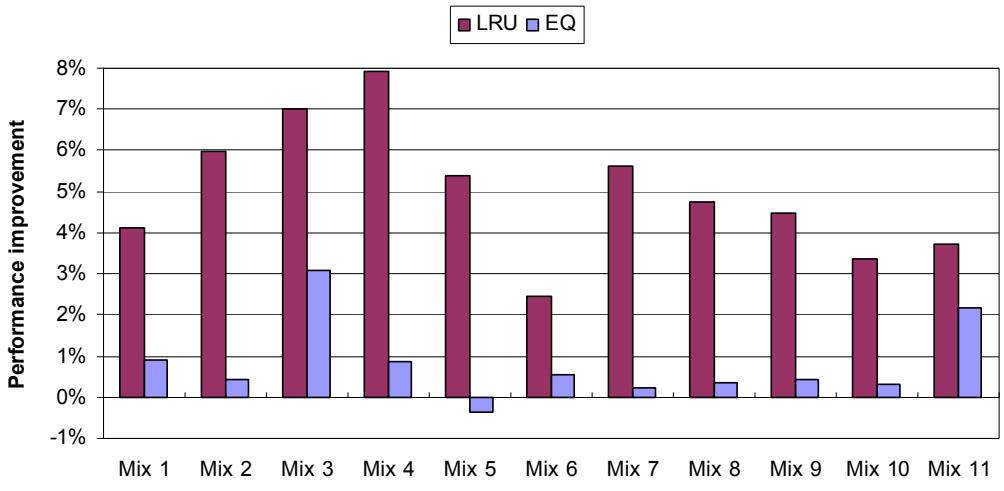


FIGURE 4.9: Performance of CPI-based partitioning scheme over LRU and EQ for a quad-core system.

It can be seen that Mix3 reached the highest performance enhancement of 3.11% over EQ. The CPI-based partitioning scheme outperforms LRU by up to 7.93%. A comparison with EQ reveals that this scheme can still exhibit performance degradation, such as experienced by Mix5. Core C1 incurred more misses than EQ, leading to about a 4% increment in the total number of L2 misses but less than 0.5% degradation in the system performance. The total number of misses produced by Mix10 is similar to EQ, but the overall IPC of the workload has achieved a small improvement of 0.4%. Consequently, the plot shows that this scheme has successfully improved the speed of the system without costing any degradation in performance metrics such as the number of cache misses

4.4 SUMMARY

Evaluation among the four partitioning schemes, LRU, EQ, UCP and CPI-based, reveals that dynamically partitioning the cache on utility-basis and CPI-basis achieved greater performance gain for the majority of simulated workload combinations.

The advantage of LRU over the other schemes is that it can respond to changes in cache demand at runtime and manage the cache resources differently in every cache set based on the behaviour of different applications. LRU may cause very little performance improvement in applications with bad behaviour such as cache contention. Applications with good behaviour may not be well-managed by LRU and may experience cache starvation. In contrast, EQ, UCP and CPI-based partitioning schemes explicitly partition the cache to improve the performance of each application in isolation, by distributing cache space among different applications executing in parallel. These schemes try to avoid cache contention and improve the overall throughput in addition to the implementation of LRU in

the system. From the results presented in this chapter, comparisons with EQ show that UCP makes more significant improvements to the number of cache misses than the CPI-based scheme. Conversely, a CPI-based scheme improves IPC throughput more than UCP.

Prior to changes in the applications' behaviour at runtime and specifically in different execution intervals, the trade-offs between cache misses and system performance improvements are crucial but difficult to control and avoid. In order to make a better partitioning decision, UCP tries to retain past information gathered from the previous execution interval. The hit counters in all UMON circuits are halved so that dynamic variation of the behavior of the applications is taken into account before an approximation of the best partition is made and implemented in the next interval. On the other hand, the CPI-based partitioning scheme updates the CPI models at every execution interval. Thus, the main data used by the partitioning algorithm is maintained because information of the current interval and the past intervals is always used.

The partitioning decision made by UCP ensures that each core will receive at least one cache way at every partitioning interval. The partitioning decision is made based on the maximum utility value that may be achieved by all the cores in the next execution interval. This means the allocation decision depends on and affects each application executing in the system. On the other hand, the CPI-based partitioning scheme focuses on two threads/applications, the slowest and the fastest threads. After the cache is repartitioned, only these two threads experience changes in the number of cache ways allocated by the partitioning algorithm.

From the simulation results presented in this chapter, it is found that the UCP scheme is better than the CPI-based scheme because of its ability to estimate the benefits of giving

more cache ways to each competing application in the system, whereas the CPI-based scheme only estimates the performance that could be gained by the slowest and the fastest applications in the system. In the context of the hardware overhead, it is observed that the cost of implementing a tag array for each application used in the UMON circuit of the UCP scheme is significantly higher than the cost of additional hardware in the CPI-based scheme. The CPI-based scheme is therefore better than UCP in structuring additional hardware in the system, but worse than UCP in evaluating and estimating the performance benefits that could be gained by all competing applications in the system. In addition, it is observed that the UCP scheme may achieve better performance in a system using a LRU-like cache replacement policy due to its strategy of monitoring the cache utility of an application when the application's data is traversing from the MRU to LRU recency positions. This is another weakness of the UCP as it may not give an accurate estimation of partitioning decisions if another type of cache replacement policy is employed in the shared cache, in which case it may be ineffectively partitioning the shared cache.

The drawbacks of the CPI-based scheme are identified in terms of its inconsistency when applied in a system running multi-application workloads and its limitation in making partitioning decisions due to the performance metrics used. From our observations, the CPI-based scheme showed inconsistent benefits to the overall throughput of the system executing multiple single thread applications. This is because the scheme was initially proposed to partition the shared cache among multiple threads that belong to the slowest application in the system. Therefore, we propose a scheme that should do better than both UCP and CPI-based schemes in Chapter 6. A scheme that can efficiently manage the cache space allocation and promise fair execution speed among different applications has become the focus of the work in this thesis. In addition, a sensible cache management scheme that could benefit as much as possible from the total cache misses and the overall system

throughput is also the interest of this thesis. It is also desirable if both CPI values and the effect of increasing or decreasing the partition size of an application are taken into account when partitioning the shared cache. It is preferred that partitioning strategies will cost insignificant hardware overhead. A partitioning scheme that meets all these criteria is proposed in Chapter 6.

CHAPTER 5

CACHE REPLACEMENT POLICY

With the increasing number of applications simultaneously sharing the last level cache, cache contention has become one of the major focuses of processor architects. Competition among multiple applications for the LLC without efficient cache management may lead to an under-utilised system. In multiprocessor systems, the cache resources are shared among competing applications, but due to the limitations discussed in Chapter 2, the performance of the system may be degraded by various forms of interference between the applications. The cache replacement policy has been identified as one of the contributors to the overall improvement of the system performance and has received much attention from researchers and computer architects. Since the conventional LRU replacement policy is widely accepted as the standard replacement policy, many attempts have been made to improve the policy in respect of the three component policies of insertion decision, promotion policy and/or even the victim selection.

In this chapter, all three components of the replacement policy are investigated for a partitioned shared cache. The aim of the work presented in this chapter is to provide an adaptive LRU replacement policy that is targeted for a system where the shared cache has been carefully partitioned and distributed among multiple cores by a well-accepted cache partitioning scheme. The results of the simulated adaptive insertion, promotion, and eviction

policies are presented, as well as their impacts on the overall performance of the multiprocessor system.

5.1 INTRODUCTION

The commonly used LRU replacement policy employed in a multiprocessor system is inherited from uniprocessor systems. For that reason, some drawbacks of the LRU replacement policy are identified which are generally due to lines that reside too long in the cache. The insertion of new lines at the highest priority position (the MRU position) in turn has caused zero-reuse lines and lines without high-locality to spend a large amount of time occupying the cache [Xie and Loh 2009; Li et al. 2011]. The promotion strategy of the LRU replacement policy that automatically moves hit lines to the MRU position have also resulted in cache waste because it maximises the time taken for dead lines to traverse from the MRU position to the lowest priority position (LRU position) and eventually be evicted. A dead line refers to a cache line that is not reused from the time of its last access until it is evicted from the cache [Xie and Loh 2009]. Even though the LRU policy is good at handling lines with high utility and temporal locality, other cores in the system have to wait longer before they can make use of the cache capacity while dead lines remain in the cache. Nevertheless, there are lines that are re-referenced immediately after they are replaced and if the LRU replacement policy can keep such lines longer in the cache before they are evicted, additional hits could be manifested. Many researchers have proposed a variety of techniques intended to improve the replacement policy so that better performance of the shared cache can be obtained.

5.2 MOTIVATION

The Pseudo Insertion/Promotion Policy (PIPP) proposed by Xie and Loh [2009] is one of the cache replacement techniques that have been augmented from the traditional LRU policy, particularly in the context of insertion and promotion policies. The main differences between PIPP and the LRU policy are (1) PIPP does not always insert incoming cache lines at the highest priority position and (2) upon a cache hit, PIPP will not automatically promote the hit line to the highest priority position. Note that the Utility-based Cache Partitioning (UCP) scheme discussed in the previous chapter is used by PIPP to determine the allocation of cache ways for competing applications. However PIPP does not explicitly partition the shared cache as UCP does. Instead it uses the cache allocation information to implicitly partition the cache. In other words, the number of ways allocated to an application is used to specify the priority position for the application to install new incoming lines and the sequence order of the insertion position starts from the lowest priority position to the highest priority position. In addition, PIPP promotes hit lines by a single priority position towards the highest priority position with a certain probability, ensuring that the priority increment remains unchanged.

Figure 5.1 shows how the insertion policy of PIPP works. In the figure, the UCP scheme has allocated 5 ways to Core 1 and 3 ways to Core 2. All new lines of Core 1 are inserted at priority position 5 while all new lines of Core 2 are installed at position 3. Note that in PIPP, evictions always choose the line at the lowest priority position. On every cache miss, the miss-causing core can evict cache lines that belong to another core in the system. This allows cache capacity stealing among the cores, which may result in an inefficient strategy in maintaining the cache ways that have been allocated to the cores.

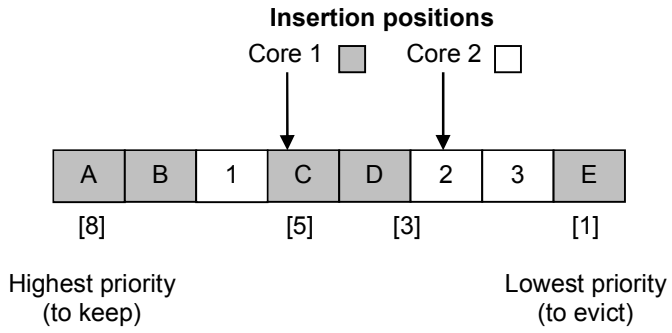


FIGURE 5.1: Example of insertion positions in PIPP. Consider Core 1 is allocated with 5 ways and Core 2 receives 3 ways from UCP scheme.

Generally, the insertion policy of PIPP seems to treat zero-reuse lines more effectively than the LRU policy. Under PIPP, the lifetime of the zero-reuse lines can be reduced due to a shorter distance for the lines to traverse towards the lowest priority position for eviction. The promotion by a single priority position of PIPP enables the single-reuse lines and the lines with a burst of references to occupy the cache long enough before they become dead, but will quickly become a potential victim once they are ready to be replaced.

Even though PIPP can improve the performance of LRU policy, it is less effective at maintaining the number of cache ways that have been allocated by the partitioning scheme to each competing applications. This is due to its pseudo insertion/promotion policy that does not explicitly enforce the partitioning. PIPP actually stimulates the system to create results similar to the hard partition, so that the cache capacity stealing allowed by PIPP among all the applications may produce benefits in the utilisation of the available cache resources. However, this also may lead to a large number of lines being taken away from any core(s) in the system to be used by another core(s).

The algorithms described in this chapter are intended to strictly partition the cache and explicitly enforce the original way-partitioning decisions. One may say that some of the cache capacity allocated to the cores may be under-utilised, but such a situation is not likely to happen if the cache is properly partitioned by the cache partitioning scheme. If all components of the cache replacement policy are improved accordingly, the occupancy of the partitioned shared cache can be controlled thereby improving the overall performance of the system. Since UCP is widely accepted, it is used in the works presented in this chapter with the implementation of an adaptive LRU replacement policy that has been adopted from the concept of PIPP. The adaptive insertion and promotion strategies of the new policy are augmented from the concept introduced by PIPP in order to maintain the benefits received by zero-reuse lines, single-reuse lines and multiple-reuse lines at a short distance. Meanwhile, the eviction policy of the traditional LRU policy is improved so that it can fit the environment of a multiprocessor system so that the cache capacity stealing can be controlled and avoided among the cores.

5.3 Adaptive Insertion and Promotion Policy

5.3.1 Description

The insertion and promotion policies of the traditional LRU give each line more chances to obtain a hit while it traverses from the MRU position to the LRU position. However, this strategy may not be very good for workloads in which the working set size is greater than the available cache space. It has been identified that issues such as cache thrashing can be improved by retaining some fraction of working set long enough in the cache in order to manifest more cache hits [Qureshi et al. 2007; Jaleel et al. 2008].

Earlier studies have proposed to insert a new line either at the MRU position or the LRU position. When the new line is inserted at the MRU position, frequent accesses of the line while it traverses towards LRU position can derive additional hits, but it will also waste cache capacity in the case of dead-on-arrival lines. By comparison, insertion at the LRU position has shown substantial benefits on the zero-reuse lines as the lines can be evicted immediately after they are used [Jaleel et al. 2008]. On every cache hit, both strategies automatically move all the hit lines to the MRU positions. This may be good for cache lines with multiple-reuse, but not for single-reuse lines or lines with many references at greater distance (particularly after these lines become dead). It would be desirable if the cache spaces that these lines are occupying are used by other lines to generate additional hits.

To avoid such circumstances, the existing strategies are improved in a new approach proposed in this section, namely Middle Insertion 2 Positions Promotion (MI2PP) policy. The objective of MI2PP policy is to provide better thrashing resistance to a partitioned shared cache by maintaining an equal number of cache ways received by all cores to the number of ways that have been allocated originally. The MI2PP policy is targeted to improve the insertion and promotion strategies of a cache replacement policy so that the cache occupancy of zero-reuse and single-reuse lines is not wasted, while multiple-reuse lines could be retained in the cache long enough to incur additional cache hits before eviction. In other words, the MI2PP policy tries to retain as many of the beneficial cache lines as possible, so that they could manifest more cache hits in the allocated cache space and minimise the residency of zero-reuse and single-reuse lines in the allocated space without yielding significant hardware overhead. Therefore, the MI2PP policy chooses to insert the new lines at the middle position of the priority stack by using the following formula:

$$InsertionPosition = \frac{set\_associativity}{numberOfcores} \quad (5.1)$$

MI2PP may not be as good as LRU insertion in minimising the residence time of the zero-reuse lines in the cache, but it could reduce the time spent by the zero-reuse lines by approximately half compared to those installed at the MRU position. MI2PP could also reduce the number of misses incurred due to dead-on-arrival lines. For the lines with single-reuse and multiple-reuse, they can be protected and have more chances for further hits while they traverse from the middle position to the LRU position. This is in contrast to the immediate disadvantage if they are inserted at the LRU position. However, if the next references are not in the near-future, LRU insertion will remain a better policy than MI2PP.

On a cache hit, MI2PP implements a different promotion policy from that of LRU. The hit line will automatically be moved by two priority positions towards the highest priority position. The reason for this change is that if the lines were automatically shifted to the MRU position, as in LRU policy, single-reuse lines and lines with multiple-reuse at a greater distance may consume cache capacity without providing any benefit. MI2PP will shorten the distance for the dead lines to traverse towards the LRU position, meaning the lines will have high potential to be evicted sooner than in the conventional MRU promotion policy. This strategy will also slow down the movement of the lines with multiple-reuse towards the MRU position if the next accesses of other cache lines happen at priority positions lower than the positions of the multiple-reuse lines, providing the multiple-reuse lines with more opportunity to derive additional hits along the way. Given that all new lines are installed at the middle priority position and get promoted by only two positions upon cache hits, this approach does not cause any disadvantage to the many frequent access lines. This is because these lines are preserved at priority positions which are high enough for

them to be kept in the cache long enough to generate more hits before eviction.

Since the target of the MI2PP policy is to ensure that the number of cache ways received by all cores is equal to the number of ways they have been allocated, victim selection for the cache line replacement is modified by taking into account the total number of ways belonging to the miss-causing application/core. If the miss-causing core gets more ways than it is allocated at the time of the cache miss, the LRU line of the core will be selected as an evictee. Otherwise, the LRU line in the set that does not belong to the miss-causing core will be replaced. This technique is used to enforce the cache partition and helps to preserve the benefits of partitioning the cache by sustaining adequate cache resources among all the cores.

In summary, the intention of MI2PP policy is to provide better thrashing resistance to a partitioned shared cache. The aim was to provide an effective yet not so complex cache replacement policy that works well in a partitioned cache, since an optimal cache partitioning scheme used in the system may cost significant hardware overhead. Therefore, MI2PP policy uses information already available from the baseline LRU replacement policy and exploits the information to make better use of the partitioned shared cache resources to gain better overall performance of the system.

5.3.2 Comparing Operation of MI2PP policy to PIPP

Since MI2PP policy was adopted from PIPP, a simple example used by Xie and Loh [2009] is referenced, extended and presented in this section to deliver a detailed explanation of MI2PP. Figure 5.2(a) illustrates a simple operation of PIPP, while Figure 5.2(b) demonstrates the operation of MI2PP policy on a dual-core system.

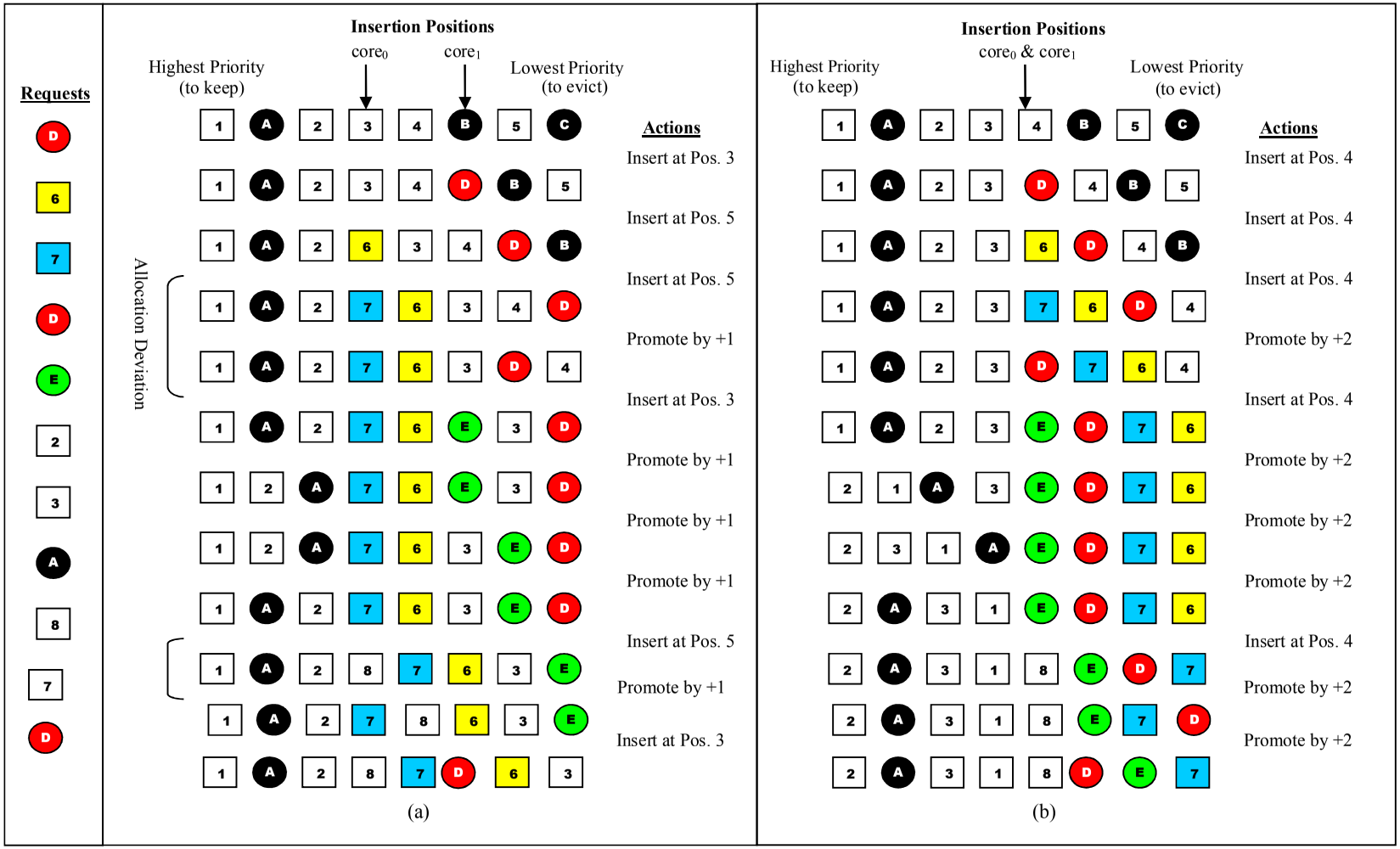


FIGURE 5.2: Example operation of (a) PIPP from Xie and Loh [2009], and (b) MI2PP policy for a variety of insertions and promotions activities. Assume evictions for both policies always choose the lowest-priority cache line in the set.

The two cores, namely $core_0$ and $core_1$ are allocated with five and three cache ways, respectively. To differentiate the lines occupied by the cores, the cache lines of $core_0$ are symbolised by numerals in squares, while letters in black circles represent cache lines of $core_1$. For both of the policies, the insertion position of each core is shown in the figure. For PIPP in Figure 5.2(a), when $core_1$ makes a first request for line D (symbolised in red circles) which is not in the cache, the line is installed at position 3. Similarly with $core_0$, when references to line 6 (symbolised in yellow squares) and 7 (symbolised in blue squares) during the second and third requests incurred cache misses, both lines are inserted at position 5. Conversely, MI2PP in Figure 5.2(b) always inserts all the new lines requested by any core at the same location, which is at position 4. When the next reference of line D (the fourth request) by $core_1$ creates a hit, PIPP will automatically move the line by a single position. On the other hand, the line will be shifted slightly higher by MI2PP, which is by two priority positions.

One of the significant differences between the two policies that can be observed from the diagram is in the context of cache capacity management. At some execution intervals, it can be seen that PIPP demonstrates cache allocation deviation. This is due to the highlighted fact that the policy does not explicitly enforce the cache partitions, but only pseudo-partitions the cache. Conversely, MI2PP strictly enforces the cache partitions and always maintains the target allocation of 5 lines to $core_0$ and 3 lines to $core_1$ in every execution interval.

To better understand the similarities and differences between PIPP and MI2PP policy, the discussion is narrowed down to the operation of both policies towards zero-reuse, single-reuse and multiple-reuse lines. Assume that line 6 (symbolised in yellow squares) that belongs to $core_0$ does not exhibit any reuse. PIPP inserts the line at position 5, while

MI2PP installs at position 4. The figure illustrates that after several hits and misses, MI2PP eliminates the line from the cache more quickly than PIPP. This shows the insertion at priority position 4 (at the middle of the priority stack) can reduce the lifetime of dead lines in the cache. However, if MI2PP is compared with the LRU insertion such as implemented by Jaleel et al. [2008] in TADIP, MI2PP is not efficient in minimising the age of the dead lines although it has been shown in the figure to perform better than PIPP.

Now consider line 7 (symbolised in blue squares) as a single-reuse line which is reused while it is still in the cache. MI2PP will promote the line by two priority positions, which is slightly higher than PIPP but the lower insertion at position 4 by MI2PP helps to push the line downwards to the end of the priority order sooner than PIPP. However, if line E (symbolised in green circles) is also considered as a single-reuse line which is installed by PIPP at a lower priority position than MI2PP, the figure shows that a cache miss would be incurred in PIPP if the next reuse happens at the final interval. This is because the line is already replaced by another line, while MI2PP retains the line in the cache. Thus, the promotion by 2 positions in MI2PP is beneficial in keeping the single-reuse line long enough for the next reuse, while insertion at the middle position is an advantage to shorten the lifetime of the single-reuse line after the line has been reused.

Finally, as seen in the figure, line D represents a multiple-reuse line. After line D is brought into the cache, MI2PP keeps the line long enough to expose it to cache hits on every next reference. However, PIPP eliminates the line more quickly than MI2PP after the line has been first reused, which costs a cache miss upon second reference to the line. The treatment of line D illustrates how the promotion by two priority positions of MI2PP can be better than PIPP in maximising the amount of time for multiple-reuse lines residing in the cache. Overall, the example demonstrates that MI2PP can save more cache misses

compared to PIPP.

5.3.3 Simulation Results

The proposed MI2PP policy was evaluated on a quad-core system sharing a last level L2 cache, with UCP used to partition the shared cache. The L2 shared cache was warmed up for 10 million instructions before UCP was invoked every 15 million instructions. Note that the size of the execution intervals is similar to the one that was used in the experimentation of CPI-based cache partitioning scheme presented in section 4.3. The reason for this is because according to the investigations by Muralidhara et al. [2010], the system performance overhead that was incurred due to the invoking of the dynamic scheme during runtime, specifically in every 15 to 20 million instructions or higher is very small (less than 1.5%) when compared to the overall system execution time of 5 billion instructions. As each application used to evaluate MI2PP executed around 1 billion instructions, 15 million instructions intervals were used in this work.

The results of MI2PP were compared with three multiprocessor systems: a system with an unmanaged shared cache using a traditional LRU replacement policy as the baseline, a system implementing UCP as presented in the previous chapter and a system employing PIPP. For all of the evaluations, three metrics proposed by Luo et al. [2001] were used to quantify the performance of the system. The performance metrics reported in this section are the *IPC throughput*, *weighted speedup* which indicates reduction in execution time and the *harmonic mean* of weighted speedup which accounts for and balances both fairness and performance of the system.

Let IPC_i denote the IPC of the ith application, $SingleIPC_i$ the stand alone IPC of the same application if it is executed in isolation and N the number of threads the system

executes concurrently. The formulae for the aforementioned metrics are:

$$IPC\ Throughput = \sum_{i=0}^{N} IPC_i \tag{5.2}$$

$$Weighted\ Speedup = \sum_{i=0}^{N} \frac{IPC_i}{SingleIPC_i} \tag{5.3}$$

$$Harmonic\ Mean = \frac{N}{\sum_{i=0}^{N} \frac{SingleIPC_i}{IPC_i}} \tag{5.4}$$

Figure 5.3 shows the IPC throughput of UCP, PIPP and MI2PP policies compared with the baseline system using the traditional LRU policy. For all of the simulated workloads, MI2PP consistently outperforms the LRU policy by up to 7.3% in the case of Mix4. It also can be seen that MI2PP in most cases works better than UCP and PIPP with the maximum achieved improvement of 3.1% in Mix11 and 1.7% in Mix4, respectively. For eight out of eleven workloads, PIPP improves on UCP performance up to 2.8% in Mix11. However for the remaining workloads, it slightly degrades the cache performance. It can be observed that for Mix3, Mix6 and Mix9, the performance gain of PIPP is similar to UCP, whereas MI2PP improves on both UCP and PIPP for both workloads. This may be because the cache allocation deviations due to the pseudo-partitioning strategy employed by PIPP end up harming the system. Even though the replacement strategy of PIPP is expected to work better than LRU policy in UCP, it evidently does not overcome the defects that PIPP has created. The adaptive insertion and promotion strategies of MI2PP, on top of the benefits it has gained from maintaining the target partitions among the cores, are able to outperform both UCP and PIPP.

Note that PIPP experiences a small deterioration on the performance of Mix4 over

UCP, however MI2PP shows about a 1.5% improvement over UCP in this case and as much as an almost 2% gain over PIPP. However later (Figure 5.6), it will be evident that PIPP has outperformed the UCP in the L2 miss rates improvement of Mix4, possibly by retaining many of the cache lines belonging to the high utility applications in the cache. PIPP continually promotes useful cache lines that will continue to provide cache hits to the higher priority positions and pushes the low utility cache lines that were installed at lower priority positions down the priority ordering. This results in the low utility lines being quickly evicted, allowing the possibility that the cache occupancy of the application(s) in the system may not match the target allocation. This scenario has been discussed in Xie and Loh [2009].

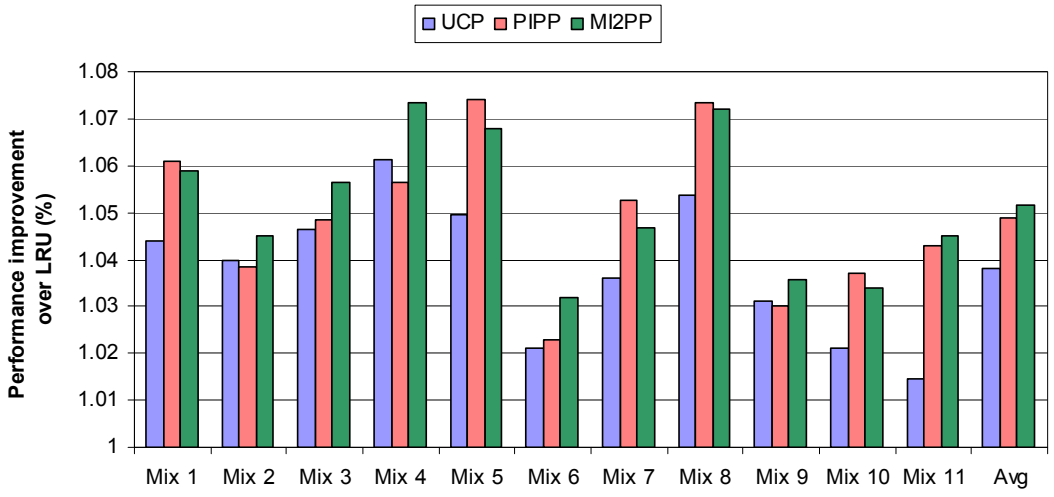


FIGURE 5.3: Performance result for IPC throughput of UCP, PIPP, and MI2PP policy normalised to an LRU-managed cache.

The allocation deviation in the cache could be the reason for the degradation in the IPC performance of PIPP in Mix4 over both UCP and MI2PP where the actual cache occupancy

of all applications always match their target allocations. Due to deviation from the cache ways allocation in PIPP, the increased cache contention could harm the performance of PIPP. This observation explains the importance of sustaining the cache allocation between the competing applications, which may cause significant impact on the overall system performance.

There are a few workloads, namely Mix5, Mix7 and Mix10, in which MI2PP is outperformed by PIPP. The stronger performance achieved by PIPP comes from its effective cache insertion strategy that produces shorter lifetimes for zero-reuse lines than MI2PP. PIPP's promotion policy also keeps the single-reuse and multiple-reuse lines for the high utility applications resident in the cache long enough to expose them to additional hits, but not too long to be evicted once the lines become dead. On average, the proposed MI2PP improves the throughput by 5.2% over LRU policy, about 1.5% over UCP and about 0.4% over PIPP. Most importantly, MI2PP improves significantly on PIPP in a few cases in which PIPP does less well than UCP, but does not significantly underperform on PIPP in any case. It appears that MI2PP may achieve a more consistent performance across different workloads.

Figure 5.4 presents the performance of MI2PP policy in terms of the weighted speedup metric. The values are normalised to the weighted speedup of the baseline LRU policy and MI2PP is compared with UCP and PIPP. MI2PP improves the performance of LRU policy by more than 5% for eight out of the eleven workloads, with the maximum performance gain of almost 9%. Again, MI2PP consistently outperforms UCP for all the workloads by up to 3%. This is contrary to the performance of PIPP, which is slightly worse than UCP for Mix2, Mix4 and Mix9. On average, MI2PP improves the weighted speedup by about 6% over LRU and 1.6% over UCP, but achieved similar performance with PIPP. Again, the

more consistent improvement in performance of MI2PP across all workloads is evident.

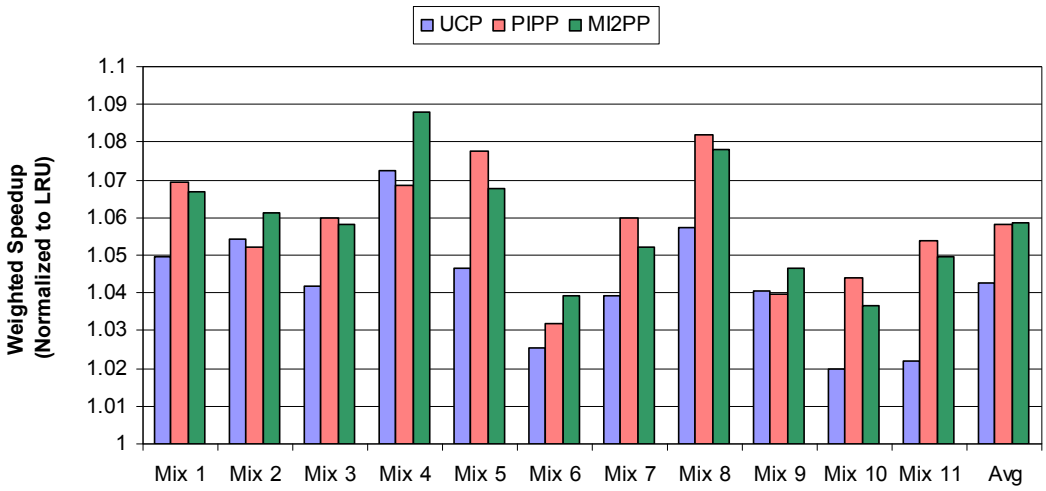


FIGURE 5.4: Performance as measured by the weighted speedups of IPC for UCP, PIPP and MI2PP policy compared to baseline LRU.

The harmonic mean metric was introduced to evaluate both fairness and performance of the simulated workloads in the multiprocessor system. The metric captures the notion of fairness better than the weighted speedup. If one or more applications in the system generate lower IPC speedup, the end results tend to produce small values. Figure 5.5 shows the performance of the UCP, PIPP and MI2PP for the fairness metric normalised to the traditional LRU policy. UCP has an average value of 1.05, while PIPP and MI2PP share a value of 1.07. It is noticeable that MI2PP is as good as UCP for Mix2 and Mix9 and performs better for the rest of the workloads. On average, MI2PP improves the fairness metric of PIPP for seven out of eleven workloads. Thus, MI2PP can improve the performance of some benchmarks compared to UCP and PIPP at the expense of hurting other benchmarks in the simulated workload mixes.

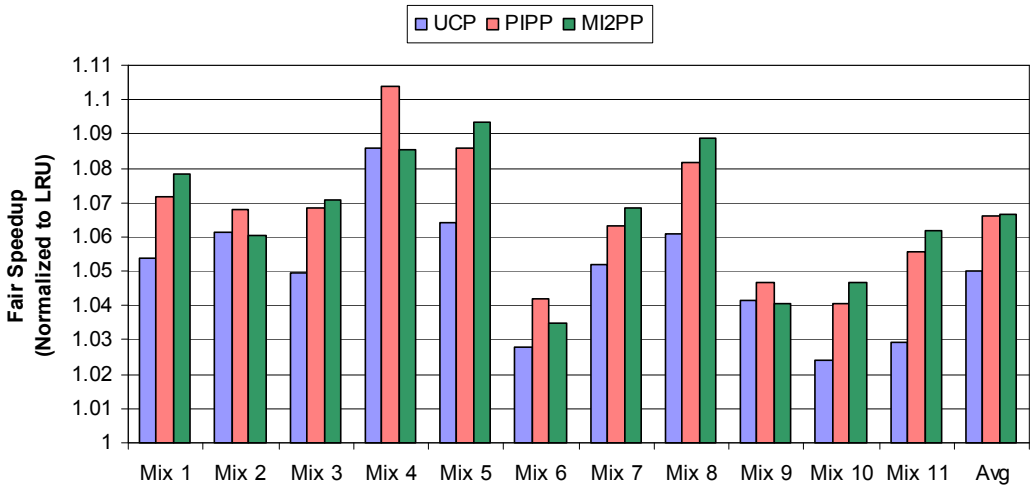


FIGURE 5.5: Comparison of UCP, PIPP and MI2PP policy over traditional LRU for the harmonic mean fairness metric.

In the figure, this metric shows that PIPP never underperforms on UCP, whereas MI2PP does in the case of Mix2, Mix4, Mix6 and Mix9. This is because the IPC performance of one or two applications in the mixes is degraded by MI2PP. Note that in this case, the IPC performance of each application is defined as (IPC_{ISO}/IPC_{CONC}), where IPC_{ISO} is the total IPC of an application if it is executed in isolation and IPC_{CONC} is the total IPC of the same application if it is executed concurrently with other applications in the system. From our observation for Mix4 on MI2PP, the IPC performance of the application executed on core C4 alone was outperformed by PIPP by about 5%. Thus, the core has become the largest contributor to the smaller overall fairness speedup of Mix4 compared to PIPP. Meanwhile, in the case of Mix2, Mix6 and Mix9, it is found that one or two applications of the mixes on MI2PP have each recorded around 1.5% to 3% lower IPC performance than on PIPP. As a result, the fairness among applications in the mentioned workloads on MI2PP is outperformed by PIPP.

Figure 5.6 depicts the reduction of L2 miss rates measured in MPKI (misses per thousand instructions) for all of the eleven workloads. MI2PP improves on the MPKI reduction of baseline LRU by up to 52% and by as much as 10.5% over UCP. The patterns of MPKI reduction for all the simulated workloads are similar to their achieved IPC throughput improvements, except for Mix3, Mix4, Mix6 and Mix11. Figure 5.3 has shown that PIPP benefits Mix3 and Mix6 with small improvements over UCP, but MI2PP further improves on this. Meanwhile, MI2PP performed similarly to PIPP for Mix11 and outperformed the IPC throughput of UCP. However, as can be seen in Figure 5.6, the MPKI reduction of PIPP for these three workload mixes is larger than MI2PP, about 4.5% for Mix3, 2% for Mix6 and 3.9% for Mix11, respectively. This could be due to the dead time of many single-reuse lines and multiple-reuse lines in the workloads that have actually been increased by the promotion policy of MI2PP.

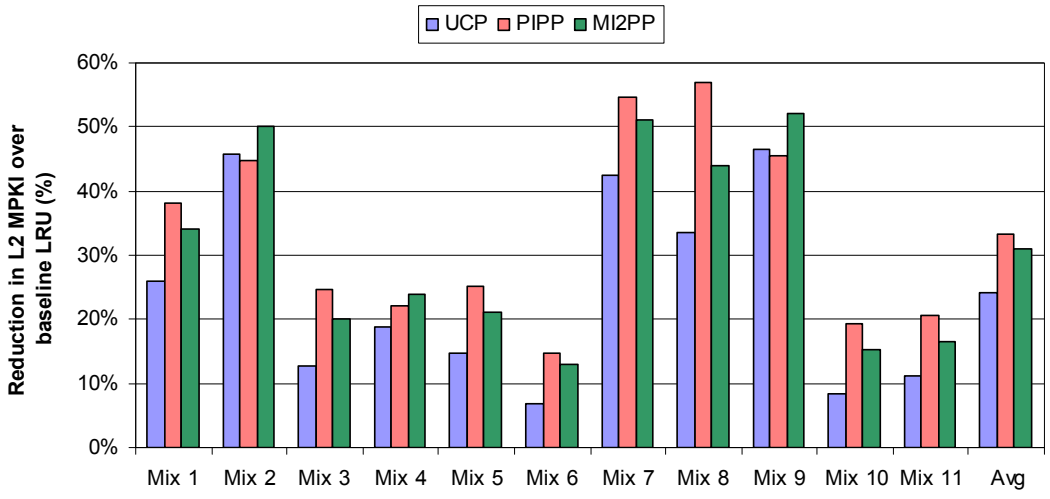
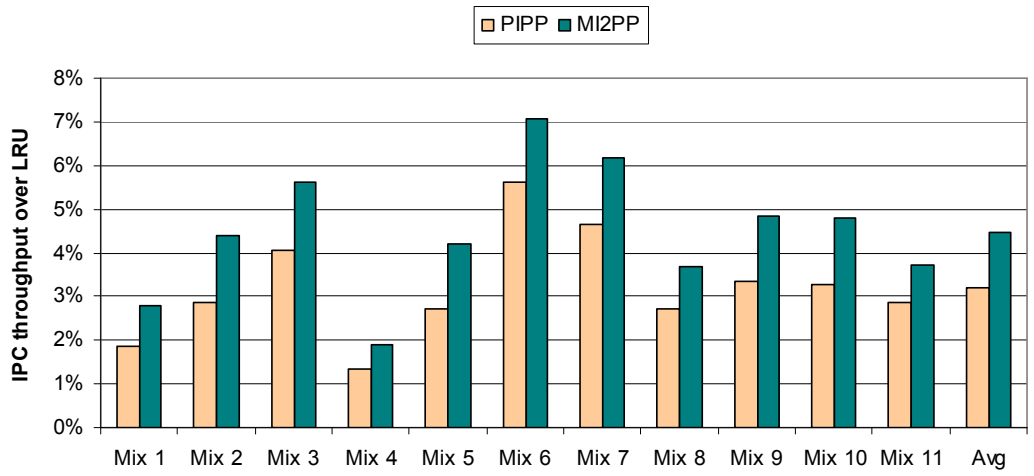


FIGURE 5.6: L2 miss rate in MPKI of UCP, PIPP and MI2PP compared to baseline LRU policy.

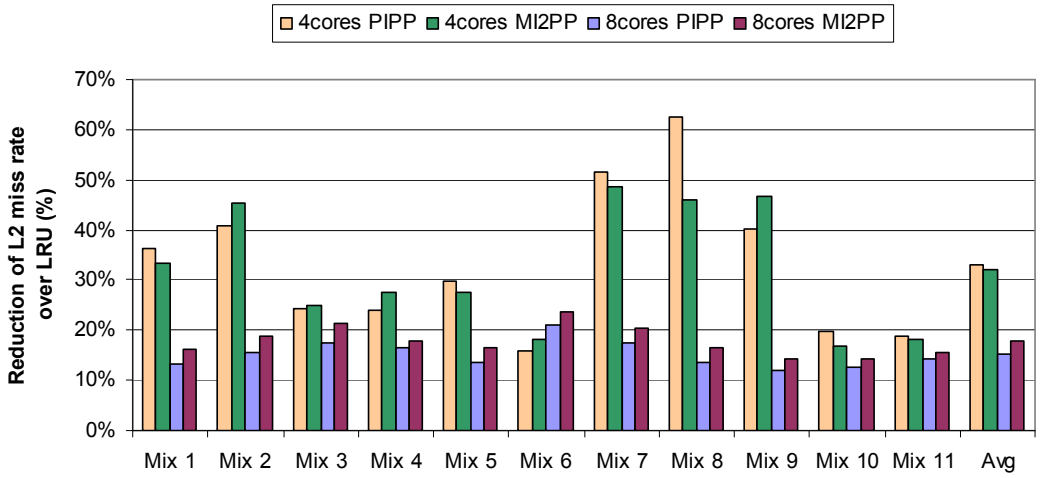
MI2PP consistently improves the cache performance and reduces the L2 miss rate of Mix4 due to its ability to retain the working sets in the cache. The permanent insertion position may be higher than PIPP, as well as the promotion policy, which moves the lines high enough in the cache to allow multiple-reuse lines to be exposed to many additional hits. PIPP on the other hand was outperformed by UCP in the IPC throughput improvement, but is able to achieve a miss rate reduction of about 3.2% over UCP. In this case, PIPP improved the MPKI reduction by retaining many cache lines belonging to the high utility applications in the cache. This is done by installing the lines of low utility applications at lower priority position, close to the LRU position. From this observation, PIPP may hurt the system performance if the number of cores is increased because most of the cache lines will be inserted at the positions very near to the LRU position. On average, MI2PP achieved MPKI reduction of 31% over LRU policy, 6.8% over UCP and a performance penalty of 3% over PIPP.

It is known that accesses to main memory due to misses occurring in the L2 cache require more power/energy dissipation. Hence, the cache miss reductions in L2 reported in Figure 5.6 can be expected to reflect an improvement in power/energy dissipation. As depicted in the figure, MI2PP has significantly reduced the L2 misses over LRU compared to UCP and PIPP. Thus, it can be concluded that power/energy dissipation by MI2PP could also be significantly reduced compared to LRU. However, the reduction may not very significant compared to UCP and PIPP, due to insignificant miss reduction of MI2PP over these two schemes. Note that the amount of power/energy dissipation reduced by MI2PP over the three schemes could be more or less than the percentage of miss rate reduction in L2 cache reported in Figure 5.6. The actual amount of power/energy reduction by MI2PP will be investigated in our future work. Figure 5.7 illustrates the IPC improvement and the reduction of miss rate by PIPP and MI2PP, relative to LRU policy in an 8-core system

sharing 512KB L2 cache.



(a)



(b)

FIGURE 5.7: (a) IPC throughput of the 8-core system, and (b) L2 miss rate of the 4-core and 8-core systems, using PIPP and MI2PP, relative to the baseline LRU policy.

Note that, the miss rate reported in this figure is defined as $\sum_{i=1}^{N} missRate_i / totalAccesses_i$, where $missRate_i$ is the total miss rate of the *i*th application and $totalAccesses_i$ is the total number of L2 cache accesses by the same application. The performance comparison between 4-core system and 8-core system were evaluated to demonstrate whether insertion and promotion policies of PIPP and MI2PP could harm the system if the number of cores is increased. This is because a larger system will increase the competition among cores for the shared cache resources and the intention to retain as much useful data as possible in the cache will become harder to achieve.

Figure 5.7 shows that MI2PP consistently outperformed PIPP for both the IPC improvement (Figure 5.7(a)) and the miss rate reduction (Figure 5.7(b)) of the 8-core system. In several cases, it is observed that MI2PP was outperformed by PIPP in both the IPC throughput (Figure 5.3) and L2 miss rate (Figure 5.7(b)) of the 4-core system. In the 8-core system, MI2PP has achieved better performance than PIPP. In several other cases (Mix2, Mix3, Mix6, Mix9, and Mix11), it can be seen that the IPC improvement differences between MI2PP and PIPP are larger in the 8-core system compared to the performance differences in the 4-core system. On average, MI2PP has improved the IPC throughput of 8-core system by around 1.3% compared to PIPP, while in the 4-core system, the average improvement by MI2PP was about 0.3% better than PIPP.

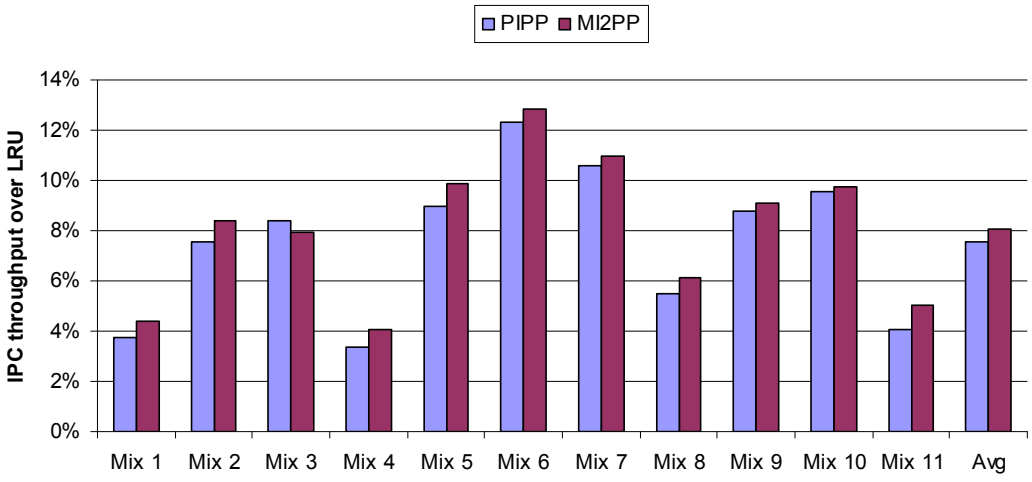
In miss rate reduction, MI2PP was outperformed by PIPP by on average around 1% in the 4-core system. On the other hand, MI2PP has achieved about 3% improvement on average over PIPP, in the 8-core system. Thus, the insertion of new lines at the middle priority position implemented by MI2PP is more effective than insertion at lower priority positions implemented by PIPP. This is because the results presented in Figure 5.7 show that for a large system, PIPP tends to insert new lines of many cores at lower priority

positions than MI2PP which always installs new lines at the middle priority position. PIPP could therefore degrade the system performance compared to MI2PP. Table 5.1 shows the workload mixes used in the 8-core system.

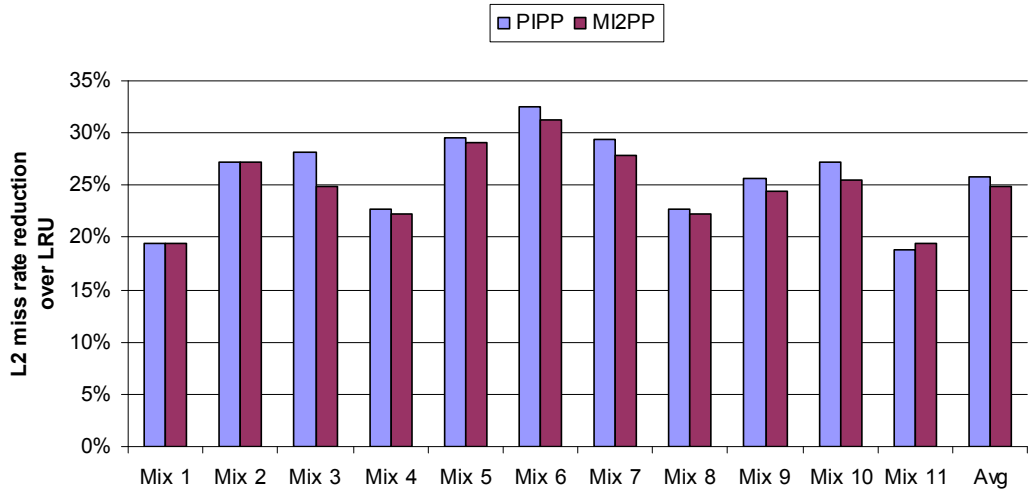
TABLE 5.1: Workload mixes for the 8-core system.

Workload	Applications
Mix-1	pr, bzip2, art, mcf, art, mcf, twolf, bzip2
Mix-2	mcf, twolf, bzip2, vpr, art, twolf, bzip2, mcf
Mix-3	vpr, mcf, twolf, bzip2, equake, twolf, vpr, mcf
Mix-4	art, bzip2, vpr, mcf, equake, art, bzip2, twolf
Mix-5	twolf, mcf, vpr, bzip2, equake, bzip2, twolf, mcf
Mix-6	bzip2, vpr, twolf, mcf, parser, twolf, bzip2, mcf
Mix-7	vpr, bzip2, mcf, twolf, parser, mcf, vpr, bzip2
Mix-8	art, vpr, bzip2, mcf, parser, bzip2, twolf, mcf
Mix-9	twolf, vpr, bzip2, mcf, ammp, vpr, mcf, bzip2
Mix-10	mcf, twolf, bzip2, vpr, ammp, mcf, twolf, bzip2
Mix-11	vpr, bzip2, art, mcf, ammp, twolf, bzip2, mcf

Figure 5.8 shows the performance of MI2PP and PIPP over LRU policy in a 16-core system sharing a 32-way, 512KB cache. The applications of each workload mix used in the 16-core system are the same applications used in the 8-core system, with the number of each application in each workload mix doubled. The system performance illustrated in the figure is in terms of IPC improvement (Figure 5.8(a)) and L2 cache miss rate reduction (Figure 5.8(b)).



(a)



(b)

FIGURE 5.8: (a) IPC throughput, and (b) L2 miss rate of the 16-core system, using PIPP and MI2PP, relative to the baseline LRU policy.

The figure shows that for MI2PP the IPC throughput improvement of the 8-core system relative to the LRU policy is maintained to be better than PIPP for the majority of the workload mixes. Note that, the maximum IPC improvement achieved by MI2PP is about 1% larger than PIPP in Mix2, Mix5 and Mix11. However, it is observed that there is a small

degradation of MI2PP IPC improvement compared to PIPP in Mix3 by approximately 0.5%. This is due to the additional 3.5% miss rate of MI2PP compared to the miss rate of PIPP (Figure 5.8(b)). On average, MI2PP achieved approximately 0.5% improvement than PIPP in improving the IPC throughput of the 16-core system.

For the majority of workload mixes, PIPP has outperformed MI2PP in the L2 miss rate reduction, with the maximum improvement in Mix3, while in Mix1, Mix2 and Mix11, MI2PP achieved similar miss rate reductions. Note that, for a 32-way cache shared by 16 cores, the maximum number of ways that can be allocated to a core in the system is 17 ways while the remaining cores will each receive only 1 way as the UCP partitioning scheme used in the system will always guarantee each core at least 1 way. However, from our observation, this is not always the case but the cores in the 16-core system tend to receive fewer ways compared to the 8-core system. Thus, the insertion at the lower priority position by PIPP in the 16-core system is efficient in quickly eliminating dead-on-arrival lines and single-reuse lines compared to the middle priority position insertion by MI2PP in the 8-core system. In addition, the single promotion strategy of PIPP has an advantage over MI2PP in shortening the age of dead lines in the cache, thereby the cache waste problem can be reduced by evicting the dead lines sooner than MI2PP. The larger miss rate recorded by MI2PP relative to PIPP could also be due to extra misses from the additional instructions executed by MI2PP, which are not executed by PIPP.

Figure 5.9 shows the miss rate distribution of all cores in the system for both PIPP and MI2PP. MI2PP has a noticeably reduced the miss rate of at least one or two cores in the majority of workload mixes relative to PIPP. For Mix11 it can be seen that MI2PP in core C6 has a miss rate reduction of 20% compared to PIPP. It can therefore be asserted that the total IPC throughput improvement of MI2PP is better than that of PIPP.

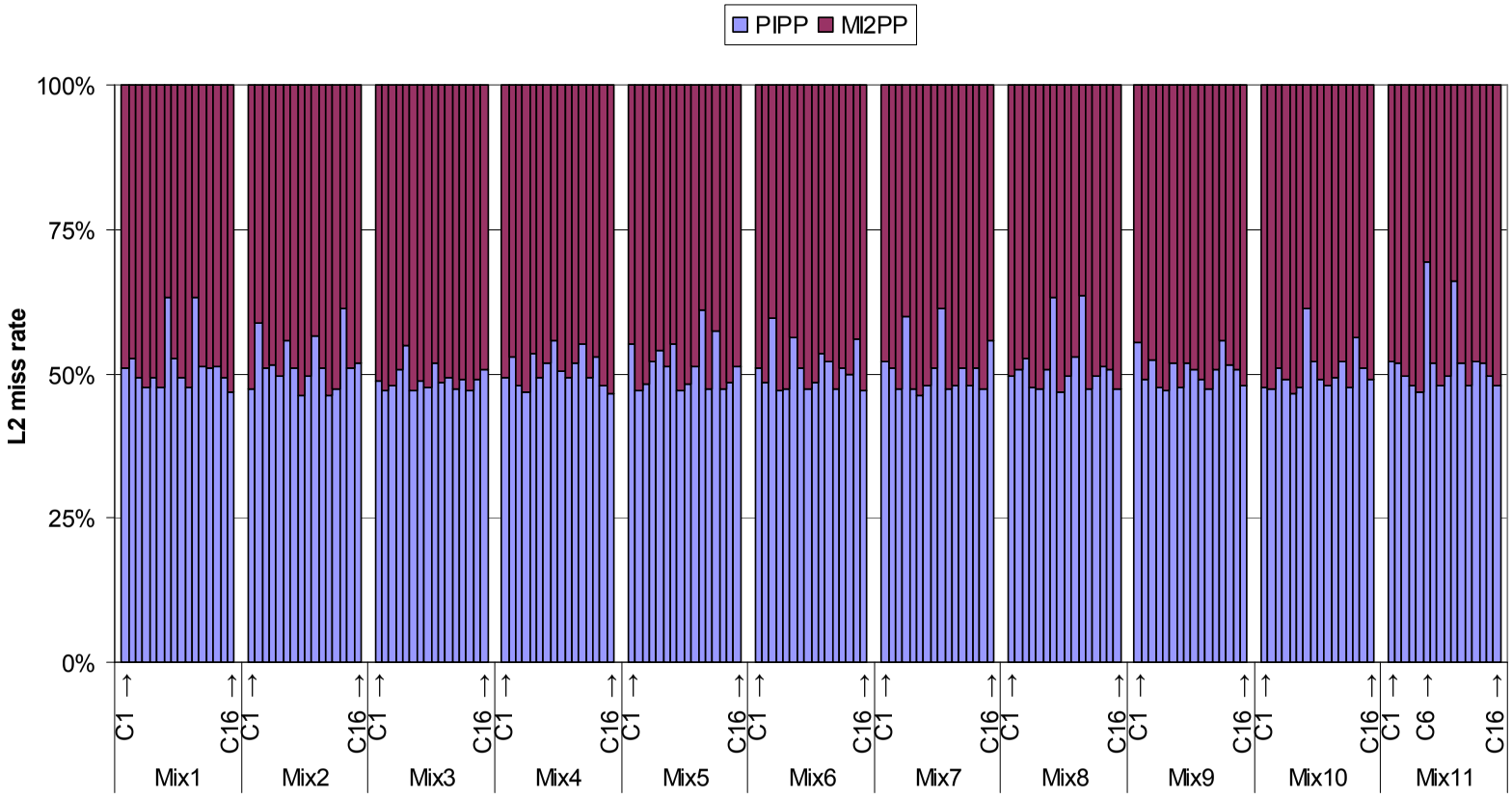


FIGURE 5 9: Distribution of L2 miss rate of each core in the 16-core system.

The 16-core system simulation shows that the IPC throughput of MI2PP is better than PIPP due to the additional instructions executed by MI2PP. However, MI2PP was outperformed by PIPP in reducing the L2 miss rate because of its insertion and promotion strategies. However, the miss rate reduction by MI2PP is better in the 8-core system compared to PIPP. This could be due to the larger number of ways allocated to each core in the 8-core system compared to the 16-core system. Therefore, MI2PP could perform better in the 16-core system if all cores are allocated with more cache ways. This could be done by increasing the associativity of the shared cache. As a conclusion, MI2PP is observed to be able to maintain its performance over PIPP in a larger system. However, for MI2PP to achieve better performance in terms of both IPC throughput and L2 miss rate compared to PIPP, all cores in the system could be arranged in clusters. For example, in the 16-core system, all the cores can be arranged in the clusters of 8 cores. This is because the performance of MI2PP in the 8-core system shows better improvement in both IPC throughput and L2 cache miss rate, relative to PIPP.

In order to further improve the performance of the multiprocessor system, the use of pre-execution profiling in partitioning the L2 shared cache is beneficial in exploring all possible static partitioning decisions of the shared cache. This is because profiling will search the entire cache space thoroughly before the partition decision for each core/application in the system, which are tightly dependent on system's applications and cache size can be determined. Hence, it is expected that the multiprocessor system throughput can be improved. However, implementation of MI2PP with shared cache partitioning decisions obtained via profiling would not be expected to make significant improvement to the MI2PP performance as the adaptive insertion and promotion strategy of MI2PP does not directly depend on the changes in the cache partition size of each core/application during runtime. Thus, pre-execution static profiling is not expected to be of

significant benefit in a system using MI2PP.

Conversely, a compiler assisted cache partitioning is expected would help increasing MI2PP performance by having less complexity and time taken to calculate new partition sizes of each core/application dynamically. Special code that will monitor cores and applications' behaviors at runtime would be used by a compiler so that new partitions can be determined by the code rather than by a dedicated partitioning hardware. For that reason, additional hardware cost and power consumption in the system would significantly reduce. Hence, the dynamic partitioning strategy by the compiler assisted cache partitioning may become superior to the pre-execution static profiling. However, the benefit of compiler assisted cache partitioning may not directly affect MI2PP performance, but the optimized partitioned shared cache by the compiler would help to increase the effectiveness of MI2PP in the partitioned cache. Note that even though the compiler assisted cache partitioning and pre-execution profiling would improve MI2PP performance, this can only be proved by performing experiment simulations. Thus, the effects of these techniques in a system using MI2PP will be explored in future work. Overall, MI2PP could achieve better performance in a larger system, but could need some considerations of the shared cache associativity or the core arrangement in the system, as well as the necessity and advantage of implementing the pre-execution profiling and compiler assisted cache partitioning.

5.3.4 Hardware Overhead

Since UCP is used in the implementation of the MI2PP policy, the Utility Monitor (UMON) circuit containing the shadow tags of 32 sampled sets is maintained for each core in the system. The UMON circuits are used to record the utility information of the shared cache, which is accessed by each particular core. Each UMON requires an adder (used to update the hit counters upon every cache hit), which will be used to create the utility curve for

cache partitioning purposes. Additionally, a shifter is necessary to halve the counters after each partition decision is made. Therefore, each UMON associated with each processor in the system has increased 0.39% of the L2 cache area (assuming 32-bit tag entries). Table 5.2 shows the detailed calculation to determine an estimation of the hardware overhead. In addition to the storage overhead of the conventional LRU replacement policy, MI2PP needs to store the owner of each line in the cache. Thus, for a system with N cores and ω-ways set associative, an additional $\log_2 N$ bits and ($\omega * \log_2 \omega$) storage are required. The accumulated information will be used by a comparator to identify a victim for replacement upon a cache miss. For 32-way set-associative cache, each core requires a 5-bit counter to record the number of cache ways belonging to the core and another 5-bit register to store the number of target partitions defined by UCP, as well as to store the predetermined insertion position of new lines. In total, the hardware overhead of the system due to the monitoring components is 9.5KB, a 1.56% increment in the L2 cache area.

TABLE 5.2: A hardware cost of an UMON circuit associated to a single processor.

Shadow tag array per cache set (18 bits * 32 ways)	**72B**
Sampled set size (1024/32 sets * 72B)	**2,304B**
Hit counters overhead (32 counters * 4B)	**128B**
Total UMON overhead (2,304B + 128B)	**2.375KB**
Total bits of a L2 cache tag entry (18-bit tag + 1 valid bit + 1 dirty bit + 2 recency bits + 2-bit for core ID)	**24 bits**
Area of baseline L2 cache (96KB tag + 512KB)	**608KB**
% increase in L2 cache area due to a single UMON circuit	**0.39%**

Aside from the small increment in the hardware overhead of the system, MI2PP is significantly better than LRU and PIPP in terms of the replacement logic complexity. Table 5.3 shows the number of bits in the priority stack that must be updated during a hit in a cache. For a 32-way cache (i.e. *A=32*) shared by two cores, LRU policy requires significantly high number of bits to be updated compared to PIPP and MI2PP. However, as MI2PP promotes a hit line one extra position compared to PIPP, the difference in the number of bits to be updated in the priority stack for the affected lines by both policies is very small. That is PIPP requires 10 bits to update priority stack of two affected lines, whereas MI2PP needs to update 15 bits of three affected lines.

TABLE 5.3: Replacement logic complexity of the LRU, PIPP and MI2PP policies.

Event	**Replacement logic**		
	LRU	**PIPP**	**MI2PP**
Promotion	$A \times \log_2(A)$	$(1+1\,position) \times \log_2(A)$	$(1+2\,positions) \times \log_2(A)$
Insertion	$A \times \log_2(A)$	$(Partition\_size) \times \log_2(A)$	$\left(\frac{A}{2}\right) \times \log_2(A)$

On the other hand, the insertion of a new line in the cache requires MI2PP to update only half number of bits that must be updated by LRU. This is due to the advantage of insertion at the middle position applied by all cores in the system using MI2PP. Hence, MI2PPP needs to update 80 bits compared to 160 bits by LRU. Conversely, the insertion position of PIPP, which depends on the cache partition sizes of different cores in the system, requires different number of bits to be updated on the arrival of a new line in the cache. In other words, the number of bits to be updated because of the insertion of a new line depends

on the partition size of the miss-causing core. In a worst case scenario, in which Core1 receives 31 ways and Core2 gets only 1 way, PIPP needs to update 155 bits for Core1. However, in the best case for PIPP in which each core receives the same number of cache ways (i.e. $A/2=16$ ways per core), the number of bits to be updated would be similar to MI2PP. Additionally, when the number of cores in the system is increased, the number of bits that must be updated on the arrival of a new line would be less than MI2PP as it will always insert a new line in the middle priority position. Therefore, the complexity of MI2PP is about the same as PIPP, and it is expected that on average there would be no significant difference in the replacement logic complexity between both policies.

5.3.5 Performance Analysis

5.3.5.1 Comparison with LRU, UCP, and PIPP

The performance benefits of MI2PP policy appear to demonstrate that the adaptive replacement policy is able to improve the overall performance of the baseline LRU replacement policy, as well as UCP. The evaluation of MI2PP across all the workloads compared with the LRU policy provides evidence that MI2PP is good at minimising the length of retention of dead lines in the cache. Hence, MI2PP reduces the potential for cache waste that leads to significant cache misses. In general, the insertion policy of MI2PP is beneficial in its treatment of dead-on-arrival lines, reducing the amount of time they spend in the cache. Along with the two positions promotion strategy, MI2PP also offers a shorter distance for single-reuse lines and multiple-reuse lines to traverse towards the lowest priority position for eviction. However, this latter benefit of MI2PP's promotion policy is only applicable if the reused lines become dead at any position lower than MRU position. Otherwise, MI2PP would not give any advantage over the hit promotion strategy of the original LRU policy.

The distribution of total cache misses among all of the cores in the system gives a better explanation of how MI2PP provides significant improvement to the original LRU policy. For all the workloads shown in Figure 5.10, it can be seen that there is at least one core that achieves significant benefits from MI2PP, while the remaining cores in the system do not demonstrate a large margin of improvements. This could be due to the different number of instructions executed by each core, which may cost additional misses from the extra executed instructions. Significant benefits gained by the core(s) in every workload combination indicate that both insertion and promotion policies of MI2PP is better than LRU in reducing the total miss rate of the cache.

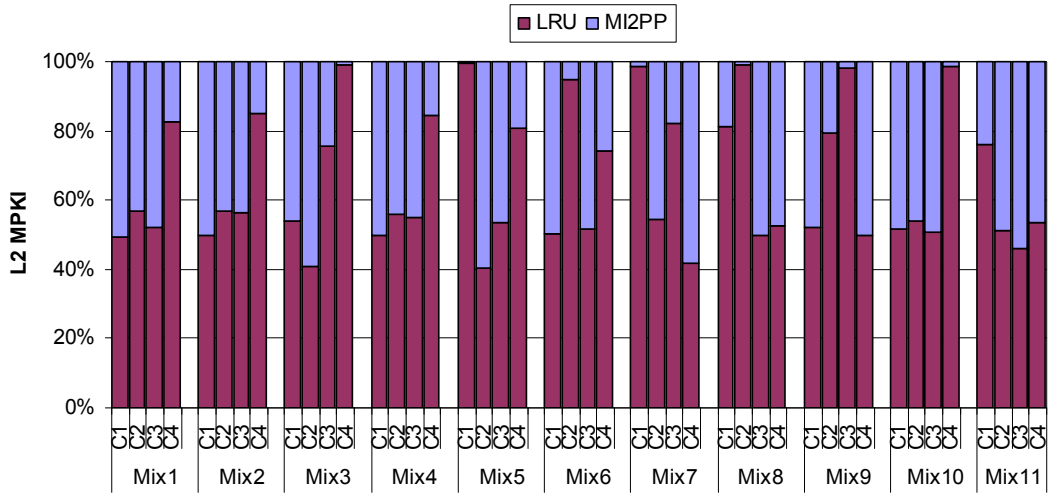


FIGURE 5.10: Distribution of L2 miss rate between MI2PP and LRU.

From the results that have been presented so far, the relative performance variations observed between MI2PP and PIPP may be caused by several possible reasons. PIPP employs a variety of insertion positions based on the output of UCP. Inserting new lines at

lower priority positions gives an advantage to PIPP in evicting dead-on-arrival lines more quickly than MI2PP. However, insertion at a higher position than the predetermined insertion position of MI2PP would cause PIPP to increase the age of zero-reuse lines in the cache, thereby incurring more cache misses. For single-reuse or multiple-reuse lines, the insertion at lower priority positions causes the lines to generate many misses on their subsequent references. This is because the lines have been replaced before their next references. Insertion at higher priority positions is desirable to overcome this deficiency. Figure 5.11 depicts the frequency with which PIPP installs new lines at priority positions very close to LRU position. For a 32-way associative L2 cache shared by four cores, positions with priority 1-8 are considered as the closest positions to the lowest priority position. The figure shows the number of cores in the first 15 execution intervals that have been selected by PIPP to use the lower priority positions as insertion position.

The conclusion from the figure is that PIPP tends to insert new lines at lower priority positions than MI2PP, thereby providing benefit to the dead-on-arrival lines. For the cores that are running application(s) with many zero-reuse lines, this could be the reason why in some cases PIPP shows it had better overall miss rate reduction than MI2PP. This also could be another reason for the better overall IPC throughput achieved by PIPP compared to MI2PP in some cases. On the other hand, the insertion at lower priority positions that provides shorter distance for new lines to travel towards the lowest priority position than MI2PP has quickly pushed the new spatial-locality lines towards the end of priority ordering for eviction while the temporal-locality lines have been continually promoted to the higher priority positions. This has allowed application(s) with more temporal-locality lines in the cache to steal cache capacity from the application(s) with more spatial-locality lines. As a result, the cache allocation deviations at runtime could be the reason for the degradation in the performance of PIPP compared to MI2PP.

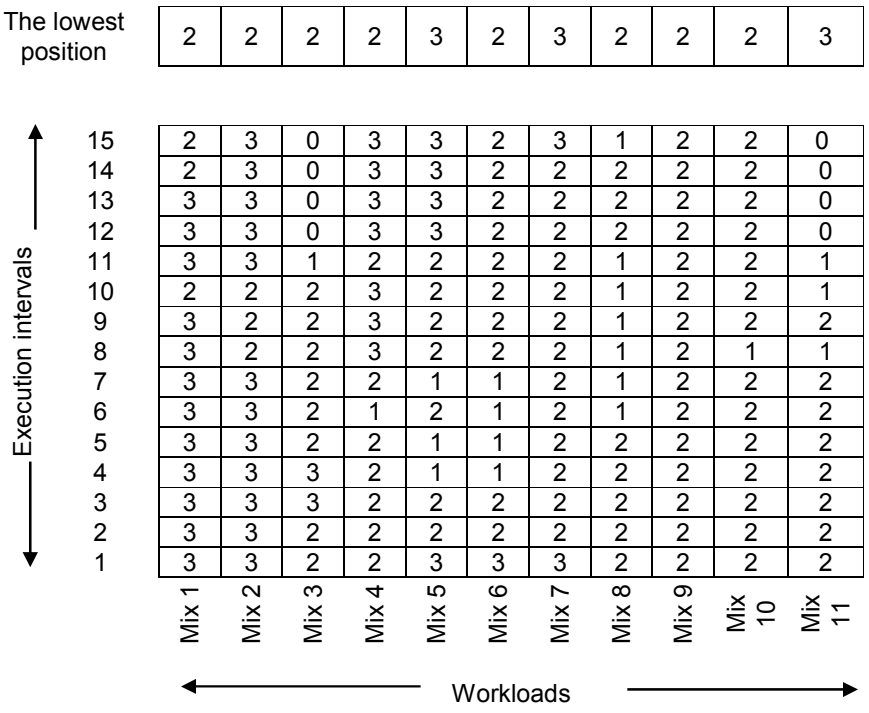


FIGURE 5.11: Number of cores using lower priority positions as the insertion location.

Another possible reason for the inconsistent improvement of MI2PP over PIPP could be the promotion policy of PIPP, which is not effective for the lines with many references at a far distance. This means that PIPP is unable to keep the lines long enough for additional hits. For such situations, MI2PP provides more chances for the lines to be exposed for extra hits and could reduce the miss count. Unfortunately, MI2PP could also occasionally increase the amount of time taken for the lines to move towards the end of the priority stack after the lines become dead. Figure 5.12 depicts the L2 miss rate of each core in the system implementing UCP, PIPP and MI2PP.

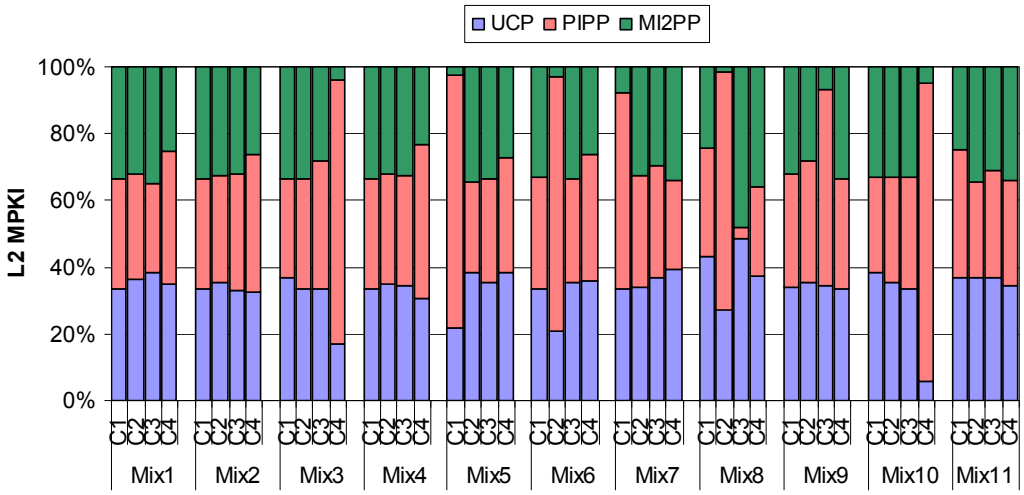


FIGURE 5.12: Distribution of L2 miss rate among UCP, PIPP and MI2PP.

According to the results obtained from all the simulations and the preceding explanations, it appears that MI2PP was outperformed by PIPP due to the differences in managing the insertion positions of new incoming lines and the promotion policy implemented in the system. The predetermined insertion position of MI2PP has pros and cons compared to PIPP, but overall, it is still considered to be better than PIPP because it will never give any chance for the new lines to be installed at any positions very near to the MRU or LRU position. In addition, the two position promotion policy of MI2PP is able to provide more chances for single-reuse and multiple-reuse lines to manifest extra cache hits before evictions compared to PIPP. Note that Xie and Loh [2009] claimed that their PIPP is as good as or better than UCP in controlling the cache occupancy. This can be seen in the results presented so far. MI2PP has shown that by enforcing the appropriate cache partitions, the total improvement can sometimes surpass the performance of PIPP over UCP and tends to be more consistent across the mix of workloads.

5.3.5.2 Comparison with another cache partitioning scheme

The MI2PP policy was proposed to improve the performance of a partitioned L2 shared cache. Therefore, as well as the utility-based cache partitioning scheme proposed by Qureshi and Patt [2006], MI2PP was compared against a CPI-based cache partitioning scheme introduced by Muralidhara et al. [2010] to observe its abilities in a system using a different type of partitioning scheme. For a quad-core system sharing a L2 cache, the CPI-based cache partitioning scheme was used to distribute the cache space among the competing cores. Two systems have been evaluated: (1) a system using the original CPI-based cache partitioning scheme, as explained in Chapter 4 and (2) a system employing MI2PP as the cache replacement policy. The baseline results used in this section were retrieved from the works presented in the previous chapter, particularly the results under section 4.3, "CPI-based Cache Partitioning Scheme".

The performance improvements gained by MI2PP compared to a system using the CPI-based partitioning scheme is illustrated in Figure 5.13. It can be seen that MI2PP consistently improves on the performance of the CPI-based partitioning scheme, in terms of all of the previously defined metrics. However, there is inconsistency particularly in the fairness speedups across all of the simulated workloads when compared to the improvement in overall IPC throughput. The performance improvements measured in the harmonic mean metric were expected to be higher than total IPC throughput because all of the applications in each workload gain benefits from IPC throughput and thus tend to produce a large value of harmonic mean.

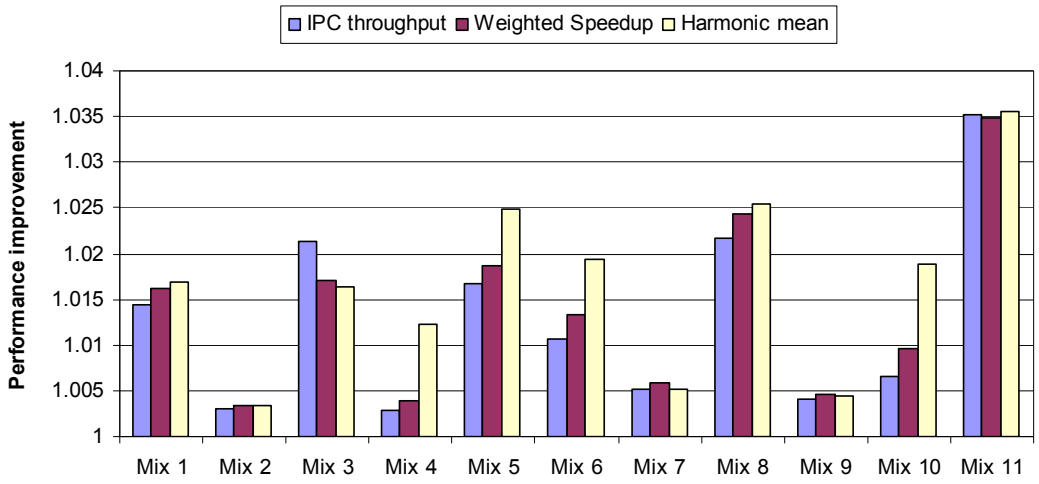


FIGURE 5.13: Performance gain of MI2PP over a system using CPI-based cache partitioning scheme.

The results show that in Mix3 the IPC speedups of all the applications are not fairly improved. This means that one or more applications in Mix3 have generated lower IPC speedup and in turn degrade the fairness of the overall system performance. The results also show that six out of eleven workloads yield better fairness speedups among applications in the workloads, whereas the remaining workloads demonstrate no significant difference between the improvements to the IPC throughput, weighted speedup and harmonic mean. From the figure, it can be concluded that MI2PP consistently benefits the IPC throughput of the system using a CPI-based partitioning scheme, with the greatest improvement of 3.5% and can yield similar or better performance to the harmonic mean of the system with an average of 1.7%.

Figure 5.14 presents the miss rate reduction (measured in number of misses per total number of cache accesses) between a system using the original CPI-based partitioning scheme and a system using MI2PP. The performance of both systems was normalised to an

unmanaged system using the LRU replacement policy. It is observed that the ability of MI2PP to reduce the miss rate of the shared cache across the simulated workloads is maintained from the performance of MI2PP on the system using UCP, as shown in Figure 5.6. That is, MI2PP exhibits consistent significant reductions in the miss rate yielded by the system using a CPI-based cache partitioning scheme. There is a maximum value of 34.7% reduction over the CPI-based partitioning scheme and an average of 12%.

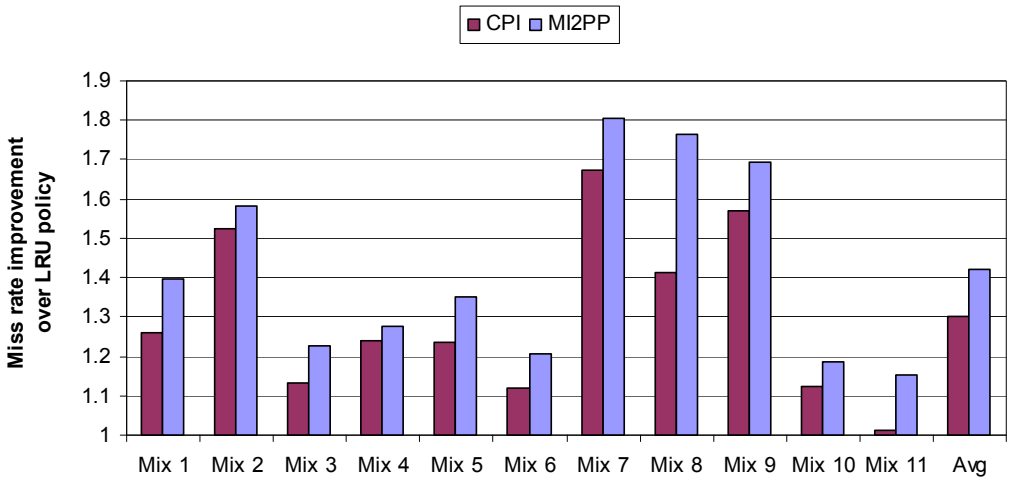
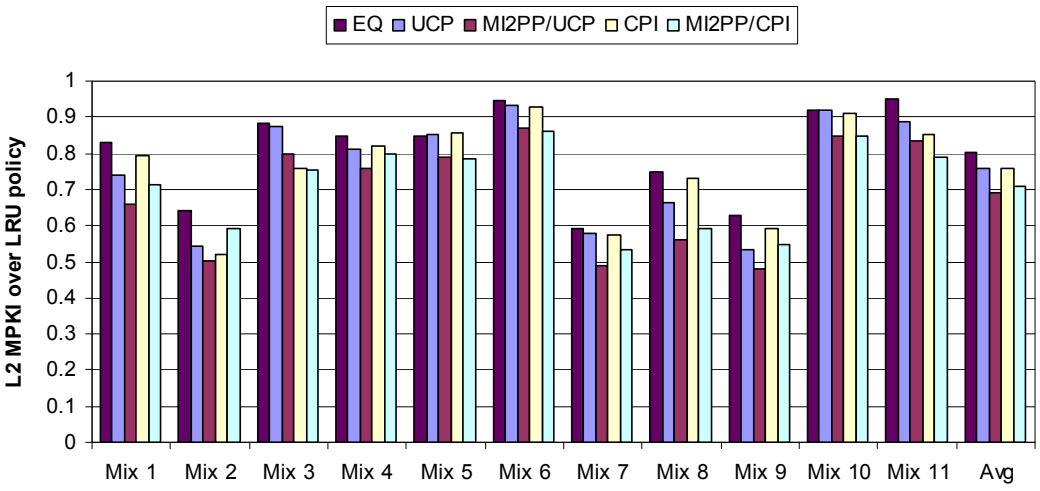


FIGURE 5.14: Total miss rate reduction normalised to the original LRU replacement policy.

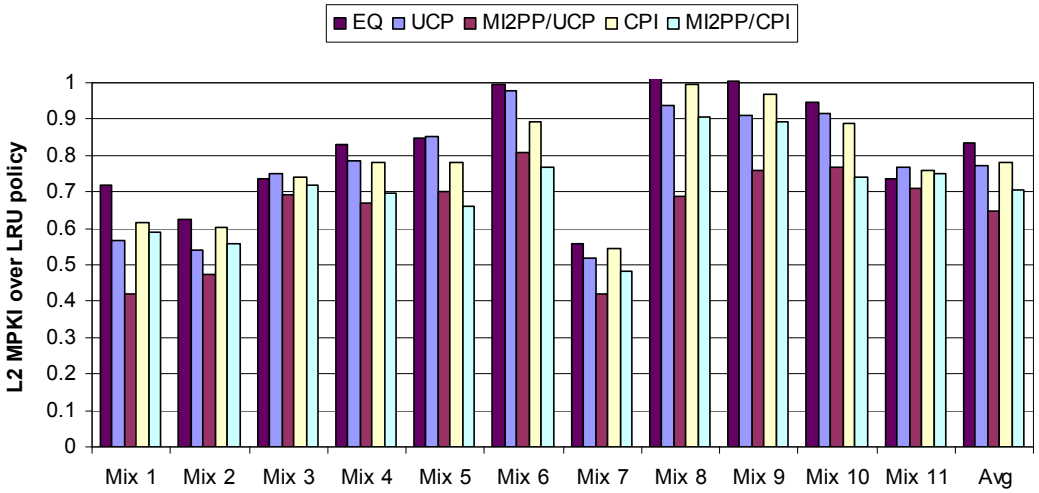
5.3.5.3 Cache size sensitivity study

To study the sensitivity of MI2PP policy to different cache sizes, the eleven workload combinations were simulated in a system with a shared cache size of 1MB and 4MB. Comparisons were made with the statically (equal) partitioned shared cache (as presented in Chapter 4), as well as the original UCP and CPI-based partitioning schemes. The miss rate of MI2PP policy in a 512KB L2 cache is shown in Figure 5.15(a) with the results of the

system using the other cache partitioning schemes.



(a)



(b)

FIGURE 5.15: Miss rate relative to the baseline LRU policy of a system with shared cache sizes of (a) 512KB, and (b) 1MB using different cache partitioning schemes and MI2PP policy.

It can be seen that for the majority of the simulated workloads, MI2PP further improves on the performance of the other cache partitioning schemes. However, the implementation of MI2PP in the system using the CPI-based partitioning scheme (MI2PP/CPI) exhibits slight improvement to the performance of Mix3 and yields about 8% more misses for Mix2. As depicted in Figure 5.15(b), when the cache size is increased to 1MB, both Mix2 and Mix3 show a noticeable reduction in the cache miss rate. Moreover, MI2PP/UCP has significantly reduced the cache miss rate for several workload mixes, with a maximum reduction of 25%. The average cache miss rate incurred due to MI2PP/UCP for the cache size of 1MB is decreased by almost 15% compared with the cases when MI2PP is not used. This value doubled the reduction of misses saved by MI2PP in the 512KB L2 cache.

To further investigate the effects of MI2PP towards different cache sizes, this new replacement policy is employed in a 4MB L2 cache that used the UCP scheme to partition the shared cache resources among four competing applications in the system. Figure 5.16 shows the overall performance of MI2PP/UCP in reducing the miss rate of the partitioned shared cache, relative to UCP, over different cache sizes. As can be observed from the plot, the implementation of MI2PP/UCP in all of the cache sizes has demonstrated a reduction in the number of cache misses for all of the simulated workloads. As expected, for the 1MB cache the miss reductions are better compared to 512KB cache, but for 4MB cache the reductions sometimes are lower than 1MB cache.

The figure also shows that MI2PP/UCP reduces the miss rate of the 4MB cache by an average of 23% even though it yields less than 10% of reduction in four of the workload mixes. This may be due to the large cache size, which the UCP scheme has distributed among the competing applications in the system being sufficient to accommodate the working set of the application(s). In this case, MI2PP could not provide further significant

improvement to the cache that has been well managed by the UCP scheme in the context of minimising the total number of cache misses. Overall, it can be clearly seen from the figure that the average miss rate reduction of all the simulated workloads generally increases with the increasing cache size, although the miss reduction of some workloads may not follow the cache size. This suggests that MI2PP/UCP may be more effective in reducing miss rates than UCP in a system constrained by cache size.

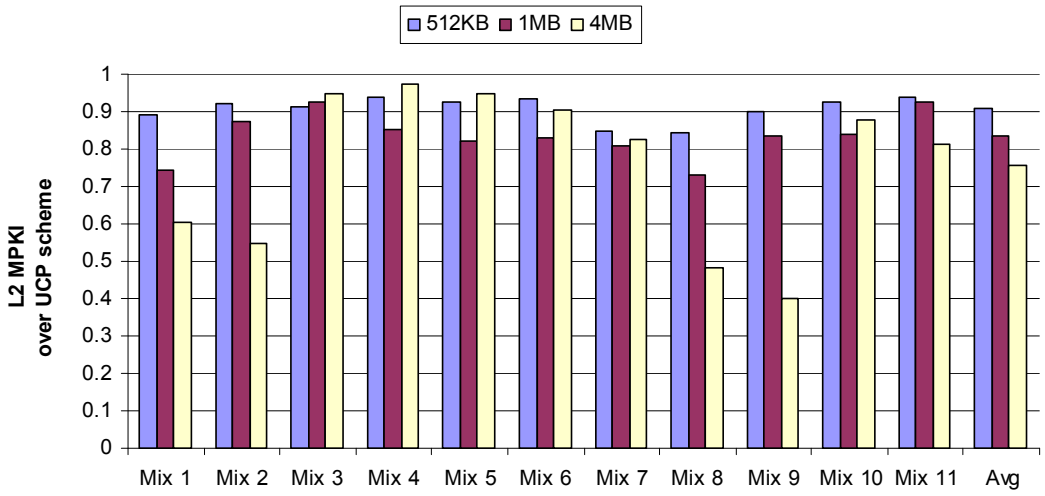


FIGURE 5.16: Miss rate, relative to UCP, of MI2PP/UCP policy implemented in different cache sizes.

From the results presented so far, it is interesting to observe that the average reduction in the L2 miss rate due to the implementation of MI2PP in the system using UCP is better than in the system using a CPI-based partitioning scheme. The reason could be due to the methods implemented by both cache partitioning schemes in making their decisions to partition the shared cache. While the CPI-based partitioning scheme uses CPI values to decide the target partition for each of the competing cores in the system, UCP takes miss

counts and hit counts of the shared cache into consideration to determine the partition allocations. The latter method obviously attempts to directly reduce the number of cache misses in its partitioning strategy. This is not the case in the CPI-based partitioning scheme. Therefore, the CPI-based cache partitioning scheme does not significantly assist MI2PP to save more cache misses in the system in the way that UCP does. The results in this section also demonstrate that UCP can outperform the scheme proposed by Muralidhara et al. [2010] in minimising the number of misses incurred in the L2 shared cache. From all the comparisons that have been made in this section, the impact of MI2PP in minimising the cache miss counts is significant in the partitioned cache, even though the IPC improvement variation shows lower level of improvement.

5.3.5.4 Implementation in the unpartitioned shared cache

Even though the MI2PP policy achieves better improvements in reducing the miss rate in a partitioned shared cache, it is informative to investigate its performance in an unpartitioned cache – that is, to see how well it performs over the baseline LRU policy. The miss rate reduction of MI2PP over LRU policy for 512KB L2 cache is shown in Figure 5.17 and as can be seen, MI2PP performs better than LRU for most of the simulated workloads, with the maximum achievement of 13%.

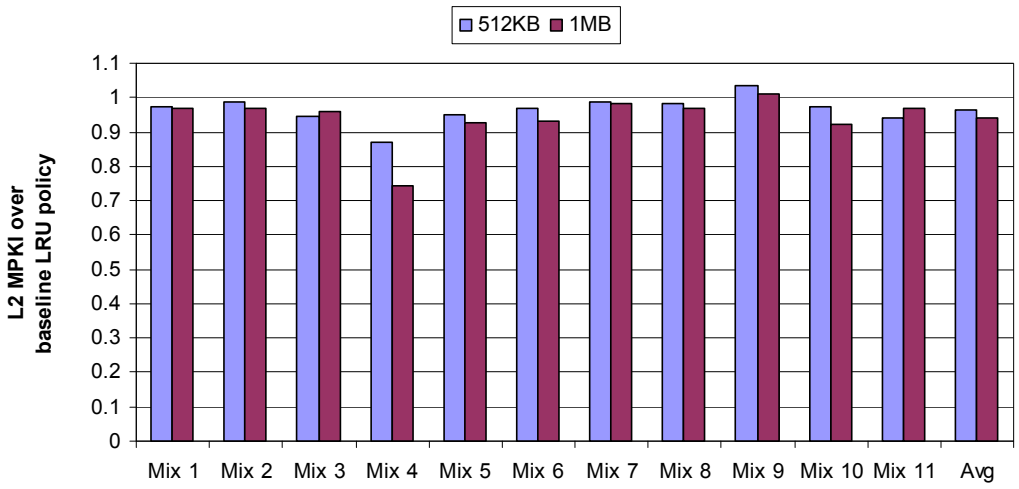


FIGURE 5.17: Miss rate reduction of MI2PP policy implemented in different sizes of unmanaged shared cache.

In the figure, MI2PP incurred about 3% more misses than LRU policy for Mix9. It is found that the only low utility application in Mix9, namely "equake" has incurred around 18% more misses than LRU policy and generated the highest number of misses compared to other high utility applications in the mix. This could be due to multiple-reuse lines of the low utility application that are quickly pushed to the lowest priority position because of the aggressive competition among the high utility applications that were trying to promote their lines at the higher priority positions. The multiple-reuse lines of the low utility application were evicted before their next references, incurring significant additional misses in Mix9 compared to the LRU policy.

To further study the sensitivity of MI2PP to cache size, the performance of MI2PP when it is employed in the 1MB unpartitioned shared cache is also plotted in Figure 5.17. It is shown that in general, MI2PP slightly outperforms the original LRU policy for most of the simulated workloads in both 512KB and 1MB L2 caches and, as expected, MI2PP

further reduced the miss rate when the cache size is increased from 512KB to 1MB.

5.3.6 Conclusion

The existing cache partitioning schemes used in the studies discussed in this section are generally very effective in managing the cache resources among different applications executed in the system. However, the benefit that each application can get from the given amount of cache space may be further enhanced if the cache lines of the applications can be retained long enough in the cache so that more cache hits can be generated. In this section, the MI2PP policy is proposed to provide more benefit to a partitioned shared cache. The effect of this new cache replacement policy shows that MI2PP is able to enhance the overall performance (through a reduction in miss rate) of the system by retaining the portion of the cache resources of each application in the cache long enough to exhibit additional cache hits. The MI2PP policy also shows that it can produce further improvements over well-accepted cache partitioning schemes such as UCP and maintains its benefits across various cache sizes. The adaptive insertion and promotion policies employed in the MI2PP policy can enhance the overall system performance significantly compared to the LRU policy implemented in either an unmanaged cache, or a partitioned shared cache. Finally, the proposed MI2PP is also able to achieve better throughput compared to pseudo-partitioning schemes such as PIPP.

However, the several abovementioned limitations of MI2PP that sometimes cost degradation in system performance require some improvements in order to make this new cache replacement policy more robust. It is important to understand the effect of access patterns of all the workloads on each other to achieve better overall use of the cache space. As a result, if there is any enhancement to be made on MI2PP, the improvement is likely to be some form of dynamic promotion policy.

5.4 PARTITION-BASED REPLACEMENT POLICY

5.4.1 Description

One of the methods used to determine a victim for replacement in a partitioned shared cache is by selecting the LRU line in the set, or the LRU line of the miss-causing application/core. The decision on which of these methods is used is usually made according to the number of cache ways allocated by the cache partitioning scheme, as well as the number of ways currently belonging to each core in the system. However, if the total number of ways owned by the miss-causing application is smaller than that allocated, selecting the LRU line in a set that belongs to another core creates some drawbacks. Assume that the selected LRU line in the set belongs to the application with the lowest allocation. The respective core may continuously suffer from insufficient cache space and thereby compromise overall cache performance.

The Partition-based Replacement Policy (PRP) is proposed and presented in this section to investigate the impact of an adaptive eviction policy, introducing the consideration of whether an application is over-allocated as another metric to determine the victim in the cache replacement strategy. In general, an application running on an over-allocated core may indicate that the working set of the application exceeds the allocated cache space, therefore tending to cause line stealing from other cores in the system. If the cache replacement policy fails to maintain adequate cache resources of cores with lower allocations due to the conflict created by the over-allocated core, issues such as cache thrashing may further hurt the overall performance of the core, as well as the whole system.

As a result, a new way of managing the shared cache particularly in the context of the insertion decision is introduced in PRP. The insertion position depends on the victim

selection performed upon a cache miss. Assume the number of cache ways assigned to the miss-causing application is smaller than was allocated, PRP will then select an evictee from the over-allocated core. The new incoming line will then be installed at the highest priority position to assist the under-allocated core (the miss-causing application) to retain its lines in the cache for a longer period of time. The lines of the under-allocated core can therefore be protected from being evicted too soon and could be saved to expose additional hits. Conversely, if the miss-causing application has a larger number of lines than it has been allocated, the LRU of that application will be selected as an evictee. In that case, the insertion of the new line will be performed at the lowest priority position.

The selection between LRU and MRU positions in the insertion policy of PRP is similar to the Dynamic Insertion Policy (DIP) proposed by Qureshi et al. [2007] but differs in the metric used for the selection decision. Since PRP uses the allocation status to decide the insertion position, it requires no additional information, such as the number of misses incurred by all applications, to choose the appropriate insertion position. All the information gathered by the cache partitioning scheme and the traditional LRU policy is sufficient for PRP to decide on the insertion position.

The promotion policy implemented in PIPP is employed and maintained by PRP. Upon a cache hit, the hit line is automatically shifted by a single priority position towards the MRU position. The intention of the single step promotion is to cluster the lines near the low priority positions, mainly to reduce the residence time of dead lines. Note that while our proposed two-priority positions promotion policy of MI2PP seems to have been beneficial to system performance, it is not implemented in PRP so that the effects of PRP's replacement policy can be isolated and evaluated against MI2PP.

5.4.2 Evaluation

The proposed PRP was evaluated on a quad-core system sharing a 512KB L2 cache that has been partitioned using a UCP scheme. Figure 5.18 shows the performance (in terms of IPC throughput) of PRP over a system with a shared cache using a normal LRU policy and UCP. For all the simulated workloads, PRP achieved performance improvements over LRU with the maximum improvement of about 7% in Mix4. This performance gain is slightly lower than the proposed MI2PP described in the preceding sections. On average, PRP outperformed LRU by 4.2% and for the majority of cases, the performance of PRP was slightly improved by an average of 1% over UCP. However, there are a few cases the overall performance gained no benefit or degraded, with the maximum degradation of 0.8%.

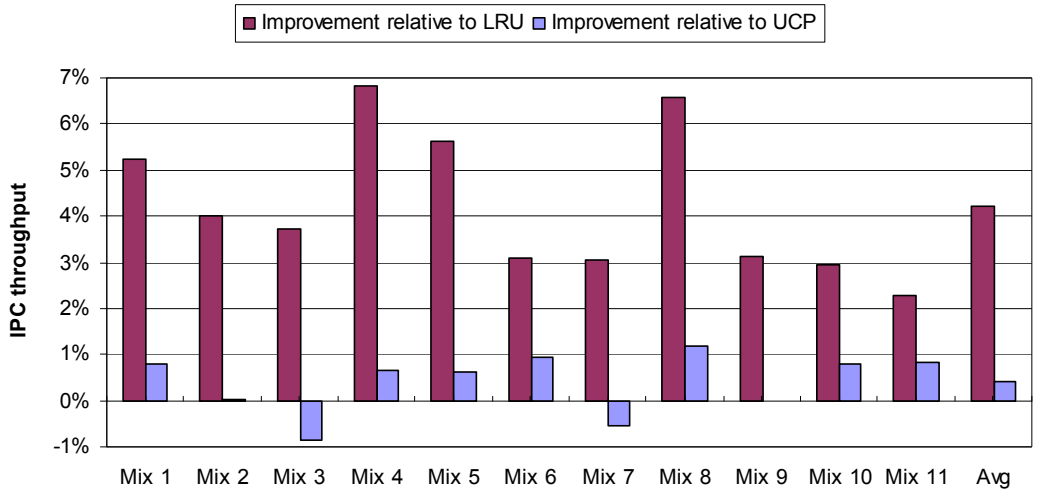
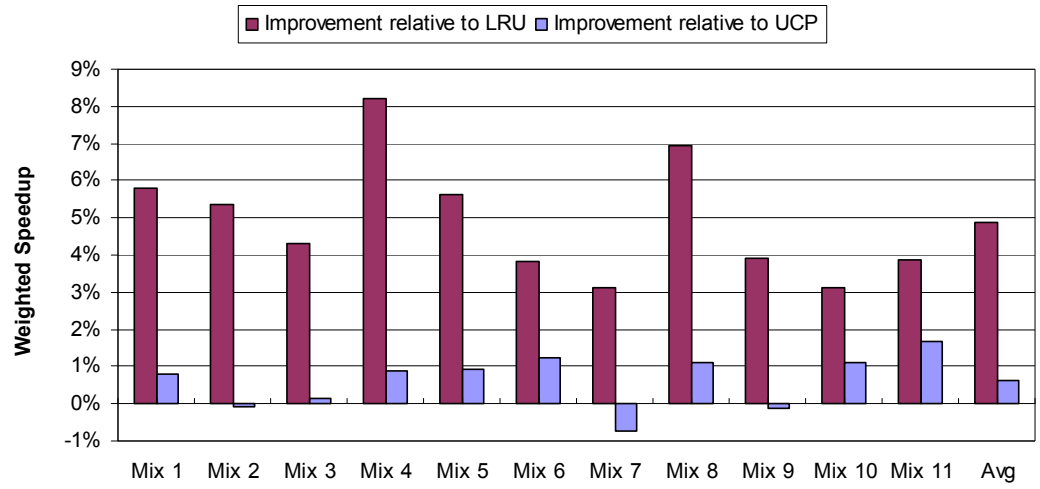


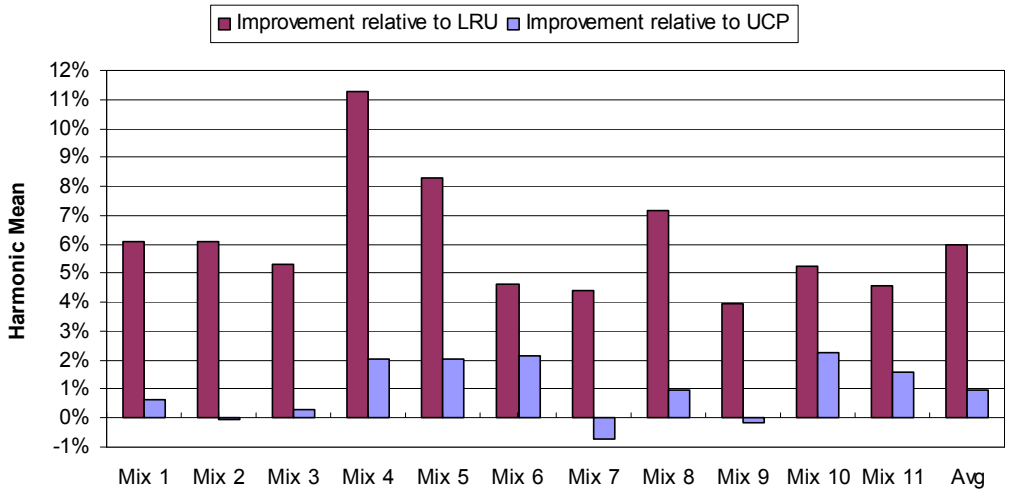
FIGURE 5.18: Performance improvement of PRP normalised to baseline LRU policy and UCP.

Since PRP was proposed to allow each core in the system to maintain its allocated cache resources, the performance fairness of all cores was investigated so that the effect of fine-grained victim selection among the core can be observed. The performance of PRP

measured in weighted speedup is depicted in Figure 5.19(a), while the harmonic mean fairness is illustrated in Figure 5.19(b). The figures demonstrate approximately identical trends of performance gain across all the simulated workloads, normalised to LRU and UCP.



(a)



(b)

FIGURE 5.19: Performance results for the weighted speedup and fair speedup metrics.

From the figure, the weighted speedup of PRP has outperformed LRU by an average of 5% and UCP by an average of 0.6%. On average, PRP shows a 6% fairness speedup compared to LRU and 1% improvement compared to UCP. Although the results show marginal improvements over the IPC throughput figures presented in Figure 5.18, it is apparent that the performance of PRP is still about the same as in UCP. Overall, the evaluation of PRP shows that the performance gain for all the three metrics were lower than the MI2PP policy proposed in the previous section. Nevertheless, the eviction policy of PRP that takes the over-allocated application into account for decision making, does provide benefit to the overall cache performance, but it is not significantly better than UCP.

To gain a better insight into the behaviour of PRP, the impact of L2 miss rates measured in MPKI for all the workloads is examined. Figure 5.20 illustrates significant MPKI reductions by PRP for all the workloads over LRU, two of the simulated workloads incurring more than 5% misses compared to UCP. This is due to MRU-insertion and LRU-insertion strategy employed by PRP, which appears to harm the partitioned shared cache, especially due to the multiple-reuse lines that were inserted at the LRU position.

The proposed PRP was expected to reduce the L2 miss rate of the partitioned cache, but the simulations indicate that PRP is not better than UCP. The insertion position of the single-reuse and multiple-reuse lines at the LRU position due to the over-allocated miss-causing application has caused the lines to incur many cache misses upon their next reuse. Such situations are exhibited by Mix7 and Mix9, which execute three high utility workloads in which the LRU insertion strategy of PRP is unable to beat the MRU insertion employed by the LRU policy in UCP. On average, the L2 miss rate of PRP outperforms LRU by 28%, but only 3.6% over UCP.

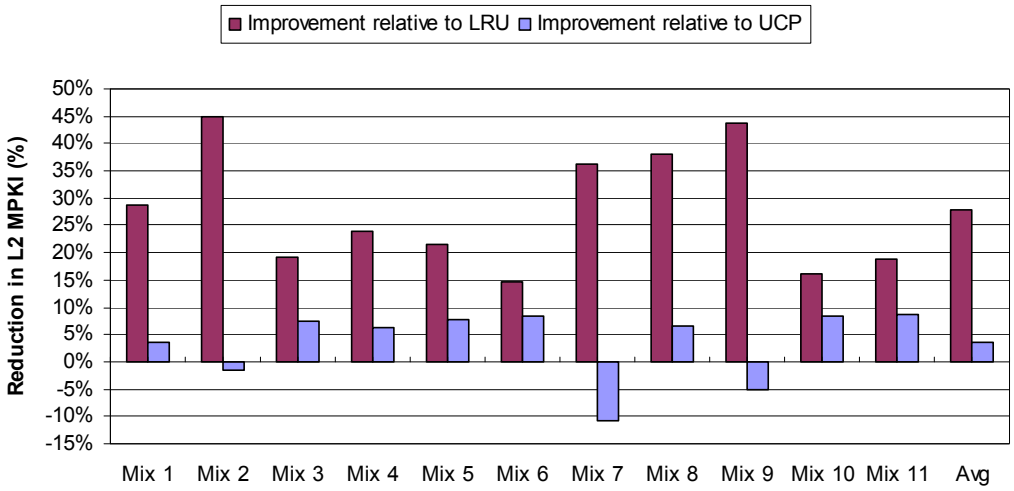


FIGURE 5.20: L2 miss rate of PRP normalised to LRU and UCP.

Given that insertion policy implemented by PRP is decided according to the victim selection, which is based on whether the victim is taken from an over-allocated or under-allocated core, PRP requires negligible hardware overhead. This is contrary to the PSEL counters needed by TADIP to determine the insertion position of new lines. PRP only costs extra logic to take into account the over-allocated application in its victim selection decisions. However, the storage overhead of PRP is similar to MI2PP, as PRP also needs to implement the UMON circuits and to record the owner of the lines in the cache.

5.4.3 Conclusion

The proposed PRP appears able to improve upon the original LRU replacement policy, but comparison with UCP shows that the scheme is unable to produce improved performance across all of the simulated workloads. Even though the degradations are small – except for Mix7, which incurred about 12% L2 misses – it shows that PRP needs improvement. It is assumed that the worse performance was due to an ineffective balance between LRU insertion and MRU insertion. MRU insertion is not beneficial to dead-on-arrival lines,

contrary to the LRU insertion. Unfortunately, for multiple-reuse lines, insertion at the LRU position kills the line too soon, resulting in a series of cache misses for following references. In addition, the remaining lines of the over-allocated application, which are the longer-standing residents of the cache, will spend a longer time occupying the cache space, preventing other cores from inserting new lines in the cache.

This suggests that an improvement can be made to the insertion policy of PRP. From the results presented in the previous section, particularly in the performance of MI2PP policy, it is expected that insertion at the middle position of the priority stack could reduce the potential for the conflicts that have occurred in PRP. This means that instead of inserting a new line at LRU position, the line should be inserted at the middle of the priority stack. Doing so will also provide better fairness in capacity sharing among the cores in the system, further improving the performance of the partitioned shared cache.

Figure 5.21 shows the IPC performance of the extended PRP (EPRP) that uses the middle insertion priority position in a 512KB cache, relative to LRU policy. It is found that EPRP has consistently outperformed PRP and UCP in all workload mixes, with the maximum improvement of about 1.5% over PRP and around 2.5% over UCP, in Mix11.

Interestingly, for Mix3 and Mix7 PRP was outperformed by UCP, but EPRP has consistently demonstrated better performance than UCP for both cases. This shows that the middle insertion position implemented by EPRP has saved many of the cache misses that occurred in PRP due to the early evictions of multiple-reuse lines (the lines were inserted by PRP at the LRU position and were quickly evicted before their next references). This gives more chances for EPRP to execute more instructions compared to PRP, causing the IPC throughput of both Mix3 and Mix7 in EPRP to be better than PRP. Overall, EPRP has

slightly outperformed PRP and UCP IPC throughput by 0.7% and 1.2%, respectively.

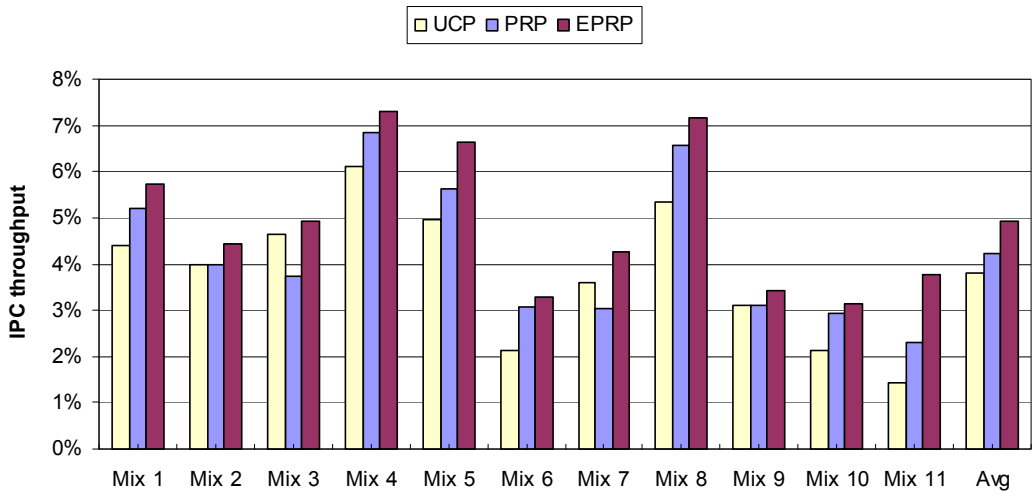


FIGURE 5.21: Performance improvement of UCP, PRP and EPRP in a 512KB cache, normalized to LRU policy.

Figure 5.22 illustrates the miss rate reduction of UCP, PRP and EPRP in a 512KB cache, relative to the LRU policy. It can be observed in the figure that the number of misses saved by EPRP is consistently larger than UCP. Comparison with the original PRP shows that EPRP has outperformed the miss rate reduction of PRP in the majority of the cases.

There are several interesting points identified in this figure. First, in Mix11, EPRP showed a similar miss rate reduction to PRP. Second, EPRP was slightly outperformed by PRP in Mix10. However, for both cases, the IPC throughput improvement of PRP was outperformed by EPRP. The insignificant performance of EPRP could be due to the additional misses incurred by EPRP from the extra instructions of the mixes that the EPRP has executed, which were not processed by PRP. Hence, EPRP achieved better IPC

improvement from the extra instructions that it has executed, but incurred more misses from the additional executed instructions. Also, again in Mix7, EPRP achieved the highest miss rate reduction over PRP – an improvement of around 11%. Finally, EPRP shows that it is able to reduce the miss rate of UCP, with the maximum miss rate improvement of around 10.5% in Mix8. On average, EPRP has outperformed the miss rate reduction of UCP and original PRP by 6.8% and 3.3%, respectively.

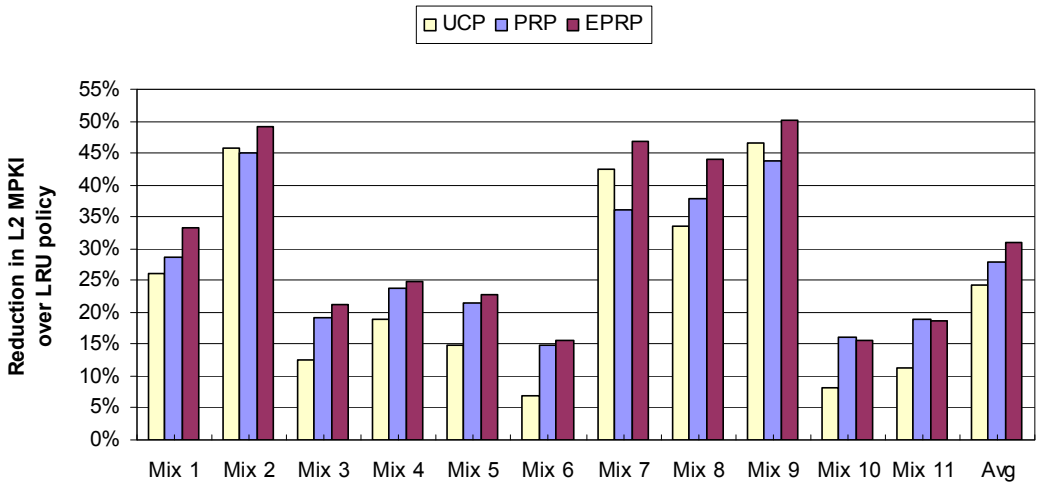


FIGURE 5.22: L2 miss rate of PRP and EPRP, relative to LRU policy.

The performance of EPRP shows that the original PRP can be improved by using a more efficient insertion policy. The middle insertion position implemented in EPRP has demonstrated that many misses can be saved, giving more chances to the system to execute more instructions. As a result, EPRP could achieve higher IPC throughput and miss rate reduction, so that overall, EPRP has performed better than PRP.

5.5 SUMMARY

This chapter presented two schemes designed to improve the cache replacement policy, which are particularly to be used in a partitioned shared cache. The schemes made use of the three components of replacement policy: the insertion policy, promotion policy and eviction policy. The MI2PP scheme was focused on the insertion and promotion strategies. This scheme performed better than the traditional LRU policy that was used in both unmanaged and well-managed caches. Additionally, this scheme showed that strictly enforcing the cache partitioning remains a better partitioning strategy than the pseudo-partitioning proposed by PIPP.

To make better use of the partitioned cache resources among all of the cores at every execution interval, a fine-grained victim selection is required to guarantee no cache resources are taken away from the respective owners. Thus PRP, the second scheme presented in this chapter introduced the use of the allocation status of applications as another metric that must also be taken into consideration by the eviction policy in deciding the potential line to be replaced. The goal of this scheme was to avoid the possibility of eliminating any lines that belong to the less-allocated application, which will further cause the application to suffer from inadequate cache resources. However, overall this scheme did not demonstrate any performance improvement compared to UCP.

To improve the performance of PRP, the LRU insertion policy implemented in PRP was replaced by the middle position insertion strategy. The modified PRP is called EPRP and the aim of EPRP is to reduce the number of misses that could be due to the multiple-reuse lines in the cache. This is because, in PRP, the lines are inserted at the LRU position causing the line to quickly evict before their next reuses. On average, the EPRP showed that

the use of the middle position insertion policy has consistently outperformed the UCP scheme and is superior to the original PRP.

CHAPTER 6

SHARED CACHE PARTITIONING BASED ON PERFORMANCE GAIN ESTIMATIONS

Different applications competing for cache resources in CMP require a very effective sharing mechanism to optimise the utilisation of a shared cache. Several schemes were presented in Chapter 4 in an attempt to partition the cache at runtime, but it was found that there were several trade-offs between the different schemes in optimising system performance.

From the evaluations on the simulated schemes presented in Chapter 4, the behaviour of the applications in the previous execution interval has been shown to be beneficial in providing better information about the cache partition decision for the next interval. If the history of each application is recorded sufficiently, the partitioning decision could be more effective. However, different metrics recorded at runtime and used to decide the partition sizes give different impacts on the overall system performance of different applications. Furthermore, it was also shown that the hardware cost could increase due to the need to obtain the required application's information at every execution interval.

Based on the observations in Chapter 4, another cache partitioning scheme was designed and is presented in this chapter. The scheme was expected to provide better

utilisation of the shared cache resources using important runtime information of the cache without incurring significant hardware overhead. The new scheme also was proposed to provide better estimations on the cache partitioning decisions of the competing applications in the system. This chapter provides the description and the evaluations of the proposed new scheme.

6.1 INTRODUCTION

Various techniques have been proposed to enhance the performance of multiprocessor systems, as well as to improve the overall shared cache throughput. In general, the parallel applications in the system sharing the shared memory resources have different behaviours and requirements throughout the execution processes. The performance of the applications can be characterised by many factors such as the speed of the slowest application, or the maximum number of misses incurred by an application in the system.

The cache partitioning scheme developed by Qureshi and Patt [2006] uses a monitoring mechanism to record the behaviour of each competing application separately. For each application, the distribution of cache hits is recorded across the recency positions ranging from the MRU to the LRU position. On the other hand, Muralidhara et al. [2010] proposed a CPI-based cache partitioning scheme that dynamically partitions the cache space among applications according to analytical models which use the CPI values of each application recorded over past execution intervals.

The simulation results presented in Chapter 4 showed that the UCP scheme by Qureshi and Patt [2006] is more effective than the CPI-based scheme due to its ability to identify the

benefits of giving more cache resources to each competing application at finer grain, in the form of cache ways. In this way, UCP is superior to the CPI-based scheme in providing an estimation of the effectiveness of different partitioning decisions. The trade-off between the two schemes however, is in the context of the hardware overhead. It is observed that implementation of a tag array for each application used in the UMON circuit of the UCP scheme is significantly higher than the CPI-based scheme, which only uses two additional counters per application to record cycle counts and instruction counts in every execution interval.

A different approach was proposed by Dybdahl et al. [2006], in which a shadow tag is assigned to each application executed in the system. The shadow tag is used to record the effect of adding (for performance gain) or removing (causing performance loss) a cache way in a set that belongs to an application. Each application is assigned only one shadow tag register, a shadow tag hit counter and a hit-in-LRU counter. This scheme works when a line is evicted from the cache and the tag of the line is saved in the shadow tag register. On a cache miss, the tag of the miss line is compared with the content of the shadow tag register. If there is a match, the shadow tag's hit counter is incremented by one, whereas the hit-in-LRU counter is updated every time a hit incurs in the LRU entry of the respective application. The content of the latter counter indicates the number of misses that would be incurred if the cache allocation of the application were to be reduced by one way per set.

Next, the new cache allocations for the applications in the system are determined based on the values of the hit counters. If the maximum value of the shadow tag counter of an application is larger than the minimum value in the hit-in-LRU entry counter of another application in the system, a cache way is given to the application with the maximum hit counts in the shadow tag's hit counter. In other words, this scheme uses predicted cache

performance gain and performance loss information in making decisions about cache distributions. Comparison with the UCP scheme performed by Dybdahl et al. [2006] showed that this scheme yields less hardware overhead due to the smaller number of memories and counters required for the purpose of monitoring the behaviour of each competing application. Let us say that *s* is a number of sets, *w* is a number of ways, *c* is a number of cores and *t* is the number of bits per tag. The UCP scheme uses $s \times w \times c \times t$ bits for its shadow tag arrays, whereas the scheme propose by Dybdahl et al. only uses $s \times c \times t$ bits. Even though the UCP scheme introduced the use of dynamic 32 sets sampling to reduce the hardware overhead in order to monitor the overall cache behaviour, the cost difference between the two schemes is still considerable.

It is also found that the proposed cache partitioning scheme by Muralidhara et al. [2010] costs less in hardware overhead than the UCP scheme, although from our investigation in Chapter 4, the scheme occasionally was not as accurate as UCP and failed to identify the requirements of each application at runtime. Therefore, a similar technique used by these schemes was combined and proposed in a new cache partitioning mechanism, presented in this chapter. The aim was to improve the overall throughput of the system at minimal complexity without yielding significant hardware overhead and increase the execution speed of the slowest application in the system.

6.2 MOTIVATION

As described earlier, different performance metrics used to partition the shared memory resources can produce various impacts on the multiple applications. It was observed that the UCP scheme is highly dependent on the system or applications using a LRU-like cache

replacement policy. This is due to the applied concept of monitoring an application's cache utility while the data brought in by the application is traversing across the recency positions ranging from MRU to LRU. Therefore, the UCP scheme may not give an accurate estimation and may be ineffective in partitioning the shared cache if another type of cache replacement policy is employed in the cache.

On the other hand, the CPI-based scheme developed by Muralidhara et al. [2010] was initially proposed to partition the shared cache among multiple threads that belong to the slowest application in the system. However, it showed inconsistent benefits on the overall throughput of the system when the scheme was applied to partition the cache among multiple single thread applications. In addition, the efficiency of this scheme in utilising the partition cache is questionable since the partitioning process is solely based on the history of the CPI values of each application in the system and the analytical model built at runtime. This approach may not effectively estimate the future performance gain of an application in response to the additional cache space allocated to it. As a consequence, it is desirable that both CPI values and the effect of increasing or decreasing the partition size of an application are taken into account in performing the partition decisions.

Another important point to consider is the limited ability of a cache replacement policy to resolve cache misses incurred by multiple-reuse lines at greater distance. The existing LRU insertion strategy of a cache replacement policy is unable to keep the multiple-reuse lines longer in the cache before their next references, therefore incurring more misses. The MRU insertion strategy is not effective for zero-reuse, single-reuse and dead-on-arrival lines, as they will be kept in the cache longer before eviction without providing any benefits, causing an increase in cache waste and competition among applications in the system. A partitioning strategy that could improve the performance of a shared cache by allocating

sufficient cache resources is expected to overcome the limitations of the cache replacement strategy implementing the shared cache. This could be done by estimating the future performance gain of an application in response to the additional cache space allocated to it so that an effective allocation can be fully utilised by the application.

It is clear that each application in the system needs a sufficient cache resource in order to avoid contention and thrashing in the shared cache. A cache partitioning scheme must allocate adequate cache sizes for the applications in the system. If the amount of cache an application has been assigned is small, it will be difficult for any cache management scheme to enhance the performance of that application. A partitioned shared cache based on effective performance metrics can therefore assist the cache replacement policy to fully utilise the cache space across multiple applications, minimising the total number of cache misses.

In this chapter, a dynamic cache partitioning scheme is proposed to enhance the overall throughput of multiple applications in the system. This scheme dynamically determines the cache requirements of each individual application based on its latest performance. The shared cache is partitioned with respect to the CPI values of each application. The application with the highest CPI (the slowest application) will be allocated with more cache ways for the subsequent execution interval. The amount of the additional cache received by the slowest application is determined by the size of the performance gain if the application was assigned with extra cache ways. The approach used to estimate the performance gain of each application is based on the technique used by Dybdahl et al. [2006], which monitors the performance gain and performance loss of each application in the system in order to determine the partition size.

The goal of the new scheme is to ensure that all of the applications progress at approximately the same speeds, accelerating the slowest application without significantly slowing the faster applications, by allocating more cache ways to the slowest application. The focus of this scheme is to investigate the effectiveness and efficiency of using more than one performance metric to determine the cache partition sizes. Most of the existing cache partitioning schemes try to improve the overall system using a global performance metric, whereas the new scheme is based on the hypothesis that if more metrics are used, the partition decisions would be more effective and the system performance enhanced. Moreover, the techniques used in the new scheme are able to be used with any cache replacement policy employed in a shared cache, since the estimation of an application's performance gains are based on the cache lines that have been evicted from the cache.

6.3 Structure of The New Scheme

Initially, the new scheme needs to extend the tag of each cache line with a core identification (core ID). The intention is to record the core that brings the line into the cache so that the cache occupancy of each core can be monitored at runtime.

Figure 6.1: Each cache line is extended to include core identification.

The implementation of the core ID is also capable of ensuring that no application in the

system evicts any cache lines that belong to another application. For example, an application with high demand is restricted from eliminating the LRU entry of another application, if the LRU replacement policy is used in the system. Algorithm 1 describes the augmented LRU replacement policy that restricts the eviction policy of each application.

ALGORITHM 1: Pseudo code for finding a line for eviction. The function returns the position of the line to be replaced.

```
function FIND A LINE TO REPLACE
    if (num_of_lines_owned_by [coreID] ≥ num_of_lines_allocated [coreID]),
      then
        for each line i = 1 to N, do        // N-way set associativity
            coreID_lru_entry ← get line owned by coreID that is the least
                               recently used in the set
                               (LRU_line_stack_position == could be 2, 3, ... N)
        end for
    else
                    lru_entry ← find the least recently used line in the set
                                (LRU_line_stack_position == N)
    end if

    return LRU_line_stack_position
end function
```

* The priority position of the line to be replaced will be updated when it is replaced by a new line. A value of 1 will be assigned to *LRU_line_stack_position* if the new line is installed at MRU position, whereas a value of *N* will be used for insertion at LRU position. In addition, the priority positions of the remaining lines in the set will be updated accordingly.

6.3.1 Monitoring Environment and Scheduler

The first process in this dynamic cache partitioning scheme is to monitor and record the performance of each individual application at every execution interval. The cache resource is repartitioned based on the application's performance at the end of each interval. Since this new scheme is intended to speed up the slowest application so that the progress of all applications are at approximately the same speed, the CPI values become an important reference in the evaluation of the performance of all applications at runtime. Therefore, two counters are assigned to each application to record the cycle counts and the instruction counts during each interval. The contents of the counters are then used to calculate the CPI value of each individual application in order to identify the slowest and the fastest applications. It is expected that the CPI value of the slowest application could be enhanced by allocating more cache ways, since the CPI value increases with the miss rate. The fastest application will have to give away a number of cache ways to the slowest application in the system. Another reason why the CPI value is used to determine the potential donor and recipient of the extra cache ways is that the CPI value is more accurate in reflecting the application's performance compared to the miss rate counts. This is because the cache miss rate does not consider the performance impact of the variable number of memory accesses made by an application during an execution interval.

The scheme also needs to determine the number of additional ways that will be received by the slowest application. The performance gains of the slowest and the fastest applications are used, had both applications have been allocated with more cache ways ranging from one to four ways (assuming a cache organisation as in Table 3.2). To record the performance gain, four counters and a shadow tag array per core are required. The shadow tag array consists of four registers for each set, which are used to store the tag of a line that is just

evicted from the cache. This means that when a new line is brought into the set because of a cache miss event, the tag of the line that is going to be replaced will be saved in the shadow tag array of the core that fetched the line. The traditional LRU replacement policy is implemented in the shadow tag arrays, while the new tag brought into the array is stored at the MRU position. The counter associated with the *p* position at which the tag resides in the shadow tag array represents the number of hits, had the application had an additional *p* cache ways.

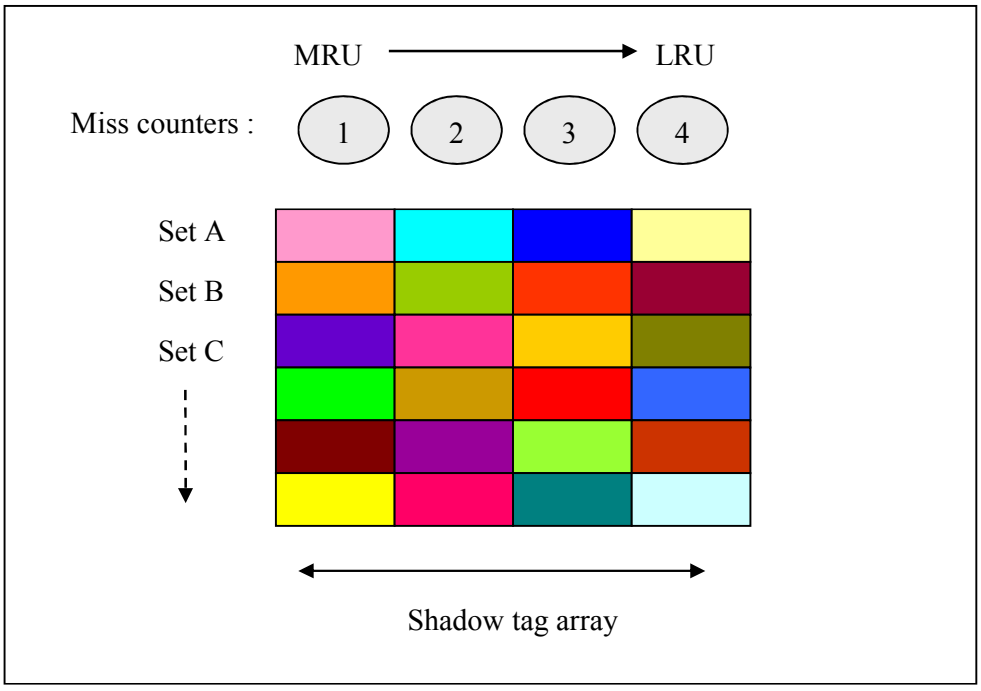


FIGURE 6.2: The shadow tag array and four miss counters associated with each recency position in the monitoring hardware structure of each core.

On every cache miss, the tag of the miss-causing line will be compared with the contents of the shadow tag array belonging to the miss-causing application. If there is a tag

match in the shadow tag array at position *p*, the respective miss counter *p* where the match is happening will be incremented by one. This indicates that the miss would have been a hit if the partition of the application had been allocated with *p* more ways. The reason this scheme evaluates the performance gain by adding up to four ways to an application is due to the expectation that by allocating an additional four ways, the cache misses incurred by lines with multiple-reuse at greater distance could be reduced or eliminated. In other words, monitoring four extra ways at most, is sufficient to estimate further misses that may have been hits if the application had been allocated with additional one, two, three or four ways. In order to reduce the hardware overhead, all the sets in a shadow tag array share the same miss counters that are associated with the recency positions of the tag registers.

6.3.2 Identifying the Partitioning Dependence

The new scheme proposed in this chapter differs from the existing partitioning schemes of Dybdahl et al. [2006], Nikas [2008], Nikas et al. [2008], and Chaturvedi et al. [2010], which use performance gain and performance loss to decide the partition sizes. The new scheme does not consider performance loss if one or more ways are taken away from an application, for the following two reasons. Firstly, to estimate the performance loss, a hit counter(s) must be updated on every cache hit that occurs at the LRU position or any positions near to LRU. To perform this update, very fast logic is required, because it is desirable that the counter update in parallel with the occurrence of cache hit in the cache set (i.e. in the same time taken to process the cache hit). Secondly, to identify if the hit is happening at the LRU position, the scheme has to access all the lines in the set costing extra complexity in the partitioning scheme.

While the CPI value is able to identify the donor and the recipient applications of the extra cache space, the use of the CPI values as an alternative metric to support the

estimation of suitable cache sizes of the applications in the system is promising. Even though this approach is very different in the context of determining the performance loss, it could also approximate the applications that could gain benefits or experience performance loss from the increasing or decreasing cache ways. This is because the CPI value positively correlates with the cache miss rate.

Table 6.1 shows the correlation between CPI value and performance gain in making the partitioning decision. The data presented in this table was taken from one of the simulations performed in Chapter 4 – benchmark Mix3 implementing a CPI-based cache partitioning scheme in a 512KB L2 cache. It was found from the simulation that the application with the highest CPI value would not always benefit from the allocation of additional cache ways. Core1 is identified as the slowest core according to the CPI values, but none of its miss counters recorded any changes. Therefore, allocating extra ways to Core1 would not provide any advantage to the application running on the core, or to the system. However, it is also observed that Core2 has the highest miss counts recorded in C[1], so Core2 would benefit from additional cache ways.

TABLE 6.1: A snapshot of the information recorded during one of the execution intervals.

Core ID	**CPI**	**Miss Counters**			
		C[1]	**C[2]**	**C[3]**	**C[4]**
Core 1	74.00	0	0	0	0
Core 2	12.27	52870	41935	42245	41700
Core 3	12.73	21694	13211	12881	11706
Core 4	9.14	9683	10293	7691	4813

Table 6.1 also shows an interesting relationship between Core2 and Core3. Both cores have approximately equal CPI values, but miss counter C[1] tells us that the number of Core2 misses that could be saved is substantially higher than in Core3, even though the CPI value of Core2 is slightly lower. Consequently, if the new partitioning scheme solely depends on the CPI to determine the best donor and recipient cores of the additional cache ways, better system performance would not necessarily result. The use of miss counter values that represent the performance gain is therefore very important in selecting the recipient of the extra ways, so that the recipient's performance can be enhanced. The following section will discuss the process of identifying the donor and the recipient applications by using both CPI and miss counter values.

6.4 Adaptive CPI-based Cache Partitioning Scheme

A new Adaptive CPI-based Cache Partitioning (ACCP) scheme is proposed to modify the cache space allocation of applications in the system by up to four ways at the end of every execution interval. At the end of each interval, the instruction counts and cycle counts of all applications are gathered and the CPI values calculated. The shared cache space is partitioned based on the CPI values in such a way that the application with the highest CPI (slowest application) will receive extra ways from the application with the lowest CPI (fastest application). The goal to minimise the difference between the CPI values of all of the applications by allocating more cache ways is met, along with achieving a reduction in cache misses and increasing throughput.

In order to make sure the slowest application will get further benefits from the additional cache ways and avoiding the anomalies illustrated in Table 6.1, its miss counter associated with the MRU position is compared with the MRU miss counter of the fastest application. If the value of the counter belonging to the slowest application is less than the

value in the miss counter of the fastest application, another application in the system will be selected as the slowest application – that is, the application with the highest MRU miss count. The application with the next highest CPI value will not be selected because there is a possibility that the MRU miss counts of this application is less than the MRU miss counts of the fastest application or the remaining applications in the system. As shown by the data presented in Table 6.1, there is no guarantee that an application with a higher CPI value will get a higher benefit from receiving extra cache ways than another application with lower CPI values receiving the additional ways. This is because the MRU miss counts of the lower CPI application may be higher than the MRU miss counts of the higher CPI application. Therefore, the identified application with the highest MRU miss counts will be assigned as the new slowest application and will receive additional cache ways.

Algorithm 2 describes the process of determining the number of cache ways to be given to the slowest application. To estimate the performance gain, the miss counters of the fastest and the slowest applications are read. The miss counts in the miss counters are correlated with the number of hits, had the respective application been given additional cache ways. In this work, the performance loss that will cost the fastest application is not estimated. This is because, in order to estimate the performance loss of an application, several more counters will be required to record the number of hits incurred at each priority position of the lines in a set. The number of hits recorded in these counters will represent the number of misses that will be received by an application if a number of ways are taken away from it. That is, the hit counts at the LRU position represents the number of additional misses that will be incurred by an application if a cache way is removed from it. If two cache ways are taken away from the application, the number of hits incurred at the second LRU position will represent the additional number of misses to be received by the application [Dybdahl et al. 2006; Nikas 2008; Nikas et al. 2008; Chaturvedi et al. 2010]. The use of hit counters to

ALGORITHM 2: Adaptive CPI-based Cache Partitioning scheme.

- Start with equal partition for each competing application *i*, at the first interval. $allocation[i] = \frac{total\_cache\_ways}{number\_of\_cores}$
- At the end of each execution interval, do
  - ✓ Read instruction counts, $C_{INS}[i]$ and cycle counts, $C_{CYC}[i]$ for each application and calculate the CPI values.
  - ✓ Determine the slowest application, *slowest_app* and the fastest application, *fastest_app* based on the calculated CPI values.
    - Check the first miss counter, $C_{MISS}[1]$ of the *slowest_app* and the *fastest_app*.
    - If $C_{MISS\_slowest\_app}[1] < C_{MISS\_fastest\_app}[1]$, then another application in the system that has the highest value of $C_{MISS}[1]$ will be assigned as the new *slowest_app*.
  - ✓ Next, evaluate *j* miss counters, $C_{MISS}[j]$ that belongs to the *slowest_app* and the *fastest_app* to determine the number of ways to modify.

```
function PARTITION THE SHARED CACHE
max_gain = 0 ← To check if adding one more way can provide further
                benefits
extra_ways = 0 ← Number of ways to modify.
for each miss counter j = 1 to 4, do
        new_max_gain = max_gain + C_MISS_slowest_app[j]
        if (new_max_gain > max_gain) and
            (C_MISS_slowest_app[j] > C_MISS_fastest_app[j]), then
                max_gain = new_max_gain
                extra_ways += 1
        else
                break;
        end if
end for
if (allocation[fastest_app] – extra_ways > 0)
    allocation[fastest_app] -= extra_ways
    allocation[slowest_app] += extra_ways
end if
return allocation
end function
```

estimate the performance loss of each application seems beneficial, but it will cost additional hardware and processes at every execution interval. We chose to implement an algorithm that does not require this additional overhead.

In order to estimate performance gain from additional cache ways, the miss counters of the slowest application are compared one by one with the counters of the fastest application. The first counter to be checked is the one associated with the MRU position. This represents the performance gain if the application were to be given an extra one way. In sequence, the counters are compared from the one associated with the MRU position to the one associated with LRU position. Each time a counter is checked, if the miss counts of the slowest application are greater than the miss counts of the fastest application, an extra way is given to the slowest application. This process is repeated for all the four counters, or until the value of the miss counts belonging to the slowest application is less than the miss counts of the fastest application. Note that this scheme guarantees that each application owns at least one cache way. This is done by making sure that the total number of ways given to the slowest application by the fastest application will leave at least one cache way to the fastest application – otherwise the cache will not repartition. After each partitioning interval, all the miss counters of all applications are reset to zero.

6.5 Experimental Evaluation

This section briefly explains the simulation methodology and summarises the main evaluation results of the adaptive CPI-based cache partitioning scheme, ACCP.

6.5.1 Simulation Methodology

The ACCP scheme was evaluated on a variety of quad-core workload mixes sharing a 512KB, 32-way associative L2 cache. For each workload, the shared cache was warmed up for 10 million instructions and the partitioning algorithm was invoked after every execution interval of 10 million instructions. The results of the simulated ACCP were analysed and compared against two existing cache partitioning schemes, namely the UCP scheme and a CPI-based scheme. For all experiments, the overall system performance is reported in MPKI (misses per 1000 instructions), IPC throughput, weighted speedup and the harmonic mean of weighted speedup.

6.5.2 Simulation Results

For all of the performance comparisons, the conventional unmanaged shared cache using a LRU replacement policy is used as the baseline. Figure 6.3 shows the miss rate of ACCP, when compared with the performance of UCP and CPI. While ACCP was also proposed to simplify the complex partitioning decision logic in UCP, as well as to improve the partitioning conditions in the CPI-based scheme, ACCP on average has yielded a similar performance rate with both schemes. However, ACCP in a few cases is able to provide miss rate improvements over UCP and CPI-based schemes by at most 5% and 9% respectively. The figure also shows that for the majority of the workloads, ACCP incurred approximately similar miss counts over committed instructions to UCP and CPI-based. The plotted results show that the new approach is able to achieve cache misses reductions as much as UCP, with less hardware overhead. Furthermore, comparison with the CPI-based scheme shows the ability of ACCP to improve miss rates by considering both CPI values and miss rate.

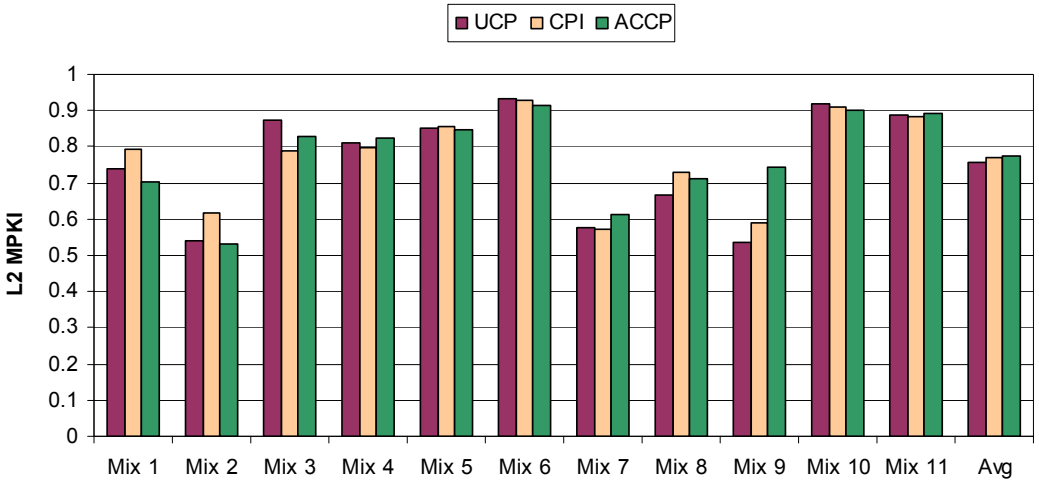


FIGURE 6.3: Miss rate improvements relative to unmanaged traditional LRU cache.

The findings reported in Figure 6.3 are supported by the IPC throughput observed across the simulated workloads. In Figure 6.4, the IPC throughput of Mix3 that employed ACCP was improved by 4.3% compared to the IPC throughput reported by the CPI-based scheme, although it was observed that ACCP has incurred a 5% additional miss rate. A similar pattern is demonstrated by Mix11, for which ACCP yields a 2% improvement in IPC over CPI-based scheme, whereas the MPKI recorded by both schemes show a negligible difference. The reason that ACCP performs better than CPI-based scheme is that while the use of IPC values can give good predictions in partitioning the shared resources, the estimation of performance gain due to the increasing of cache resources can result in better partition decisions of the slowest application and further improve the system performance.

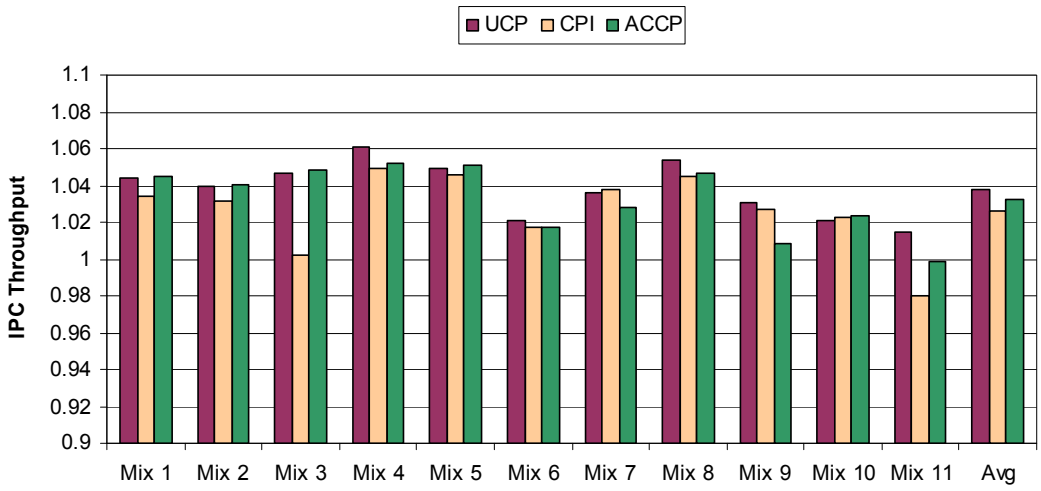
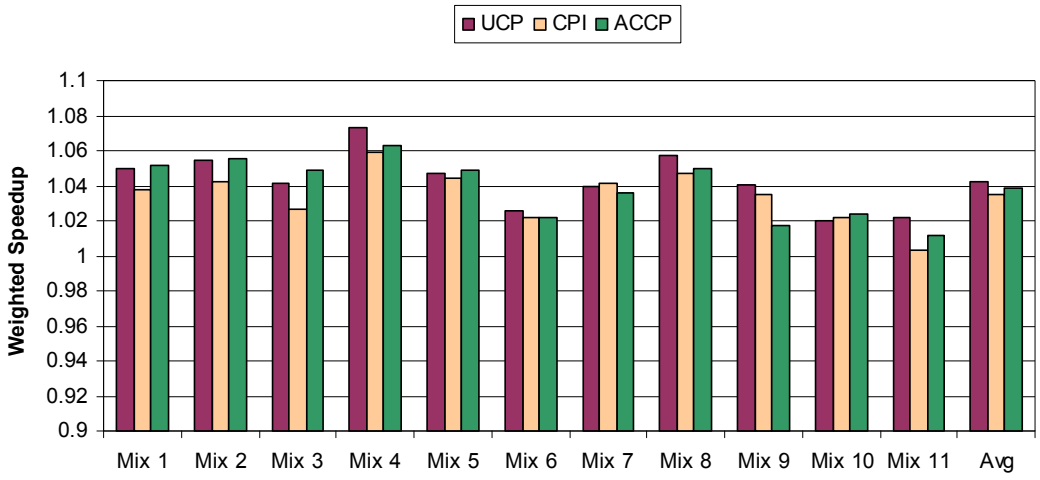
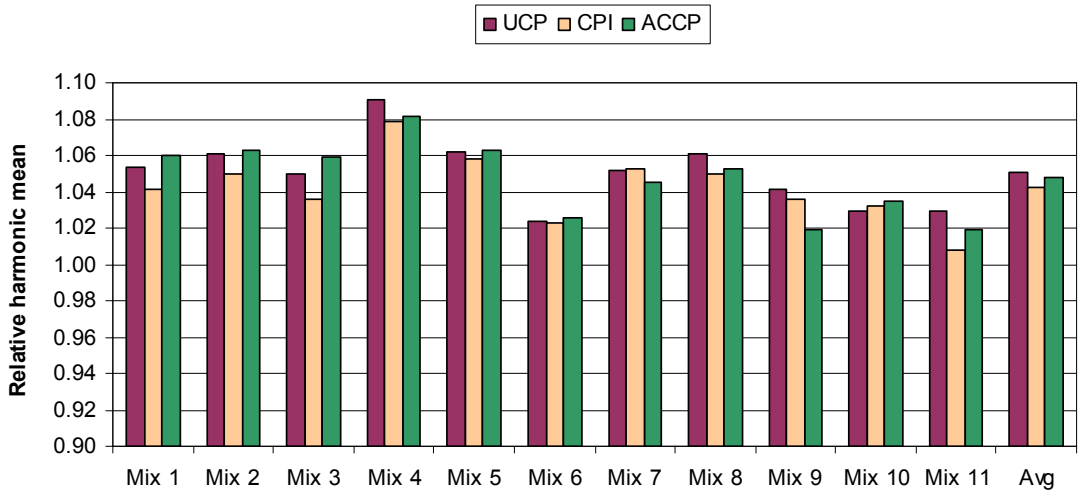


FIGURE 6.4: Performance comparison of UCP, CPI and ACCP over unmanaged LRU L2 shared cache.

From the figure, ACCP has demonstrated that for the majority of the workloads, the combination of CPI values and performance gain estimation could be used effectively to partition the shared resources among applications. The results in Figure 6.4 also show that a simpler partitioning algorithm and a lower implementation overhead do not limit the ability of ACCP to achieve similar performance with UCP. Moreover, ACCP has an advantage over UCP, in that it can be employed with any type of cache replacement policy. To investigate the effect of ACCP on each application of a workload, Figure 6.5 shows speedups achieved by ACCP compared to UCP and a CPI-based scheme.



(a)



(b)

FIGURE 6.5: Performance comparison using the (a) weighted speedup, and (b) harmonic mean of weighted IPC metrics.

It was observed that even though the patterns for both weighted speedup and harmonic means of the IPC are almost similar to the achieved IPC throughput reported in Figure 6.4, there are further improvements exhibited by ACCP. In addition, ACCP performs similarly

to or slightly better than UCP for the fairness performance of Mix1, Mix2, Mix3, Mix5 and Mix10 as illustrated in Figure 6.5 (a). The performance is further enhanced in Figure 6.5 (b) and again, the results show that ACCP in some cases is able to outperform UCP with its simpler approach and has a positive impact on the applications' throughput of the workload mixes. In addition, the fairness speedup of ACCP is better than its weighted speedup by up to 2% compared to the CPI-based scheme. This confirms that ACCP is able to assist the fairness of the workload mixes and therefore meets the objective of the scheme which is to reduce the speed difference of running applications.

6.6 Performance Analysis

The sensitivity of the ACCP scheme to the size of shared cache, the number of sampled sets and the effect of a cache replacement policy were evaluated in a quad-core system. The findings from these investigations are reported in this section.

6.6.1 Cache Scaling and Sensitivity Analysis

The performance results presented in the previous section demonstrated that for the majority of the workloads, the ACCP has slightly outperformed UCP and CPI-based schemes. Further analysis was made on the effect of different L2 cache sizes, which are the size of 1MB and 4MB shared caches. The data presented in Figure 6.6 shows that the performance of ACCP is unaffected by changes in cache sizes. This means that for any size of the shared cache, ACCP retains similar performance trends and consistently achieved similar performance to UCP. In addition, ACCP consistently outperforms the CPI-based scheme for the majority of the simulated workloads. The highest performance improvement recorded in a 1MB cache by ACCP is 1.5% over UCP and 3.72% over a CPI-based scheme for Mix11.

Meanwhile, comparison with a CPI-based scheme on a cache size of 4MB shows that ACCP outperforms the scheme by the maximum value of 5.63% for Mix10, while the performance improvement over UCP was a maximum of 6.91% for Mix10. This could be due to the total benefits of the larger cache size received by all high utility applications in the mix that contribute to the significant overall performance compared to other workload mixes. That is, ACCP has provided sufficient cache space to the high utility applications in Mix10 to achieve higher overall performance improvement.

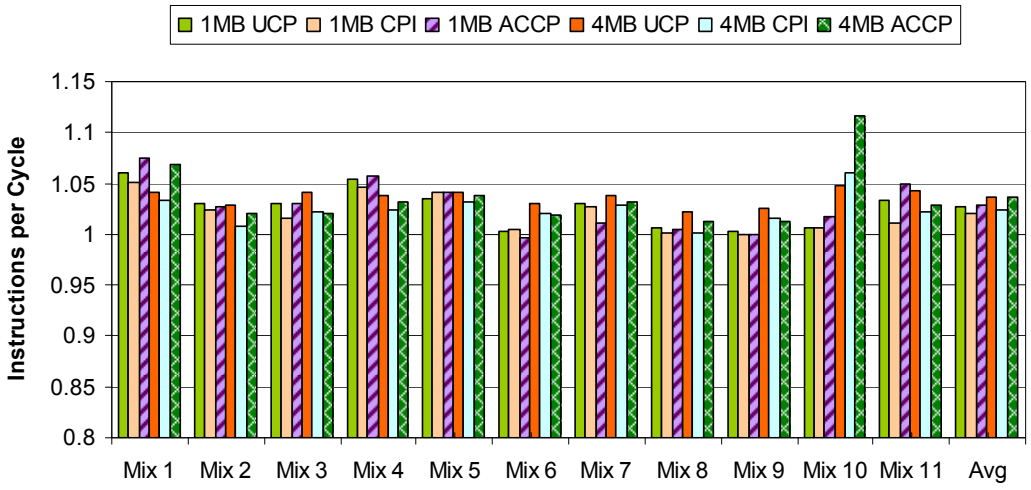


FIGURE 6.6: Performance comparison of ACCP, UCP and CPI-based schemes on different sizes of L2 shared cache, relative to baseline LRU.

Since the scale of performance improvements is very small, the effect of different cache sizes is therefore investigated in terms of the MPKI metric. Figure 6.7 depicts the MPKI of ACCP normalised to MPKI of UCP and CPI-based schemes. An interesting point to observe in the figure is that in the majority of cases, ACCP performs better over CPI-based scheme than it does over UCP. For the 1MB cache, the reduction over a CPI-based scheme is

between 5% and 10% greater than to the reduction relative to UCP. The reduction of ACCP recorded over a CPI-based scheme for the 4MB cache is more than 10% over the MPKI of ACCP normalised to UCP. The most significant reduction exhibited by ACCP occurs for Mix10, in the case of the 4MB cache size, in which the recorded MPKI is less than 20% relative to UCP. This is correlated with the performance value recorded in Figure 6.6, in which ACCP also has reported the highest improvement over UCP in the case of Mix10. On average, ACCP shows a reduction of less than 5% in the 1MB cache and about 5% to 10% in the 4MB cache over the UCP and CPI-based scheme, respectively.

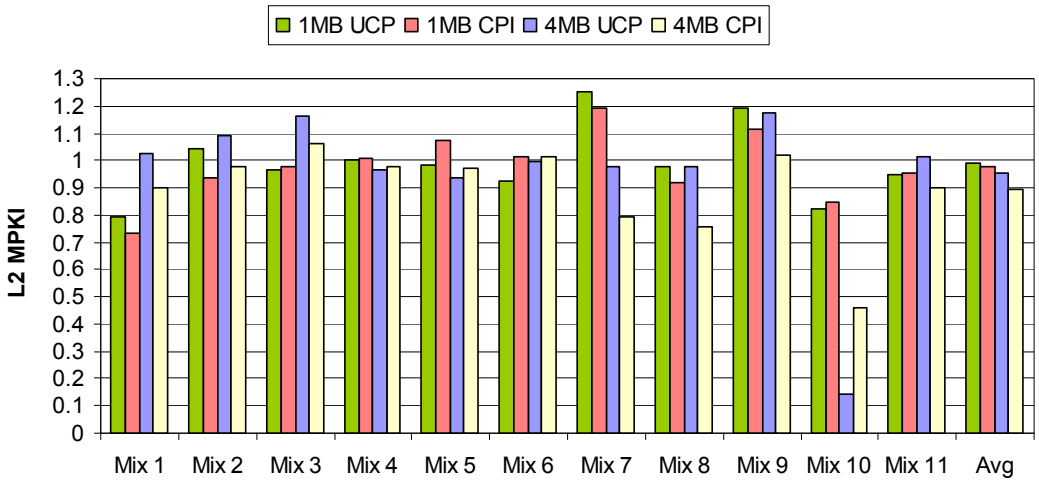


FIGURE 6.7: Miss rate reduction of ACCP over UCP and CPI-based schemes.

From Figure 6.7, it is observed that ACCP performed poorly in Mix7 (for the 1MB cache), in which it incurred around 20% more misses compared to UCP and CPI-based schemes. Meanwhile, for the 4MB cache, ACCP achieved the highest improvement in the miss rate compared to UCP and CPI-based scheme in Mix10. Figure 6.8 shows the

distribution of the miss counts among the cores in Mix7 (for 1MB cache) and Mix10 (for 4MB cache) to identify the contributing factor to the significant performance variation of ACCP in both cases. For Mix10 in the 4MB cache, it can be seen from the figure that ACCP has significantly reduced the number of misses in core C2, which executed a high utility application. The application could have received most of the 4MB cache space from ACCP, causing the significant reduction of the miss counts in this core to achieve the highest overall miss reduction of Mix10, compared to UCP and CPI-based schemes.

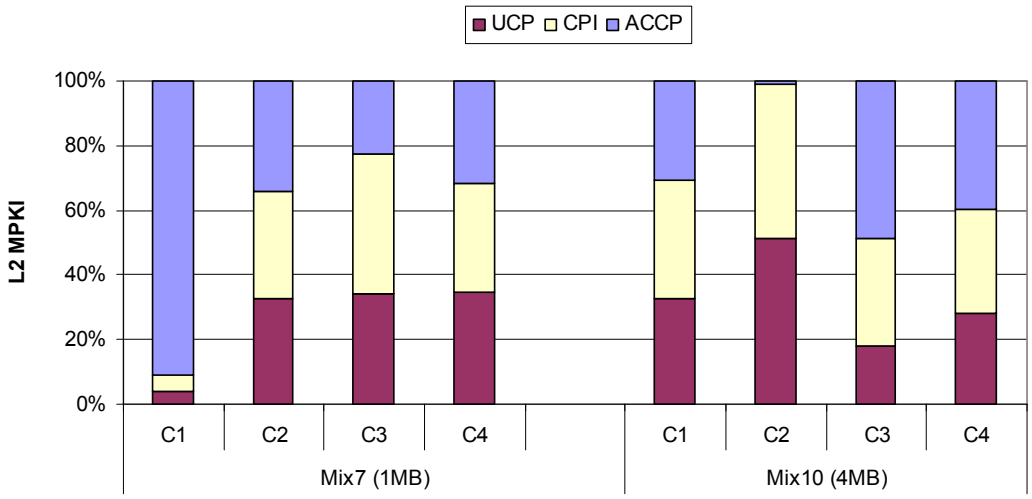


FIGURE 6.8: Distribution of L2 miss rate among UCP, CPI and ACCP.

Meanwhile, for the 1MB cache, it is found that core C1 of Mix7 incurred the highest number of misses compared to other cores in the system. Even though core C1 also executed a high utility application, the figure shows that the core did not get much benefit from the 1MB cache. This could be due to competition with other high utility applications executed by cores C3 and C4. Therefore, the significant number of misses incurred by core

C1 has contributed to the worse performance of Mix7 compared to UCP and CPI-based schemes. Note that an application could perform better or worse due to competition with other cores in the system. That is, the application could be treated differently from its type. This means that a high utility application could be treated as a streaming process in one mix, but could get many advantages from the available cache spaces in another workload mix [Suh et al. 2004]. This could account for the different results produced by an application in different mixes that led to variation of performance improvements among mixes.

6.6.2 Effect of Dynamic Set Sampling

One of the methods to reduce hardware overhead in cache partitioning schemes is derived from the Dynamic Set Sampling (DSS) implemented in the UCP scheme. By default, all sets in the cache are used to monitor the behaviour and performance of the applications in the system. This default approach is known as UMON-global. DSS only uses a limited number of sets in the cache for the monitoring purposes, reducing the hardware cost.

In the new ACCP scheme, the idea of DSS is applied and the results obtained from the simulations are presented in Figure 6.9. Initially, all sets are monitored to retrieve the estimation of performance gain if the applications in the system get extra cache ways, which means the concept of UMON-global was implemented in ACCP. Then, with the goal of reducing the hardware cost, the number of sampled sets to monitor the applications' performance is decreased. For all workloads, set 0 and every 32^{nd} sets in the shared cache are chosen to monitor the applications' behaviour and to estimate the performance gain/loss.

Figure 6.9 shows the performance improvement of ACCP implemented with both UMON-global and DSS (sampling every 32^{nd} set) in different cache sizes. From the above figure, the performance of ACCP on average is relatively insensitive to the number of sets

sampled to monitor the application's progress and the performance of the different workload mixes. In detail, for each cache size, ACCP with sampled sets labeled as ACCP_S shows almost identical performance to the default ACCP.

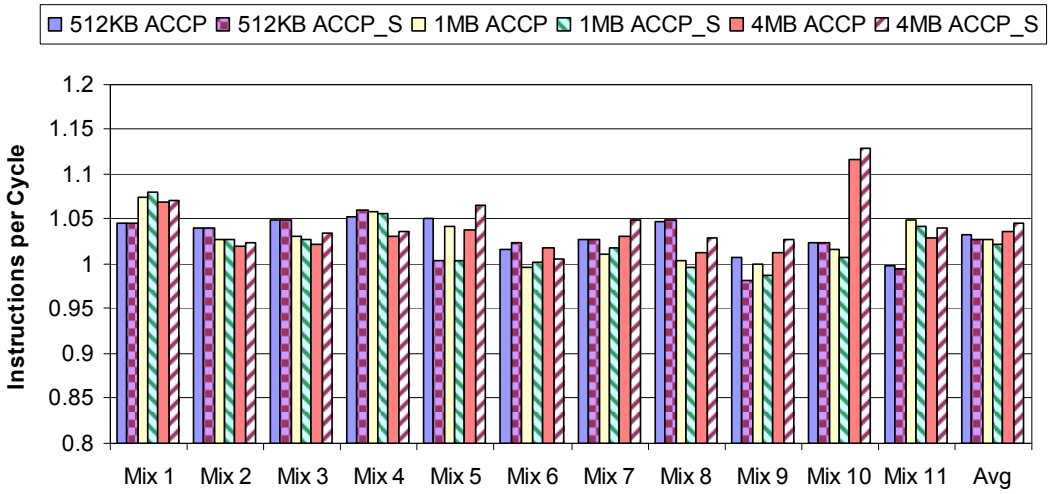


FIGURE 6.9: IPC performance improvement of ACCP using (1) DSS (labelled as ACCP_S), and (2) UMON-global (labelled as ACCP) for different cache sizes, relative to baseline LRU.

The consistency of the ACCP_S in maintaining its performance in different cache sizes has demonstrated that the higher hardware overhead in the default ACCP can be avoided with the implementation of sampled sets. This consequently will make the implementation of ACCP practical for any cache size, and importantly will further reduce the hardware overhead compared to UCP scheme. The cost of hardware overhead will be quantified in the following section.

6.6.3 Comparison with Different Cache Replacement Policies

While the ACCP cache partitioning scheme has been developed to fairly manage the shared cache among multiple cores, thrashing and contention problems still could happen in a shared cache due to competition between multiple cores for cache resources. The investigations conducted in Chapter 5 show that this scheme in combination with an optimal cache replacement policy could help to avoid this problem and further improve the overall system performance. On top of the conventional LRU policy, the ACCP was simulated with the new MI2PP replacement policy proposed in the previous chapter to evaluate the performance of the scheme with different types of cache replacement policy. The performance of ACCP with the employment of MI2PP was analysed in order to investigate the robustness of the scheme and determine whether it will perform better with any kind of cache replacement policy compared to UCP scheme. Figure 6.10 shows the performance of MI2PP policy implemented in different cache partitioning schemes, namely the ACCP, UCP and CPI-based partitioning schemes. Each bar represents the IPC performance of selected replacement policy used in different cache partitioning schemes.

The impact of MI2PP policy in ACCP (MI2PP/UCP) is first compared with the performance of the conventional LRU policy in ACCP (LRU/ACCP). It is found that MI2PP has outperformed or equaled LRU in all of the workload mixes, with the maximum value of performance improvement of about 4%. Now consider the implementation of MI2PP in UCP scheme (MI2PP/UCP). While the MI2PP in ACCP (MI2PP/ACCP) has consistently surpassed the performance of MI2PP/CPI, comparisons over MI2PP/UCP show inconsistent improvements and on average the performance is 0.3% below MI2PP/UCP. Then the miss rate of the ACCP scheme is compared with UCP and CPI-based schemes. Again, as illustrated in Figure 6.11, the improvement achieved by ACCP when the LRU

policy is replaced by MI2PP policy has been consistent.

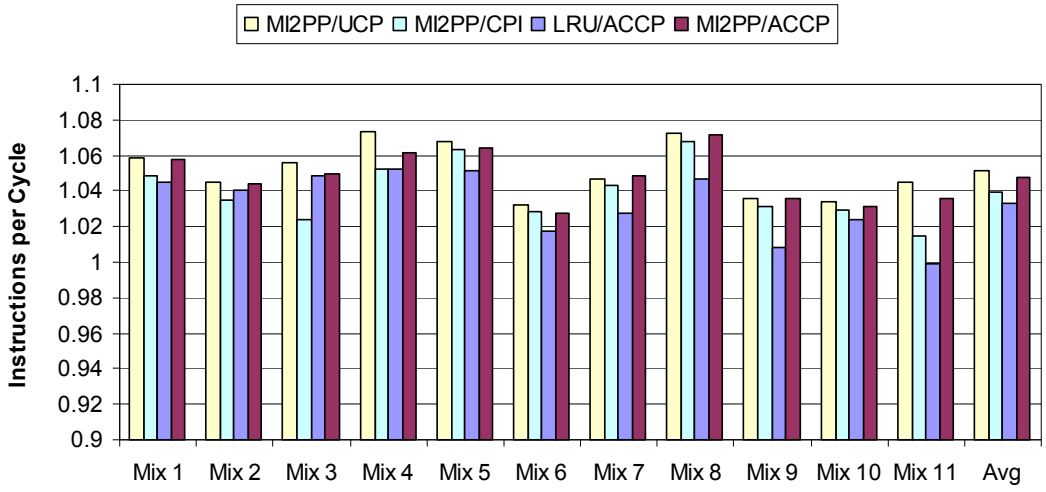


FIGURE 6.10: Performance comparison of MI2PP replacement policy employed on ACCP, UCP, and CPI-based schemes, relative to baseline LRU.

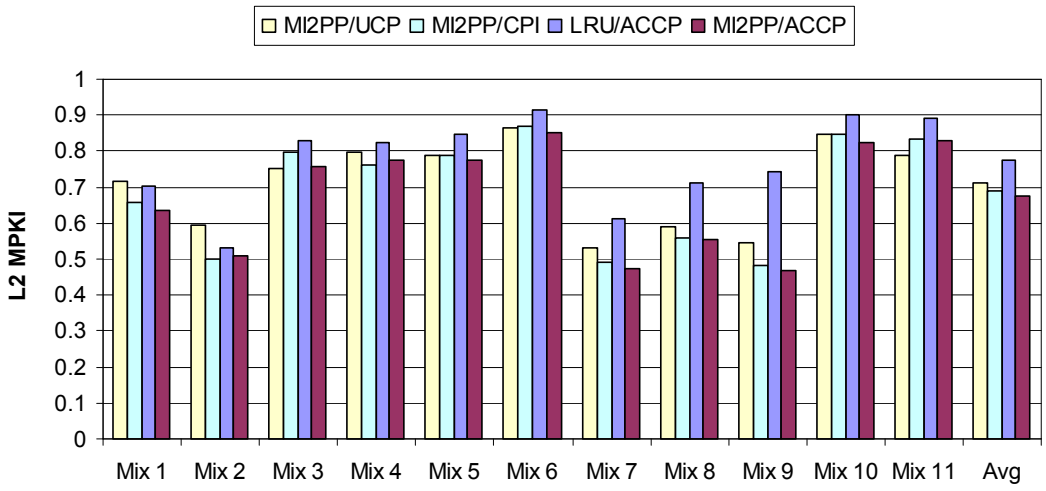


FIGURE 6.11: Miss rate of MI2PP replacement policy employed on ACCP, UCP and CPI-based schemes, relative to baseline LRU.

The highest miss rate reduction exhibited by MI2PP and ACCP is approximately 25% over the miss rate incurred by LRU/ACCP. On average, MI2PP outperforms LRU in ACPP by 10%. When the performance of MI2PP/ACCP is compared with MI2PP/UCP, the data shows that ACCP outperforms UCP for eight out of eleven simulated workloads. ACCP also demonstrates that for the remaining workloads, the performance is either similar to or only slightly worse than UCP. On average ACCP has further reduced the MPKI recorded by UCP. Nevertheless, when it comes to comparing the impact of MI2PP between its usage with ACCP and CPI-based schemes, the miss rate improvement reported by ACCP is always greater than or at least equal to the CPI-based scheme.

The results from the analysis demonstrate that UCP is well suited to LRU-like cache replacement policies, where the cache lines in the sets have a uniform access pattern. In particular, this refers to any replacement policy that will traverse the lines in the cache from the top to the bottom priority position, before the lines are being evicted. On the other hand, ACCP is focused on the access patterns of cache lines after the lines are evicted from the sets. Therefore, the scheme has demonstrated an additional benefit from its partitioning algorithm as information about the evicted lines is managed separately from the lines that are residents of the shared cache space. This means the movements or changes of the information about the evicted lines will not be affected by any adjustment happening in the cache sets. Consequently, ACCP can easily adapt to the environment of any optimal cache replacement policy, or vice versa, and can provide further advantages to the shared cache and overall system performance.

Furthermore, evaluations made on ACCP employing the MI2PP policy, which uses a static predefined insertion position and non-uniform eviction selection among applications even better illustrates that ACCP is insensitive to the environment of the cache replacement

policy used in the system. We conclude that changes to the cache replacement policy employed in the cache would not affect the partitioning decisions made by ACCP, but the replacement policy could support ACCP to achieve further improvement. However, this needs to be confirmed by future study using different cache replacement policies than the ones implemented in this work.

6.7 Complexity Management and Hardware Implementation

Algorithm 2, as presented in Section 6.4, begins by choosing the fastest and the slowest applications based on the CPI values. Then, the four miss counters of the two applications are read and compared. The values recorded in the counters represent how much the applications could benefit from the allocation of another 1, 2, 3 or 4 ways. Primarily, the partitioning algorithm of ACCP requires the implementation of addition, subtraction, and comparisons. Referring to the new algorithm, ACCP does not employ any complex control logic due to the use of few *if* statements. The *if* statements can be directly implemented in hardware by using a compare-and-multiplex operation. Meanwhile, the number of iterations that the miss counters need to decide the number of ways to reallocate can be determined in advance. Hence, comparisons between each miss counter can be done in parallel. There is no control-flow dependence on the input values of the iteration counts, except the dependence on the slowest application identifier. This scenario therefore permits the implementation and parallelisation of simple control logic for the algorithm to make the partitioning decisions.

The strategy of examining each application's performance gain by increasing up to four cache ways has reduced the hardware cost of monitoring and gathering information on all

applications in the system. Comparison with a well-established scheme, namely UCP, shows that the ACCP scheme uses considerably less storage and fewer counters than UCP. In order to monitor the behaviour of each competing application, had the applications received four additional cache ways, ACCP needs only four tag registers per set and four miss counters assigned to each application. On the other hand, the UMON-global of UCP requires *w* tag registers per set and *w* counters for each *N* applications, where *w* is the cache associativity and *N* is the number of cores. The counters used in this work were assumed to be 32 bits long, but from our observation, the size can be reduced in practice for the following reasons. Firstly, the monitoring period was reset every ten million cycles. Hence, 24-bit counters are sufficient to be used for the cycle counters. Secondly, the miss counts of the miss counters recorded in the simulation results showed that the numbers of misses could be contained within 24-bit counters.

Now consider a 4MB, 32-way associative cache assuming a 32-bit physical address space. For simplicity, a very approximate estimation of the storage overhead that will be incurred for a quad-core system if all sets are monitored and evaluated by ACCP is presented in Table 6.2.

TABLE 6.2: Hardware overhead per processor.

Shadow tag array entry ((15 bits tag + 2 bits LRU) * 4 positions)	**8.5B**
Total shadow tag array overhead (8192 sets * 8.5B)	**69,632B**
Miss counters overhead (4 counters * 3B)	**12B**
CPI counters overhead (2 counters * 3B)	**6B**
Total overhead	**68.02KB**
Area of baseline L2 cache (736KB tags + 4MB)	**4832KB**
% increase in L2 cache area due to a shadow tag array	**1.4%**

Note that, as the ACCP requires each cache line to have a core ID, the length for the identification is $\log_2 N$ bits. Thus, the overhead due to the identifications is $8192 \times 16 \times 2bits = 32KB$, an increase of about 1%. In total, the storage overhead for a quad-core system is $4cores \times 68.02KB + 32KB = 304.08KB$, an increment of about 6% over the L2 cache area. Moreover, the aforementioned adders and comparators are needed to update the counters and to perform partitioning decisions in the ACCP. This will increase the total hardware overhead slightly over the indicated value even though the additional overhead is relatively small. However, by using DSS to reduce the hardware overhead, the actual hardware cost for monitoring 256 sets by ACCP is reduced. From the calculation, to achieve such a low hardware overhead, the ACCP will cost each core 2.142KB of additional hardware due to the shadow tag arrays and counters to record the CPI values, which is an increment of 0.044% in the L2 cache area. As a result, implementation of ACCP will add an extra 0.198% area overhead towards the total shared cache area of a quad-core system. A detailed hardware simulation to evaluate ACCP will be done in our future work.

6.8 SUMMARY

This chapter has presented a new cache partitioning scheme that is intended to improve the utilisation of the shared cache among multiple cores. It also attempts to manage the progress of running applications in the system at approximately the same speed, by accelerating the slowest application without significantly slowing any others, with low hardware overhead. In essence, the new ACCP scheme uses registers to keep track of the cache activities, particularly the misses of each application in the system. Performance gain counters are used to record the cache misses for each application that could have been hits if the application had been allocated with more cache space. Additionally, the ACCP makes use of

the instruction and cycle counts of each application at the end of every execution interval to identify the slowest and fastest applications.

The rule that was followed in the new scheme is that the slowest application based on the CPI values, would most probably get maximum benefit from additional cache ways. Meanwhile, the speed of the fastest application is expected to slow down due to sacrificing cache space that was allocated to the slowest application. Therefore, a number of cache ways will be taken away from the fastest application and will be given to the slowest application if the expected hit count of the slowest application is greater than the fastest application, had the slowest application been assigned with additional cache ways. Based on this rule, the partitioning algorithm of the new scheme reevaluated the partition sizes of each core at every execution interval based on the performance gain estimation retrieved from the miss counts recorded at runtime. The new scheme only considers the performance gain of an application and does not take into account the performance loss of the fastest application, had it been allocated with fewer cache ways. In this context, the new scheme only considers the difference between the performance gain of the slowest and the fastest applications.

From the simulations that have been conducted, ACCP has achieved better or at least similar performance with a system using existing cache partitioning schemes. Comparison with the complex UCP scheme, which requires higher hardware overhead than ACCP, has shown that the new scheme is capable of achieving similar performance, even though ACCP focused on adjusting the cache allocation of only two cores in the system. The performance gain estimation used in the ACCP also has outperformed the ability of the CPI-based scheme to provide effective partitioning decisions. While it was proven by the CPI-based scheme that CPI values could be solely used as a metric to determine better partition sizes, it is found from the investigations of ACCP that the performance gain estimation can produce

significantly more effective partitioning decisions. This is because the performance gain estimation reflects how much an application would benefit from additional cache ways.

Overall, the proposed scheme shows that referring solely to CPI values to determine the cache partition sizes is not completely effective, since this approach does not use the important information provided by historical memory accesses made by each application in the system. Evaluation on the lines that have been evicted from the cache in order to make better partitioning decisions can give more flexibility to the partitioning scheme and the system to gain further benefits from the employment of both an optimal partitioning scheme and an optimal cache replacement policy. This is because the applications' behaviours and their cache activities will not incur any impact to the estimation of performance gain. Finally, the analysis made on the hardware cost has demonstrated that the new ACCP scheme incurs a very low hardware overhead. Hence, the proposed scheme can be effective in improving the CMP system performance without incurring significant hardware overhead.

CHAPTER 7

CONCLUSION

The management of shared memory has become a key performance issue in the organisation of Chip Multiprocessors. A number of techniques have been introduced to provide optimal cache resources to the applications in the system. Since multiple cores may execute different applications that have different behaviours and memory access patterns, several mechanisms have been investigated and proposed to improve the resource sharing among competing cores in the CMP across a variety of workload mixes. This thesis addresses the distribution of the shared cache resources among the cores to minimise the adverse effects on performance of competition for the cache. This work has included the development and evaluation of new eviction, insertion and promotion policies in the cache.

The main contributions of this thesis, derived from the goal to improve the cache resource sharing among competing application are summarised in this chapter. The new techniques introduced to manage the cache replacement policy in a partitioned shared cache are summarised in Section 7.1. In Section 7.2, a mechanism used to partition the shared cache space without incurring significant hardware overhead is reviewed. Section 7.3 lists the contributions of this thesis. Section 7.4 discusses the limitations of the studies and investigations presented in this thesis and Section 7.5 suggests future works.

7.1 Cache Replacement Policy

When a cache is full and a new cache line needs to be inserted in the cache, a cache replacement policy will play its role in choosing which line to discard so that space can be made for a new incoming line. For many existing cache replacement policies, several extra features have been included in an attempt to give each application running in the system an equal opportunity to have its information residing in the cache for as long as possible or at least long enough to reduce the potential of incurring avoidable misses. In general, a cache replacement policy includes three components: the insertion policy, promotion policy and eviction policy.

In a CMP, different applications are executed in parallel and each one of them has different cache behaviour at different execution stages. While existing replacement policies have introduced various techniques to select the cache line that should be ejected when a miss occurs (regardless of the application that has brought the line into the cache), a uniform victim selection strategy employed for all the competing applications in the system has been found to increase the cache throughput. That is, the concept of ownership in the partitioned shared cache can be used to avoid resource stealing among applications in the system and thereby enhance the eviction strategy in a cache replacement policy. Additionally, a replacement policy also needs to retain adequate cache resources for each application in the system. This can be done by refining the insertion and promotion policies because both policies have significant impact on the movements of cache lines once placed in the cache and implicitly become the indicator of the life span of each line in the cache.

A new cache replacement policy, named Middle Insertion 2 Positions Promotion (MI2PP) proposes a new static predefined insertion policy and a short distance promotion strategy. The intention of this new replacement policy was to increase the amount of time

multiple-reuse line resides in the cache, thereby reducing the cache misses due to multiple references to the same information at a large distance. The new policy also tries to shorten the lifetime of the zero-reuse lines in the shared cache. The original contribution of this new scheme is the modifications to the static insertion, dynamic promotion and eviction strategies of a cache replacement policy that is targeted for a partitioned shared cache.

The new MI2PP policy was evaluated using cycle-accurate simulations of a quad-core system and achieved better overall performance than the conventional LRU policy in a partitioned cache. Evaluation in an unmanaged cache also demonstrated that the new policy is on average slightly better than LRU. Furthermore, the new MI2PP policy appears to be effective at improving the system performance of a shared cache by retaining more lines of the competing applications in the cache long enough so that more cache hits can be manifested.

7.2 CACHE PARTITIONING SCHEME

Initially, the effectiveness of distributing the cache resources among applications in the system was studied and the results obtained from the investigations showed that the performance of the applications can be improved significantly. Two types of partitioning mechanisms were identified: statically partitioned and dynamically partitioned. It was found that the dynamic cache partitioning approach is superior to the statically partitioning strategy. There are a variety of metrics that can be used by the partitioning scheme to evaluate the current performance of each application and to decide the partition sizes of the applications in the system. Two dynamic cache partitioning schemes that attempt to allocate cache space among applications based on the applications' dynamic changing behaviours

and characteristics, were analysed. The results from the analysis demonstrated the pros and cons of different metrics used in deciding the partition sizes and were used as a guide to develop an improved cache partitioning scheme.

A new Adaptive CPI-based Cache Partitioning (ACCP) scheme was introduced in this thesis and evaluated on a quad-core system. The scheme was proposed to ensure the progress of all applications in the system have similar execution speed or minimum slack time among the applications. For that reason, the CPI value was selected as a metric to be used by the partitioning algorithm in deciding which allocation should receive more cache resources and which one should lose resources. Comparison of CPI values alone was shown to be insufficient for the partitioning scheme to make effective decisions about reallocating cache resources. The scheme added performance gain estimations to help re-distribute the cache resources between the slowest and the fastest applications. Another objective underlying the proposal of this scheme was to develop a new partitioning scheme that will adapt easily to any cache replacement policy employed in the cache.

The evaluations made on the new scheme and comparisons with the existing UCP and CPI-based scheme showed that the new scheme consistently outperformed the CPI-based scheme, but performed only slightly better than UCP. However, in the context of hardware cost, the new scheme has significantly reduced the area overhead incurred by UCP. Thus, the new scheme showed that the combination of both CPI values and performance gain estimations used in deciding the shared cache allocations among multiple cores in CMP does not cost significant hardware overhead.

To justify the robustness of the new scheme, the new MI2PP cache replacement policy was employed together with the new scheme on a quad-core system and their combined

performance were evaluated. The results showed a similar performance pattern with the new scheme using LRU policy. However, the improvements made by both MI2PP and ACCP in the context of miss rate were substantial. Additionally, comparisons with UCP and CPI-based schemes that used MI2PP policy demonstrated that ACPP on average showed small performance improvements.

7.3 List of Contributions

Following is a list of contributions of the work presented in this thesis.

- A new cache replacement strategy, the Middle Insertion 2 Positions Promotion (MI2PP) policy, which uses a predetermined insertion position for a new line at the middle priority ordering and the two priority positions promotion strategy was introduced. The MI2PP policy showed better performance than the LRU replacement policy in both unmanaged and well-managed caches. In addition, MI2PP showed that strictly enforcing the shared cache partitioning remains a better partitioning strategy than pseudo-partitioning of PIPP.

- The Partition-based Replacement Policy (PRP) was proposed to guarantee no cache resources are taken away from the allocated resources belonging to each application sharing a L2 cache. A fine-grained victim selection algorithm implemented in PRP used the allocation status of applications as another metric to decide on the potential line to be replaced during cache miss events. It was also used to avoid the possibility of resource stealing from a less-allocated application, which causes the application to suffer from inadequate resources. The PRP was shown to improve

upon the baseline LRU policy, but did not demonstrate any overall performance improvement compared to UCP.

- An Extended Partition-based Replacement Policy (EPRP) was the original PRP enhanced by the use of the middle position insertion policy, instead of the LRU position insertion policy. The goal of EPRP was to reduce the number of cache misses in PRP that could be due to multiple-reuse lines in the cache. The EPRP demonstrated that it has improved the overall performance of both UCP and the original PRP.

- A new Adaptive CPI-based Cache Partitioning (ACCP) scheme was introduced to manage the progress of running applications in a system, keeping them at approximately the same speed by accelerating the slowest application without significantly slowing any other applications, with low hardware overhead. The use of CPI values and performance gain estimations of the running applications in determining a shared cache allocation for each application in ACCP achieved at least similar performance improvement to UCP and better performance than the CPI-based partitioning scheme, with lower hardware overhead compared to the UCP scheme.

7.4 LIMITATIONS

The proposed MI2PP replacement policy was found to occasionally retain dead lines in the cache for a long period before eviction. This may be due to the dead-on-arrival lines, which were inserted at a priority position too far from the end of the priority stack. This situation

may also be caused by the relaxed promotion strategy which, while benefitting reused lines, may cause the lines that became dead to reside longer than desirable in the cache. These issues could degrade the system performance, hence some improvements in handling the cache misses by enhancing the insertion and promotion strategies could make the new MI2PP cache replacement policy more robust.

Existing cache partitioning schemes focus on reallocating the cache space among a specific number of cores, or all the cores in the system. The new proposed ACCP scheme focuses on varying the cache allocation of only two cores in the system, the slowest and the fastest cores. There is an awareness that this technique will lead to impractical implementation if the system scales to a significantly greater number of cores. The number of ways to vary in every execution interval might need to be increased in proportion to the increasing size of shared cache or the increment number of cores in the system. This will increase the hardware cost of ACCP.

7.5 FUTURE WORK

The applications' access patterns are critical in determining how an application may affect its competitors in the system. Accurate dynamic monitoring of these access patterns is crucial so that the shared cache space could be highly utilised without degrading the performance of other applications. If the behaviour of each application is known, a dynamic promotion policy can be implemented in the proposed MI2PP, whereby it can choose between employing a relaxed promotion strategy (a short distance promotion movement), and an aggressive promotion strategy (a farther distance promotion movement). A dynamic insertion policy in which the insertion position will be determined by the amount of cache

allocation, could shorten the distance for the dead-on-arrival lines to traverse for eviction. These strategies could help the replacement policy to evict the dead lines in the cache more quickly.

To resolve the constraints in the ACCP scheme that are listed in the previous section, the scheme can be improved by reevaluating the performance of the cores in the system, thereby reallocating the cache resources to all the cores. This could be done mainly by referring to the speed of all applications executing in the system to rank the progress of the applications, beginning with the slowest to the fastest applications. The applications at the top rank will get the highest priority to receive additional cache ways, while the applications ranked at the bottom will have to give away their cache ways. A specific number of cache ways that are going to be given to or taken away from the multiple cores is preset. To determine the optimal number of cache ways to be reallocated in every execution interval, it is beneficial to take into account the size of the shared cache or the total number of cores in the system. Further analysis is needed for this purpose.

There is potential to investigate more refined strategies for using performance gain estimation to determine the number of ways to be reallocated between the fastest and slowest applications. The performance gain comparison between the donor and the recipient applications could be repeated until there are no cache ways left to give away (based on the preset number of cache ways). It would also be interesting to investigate the effect of varying execution intervals on the performance of MI2PP, PRP and EPRP policies, as well as ACPP and similar schemes. Nonetheless, investigations on various numbers of sampled sets that will be used to evaluate and determine cache partitioning decisions at the end of every execution interval would be valuable future work.

Different types of workloads such as multi-threaded applications would produce different performance results on each technique proposed in this thesis. Thus, evaluations on the implementation of multi-threaded applications on our simulated system could be performed so that performance and reliability of the proposed techniques can be observed. Moreover, different system architectures and different processors with different ISAs would also provide different insights to the proposed techniques. Hence, this will be one of our possible future works in addition to investigating energy efficiency, its considerations and its implications on the proposed techniques.

7.6 Final Remarks

Optimising the memory sharing mechanism in CMP systems has been one of the major challenges for system designers and memory architects. The work presented in this thesis demonstrated that optimisation of the cache replacement policy is a useful and efficient approach to enable the cores in the system to better utilise the allocated cache space. Meanwhile, enhancement of the cache partitioning scheme showed that the fairness among multiple cores can be improved without incurring substantial hardware overhead. Moreover, the overall shared cache throughput was also increased, as well as the system performance. Furthermore, the combination of the cache replacement policy and the cache partitioning scheme had provided significant impact on system performance by improving the throughput of the shared cache. Hence, the findings of the work discussed in this thesis provide new insights and techniques in managing memory sharing among multiple cores in CMP.

BIBLIOGRAPHY

ALBONESI-D.-H. (1999). Selective cache ways: On-demand cache resource allocation. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on* (pp. 248-259). IEEE.

AL-ZOUBI-H., MILENKOVIC-A., AND MILENKOVIC-M. (2004). Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of the 42nd Annual Southeast Regional Conference* (pp. 267-272). ACM.

BECCHI-M., FRANKLIN-M., AND CROWLEY-P. (2007). Performance/area efficiency in chip multiprocessors with micro-caches. In *Proceedings of the 4th International Conference on Computing Frontiers* (pp. 247-258). ACM.

BURSKY-D. (2013). Advances in IC development take center-stage at ISSCC. Chip Design Magazine. Retrieved from: http://www.chipdesignmag.com/bursky/?p=95

BUTKO-A., GARIBOTTI-R., OST-L., AND SASSATELLI-G. (2012, July). Accuracy evaluation of GEM5 simulator system. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on* (pp. 1-7). IEEE.

CADENCE DESIGN SYSTEMS, INC. (2013). Cadence Reports First Quarter 2013 Financial Results and Completes Acquisition of Tensilica. Press Release Date: 23 April 2013. Retrieved from: http://www.cadence.com/cadence/newsroom/press_releases/Pages/pr.aspx?xml=042413_financial&CMP=home

CHANDRA-D., GUO-F., KIM-S., AND SOLIHIN-Y. (2005, February). Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on* (pp. 340-

351). IEEE.

CHANG-J. (2007). *Cooperative Caching for Chip Multiprocessor*. Doctoral dissertation, the University of Wisconsin-Madison.

CHANG-J., AND SOHI-G.-S. (2006). Cooperative caching for chip multiprocessors. In *Proceeding of the 33rd Annual International Symposium on Computer Architecture* (Vol. 34, No. 2, pp. 264-276). IEEE Computer Society.

CHANG-J., AND SOHI-G.-S. (2007, June). Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing* (pp. 242-252). ACM.

CHATURVEDI-N., THOMAS-J., AND GURUNARAYANAN-S. (2010). Adaptive block pinning based: dynamic cache partitioning for multi-core architectures. *International Journal of Computer Science & Information Technology*, *2*(6), 38-45, December 2010.

CHEN-Y., LI-E., KIM-C., AND TANG-Z. (2009, May). Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* (pp. 1-11). IEEE.

CHENG-L., MURALIMANOHAR-N., RAMANI-K., BALASUBRAMONIAN-R., AND CARTER-J.-B. (2006). Interconnect-aware coherence protocols for chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, *34*(2), 339-351.

CHISNALL-D. (2014). *The Pitfalls of Parallelism*. Indiana. Pearson Education. Retrieved from: http://www.informit.com/articles/article.aspx?p=2233979

DUONG-N., ZHAO-D., KIM-T., CAMMAROTA-R., VALERO-M., AND VEIDENBAUM-A.-V. (2012, December). Improving cache management policies using dynamic reuse distances. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 389-400). IEEE Computer Society.

DYBDAHL-H., STENSTRÖM-P., AND NATVIG-L. (2006). A cache-partitioning aware replacement policy for chip multiprocessors. In *High Performance Computing-HiPC 2006* (pp. 22-34). Springer Berlin Heidelberg.

EZER-G. (2000). Xtensa with user defined DSP coprocessor microarchitectures. In *Computer Design, 2000. Proceedings. 2000 International Conference on* (pp. 335-342). IEEE.

FLORES-A., ARAGÓN-J.-L., AND ACACIO-M.-E. (2007). Efficient message management in tiled CMP architectures using a heterogeneous interconnection network. In *High Performance Computing–HiPC 2007* (pp. 133-146). Springer Berlin Heidelberg.

FLORES-A., ARAGON-J.-L., AND ACACIO-M.-E. (2010). Heterogeneous interconnects for energy-efficient message management in CMPs. In *Computers, IEEE Transactions on*, *59*(1), 16-28.

HENNESSY-J.-L., AND PATTERSON-D.-A. (2012). *Computer architecture: a quantitative approach*. Elsevier.

HENNING-J.-L. (2000). SPEC CPU2000: Measuring CPU performance in the new millennium. In *Computer*, *33*(7), 28-35.

HERRERO-E., GONZÁLEZ-J., AND CANAL-R. (2008, October). Distributed cooperative caching. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (pp. 134-143). ACM.

HUANG-A., GAO-J., GUO-W., SHI-W., ZHANG-M., AND JIANG-J. (2012, June). PSA-NUCA: A pressure self-adapting dynamic non-uniform cache architecture. In *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on* (pp. 181-188). IEEE.

INTEL CORPORATION. (2014). Intel® Core™ i7-920 Processor (8M Cache, 2.66 GHz, 4.80 GT/s Intel® QPI). Retrieved from: http://ark.intel.com/products/37147/Intel-Core-i7-920-Processor-8M-Cache-2_66-GHz-4_80-GTs-Intel-QPI

ITRS ORGANIZATION (2014a). 2013 Edition: Executive Summary. Retrieved from: http://www.itrs.net/Links/2013ITRS/2013Chapters/2013ExecutiveSummary.pdf

ITRS ORGANIZATION (2014b). 2013 Edition: ITRS International Roadmap Committee Overview. Retrieved from: http://www.itrs.net/Links/2013ITRS/2013Chapters/2013Overview.pdf

JALEEL-A., HASENPLAUGH-W., QURESHI-M., SEBOT-J., STEELY JR-S., AND EMER-J. (2008, October). Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (pp. 208-219). ACM.

JING-W., XIAOYA-F., HAI-W., AND MING-Y. (2007, August). A 16-port data cache for chip multi-processor architecture. In *Electronic Measurement and Instruments, 2007. ICEMI'07. 8th International Conference on* (pp. 3-183). IEEE.

JIA-G., LI-X., WANG-C., ZHOU-X., AND ZHU-Z. (2012, December). Behavior aware data locality for caches. In *Parallel and Distributed Systems, International Conference on* (pp. 514-521). IEEE.

JIANG-S., AND ZHANG-X. (2002). LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, *30*(1), 31-42.

JUAN-F., AND SHUAI-W. (2012, December). Energy-aware cache partition based on way-adaptable in CMP. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012. 13th International Conference on* (pp. 548-551). IEEE.

KAMPE-M., STENSTROM-P., AND DUBOIS-M. (2004, April). Self-correcting LRU replacement policies. In *Proceedings of the 1st Conference on Computing Frontiers* (pp. 181-191). ACM.

KĘDZIERSKI-K., CAZORLA-F.-J., GIOIOSA-R., BUYUKTOSUNOGLU-A., AND VALERO-M. (2010a). Power and performance aware reconfigurable cache for CMPs. In *Proceedings of the Second International Forum on Next-Generation Multicore/Manycore Technologies* (p. 1). ACM.

KEDZIERSKI-K., MORETO-M., CAZORLA-F.-J., AND VALERO-M. (2010b). Adapting cache partitioning algorithms to pseudo-LRU replacement policies. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (pp. 1-12).

KHAN-S., AND JIMÉNEZ-D.-A. (2010). Insertion Policy Selection Using Decision Tree Analysis. In *Computer Design (ICCD), 2010 IEEE International Conference on* (pp. 106-111). IEEE.

KIM-C.-H., CHUNG-S.-W., AND JHON-C.-S. (2006). PP-cache: A partitioned power-aware instruction cache architecture. *Microprocessors and Microsystems*, *30*(5), 268-279.

KIM-S., CHANDRA-D., AND SOLIHIN-Y. (2004, September). Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (pp. 111-122). IEEE Computer Society.

KIN-J., GUPTA-M., AND MANGIONE-SMITH-W.-H. (1997, December). The filter cache: an energy efficient memory structure. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (pp. 184-193). IEEE Computer Society.

KRON-J.-D., PRUMO-B., AND LOH-G.-H. (2008, June). Double-DIP: Augmenting DIP with adaptive promotion policies to manage shared L2 caches. In *Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects, Beijing, China* (pp. 1-9), 2008.

KUMAR-R., ZYUBAN-V., AND TULLSEN-D.-M. (2005). Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on* (pp. 408-419). IEEE.

LARUS-J.-R. (1993). Compiling for shared-memory and message-passing computers. *ACM Letters on Programming Languages and Systems (LOPLAS)*, *2*(1-4), 165-180.

LEE-J., AND KIM-H. (2012). TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on* (pp. 1-12). IEEE.

LEE-K.-H., AND LAM-C.-H. (1989). Massage-passing controller for a shared-memory multiprocessor. *SIGARCH Computer Architecture News* 17, 6 (December 1989), 142-149.

LEVERICH-J., ARAKIDA-H., SOLOMATNIKOV-A., FIROOZSHAHIAN-A., HOROWITZ-M., AND KOZYRAKIS-C. (2007). Comparing memory systems for chip multiprocessors. In *ACM SIGARCH Computer Architecture News* (Vol. 35, No. 2, pp. 358-368). ACM.

LEVERICH-J., ARAKIDA-H., SOLOMATNIKOV-A., FIROOZSHAHIAN-A., HOROWITZ-M., AND KOZYRAKIS-C. (2008). Comparative evaluation of memory models for chip multiprocessors. In *ACM Transactions on Architecture and Code Optimization (TACO)*, *5*(3), 12.

LI-C., WANG-D., XUE-Y., WANG-H., AND ZHANG-X. (2011). Enhanced adaptive insertion policy for shared caches. In *Advanced Parallel Processing Technologies* (pp. 16-30). Springer Berlin Heidelberg.

LIM-Z. (2010). *An RNS-Enabled Microprocessor for Public Key Cryptography*. Doctoral dissertation, University of Adelaide.

LUO-K., GUMMARAJU-J., AND FRANKLIN-M. (2001). Balancing throughput and fairness in SMT processors. In *Performance Analysis of Systems and Software, 2001. ISPASS. 2001 IEEE International Symposium on* (pp. 164-171). IEEE.

MAGNUSSON-P.-S., CHRISTENSSON-M., ESKILSON-J., FORSGREN-D., HALLBERG-G., HOGBERG-J., LARSSON-F., MOESTEDT-A, AND WERNER, B. (2002). Simics: A full system simulation platform.*Computer*, *35*(2), 50-58.

MAI-K., PAASKE-T., JAYASENA-N., HO-R., DALLY-W.-J., AND HOROWITZ-M. (2000, June). Smart memories: A modular reconfigurable architecture. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on* (pp. 161-171). IEEE.

MANDKE-A., VARADARAJAN-K., BHARDWAJ-A., AND SRIKANT-Y.-N. (2010). Optimizing power in tiled S-NUCA CMP architectures using remap table. *Computer Science and Automation Indian Institute of Science, India* (pp. 1-27).

MANIKANTAN-R., RAJAN-K., AND GOVINDARAJAN-R. (2012, June). Probabilistic shared cache management (PriSM). In *ACM SIGARCH Computer Architecture News* (Vol. 40, No. 3, pp. 428-439). IEEE Computer Society.

MORETO-M., CAZORLA-F.-J., RAMIREZ-A., AND VALERO-M. (2007). Explaining dynamic cache partitioning speed ups. *Computer Architecture Letters*, 6(1), 1-4.

MURALIDHARA-S.-P., KANDEMIR-M., AND RAGHAVAN-P. (2010). Intra-application cache partitioning. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (pp. 1-12). IEEE.

MUTLU-O., AND MOSCIBRODA-T. (2007, December). Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 146-160). IEEE Computer Society.

NANEHKARAN-Y.-A., AND AHMADI-S.-B.-B. (2013). The challenges of multi-core processor. In *International Journal of Advancements in Research & Technology* (Volume 2, Issue 6, pp. 36-39), June 2013.

NAYFEH-B.-A., AND OLUKOTUN-K. (1997). A single-chip multiprocessor. *Computer*, *30*(9), 79-85.

NIKAS-K. (2008). *An Analysis of Cache Partitioning Techniques for Chip Multiprocessor Systems*. Doctoral dissertation, the University of Manchester.

NIKAS-K., HORSNELL-M., AND GARSIDE-J. (2008, July). An adaptive bloom filter cache partitioning scheme for multicore architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on* (pp. 25-32). IEEE.

OLUKOTUN-K., NAYFEH-B.-A., HAMMOND-L., WILSON-K., AND CHANG-K. (1996). The case for a single-chip multiprocessor. *ACM Sigplan Notices*, *31*(9), 2-11.

PAL-R-K., PAUL-K., AND PRASAD-S. (2012, July). ReKonf: A reconfigurable adaptive manycore architecture. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on* (pp. 182-191). IEEE.

PANDA-P.-R., DUTT-N.-D., AND NICOLAU-A. (1997, September). Architectural exploration and optimization of local memory in embedded systems. In *System Synthesis, 1997. Proceedings., Tenth International Symposium on* (pp. 90-97). IEEE.

PATEL-K., PARAMESWARAN-S., AND SHEE-S.-L. (2007). Ensuring secure program execution in multiprocessor embedded systems: a case study. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2007 5th IEEE/ACM/IFIP International Conference on* (pp. 57-62). IEEE.

PATEL-K., AND PARAMESWARAN-S. (2008). SHIELD: a software hardware design methodology for security and reliability of MPSoCs. In *Proceedings of the 45th annual Design Automation Conference* (pp. 858-861). ACM.

PATTERSON-D. (2014). *Computer organization and design : The hardware/software interface / David A. Patterson, John L. Hennessy ; with contributions by Perry Alexander [and fifteen others]*. (Fifth ed.). Retrieved from: http://books.google.com.au/

PATTERSON-D. AND HENNESSY-J. (2005). *Computer organization and design: The hardware/software interface.* (3rd ed.). San Francisco, California: Morgan Kaufmann.

POWELL-M., YANG-S.-H., FALSAFI-B., ROY-K., AND VIJAYKUMAR-T.-N. (2000). Gated-V_{dd}: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design* (pp. 90-95). ACM.

QURESHI-M.-K., JALEEL-A., PATT-Y.-N., STEELY-S.-C., AND EMER-J. (2007, June). Adaptive insertion policies for high performance caching. In *ACM SIGARCH Computer Architecture News* (Vol. 35, No. 2, pp. 381-391). ACM.

QURESHI-M.-K., AND PATT-Y.-N. (2006). Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 423-432). IEEE Computer Society.

QURESHI-M.-K., LYNCH-D.-N., MUTLU-O., AND PATT-Y.-N. (2006). A case for MLP-aware cache replacement. In *ACM SIGARCH Computer Architecture News* (Vol. 34, No. 2, pp. 167-178). IEEE Computer Society.

REDDY-R., AND PETROV-P. (2010). Cache partitioning for energy-efficient and interference-free embedded multitasking. In *ACM Transactions on Embedded Computing Systems (TECS)*, *9*(3), 16.

SATO-M., TOBO-Y., EGAWA-R., TAKIZAWA-H., AND KOBAYASHI-H. (2012). A capacity-efficient insertion policy for dynamic cache resizing mechanisms. In *Proceedings of the 9th conference on Computing Frontiers* (pp. 265-268). ACM.

SCHAUER-B. (2008). Multicore Processors: A Necessity. Retrieved from: http://www.csa.com/discoveryguides/multicore/review6.php

SHARIFI-A., SRIKANTAIAH-S., KANDEMIR-M., AND IRWIN-M.-J. (2012, June). Courteous cache sharing: Being nice to others in capacity management. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE* (pp. 678-687). IEEE.

SHEE-S.-L, ERDOS-A., AND PARAMESWARAN-S. (2006). Heterogeneous multiprocessor implementations for JPEG:: a case study. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis* (pp. 217-222). ACM.

SHEE-S.-L, AND PARAMESWARAN-S. (2007). Design methodology for pipelined heterogeneous multiprocessor system. In *Proceedings of the 44th Annual Design Automation Conference* (pp. 811-816). ACM.

SHIUE-W.-T., AND CHAKRABARTI-C. (1999, June). Memory exploration for low power, embedded systems. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference* (pp. 140-145). ACM.

SRINIVASAN-K., TELKAR-N., RAMAMURTHI-V., AND CHATHA-K.-S. (2004, February). System-level design techniques for throughput and power optimization of multiprocessor SoC architectures. In *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on* (pp. 39-45). IEEE.

STALLINGS-W. (2006). *Computer Organization and Architecture 7th Edition*. Upper Saddle River, NJ: Prentice Hall.

STANDARD PERFORMANCE EVALUATION CORPORATION. (2010). SPEC CPU2000 Benchmark Description File. Online. Available : www.spec.org/cpu2000. Last viewed 2012.

SU-C.-L., AND DESPAIN-A.-M. (1995, April). Cache design trade-offs for power and performance optimization: a case study. In *Proceedings of the 1995 International Symposium on Low Power Design* (pp. 63-68). ACM.

SUH-G.-E., RUDOLPH-L., AND DEVADAS-S. (2004). Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, *28*(1), 7-26.

SUI-X., WU-J., CHEN-G., TANG-Y., AND ZHU-X. (2010, May). Augmenting cache partitioning with thread-aware insertion/promotion policies to manage shared caches. In *Proceedings of the 7th ACM International Conference on Computing Frontiers* (pp. 79-80). ACM.

TAM-S.-W., SOCHER-E., CHANG-M.-C.-F., CONG-J., AND REINMAN-G.-D. (2011). RF-interconnect for future network-on-chip. In C. Silvano et al. (eds.), *Low Power Networks-on-chip* (pp. 255-280). Springer US.

TENSILICA, INC. (2010). *Xtensa Modeling Protocol (XTMP) User's Guide*, Issue Date: 9/2010. Tensilica, Inc., Santa Clara California, USA.

TENSILICA, INC. (2011). *Xtensa LX4 Microprocessor Data Book*, RD-2011.2 Release. Tensilica, Inc., Santa Clara California, USA.

WANG-W., MISHRA-P., AND RANKA-S. (2011). Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE* (pp. 948-953). IEEE.

WU-J., SUI-X., TANG-Y., ZHU-X., WANG-J., AND CHEN-G. (2010, September). Cache management with partitioning-aware eviction and thread-aware insertion/promotion policy. In *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on* (pp. 374-381). IEEE.

XIE-Y. (2011). *Efficient shared cache management in multicore processors*. Doctoral dissertation, Georgia Institute of Technology.

XIE-Y., AND LOH-G. (2008, June). Dynamic classification of program memory behaviors in CMPs. In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects* (pp. 1-9).

XIE-Y., AND LOH-G.-H. (2009). PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News* (Vol. 37, No. 3, pp. 174-183). ACM.

XIE-Y., AND LOH-G.-H. (2010). Scalable shared-cache management by containing thrashing workloads. In *High Performance Embedded Architectures and Compilers* (pp. 262-276). Springer Berlin Heidelberg.

YANG-S.-H., POWELL-M.-D., FALSAFI-B., AND VIJAYKUMAR-T.-N. (2002, February). Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on* (pp. 151-161). IEEE.

ZHANG-C., VAHID-F., AND NAJJAR-W. (2003). A highly configurable cache architecture for embedded systems. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on* (pp. 136-146). IEEE.