NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Study, assessment and optimisation of last level cache replacement policies in CMP systems

DIPLOMA THESIS

of

Ioannis K. Agriomallos

Supervisor: Nectarios Koziris
Professor NTUA

Athens, March 2015

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
COMPUTER SCIENCE DIVISION
COMPUTING SYSTEMS LABORATORY

Study, assessment and optimisation of last level cache replacement policies in CMP systems

DIPLOMA THESIS

of

Ioannis K. Agriomallos

Supervisor: Nectarios Koziris
Professor NTUA

Approved by the three-member committee on the 4th of March 2015.

....	....	....
Nectarios Koziris	Nikolaos Papaspyrou	Georgios Goumas
Professor NTUA	Associate Professor NTUA	Lecturer NTUA

Athens, March 2015

. . .

Ioannis K. Agriomallos
Graduate Electrical and Computer Engineer of NTUA

Copyright © Agriomallos K. Ioannis, 2015
All rights reserved.

The present work may not be reproduced, stored nor distributed in whole or in part for commercial purposes. Permission is hereby granted to reproduce, store and distribute this work for non-profit, educational and research purposes, provided that the source is acknowledged and the present copyright message is retained. Enquiries regarding use for profit should be directed to the author.
The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Technical University of Athens.

Περίληψη

Η συνεχώς αυξανόμενη ζήτηση για ολοένα και αποδοτικότερα υπολογιστικά συστήματα ώθησε τους κατασκευαστές υλικού στην αναζήτηση καινοτόμων αλλαγών στην ήδη υπάρχουσα τεχνολογία, με κύριο στόχο την αύξηση στην ταχύτητα μετάδοσης,επεξεργασίας και αποθήκευσης των δεδομένων. Η μετάβαση από μονοπύρηνα σε πολυπύρηνα επεξεργαστικά συστήματα ήταν μία απ'τις κυρίαρχες αλλαγές, όπου πολλά σχεδιαστικά ζητήματα επιλύθηκαν με την υϊοθέτηση της νοοτροπίας κατασκευής των μονοπύρηνων συστημάτων. Όμως ο τρόπος λειτουργίας των πολυπύρηνων συστημάτων δεν είναι ξεκάθαρος σε όλες του τις πτυχές και δεν προσομοιάζει αυτόν των μονοπύρηνων, ανοίγοντας έτσι το δρόμο της έρευνας για την εξεύρεση νέων βέλτιστων σχεδιαστικών αλλαγών. Ένας απ'τους τομείς που χρήζει έρευνας είναι η ιεραρχία μνήμης, όπου έγινε αυτούσια μεταφορά της βέλτιστης πολιτικής αντικατάστασης -ελάχιστα πρόσφατα χρησιμοποιούμενου ή Least Recently Used (*LRU*)- για το τελευταίο επίπεδο κρυφής μνήμης (Last Level Cache [LLC]) από τα μονοπύρηνα στα πολυπύρηνα συστήματα. Μελέτες έχουν δείξει πως η *LRU* δεν είναι η βέλτιστη πολιτική αντικατάστασης LLC για τα πολυεπεξεργαστικά συστήματα, ενώ έχουν γίνει διάφορες προτάσεις οι οποίες συμπεριφέρονται καλύτερα απ'την *LRU*.

Σκοπός της συγκεκριμένης διπλωματικής εργασίας είναι η μελέτη, υλοποίηση, αξιολόγηση και βελτιστοποίηση αλγορίθμων πολιτικών αντικατάστασης μπλοκ στο τελευταίο επίπεδο κρυφής μνήμης για πολυεπεξεργαστικά συστήματα. Συγκεκριμένα γίνεται σύγκριση διάφορων υλοποιημένων πολιτικών αντικατάστασης -ενδεικτικά: *ABFCP*, *TADIP*, *TADRRIP*, *PIPP*, *UCP*- ως προς την *LRU*. Επιπλέον προτείνεται μια τροποποίηση μιας εξ'αυτών (*ABFCP*), η οποία βελτιώνει την ήδη υπάρχουσα πολιτική, σε βαθμό συγκρίσιμο με τις καλύτερες πολιτικές και σε συνδυασμό με αισθητά μειωμένο κόστος σε επιπλέον υλικό.

Λέξεις Κλειδιά — πολιτική αντικατάστασης, κατακερματισμός κρυφής μνήμης, Bloom φίλτρα, *ABFCP*, ανακατασκευή UMON, BFMON, πολυπύρηνες αρχιτεκτονικές

Abstract

The constantly increasing demand for more efficient computing systems motivated the hardware manufacturers to make innovative changes in existing technology, in order to achieve faster transmission, processing and storing of data. The transition from single-processing to multi-processing systems was one of the main changes, where many designing issues were resolved by adopting the principles of single-processing systems. The way multi-processing systems operate is yet to be fully understood and does not resemble that of single-processing systems, thus opening the path of research for new optimal design changes. Among the fields that favour research is memory hierarchy, where optimal replacement policy -Least Recently Used (*LRU*)- for the Last Level Cache (LLC) was transferred unchanged from single-processing to multi-processing systems. Researches have shown that *LRU* is not the best LLC replacement policy for multiprocessing systems, whereas several competing policies have been proposed, which perform better than *LRU*.

The major subject of this diploma thesis is the study, implementation, evaluation and optimization of last level cache replacement algorithms for multiprocessing systems. Specifically a comparison is made between several implemented replacement policies -for instance: *ABFCP*, *TADIP*, *TADRRIP*, *PIPP*, *UCP*- with respect to *LRU*. Moreover an alternative for one of these policies (*ABFCP*) is proposed, which shows an improvement at a degree comparable to the best performing ones, as well as combining considerably reduced hardware overhead.

***Keywords* —** replacement policy, cache partitioning, Bloom filters, *ABFCP*, UMON reconstruction, BFMON, estimator, multicore architectures

"Not all those who wander are lost."

J. R. R. Tolkien

Acknowledgements

This diploma thesis was conducted in the Computing Systems Laboratory of the School of Electrical and Computer Engineering of the National Technical University of Athens, under the supervision of Professor Nectarios Koziris.

Primarily, I would like to thank my supervisor, Dr. Nectarios Koziris, for his guidance during the process of this thesis and throughout my undergraduate studies.

Furthermore, I would specially like to express my appreciation and gratitude to the Post-Doctoral Researcher Konstantinos Nikas, who not only guided me appropriately, but also through his patience gave me space and *time* to find -among others- myself and manage to bring this cycle of my studies to completion. I would also like to thank all CSLab researchers for their help whenever needed.

In this journey my family could not be absent, whom I can't thank enough in these lines, for their continuous and sometimes-hard-for-the-recipient-to-perceive (that is me) selfless support. I thank my parents, Kostas and Dimitra, my grandparents and uncle Takis, who all contributed more or less to who I am today. I also thank my siblings, Petros and Katerina, who witnessed, silently endured and ultimately supported my efforts throughout the years.

Moreover i would like to 'high-five' my other family, that is my friends, for playing a significant role in my life, by being part of some of its most memorable instances up till now, as well as standing by my side on any occasion. Though I may forget some, I thank Giorgos, Nikos, Anestis, Thanos, Alkis, Eleni and Vivi for the chance to get to know each other and walk on the 'same road'.

Finally, I would like to distinctly thank 3 of my friends that played a major role both during my studies and during my experiencing of life. Firstly, I would like to thank my long-time (no see, due to his PhD obligations) school friend Evangelos Giampazolias, who apart from his against-all-odds efforts to enlighten me, was always there for support, help, suggestions, conversations and endless laughter. Subsequently, I would like to thank my colleague Theodoros Gkountouvas (also long-time no see, due to PhD obligations) for the opportunity of coming in touch with a new world, sharing some of the most awesome faculty years, as well as constantly proving that a single '++' can make all the difference. Last but not least, I would like to thank my colleague Georgios Veletzas, whom I consider as one of my role models, for his unprecedented perception, spirit and force of will, which influenced not only my evolution but contributed in some aspects to the realisation of this diploma thesis.

Contents

Περίληψη i

Abstract iii

Acknowledgements v

Contents vii

List of Figures ix

List of Tables x

List of Algorithms x

1 Introduction 1
- 1.1 Modern Computer Architectures 1
- 1.2 Memory Hierarchy 1
- 1.3 Cache 3
  - 1.3.1 Structure & Functionality 3
  - 1.3.2 Associativity 4
- 1.4 CMP's Approach 4

2 Cache Replacement Policies 5
- 2.1 Replacement Policy Properties 5
- 2.2 Commonly Used Replacement Policies 6
- 2.3 Application Profiles based on Cache Occupancy 7
- 2.4 Replacement Schemes for Shared Last Level Caches 8
  - 2.4.1 Partitioning Based Policies 8
  - 2.4.2 Insertion Based Policies 11
  - 2.4.3 Hybrid Policies 16
- 2.5 Synopsis 16

3 Experimental Methodology 17
- 3.1 CMP$im Simulator 17
- 3.2 Benchmark Suite 18
  - 3.2.1 SPEC CINT 2006 18
  - 3.2.2 SPEC CFP 2006 19
- 3.3 Classification of Benchmarks 20
  - 3.3.1 Cache-friendly 20
  - 3.3.2 Cache-fitting 20

3.3.3 Cache-thrashing & Cache-streaming 20
3.4 Multi-core Simulation Configuration 24
3.5 Synopsis 24

4 Results, Optimisation & Analysis 25
4.1 Metrics 25
4.2 Comparison of LLC Partitioning Schemes 25
4.3 Optimisation of *ABFCP* 27
4.3.1 Distribution of Ways 27
4.3.2 Repartition Phase 29
4.3.3 Set Flexibility 30
4.4 New Estimator for *ABFCP* 31
4.5 Effect of Sampling for *ABFCP* 33
4.6 Effect of Reduction of Bloom Filter's Tag 34
4.7 Comparison of all presented LLC schemes 35

5 Conclusions 39
5.1 Overview 39
5.2 Future Work 40

Appendices 41

Appendix A Details for LLC Replacement Policies 43
A.1 UCP's Partitioning Algorithms 43
A.2 ABFCP's Partitioning Algorithms 45
A.3 PIPP's stream-sensitive mechanism 46

Bibliography 47

List of Figures

1.1 Example of memory hierarchy in a 4-core CMP. 2
1.2 Main Memory and Cache Organisation. 3

2.1 Victim selection, insertion and promotion strategies of *LRU*. 5
2.2 *LRU* stack property. 8
2.3 *UCP* scheme for a dual core processor. 9
2.4 Utility Monitor. 9
2.5 *ABFCP* scheme for a dual core processor. 10
2.6 Bloom Filter. 11
2.7 *LRU* , *LIP* and *BIP* policy. 12
2.8 *DIP*. 15
2.9 *TADIP*. 15

3.1 Cache-friendly benchmarks. 21
3.2 Cache-fitting benchmarks. 22
3.3 Cache-thrashing benchmarks. 23
3.4 Cache-streaming benchmarks. 23

4.1 Throughput of LLC partitioning policies. 26
4.2 Weighted speedup of LLC partitioning policies. 26
4.3 Harmonic mean fairness of LLC partitioning policies. 27
4.4 BFMON for an 8-way cache. 28
4.5 Throughput of *ABFCP* original and per-way flexible. 28
4.6 Harmonic mean fairness of *ABFCP* original and per-way flexible. 29
4.7 Throughput of *ABFCP* versions for both resetting and halving of counters. . . 29
4.8 Harmonic mean fairness of *ABFCP* versions versions for both resetting and halving of counters. 30
4.9 Throughput of *ABFCP* versions for both per-set and uniform partitioning. . . 30
4.10 Harmonic mean fairness of *ABFCP* versions for both per-set and uniform partitioning. 31
4.11 Original vs New *ABFCP*. 31
4.12 Throughput and Harmonic mean fairness of original and new *ABFCP* versions. 32
4.13 *ABFCP* sampling versions. 33
4.14 Throughput and Harmonic mean fairness of best 3 *ABFCP* versions for *Sampling* and *GroupSampling*. 33
4.15 Throughput and Harmonic mean fairness of best *ABFCP* version, namely nA+-1uni_div2_32_s, with varying size of BF's tag. 34
4.16 Throughput of LLC replacement schemes. 35

4.17 Harmonic mean fairness of of LLC replacement schemes. 35
4.18 Throughput of *LRU* managed LLC for different cache configurations, normalised to best new *ABFCP*. 37

List of Tables

2.1 *RRIP*. 14

3.1 Workload mixes. 24

4.1 Differences between *ABFCP* and *UCP*. 27
4.2 Required Hardware Overhead of LLC schemes 36

A.1 Possible Partition Changes. 45

List of Algorithms

1 UCP's Greedy Algorithm 44
2 UCP's Lookahead Algorithm 44
3 ABFCP's Linear Algorithm 45
4 ABFCP's new Estimator Linear Algorithm 46

"Computer Science is no more about computers than astronomy is about telescopes."

Edsger W. Dijkstra

"I do not fear computers. I fear the lack of them."

Isaac Asimov

1 Introduction

It lies within the impositions of modern times, that performance and efficiency come at low cost, while such demand is ever-increasing. The trigger for this change was mainly the advent of computing era, which since its appearance has been an expanding field, becoming the point of reference while redefining our perception for most everyday aspects. Computers have changed radically everyday life, with some saying it is for the best, while others still remain suspicious. It is a truly powerful and unprecedented invention and lies within human reach to make best use of it, trying to preserve balance between one's potential and moral, while hunting for "knowledge".

1.1 Modern Computer Architectures

In an attempt to achieve better-performing computing systems, scientists used several techniques, among which was increasing processor's (CPU) frequency, introducing instruction level parallelism (via pipelining or superscaling) or thread level parallelism (via multi-threading or multi-processing). Thus, the transition from single- to multi-processing systems was realised, otherwise known as chip multiprocessors or CMPs, which is the most common case nowadays. However, in a computing system where several components must operate in coordination, CPU is -currently- the fastest working one, starving most of the time for data, waiting for their arrival from the closest memory. The speed-performance bottleneck is caused by the generally-slower operating memory, a problem that scientists tried to solve by adopting -among others- a memory hierarchy logic.

1.2 Memory Hierarchy

Memory is divided into several hierarchical layers resembling a pyramid, where closer-to-CPU and smaller-in-size memories respond faster than more distant and bigger ones. This notion would provide requested data quicker by alleviating the bottleneck problem and therefore result in a better performing system. Ahead of the main memory a group of smaller memories is placed, known as *cache* (french for 'hidden'), in order to hold some of the

more frequently used data closer to the processor, thus allowing it to complete execution of instructions sooner rather than stalling and waiting for data. The memory hierarchy is usually distributed as follows, with the top of the list comprising fast, expensive and small in size units, while at the bottom reside slower, cheaper and bigger units:

1. *Processor Registers*, which are the closest units to the CPU, containing all data necessary during execution time.

2. *Cache*, which contains copies of main memory blocks and is mainly divided into two independent categories, instruction and data cache, with data cache usually organised in more levels:

    (a) Level 1 Cache (L1C), private to each core, split into:

        i. Level 1 Instruction Cache (L1IC), which speeds up instruction fetch.

        ii. Level 1 Data Cache (L1DC).

    (b) Level 2 Cache (L2C), either private to each core or shared by every two cores, unless it is the last level cache, when it is shared by all cores.

    (c) Level 3 Cache (L3C) or last level (LLC), shared by all cores.

3. *Main Memory*, which contains instructions and data of every executing application and provides them to CPU and therefore to higher levels in memory hierarchy as well.

Every memory hierarchy layer can only trade data with the layer above and below it. The smallest amount of data that can be transferred between layers is called *block*. Figure 1.1 shows an example of a 4-core CMP's memory hierarchy.

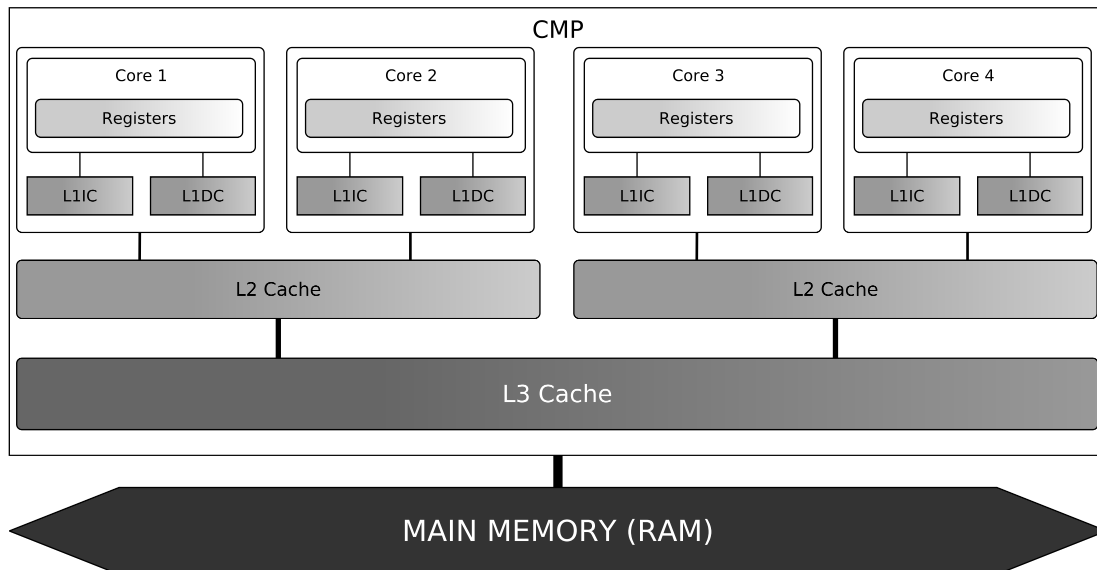


Figure 1.1: Example of memory hierarchy in a 4-core CMP.

Main memory is vital for a system, since a program is loaded there before starting execution. Main memory consists of a number of cells, whose size is called *word length* and can store a certain amount of bytes. Each cell has a unique unalterable *address*, through which a CPU can access its contents. If there are m cells and each address is represented by n digits, then the main memory addresses vary from 0 to $2^n - 1$ with $2^n = m$, as can be seen in Figure 1.2.

1.3 Cache

When the processor needs to read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory. From now on cache will denote LLC, unless stated otherwise. Some of cache's characteristic properties are presented subsequently.

1.3.1 Structure & Functionality

Cache is organised similarly to main memory and its fundamental terminology follows:

- *Entry/Block/Line* is the basic structural unit of each cache, similar to main memory cells, composed of the data block (copied group of main memory words) and the tag (identifier used for resolving an access into a hit or miss).
- *Set* is a group of entries, whose tags are checked simultaneously during cache accesses.
- *Way* is equal to the number of entries in a set and is denoted by the degree of cache associativity.

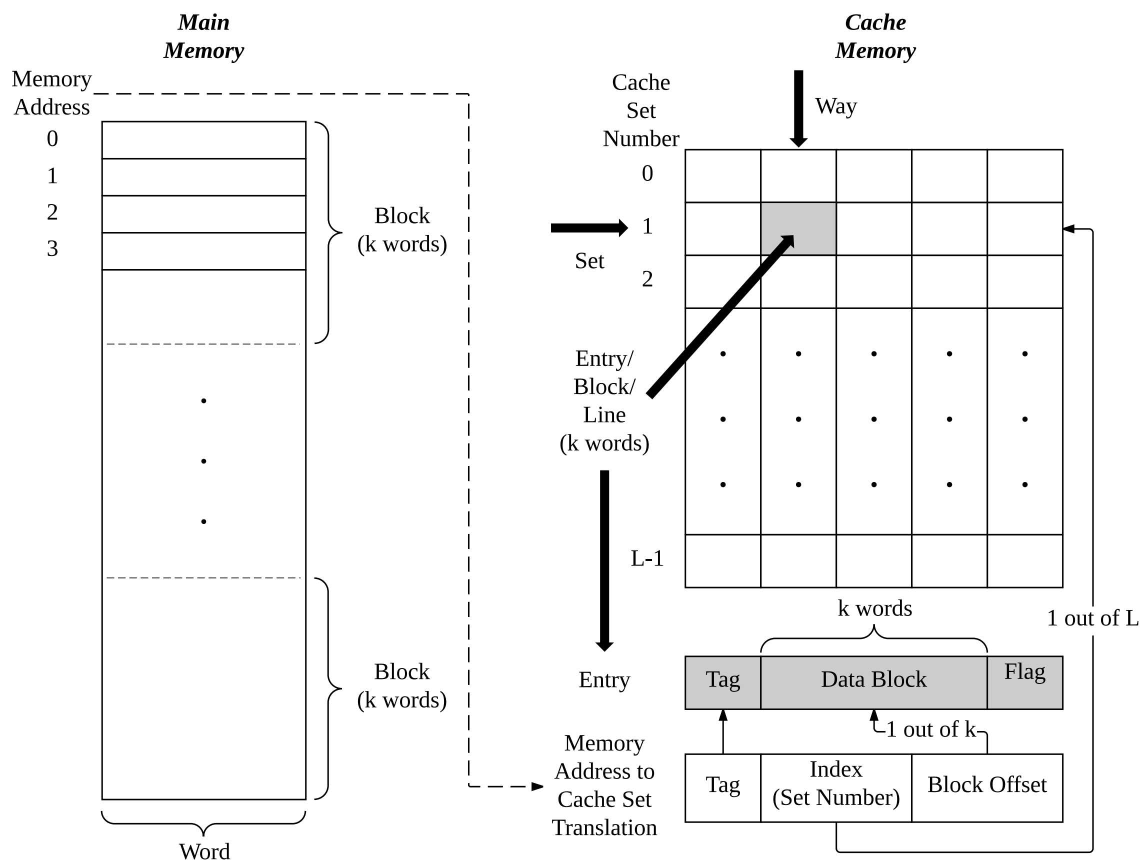


Figure 1.2: Main memory (on the left) and a 5-way set associative cache (on the right).

When the processor requires access to a location in main memory, it first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in any cache lines that might contain that address, by comparing requested with

stored tag. If the processor finds that the memory location is in the cache, a *cache hit* has occurred. However, if the processor does not find the memory location in the cache, a *cache miss* has occurred. In the case of:

- a cache hit, the processor immediately reads or writes the data in the cache line
- a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.

On a cache miss, the data arriving from main memory may have to replace another cache block, due to cache's limited storage capacity. A replacement policy denotes which among the existing cache blocks should be evicted.

1.3.2 Associativity

Considering that cache is orders of magnitude smaller than main memory, there has to be a mapping/association of main memory addresses to cache entries. There are mainly three types of cache, with respect to associativity:

- *Direct mapped*, where each main memory block is associated with exactly one cache entry.
- *Fully associative*, where each main memory block can be associated with any cache entry.
- N*-way set associative*, where each main memory block can be associated with any of the N cache entries (ways), contained in a specific cache set. For instance a N-way set associative cache containing B total entries, is organised in $L = B/N$ sets.

In fact a direct mapped resembles a 1-way set associative cache and a fully associative is like a B-way set associative cache, where B denotes the total number of entries in the cache. Figure 1.2 shows a related example.

1.4 CMP's Approach

Caches play an important role in system performance and the key aspects of reducing their response time are miss rate, miss latency/penalty and hit time. The two latter remain beyond the scope of this diploma thesis, which mainly focuses on techniques for miss rate reduction.

Last Level Cache is shared by all available cores, whether it resides within a single-core or a multi-core system, so its effective management is crucial, as it is the last available level of low-latency and fast-responsive memory, before requesting the desired data from the distant and slower main memory. It would be ideal if every request resulted in a cache hit and although this cannot be true, due to *compulsory misses* (first request to a specific memory address), hardware designers try to minimise *capacity misses* (no more available cache storing space) and *conflict misses* (that could have been avoided, had the cache not evicted an entry earlier).

Several techniques have been proposed for reduction of capacity or conflict misses, like increasing cache size and/or associativity, using hardware prefetching, using NUCA (Non Uniform Cache Access), optimising replacement policies etc. **In this diploma thesis a few LLC replacement policies and their significance in system performance are examined.**

"Computers are useless. They can only give you answers."
Pablo Picasso

"Science is what we understand well enough to explain to a computer; art is everything else."
Donald Ervin Knuth

2 Cache Replacement Policies

When a block is transferred from a lower layer to a higher layer of memory hierarchy, it is probable that it will have to take the place of another block. A *replacement policy* designates which block/victim is to be evicted. This operation concerns all caches with associativity more than one, so direct mapped caches are excluded, since only then more than one candidate evictees are available. In this chapter replacement policies for single-core and multi-core systems will be presented.

2.1 Replacement Policy Properties

Most replacement policies are based on a "priority" stack mechanism, so as to track how important each cache block is, which helps them decide the next victim. Each replacement policy usually consists of 3 distinct strategies/components:

- *Victim selection*, which indicates on a cache miss whose block turn is to be evicted.
- *Insertion*, which indicates on a cache miss where in the priority stack the incoming block will be placed.
- *Promotion*, which indicates on a cache hit where in the priority stack a block will be promoted.

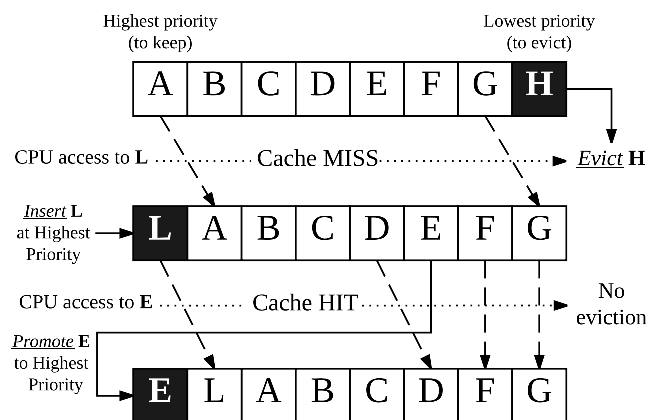


Figure 2.1: Victim selection, insertion and promotion strategies of *LRU*.

Figure 2.1 shows an example of *LRU* replacement policy, which is subsequently presented.

2.2 Commonly Used Replacement Policies

Replacement policies that are often met on single-core systems and have influenced replacement schemes for shared last level caches in CMPs are presented below, together with their strategies.

$\rightharpoonup$***LRU (Least Recently Used)*** is universally accepted as the best replacement policy for single-core systems, indicating that on a cache miss the least recently accessed block is replaced. The 3 distinct strategies of *LRU* are:

- *Victim selection*: evict block at LRU (Least Recenlty Used) recency position.
- *Insertion*: insert block at MRU (Most Recenlty Used) recency position.
- *Promotion*: promote accessed block to MRU and demote remaining blocks accordingly towards LRU recency position.

An example of its usage is presented in Figure 2.1. *LRU* generally has the best performance among all other policies, but it is quite expensive, requiring log_2N bits per block to implement the recency stack mechanism (4 bits for 16 way cache), for an N-way set associative cache.

$\rightharpoonup$***NRU (Not Recently Used)*** is a derivative of ***PLRU (Pseudo-LRU)***, which in turn is an approximation of *LRU*, that uses 1 bit to implement its recency stack and almost always discards one of the least recently used items instead of the least recently used. *NRU* is able to distinguish between two priorities: near-immediate and distant. The 3 distinct strategies of *NRU* are:

- *Victim selection*: evict one among the blocks at distant recency position (e.g. leftmost). If none found, convert all near-immediate recency positions to distant and repeat.
- *Insertion*: insert a block at near-immediate recency position.
- *Promotion*: promote accessed block to near-immediate recency position.

NRU provides adequate performance for low hardware overhead and is usually a good alternative over *LRU*, that several hardware manufacturers [1, 2] choose for their systems.

$\rightharpoonup$***FIFO (First In First Out)*** is a replacement policy, which also uses a stack logic per set, where a new block is inserted at the head, while the block at the tail is kicked out. The 3 distinct strategies of *FIFO* are:

- *Victim selection*: evict the oldest block (stack tail), thus the first to come is also the first to go.
- *Insertion*: insert at newest 'age' (stack head), namely the incoming block will be evicted last among the existing blocks.
- *Promotion*: no promotion required, since only insertion priority matters during victim selection.

Due to its simplicity, *FIFO* has a low hardware overhead, but usually performs poorly for practical applications, so it is rarely used in its unmodified form.

$\rightharpoonup$***RANDOM*** is a replacement policy, which has minimal cost on hardware overhead, since there is no need for tracking any kind of history or recency. The 3 distinct strategies of *RANDOM* are:

- *Victim selection*: evict a block, which is randomly picked.

- *Insertion*: insert anywhere in set, since victim selection is performed randomly, requiring no priority information.
- *Promotion*: no promotion required, since victim selection is performed randomly.

RANDOM policy usually performs better than *FIFO* and in some cases (e.g. big loops with memory accesses that exceed cache size) even better than *LRU*. However it generally performs worse than *LRU* so it is rarely used in shared caches. Some hardware manufacturers use it for private L1IC (UltraSPARC T2 [2]).

2.3 Application Profiles based on Cache Occupancy

Each application running on a core has its own cache profile, described by its cache access pattern. In general these access patterns can be classified in four main categories, as presented by Jaleel et al. [3]. If a_i stands for a cache line address, $(a_1, \ldots, a_k)^N$ stands for a temporal sequence of references to k unique addresses that repeats N times and $P_\epsilon(a_1, \ldots, a_k)$ denotes a temporal sequence that occurs with some probability ϵ, then the classification is as follows:

- *Recency-friendly Access Patterns*: $(a_1, a_2, \ldots, a_{k-1}, a_k, a_k, a_{k-1}, \ldots, a_2, a_1)^N$, for any value of k. This access pattern benefits from LRU replacement and may degrade performance with any other replacement policy. Applications whose behaviour could be summarised by similar access patterns are characterised as ***cache-friendly***.

- *Thrashing Access Patterns*: $(a_1, a_2, \ldots, a_k)^N$. When $k \leq$ cache size, then the working set of applications behaving in this manner fits in the cache, and such applications are known as ***cache-fitting***. However if $k >$ cache size, then LRU causes no cache hits, with applications labeled as ***cache-thrashing***. The optimal replacement policy would preserve some of the working set in the cache.

- *Streaming Access Patterns*: $(a_1, a_2, \ldots, a_k)$, where k can be infinite. Similar access patterns receive no cache hit under any replacement policy, therefore replacement decisions are irrelevant in presence of ***streaming*** applications.

- *Mixed Access Patterns*: $[(a_1, \ldots, a_k, a_k, \ldots, a_1)^A \mathbf{P}_\epsilon(\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_k}, \mathbf{a_{k+1}}, \ldots, \mathbf{a_m})]^N$ or $[(a_1, \ldots, a_k)^A \mathbf{P}_\epsilon(\mathbf{b_1}, \mathbf{b_2}, \ldots, \mathbf{b_m})]^N$, where $k <$ cache size and $0 < \epsilon < 1$. The **bold** subpatterns represent a *scan*, namely a big sequence of accesses with low chance of reusability. When $m + k <$ cache size, then both the working set and the scan fit into the cache and *LRU* works well (***cache-fitting*** applications). If however $m + k >$ cache size, then *LRU* discards the frequently referenced working set over the scan, ultimately causing any reference to the working set after the scan to always miss (***cache-thrashing*** applications). An optimal replacement policy would keep as much of the working set in cache as possible, efficiently recognising and discarding any blocks belonging to a scan, in order to provide the opportunity to the working set to be reused.

This classification helps to better understand the possible behaviour of any application either executing on its own or in the presence of others, as well as explore the boundaries of optimisation that a replacement policy may achieve.

2.4 Replacement Schemes for Shared Last Level Caches

Modern CMPs appearance triggered the need to address a bunch of hardware design issues, one of which being the efficient utilisation of shared LLC, in order to avoid long latency misses to main memory. To achieve that, LLC should retain more data from applications which benefit from it than from those who don't. However *LRU* treats every application based on demand rather than on performance benefit, thus risking to privilege with resources an application that has no gain while starving a low demand high performing application. Consequently in presence of more than one cores competing for resources, it is imperative that LLC management opts for *performance* rather than *demand*.

Since application profiles differ from each other, a thread unaware LLC replacement policy that treats all of them as one application with mixed access pattern is inefficient. On the contrary, hardware designers should orient towards more *thread aware* policies, that handle each application separately depending on whether it shows promising utilisation of cache resources or not. Previously presented common replacement policies lack on per thread information and decision making, opening the frontier for relevant research. Many of the subsequently described replacement schemes inherit *LRU* stack logic, but fine-tune accordingly insertion, promotion and victim selection policies, so as to achieve a ***performance-based*** and ***thread-aware*** management of cache.

2.4.1 Partitioning Based Policies

The first approach towards thread-aware policies for LLC was the *partitioning* technique. Macroscopicaly, partitioning distributes cache resources among all cores, instead of simply allowing every core to compete with each other under *LRU* on a demand basis. Microscopically, the partitioning decisions are enforced during victim selection. Therefore, partitioning algorithms try to compute the best distribution of cache resources among competing applications in order to achieve better performance. The underlying policy is usually *LRU* but can be altered as well.

A basic *LRU* property, which has facilitated the estimation of each core's cache profile, is the ***LRU stack property*** [4], which denotes that if an access hits on an *LRU* managed cache with N ways, then it is guaranteed to hit on a cache with more ways. Figure 2.2 shows a 4-way set associative cache and the breakdown of cache hits on the different recency positions from MRU to LRU. If cache associativity is reduced from 4 to 3 ways, then misses would increase by 10 to a total of 35. Further reducing cache associativity to 2 ways, results in 50 misses and with only 1 way 70 misses occur. Therefore if the distribution of hits is known for an N-way cache, then it is possible to compute hits and misses for all different N associativity configurations.

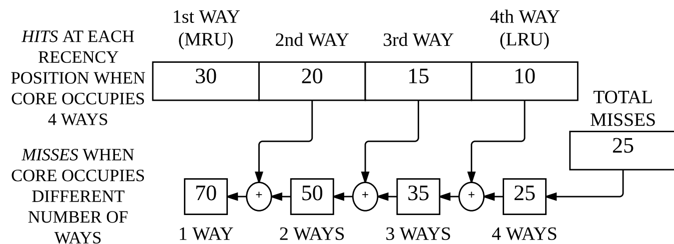


Figure 2.2: *LRU* stack property for 4-way set associative cache.

$\rightharpoonup$***UCP (Utility based Cache Partitioning)***, proposed by Qureshi and Patt [5], distributes cache resources according to the *utility*, namely the change in misses, that an application is predicted to exhibit, depending on its occupancy of cache resources (ways) at that moment.

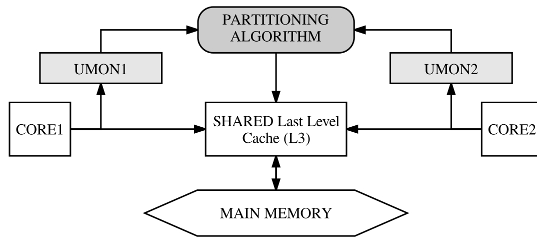


Figure 2.3: *UCP* scheme for a dual core processor.

UCP intends to provide more cache resources to whichever application has bigger utility and vice versa. In order to compute the utility of each core, information concerning each core's per-way misses must be available. For this purpose there was introduced a per-core external (not part of cache) mechanism, called *UMON (Utility MONitor)*, that tracks all LLC accesses of the corresponding core. Each UMON simulates the behaviour of its core, if it had all the cache availabe to itself, managed by *LRU*. Finally a partitioning algorithm computes the utility for each core derived from UMONs (in a way described below) and decides the distribution of cache resources, ensuring 1 way per set for each core. *UCP*, which can be seen in Figure 2.3, enforces partition during replacement and contains 3 distinct strategies:

- *Victim selection*: if the incoming block belongs to a core possessing more cache blocks than its partition indicates then the block at the LRU recency position of this core is evicted, otherwise the block at the LRU recency position among all the remaining cores is replaced.
- *Insertion*: insert at MRU recency position.
- *Promotion*: promote accessed block to MRU and demote remaining blocks accordingly towards LRU recency position.

A UMON is composed by a group of entries, called *ATD (Auxiliary Tag Directory)*, which contains as many sets and ways as LLC does, but stores only the corresponding tag, omitting the unnecessarily big -for its purposes- data blocks. Moreover each UMON's set contains N *counters*, one per way, in case of a N-way setassociative LLC, in order to store the per way hits of the application. Whenever a recorded by UMON access results in a ATD hit, the counter corresponding to the hit occurring recency/way is incremented by one. Thus, by using *LRU stack property* and by containing the necessary information for utility computation, UMON calculates the number of misses for all possible ways an application can occupy. UMON's initial design, known as *UMON-local*, opted for flexibility and comprised per way hit counters for each cache set, being capable of different partition on a per set basis. Since its implementation cost was prohibitive, attempts to reduce hardware overhead resulted in *UMON-DSS*, which monitored not all but a few cache sets, using a tecnique called *dynamic set sampling (DSS)*, as well as having a hit counter per recency positionas can be seen in Figure 2.4. As few as 32 sets is an adequate and representative sample choice for UMON-DSS, which is able to take uniform (same for all sets) partitioning decisions and is ultimately used.

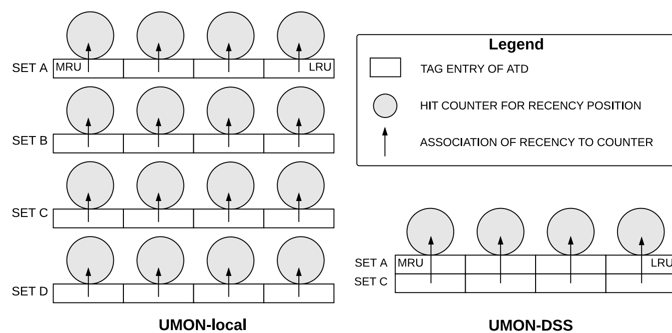


Figure 2.4: Utility Monitor.

When the partitioning algorithm is executed, it distributes cache ways among the applications, to reduce the total number of misses, which are directly correlated with the utility information in the hit counters of UMON. If $miss_a$ and $miss_b$ denote the number of misses

incurred by an application occupying a and b ($a < b$) ways respectively, then the *utility* U_a^b of increasing the ways from a to b for a N-way set associative cache with C competing cores is defined as:

$$U_a^b = miss_a - miss_b = \sum_{way=a+1}^{b} hitcounters[way], \text{ with } \begin{cases} a \in [1, N-C] \\ b \in [2, N-C+1] \end{cases} \quad (2.1)$$

If moreover A and B are two applications on a dual-core system, with utility functions UA and UB respectively, then their combined utility U_{tot} is given by:

$$U_{tot} = UA_1^i + UB_1^{N-i}, \quad \text{with} \quad i \in [1, N-1] \quad (2.2)$$

The partition with the biggest combined utility value among all possible partitions is chosen, ensuring at least 1 way to each application. This algorithm is known as *exhaustive* and finds the best among all possible partitions, but finding an optimal solution for more than 2 cores becomes prohibitively complex (NP-hard). For this reason two approximate algorithms were proposed, that find a nearly optimal solution: *Greedy* and *Lookahead* (see Appendix A.1), among which Lookahead is used due to high efficacy and low complexity. The partitioning algorithm is repeated every five million cycles and after each repartition all hit counters are reduced in half, combining information from past and present.

⇀***ABFCP (Adaptive Bloom Filter Cache Partitioning)*** was proposed by Nikas et al. [6], who wanted to retain flexibility of per set partitioning. *UCP*'s uniform partitioning is a not so flexible choice, since it enforces the same partition on all sets, as it is the only practical choice due to the hardware overhead required. If for instance competing applications demonstrate different behaviour per set, a sudden scan phase mapped on few sets would influence decisions for the remaining sets as well. *ABFCP* utilises among others a structure for maintaining probabilistic set membership, called *Bloom Filters*. Similar to UMON, Bloom Filter is a mechanism that keeps track of accesses to LLC for each core, in order to use the gathered information for partitioning decisions. *ABFCP*, which can be seen in Figure 2.5, enforces partition during replacement and contains 3 distinct strategies:

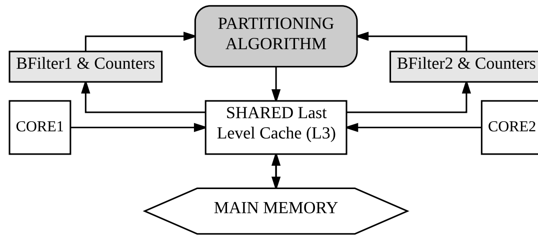


Figure 2.5: *ABFCP* scheme for a dual core processor.

- *Victim selection*: if the incoming block belongs to a core possessing more cache blocks than its partition indicates (over-allocated) then the block at the LRU recency position of this core is evicted, otherwise the block at the LRU recency position among the remaining over-allocated cores is replaced (at least one will exist).
- *Insertion*: insert at MRU recency position.
- *Promotion*: promote accessed block to MRU and demote remaining blocks accordingly towards LRU recency position.

Nikas et al. [6] proposed using *Partial-Address Bloom Filter (BF)*, developed by Peir et al. [7], to store per core information about cache blocks that were replaced from the LLC.

BF is an array containing 2^k 1-bit entries, where addressing is done by the k least significant bits of the tag of a cache block, as shown in Figure 2.6. On a cache miss the evicted cache block sets the corresponding BF entry. Moreover if the incoming block causing the LLC miss, is found in BF, namely the corresponding entry is set, then a *far-miss* is detected, denoting that this miss may have been a hit, had the core been assigned more ways. In total one BF is used for each core on a per set basis. An arising issue is that there is no certainty that a BF hit means that the block looked up was previously present in the LLC, resulting in a *false-positive*, caused by a problem knows as "aliasing", where more than one tags may be mapped to the same BF entry, as only k bits are used for the indexing. However in case of a BF miss, it is certain that the block looked upon was previously absent from the cache. Moreover the order in which the BF entries are set cannot be known and such kind of information would require additional hardware. At the same time the true entries contained in a BF could be more than cache associativity, therefore a far-miss detection cannot possibly lead to how many more ways the core should have been given for that miss to become a hit.

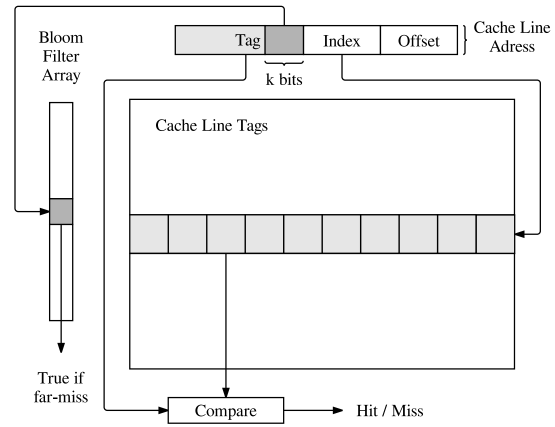


Figure 2.6: Bloom Filter.

Apart from BFs, a pair of counters for each core on a per set basis is used. One counter (C_{BFHIT}) is incremented when BF detects a far-miss. The other counter (C_{LRUHIT}) is incremented when an access results in a hit at the LRU entry owned by the core. Exploiting the *LRU stack property*, these counters can estimate possible performance gains or losses by changing a core's allocation. More specifically, if a core is deprived of one way, then the hits in the LRU recency position would become misses, therefore for this case C_{LRUHIT} estimates performance loss. Similarly, C_{BFHIT} tracks far-misses (with a possibility of false-positives), that might result in hits if the core possessed at least one more way, thus being an adequate estimation for performance gain. Since C_{BFHIT} contains information for an unknown number of true entries, a scaling factor α is added, in order to achieve a per way gain estimation. Therefore gain or loss of 1 extra way are calculated as follows:

$$\text{If} \quad \alpha = 1 - \frac{ways_{occupied}}{associativity} \quad \text{then:} \quad \begin{cases} loss_1 & = & C_{LRUHIT} \\ gain_1 & = & \alpha \times C_{BFHIT} \end{cases} \tag{2.3}$$

Combining the information of those counters, a partitioning algorithm decides per set if the allocated resources of each core, will be increased by one, reduced by one, or leaved unchanged. Rather than evaluating all possible partition changes and choosing the best choice that maximises a given metric, Nikas et al. [6] proposed a *Linear algorithm* that selects the best partition or a good approximation thereof (see Appendix A.2). The partitioning algorithm is executed every one million cycles and at the end of each repartition both counters and BFs are reset.

2.4.2 Insertion Based Policies

Considering that *LRU* is a well-suited policy for handling cache-friendly applications, hardware designers focused on how to create thrash-resistant and scan-resistant policies. Apart from the need for per thread information and management of shared LLC, the primary

idea was to vary the *insertion policy* of new blocks, instead of partitioning, in an attempt to place cache-friendly applications towards MRU and cache-unfriendly ones towards LRU recency positions.

Thread Unaware Insertion Policies

Initially a few thread unaware insertion policies suitable for single-core systems were proposed, which are shown below:

⇀***LIP (LRU Insertion Policy)*** was proposed by Qureshi et al. [8], who came to the conclusion that under *LRU*, applications with a lot of cache misses (thrashing & streaming applications) are unnecessarily kept in cache, since all new blocks are inserted at MRU. The 3 distinct strategies of *LIP* are:

- *Victim selection*: evict block at LRU recency position.
- *Insertion*: insert block at LRU recency position, which makes it candidate victim, unless it is re-referenced nearly immediately and gets promoted.
- *Promotion*: promote accessed block to MRU and demote remaining blocks accordingly towards LRU recency position.

An example of its usage is presented in Figure 2.7, for $\epsilon = 0$. *LIP* is stream- and thrash-resistant.

⇀***BIP (Bimodal Insertion Policy)*** was also proposed by Qureshi et al. [8] and came as a generalisation to *LRU* and *LIP*, after the conclusion that *LIP* treats all accesses similarly and "suspiciously", failing to favour cache-friendly applications. The 3 distinct strategies of *BIP* are:

- *Victim selection*: evict block at LRU recency position.
- *Insertion*: insert block at MRU recency position with -usually low- probability ϵ and at LRU recency position with $1 - \epsilon$.
- *Promotion*: promote accessed block to MRU and demote remaining blocks accordingly towards LRU recency position.

BIP degenerates to *LIP* for $\epsilon = 0$ and *LRU* for $\epsilon = 1$, as can be seen in Figure 2.7. Behaviour of *BIP* is therefore controlled by ϵ.

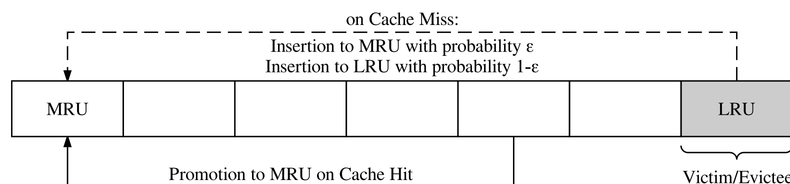


Figure 2.7: *BIP* policy, which equals to *LIP* for $\epsilon = 0$ and *LRU* for $\epsilon = 1$.

⇀***DIP (Dynamic Insertion Policy)*** was introduced by Qureshi et al. [8] in an attempt to combine two different cache management policies: cache-friendly *LRU* and thrash/stream-resistant *BIP*. *DIP* dedicates few cache sets to each of the two -*BIP* and *LRU*- competing policies and the policy that causes fewer misses at those dedicated sets is applied to the rest (follower) sets of the cache. Dedicated sets are called *SDM or Set Dueling Monitors*, half of which work with *BIP* (*BIP* SDM) and the other half with *LRU* (*LRU* SDM). Moreover

both SDM use a saturating counter, called *PSEL or Policy SELector*, which indicates which policy is better. PSEL is incremented by one on a cache miss occurring at *LRU* SDM and is decremented by one on a cache miss at *BIP* SDM. Let <P> denote the policy followed by each cache set, with *LRU* represented by <0> and *BIP* by <1>. If MSB or Most Significant Bit of PSEL counter is one, meaning it is more than half full, then *LRU* causes a lot of misses and *BIP* is chosen for the follower sets. Otherwise, if MSB of PSEL is zero, *LRU* is chosen. Figure 2.8 shows *DIP* scheme. In general <P> policy of the follower sets is specified by $P = MSB(PSEL)$. The 3 distinct strategies of *DIP* are:

- *Victim selection*: evict block at LRU recency position.
- *Insertion*:
  - if <P>=<0>, then new block is inserted at MRU recency position with -usually low- probability ϵ and at LRU recency position with $1 - \epsilon$ (*BIP* insertion).
  - if <P>=<1>, then new block is inserted at MRU recency position (*LRU* insertion).
- *Promotion*: promote accessed block to MRU and demote remaining blocks accordingly towards LRU recency position.

As few as 32 sets for each SDM are proven sufficient and hence are used for the rest of the study.

$\rightharpoonup$***RRIP (Re-Reference Interval Prediction)*** was proposed by Jaleel et al. [9] in an attempt to generalise *NRU*. *RRIP* implements its own recency stack, using M bits per block, called RRPV or Re-Reference Interval Values, being able to store 2^M different possible re-reference intervals, with highest priority predicted to be re-referenced in the *near-immediate* future and lowest priority predicted to be re-referenced in the *distant* future. If $M = 1$ RRIP degenerates into *NRU*, while if $M > 1$ intermediate re-reference intervals are enabled. Primary objective of RRIP is to prevent blocks with distant re-reference interval from polluting the cache, but in the absence of external information it can take decisions statically, therefore is also known as ***SRRIP (Static RRIP)***. The 3 distinct strategies of *SRRIP* are:

- *Victim selection*: evict one among the blocks with distant RRPV (e.g. leftmost). If none found, demote all RRPVs towards distant RRPV and repeat.
- *Insertion*: insert new block at long RRPV, namely one RRPV before distant RRPV.
- *Promotion*: promote accessed block to highest priority, that is near-immediate RRPV.

SRRIP always predicts a long re-reference interval for incoming blocks, in order to prevent cache pollution but also provide a cache-friendly application the chance to “express itself”. Table 2.1 shows an example of using *LRU*, *NRU* and *SRRIP* for the same sequence of accesses, showing the advantage of *SRRIP* that accomplishes two hits more than the other two policies.

$\rightharpoonup$***BRRIP (Bimodal RRIP)*** was also proposed by Jaleel et al. [9] as an equivalent of *BIP*, since *SRRIP* can be considered generalisation of *NRU*. The 3 distinct strategies of *BRRIP* are:

- *Victim selection*: evict one among the blocks with distant RRPV (e.g. leftmost). If none found, demote all RRPVs towards distant RRPV and repeat.
- *Insertion*: insert new block at long RRPV, namely one RRPV before distant RRPV, with -usually low- probability ϵ and at distant RRPV with $1 - \epsilon$.

Next Ref	RRIP head (MRU) ↓			RRIP tail (LRU) ↓											
A_1	Inv 0	Inv 1	Inv 2	Inv 3	miss	Inv 1	Inv 1	Inv 1	Inv 1	miss	Inv 3	Inv 3	Inv 3	Inv 3	miss
A_2	A_1 0	Inv 1	Inv 2	Inv 3	miss	A_1 0	Inv 1	Inv 1	Inv 1	miss	A_1 2	Inv 3	Inv 3	Inv 3	miss
A_2	A_2 0	A_1 1	Inv 2	Inv 3	hit	A_1 0	A_2 0	Inv 1	Inv 1	hit	A_1 2	A_2 2	Inv 3	Inv 3	hit
A_1	A_2 0	A_1 1	Inv 2	Inv 3	hit	A_1 0	A_2 0	Inv 1	Inv 1	hit	A_1 2	A_2 0	Inv 3	Inv 3	hit
B_1	A_1 0	A_2 1	Inv 2	Inv 3	miss	A_1 0	A_2 0	Inv 1	Inv 1	miss	A_1 0	A_2 0	Inv 3	Inv 3	miss
B_2	B_1 0	A_1 1	A_2 2	Inv 3	miss	A_1 0	A_2 0	B_1 0	Inv 1	miss	A_1 0	A_2 0	B_1 2	Inv 3	miss
B_3	B_2 0	B_1 1	A_1 2	A_2 3	miss	A_1 0	A_2 0	B_1 0	B_2 0	miss	A_1 0	A_2 0	B_1 2	B_2 2	miss
B_4	B_3 0	B_2 1	B_1 2	A_1 3	miss	B_3 0	A_2 1	B_1 1	B_2 1	miss	A_1 1	A_2 1	B_3 2	B_2 3	miss
A_1	B_4 0	B_3 1	B_2 2	B_1 3	miss	B_3 0	B_4 0	B_1 1	B_2 1	miss	A_1 1	A_2 1	B_3 2	B_4 2	***hit***
A_2	A_1 0	B_4 1	B_3 2	B_2 3	miss	B_3 0	B_4 0	A_1 0	B_2 1	miss	A_1 0	A_2 1	B_3 2	B_4 2	***hit***
	A_2 0	A_1 1	B_4 2	B_3 3		B_3 0	B_4 0	A_1 0	A_2 0		A_1 0	A_2 0	B_3 2	B_4 2	
	LRU stack position ↙					nrubit ↙					RRPV ↙				
	(a) *LRU*					(b) *NRU*					(c) *SRRIP*				
	Cache Hit: (i) move block to MRU Cache Miss: (i) replace LRU block (ii) move block to MRU					Cache Hit: (i) set nrubit of block to ‘0’ Cache Miss: (i) search for first ‘1’ from left (ii) if ‘1’ found go to step (v) (iii) set all nrubits to ‘1’ (iv) goto step (i) (v) replace block and set nrubit to ‘0’					Cache Hit: (i) set RRPV of block to ‘0’ Cache Miss: (i) search for first ‘3’ from left (ii) if ‘3’ found go to step (v) (iii) increment all RRPVs (iv) goto step (i) (v) replace block and set RRPV to ‘2’				

Table 2.1: *RRIP*.

- *Promotion*: promote accessed block to highest priority, that is near-immediate RRPV.

⇀***DRRIP (Dynamic RRIP)*** was finally proposed by Jaleel et al. [9] as an anology of *DIP*, which uses the already known SDM to choose the best of two competing policies, *SRRIP* and *BRRIP*, applying its decision to all follower sets. The 3 distinct strategies of *DRRIP* are:

- *Victim selection*: evict one among the blocks with distant RRPV (e.g. leftmost). If none found, demote all RRPVs towards distant RRPV and repeat.
- *Insertion*:
  - if <P>=<0>, then new block is inserted at long RRPV, with probability ϵ and at distant RRPV with $1-\epsilon$ (*BRRIP* insertion).
  - if <P>=<1>, then new block is inserted at long RRPV (*SRRIP* insertion).
- *Promotion*: promote accessed block to near-immediate RRPV.

Thread Aware Insertion Policies

Thread-aware policies for shared LLCs were influenced by thread unaware schemes and are presented below:

⇀***TADIP (Thread Aware DIP)*** was proposed by Jaleel et al. [3] as an expansion of *DIP* to shared LLCs. *DIP* performs well on single-core systems but fails to do so in multi-core in the absence of thread-aware information, treating all cache blocks the same, independently to whom core they belong. *TADIP* dedicates $2C$ SDM in total, namely two SDM for each of the C competing cores. Let <P_i> denote the policy followed by core i on a cache set, where $P_i = MSB(PSEL_i)$ and $i \in [0, C-1]$. For each core i, one SDM follows *BIP* and the other *LRU* for blocks belonging to core i, while following the best performing policy <P_j> for blocks of each j of the remaining cores. There are C PSEL counters, where each miss on LRU_SDM_i decrements $PSEL_i$ by one and each miss on BIP_SDM_i increments $PSEL_i$ by one. The follower sets use the best performing policy for each core.

TADIP decides for the best policy of one core accounting for the best policy of the remaining cores, as shown in Figure 2.9. The 3 distinct strategies of *TADIP* are:

- *Victim selection*: evict block at LRU recency position.
- *Insertion*:
  - if <P_i>=<0>, then new block of core i is inserted at MRU recency position with -usually low- probability ϵ and at LRU recency position with $1-\epsilon$ (*BIP* insertion).
  - if <P_i>=<1>, then new block of core i is inserted at MRU recency position (*LRU* insertion).
- *Promotion*: promote accessed block to MRU and demote remaining blocks accordingly towards LRU recency position.

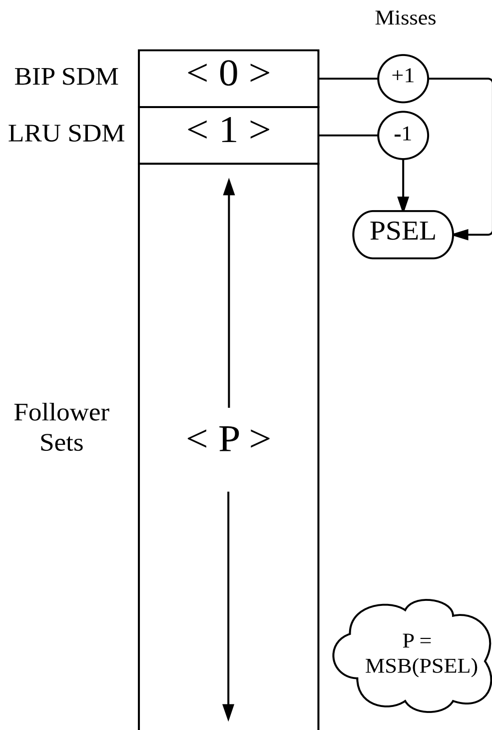


Figure 2.8: *DIP*.

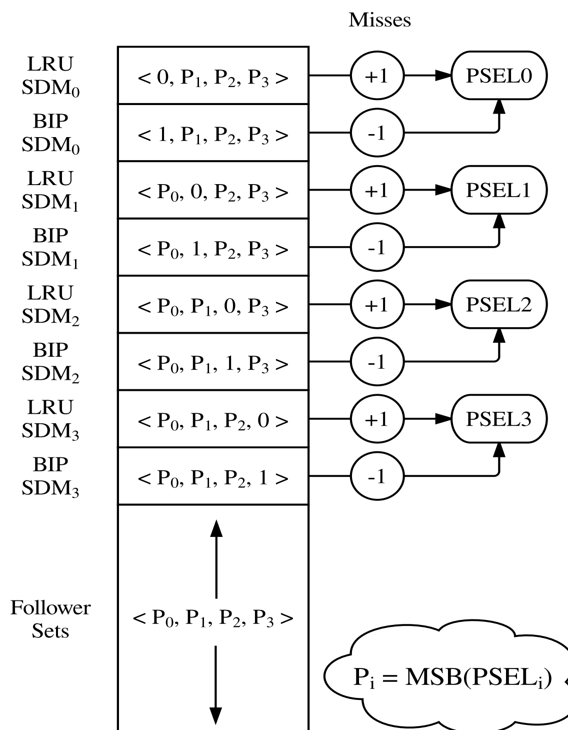


Figure 2.9: *TADIP*.

$\rightharpoonup$***TADRRIP (Thread Aware DRRIP)*** was proposed by Jaleel et al. [9] as an equivalent of *TADIP*, which uses SDM and PSEL counters the same way as *TADIP*. Apart from its low cost on hardware overhead, *TADRRIP* is able to take thread aware decisions, choosing between a stream/thrash resistant policy (*BRRIP*) and a policy that performs well in presence of cache-friendly applications (*SRRIP*), being also able to adjust the insertion of new blocks, presenting some sort of scan-resistance, by preventing cache pollution. The 3 distinct strategies of *TADRRIP* are:

- *Victim selection*: evict one among the blocks with distant RRPV (e.g. leftmost). If none found, demote all RRPVs towards distant RRPV and repeat.
- *Insertion*:
  - if <P_i>=<0>, then new block is inserted at long RRPV, with probability ϵ and at distant RRPV with $1-\epsilon$ (*BRRIP* insertion).

  - if <P_i>=<1>, then new block is inserted at long RRPV (*SRRIP* insertion).

- *Promotion*: promote accessed block to near-immediate RRPV.

2.4.3 Hybrid Policies

Finally *hybrid* LLC management schemes may be created, that combine all techniques available by exploiting their advantages to achieve better performance. One hybrid LLC policy is presented subsequently.

$\rightharpoonup$***PIPP (Promotion Insertion Pseudo-Partitioning)*** was proposed by Xie and Loh [10] and uses decisions made by a *partitioning* scheme (in this case *UCP*) in order to adjust accordingly *insertion policy* of each core. In fact UMON-DSS is used for each core, that provides a partition at the end of each repartition phase. This information could be derived from any partitioning scheme (e.g. *ABFCP*). *PIPP* doesn't enforce partitioning information during replacement, but instead uses it to during insertion. The 3 distinct strategies of *PIPP* are:

- *Victim selection*: evict block at LRU recency position.
- *Insertion*: insert at recency position equal to the indicated partition, with 1 way denoting LRU and N ways denoting MRU recency position for an N-way set associative cache.
- *Promotion*: promote accessed block one recency position towards MRU with probability p_{prom} and leave unchanged with $1 - p_{prom}$.

PIPP does not strictly enforce the target partitioning, but the combination of targeted insertion and incremental promotion creates results similar to explicit partition enforcement, hence pseudo-partitioning. There was also added a *stream-sensitive* mechanism to *PIPP* in order to recognise any cache-unfriendly applications easier (see Appendix A.3).

2.5 Synopsis

In this chapter single-core replacement policies, as well as multi-core proposed schemes have been presented. **Among these policies, *ABFCP* is chosen for further investigation, due to its promising per-set flexibility**.

"The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do."

Ted Nelson

"It's hardware that makes a machine fast. It's software that makes a fast machine slow."

Craig Bruce

3 Experimental Methodology

In order to avoid time-consuming, bug-risky and resource-hungry on-chip transfer of hardware design choices, *simulation* is an invaluable asset, that is widely used by both researchers and manufacturers. Apart from the apparent need for adaptive system simulators, various programms, with special features and behaviour have been introduced, known as *benchmarks*, in order to test thoroughly a system's components performance.

3.1 CMP$im Simulator

Jaleel et al. [11] introduced the *CMP$im Simulator*, an alternative to execution-driven and trace-driven simulation methodologies, that uses the binary instrumentation tool, *PIN*, created by Luk et al. [12]. Since CMP$im simulator is not publicly available, a free closed-code version is used for this diploma thesis, that permits alterations only to files related to the management of the LLC. This version resides online[1] and was used during a Cache Replacement Championship provided by The Journal of Instruction-Level Parallelism [13]. The provided simulation framework models a simple out-of-order processor with the following basic parameters:

- 128-entry instruction window with no scheduling restrictions (i.e., any instruction that is ready in the window can be scheduled out-of-order).
- Processor has an 8-stage, 4-wide pipeline. A maximum of two loads and a maximum of one store can be issued every cycle.
- Perfect branch prediction (i.e., no front-end or fetch hazards).
- All instructions have one-cycle latency except for cache misses. L1 cache misses / L2 cache hits are 10 cycles. L2 cache misses / L3 cache hits are 30 cycles, and L3 cache misses / memory requests have a 200-cycle latency. Hence, the total round-trip time from processor to memory is 240 cycles.

[1] http://www.jilp.org/jwac-1/

- The memory model will consist of a 3-level cache hierarchy, consisting of L1 split instruction and data caches, a L2, and a L3 (Last Level) cache. All caches support 64-byte lines. The L1 instruction cache is 32KB 4-way set associative cache with true LRU replacement. The L1 data cache is 32KB 8-way set-associative with true LRU replacement. The L2 data cache is 256KB, 8-way set-associative with true LRU replacement. The LLC's specifications (i.e. size, associativity, replacement scheme followed, etc.) are customisable and configured by the user/designer. The LLC is a non-inclusive (also non-exclusive) cache. L1 and L2 caches are private to each core, while LLC is shared by all cores.

- The number of cores is also customisable and ranges between 1, 2 and 4.

The version of PIN tool used is 2.7-31933, the version of gcc compiler is 3.4.6 and the simulator is compiled to run on a 64bit system under linux. The PIN tool is used to collect instruction-containing traces for any single-threaded application. Each trace is then in suitable form and can be feeded into the CMP$im for the simulation.

3.2 Benchmark Suite

The benchmarks used for simulations in this thesis belong to a quite popular and widely used collection, called *SPEC CPU 2006* [14], which is an industry-standardised, CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler. This benchmark suite includes the SPEC CINT (integer component) and the SPEC CFP (floating-point component) benchmarks, which contain 12 and 19 different benchmark tests respectively. Among these benchmarks, 17 are used for the simulations, the same way as *TADIP* [3] does. These benchmarks are described subsequently, whose notation comprises an identification number, a name and a letter denoting the input used.

3.2.1 SPEC CINT 2006

400.perlbench.d belongs to the general category of *programming languages*, written by Larry Wall et al., in C. It is a cut-down version of Perl v5.8.7, the popular scripting language, but SPEC's version has had most of OS-specific features removed. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs). The used input is 'diffmail.pl 4 800 10 17 19 300'.

401.bzip2.c belongs to the general category of *compression tools*, written by Julian Seward, in C. It is based on bzip2 version 1.0.3 of the same author, modified to do most work in memory, rather than doing I/O. The used input is 'chicken.jpg 30'.

403.gcc.s belongs to the general category of *Compilers for C*, written by Richard Stallman and others[2], in C. It is based on gcc version 3.2 and generates code for an AMD Opteron processor. The benchmark runs as a compiler with many of its optimisation flags enabled. The used input is 'scilab.i'.

429.mcf.r belongs to the general category of *combinatorial optimisation*, written by Andreas Löbel, in C. It is derived from MCF and uses a network simplex algorithm (which is also used in commercial products) to schedule public transport. The used input is the default reference 'inp.in'.

[2] http://gcc.gnu.org/onlinedocs/gcc/Contributors.html.

456.hmmer.r belongs to the general category of *search in a gene sequence database*, written by Sean Eddy, in C. It is used for protein sequence analysis using profile hidden Markov models (profile HMMs), which are statistical models of multiple sequence alignments, that are used in computational biology to search for patterns in DNA sequences. The used input is '–fixed 0 –mean 500 –num 500000 –sd 350 –seed 0 retro.hmm'.

462.libquantum.r belongs to the general category of *physics and quantum computing*, written by Björn Butscher and Hendrik Weimer, in C. It is a library for the simulation of a quantum computer, which is based on the principles of quantum mechanics and can solve certain computationally hard tasks in polynomial time. It runs Shor's polynomial-time factorisation algorithm. The used input is the default reference '1397 8'.

464.h264ref.f belongs to the general category of *video compression*, written by Karsten Sühring et al[3] , in C. It is a reference implementation of H.264/AVC (Advanced Video Coding), the latest state-of-the-art video compression standard, that encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2. The used input is '-d foreman_ref_encoder_main.cfg'.

473.astar.r belongs to the general category of *computer games, artificial intelligence and path finding*, written by Lev Dymchenko, in C++. It is derived from a portable 2D pathfinding library that is used in game's AI and implements among others the well-known A* algorithm. The used input is 'rivers.cfg'.

483.xalancbmk.r belongs to the general category of *xml processing*, written by IBM's Michael Wong et al., in C++. It is a modified version of Xalan-C++, an XSLT processor written in a portable subset of C++, which transforms XML documents to other document types. The used input is the default reference '-v t5.xml xalanc.xsl'.

3.2.2 SPEC CFP 2006

416.gamess.c belongs to the general category of *quantum chemical computations*, written by Gordon Research Group of Iowa State University, in Fortran. It implements a wide range of quantum chemical computations. For the SPEC workload, self-consistent field calculations are performed using the Restricted Hartree Fock method, Restricted openshell Hartree-Fock, and Multi-Configuration Self-Consistent Field. The used input is '< cytosine.2.config'.

433.milc.s belongs to the general category of *physics and quantum chromodynamics (QCD)*, written by Steven Gottlieb, in C. It is a gauge field generating program for lattice gauge theory programs with dynamical quarks. The used input is '< su3imp.in'.

434.zeusmp.z belongs to the general category of *physics and magnetohydrodynamics*, written by Michael Norman, in Fortran. It is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for the simulation of astrophysical phenomena. It solves problems in three spatial dimensions with a wide variety of boundary conditions. This program doesn't need any input.

450.soplex.p & **450.soplex.r** belong to the general category of *linear programming and optimisation*, written by R. Wunderling et al., in C++. It is based on SoPlex version 1.2.1 and solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models. The used inputs are '-s1 -e -m45000 pds-50.mps' and '-m3500 ref.mps' respectively.

[3] http://iphome.hhi.de/suehring/tml/doc/lenc/html/contributors_8h.html

459.GemsFDTD.r belongs to the general category of *computational electromagnetics*, written by Ulf Andersson et al., in Fortran. It solves the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method. This program doesn't need any input.

470.lbm.l belongs to the general category of *fluid dynamics*, written by Thomas Pohl, in C. It implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D. The used input is '3000 reference.dat 0 0 100_100_130_ldc.of'.

482.sphinx3.a belongs to the general category of *speech recognition*, written by Sphinx Speech Group, in C. It is a widely-known speech recognition system from Carnegie Mellon University. The used input is 'ctlfile . args.an4'.

3.3 Classification of Benchmarks

CMP$im simulator was executed for all traces generated for each one of the 17 benchmarks described above, for a range of LLC configurations ($size_associativity \in \{256_1, 512_2, 1024_4, 2048_8, 4096_16, 8192_32\}$), for 1 billion instructions, under *LRU* replacement policy, in order to extract single-core IPC (Instructions per Cycles) and MPKI (Misses Per Kilo Instructions) metrics. The different LLC configurations were created by having a fixed number of sets (namely 4096) and adjusting the associativity accordingly (from 1 to 32, with $\times 2$ step) to achieve desired cache size ($size = sets \times associativity$). Each trace was initially generated using PIN tool by running each benchmark with its input, for 1 billion instructions, after fast-forwarding (warming-up cache) 40 billion instructions.

Based on the classification of applications described in section 2.3, the selected benchmarks can be divided in the following three categories: *cache-friendly*, *cache-fitting* and *cache-thrashing* & *streaming*. Since benchmarks may reside in more than one categories, the classification is not absolute, but helps to intuitively understand the behaviour of each benchmark.

3.3.1 Cache-friendly

Cache-friendly benchmarks, which are shown in Figure 3.1, demonstrate good use of cache resources, reducing the misses and increasing performance, if they are given more ways. Sphinx3.a shows low utility of cache resources, so it could also be characterised as *cache-thrashing*, for smaller cache configurations (size_associativity). Some benchmarks, like bzip2.c, gcc.s and xalancbmk.r could also be characterised as *cache-fitting* for bigger cache configurations.

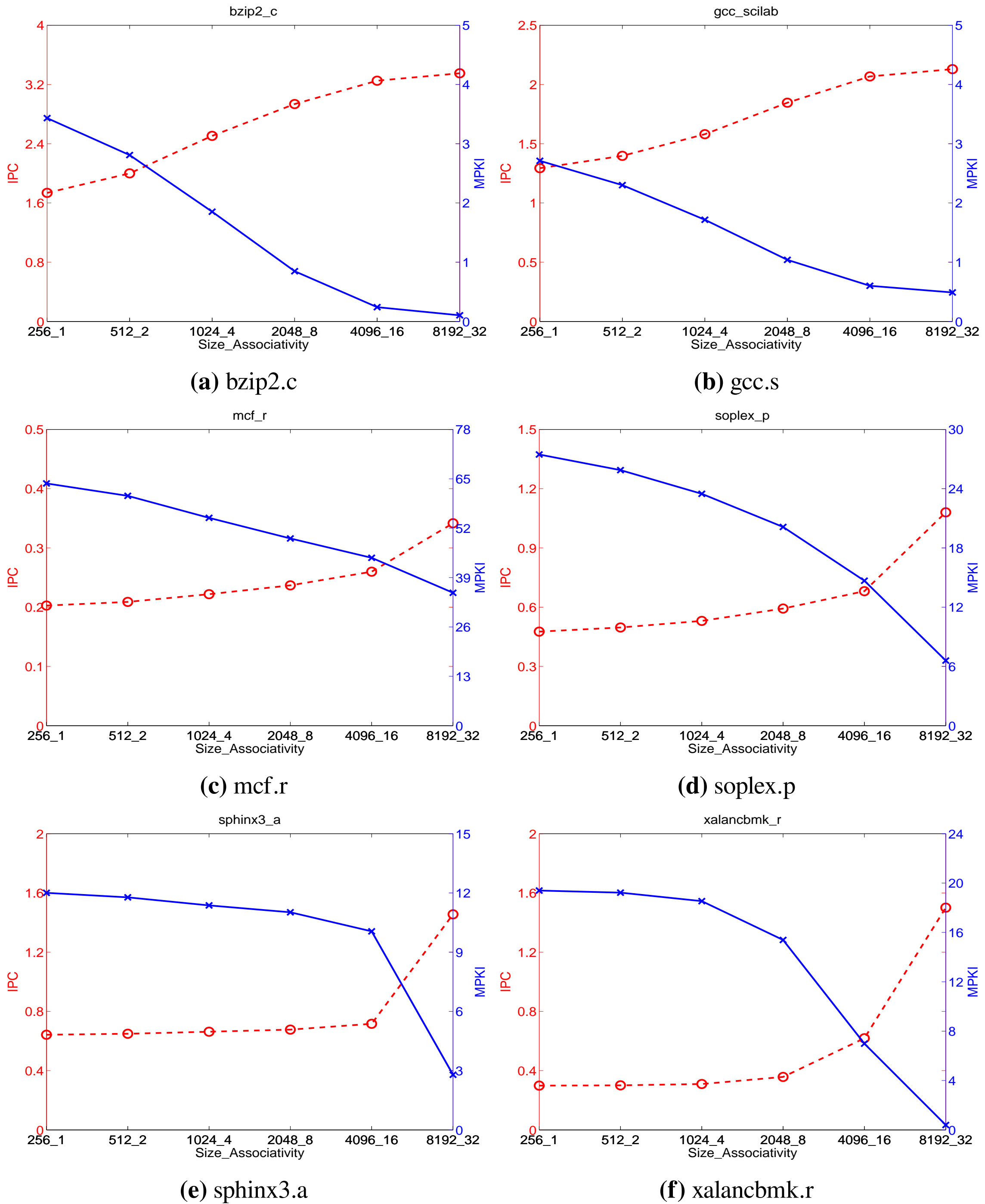
3.3.2 Cache-fitting

Cache-fitting benchmarks, which are shown in Figure 3.2, have a relatively small working set, that fits into the existing cache, therefore these benchmarks exhibit few misses with little allocated resources.

3.3.3 Cache-thrashing & Cache-streaming

Cache-thrashing benchmarks, which are shown in Figure 3.3, have several misses and exhibit little to no performance gain, if they allocate more resources. *Cache-streaming* benchmarks, which are shown in Figure 3.4, have several misses and show no performance gain,

regardless of how many resources they allocate. Usually the distinction between streaming and thrashing behaviour is difficult, therefore for the scope of this diploma thesis they are all characterised as thrashing.



(a) bzip2.c

(b) gcc.s

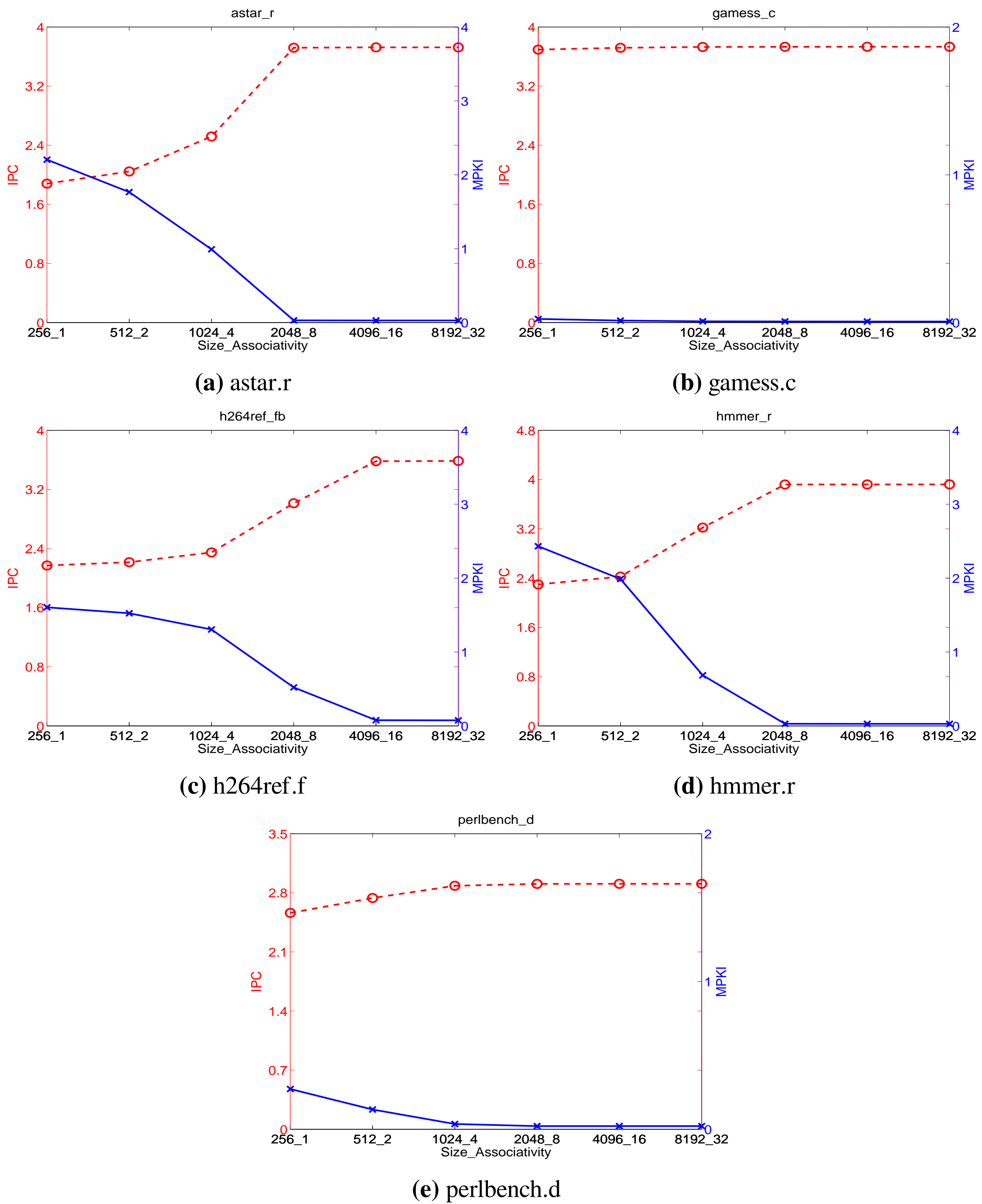
(c) mcf.r

(d) soplex.p

(e) sphinx3.a

(f) xalancbmk.r

Figure 3.1: Cache-friendly benchmarks.



(a) astar.r

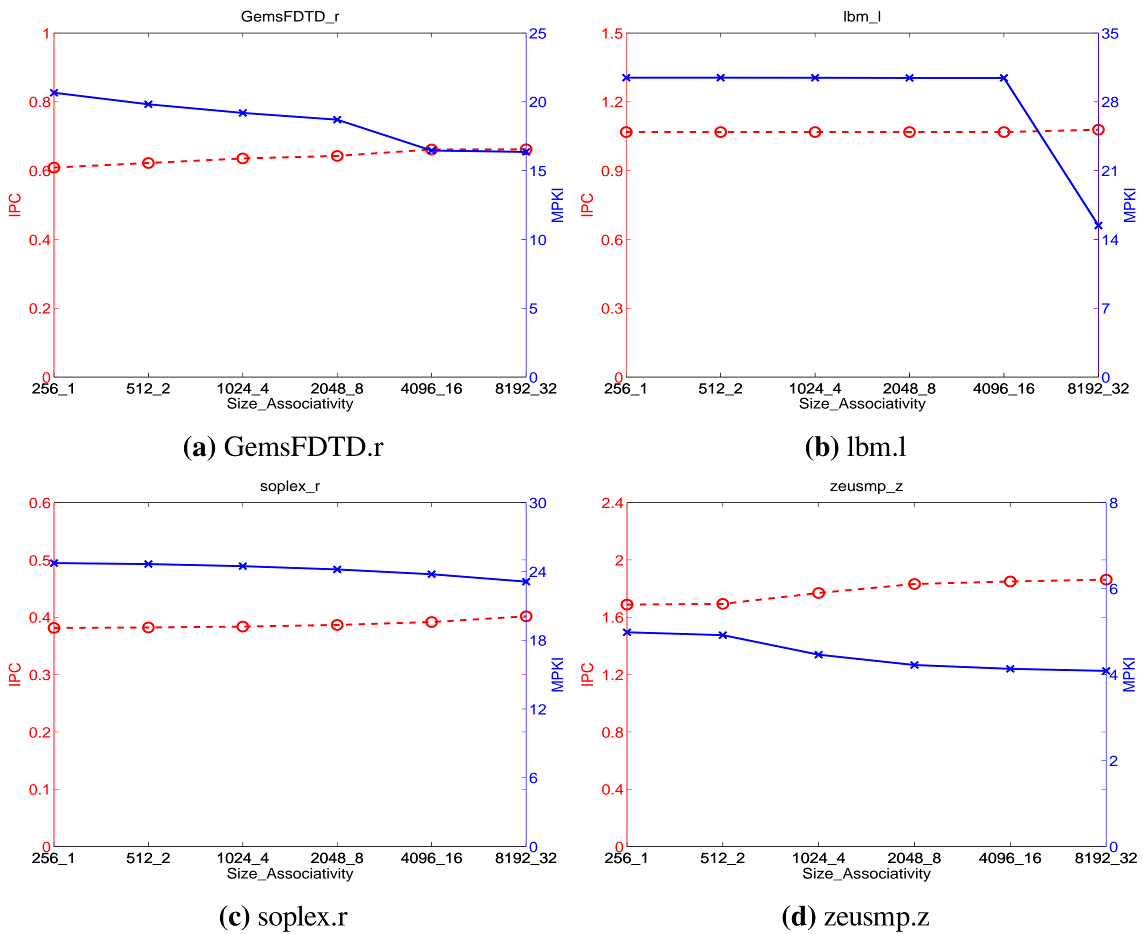
(b) gamess.c

(c) h264ref.f

(d) hmmer.r

(e) perlbench.d

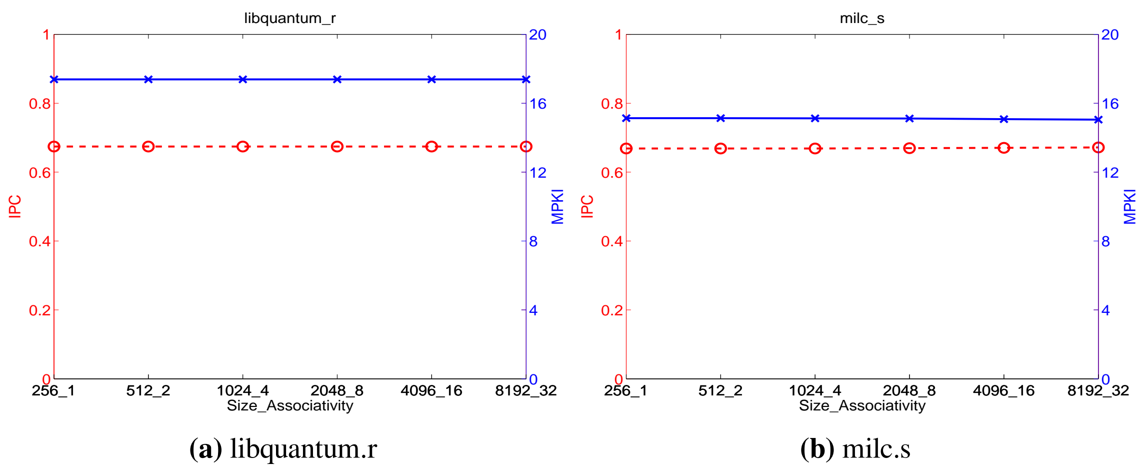
Figure 3.2: Cache-fitting benchmarks.



(a) GemsFDTD.r (b) lbm.l

(c) soplex.r (d) zeusmp.z

Figure 3.3: Cache-thrashing benchmarks.



(a) libquantum.r (b) milc.s

Figure 3.4: Cache-streaming benchmarks.

3.4 Multi-core Simulation Configuration

For the scope of this diploma thesis, simulations were run for a four-core system, with 4MBytes 16 way set-associative LLC, with 64 bytes cache line size, running on each one of LLC management schemes described in section 2.4. The inputs used for the simulations are 15 workload mixes, which combine the 17 benchmarks per 4. For each workload mix, each benchmark is executed for 1 billion instructions and repeats execution until all benchmarks have reached that number. The workload mixes were taken in accordance to [3] and are listed at Table 3.1, where abbreviations FR, FI and TH denote cache friendly, fitting and thrashing or streaming profiles respectively.

Table 3.1: Workload mixes.

Mix Name	Benchmarks in Mix	Benchmark Profiles
MIX_00	xalacbmkr.r | mcf.r | milc.s | gcc.s	FR | FR | TH | FR
MIX_01	sphinx3.a | mcf.r | hmmer.r | soplex.r	FR | FR | FI | TH
MIX_02	bzip2.c | zeusmp.z | lbm.l | hmmer.r	FR | FR | TH | FI
MIX_03	soplex.r | xalancbmk.r | h264ref.f | astar.r	TH | FR | FI | FI
MIX_04	libquantum.r | milc.s | soplex.p | bzip2.c	TH | TH | FR | FR
MIX_05	soplex.r | sphinx3.a | soplex.p | bzip2.c	TH | FR | FR | FR
MIX_06	zeusmp.z | h264ref.f | gcc.s | soplex.p	FR | FI | FR | FR
MIX_07	astar.r | GemsFDTD.r | hmmer.r | gcc.s	FI | TH | FI | FR
MIX_08	zeusmp.z | xalancbmk.r | sphinx3.a | soplex.r	FR | FR | FR | TH
MIX_09	mcf.r | xalancbmk.r | astar.r | perlbench.d	FR | FR | FI | FI
MIX_10	libquantum.r | soplex.p | perlbench.d | hmmer.r	TH | FR | FI | FI
MIX_11	soplex.p | milc.s | libquantum.r | zeusmp.z	FR | TH | TH | FR
MIX_12	lbm.l | xalancbmk.r | soplex.r | milc.s	TH | FR | TH | TH
MIX_13	zeusmp.z | libquantum.r | soplex.r | milc.s	FR | TH | TH | TH
MIX_14	gamess.c | perlbench.d | h264ref.f | hmmer.r	FI | FI | FI | FI

3.5 Synopsis

In this chapter CMP$im has been presented, which is used for the simulation of LLC replacement schemes presented in section 2.4. In order to understand the behaviour of a real 4-core system with common workloads, benchmarks from SPEC CPU 2006 suite were used, by creating 15 different combinations/mixtures of 4 out of 17 selected benchmarks. The simulation results and further analysis follow in the next chapter.

"Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest."

Isaac Asimov

"I'm sorry, Dave, I'm afraid I can't do that."

HAL, a spaceship computer, 2001: A Space Odyssey

4

Results, Optimisation & Analysis

The necessary infrastructure has already been demonstrated, upon which all the following results and analysis will be based. In this chapter several LLC replacement policies are compared and their advantages are being examined.

4.1 Metrics

In order to measure performance of multiple concurrently executing applications several metrics have been proposed, that account for fairness, performance or both. In this diploma thesis three commonly used metrics are preferred: throughput, weighted speedup and harmonic mean fairness. *Throughput (TH)* is a performance metric, showing how many total instructions are executed in a given amount of cycles (IPC). *Weighted Speedup (WS)* [15] is a fairness metric, showing how close to the performance when executing alone gets each core in the presence of others, by evaluating the sum of the normalised IPCs. *Harmonic Mean Fairness (HMF)* [16], which is harmonic mean of normalised IPCs, balances both fairness and performance. These metrics are given by:

$$Throughput = \sum(IPC_i) \tag{4.1}$$

$$WeightedSpeedup = \sum(IPC_i/SingleIPC_i) \tag{4.2}$$

$$HarmonicMeanFairness = N/\sum(SingleIPC_i/IPC_i) \tag{4.3}$$

where IPC_i is the IPC of the ith application when it concurrently executes with other applications and $SingleIPC_i$ is IPC of the same application in isolation, namely having all the cache for itself.

4.2 Comparison of LLC Partitioning Schemes

Initially a comparison of all LLC partitioning schemes (see subsection 2.4.1), namely *UCP* and *ABFCP*, is presented. The simulations were run for the 15 workload mixes, for a

16-way set associative 4MB Last Level Cache. Each benchmark repeats execution if needed, until all have executed 1 billion instructions. Statistics for each benchmark are collected only for the first 1 billion instructions, also known as region of interest. For the described configuration, Figures 4.1, 4.2, 4.3 show throughput, weighted speedup and harmonic mean fairness metrics normalised to *LRU* respectively. Apart from results for the 15 workload mixes, the geometric mean ($gmean = \sqrt[n]{x_1 \times x_2 \times \ldots \times x_n}$) is also depicted.

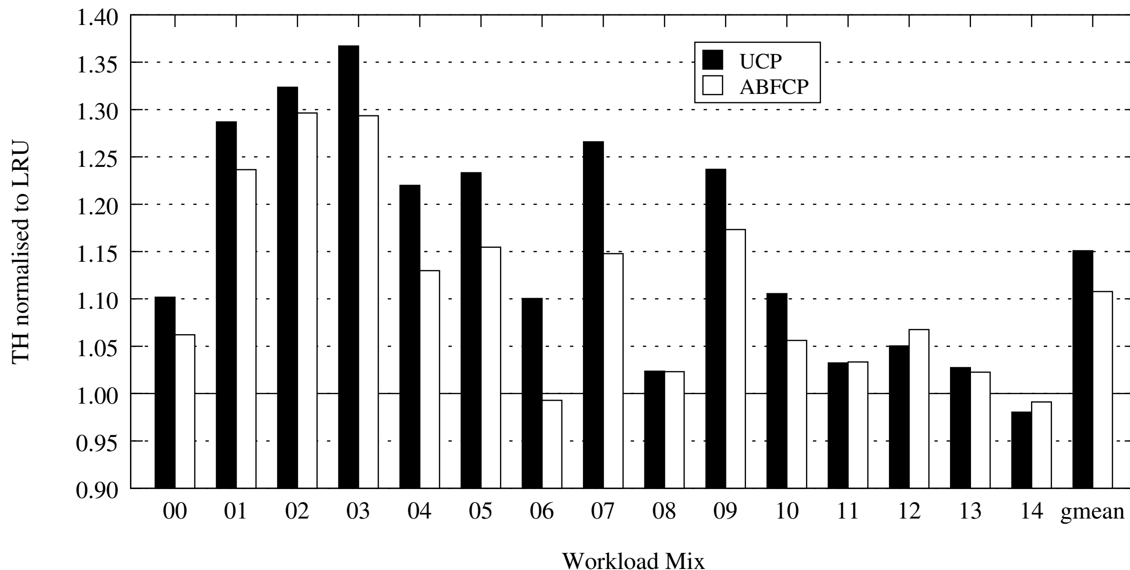


Figure 4.1: Throughput of LLC partitioning policies.

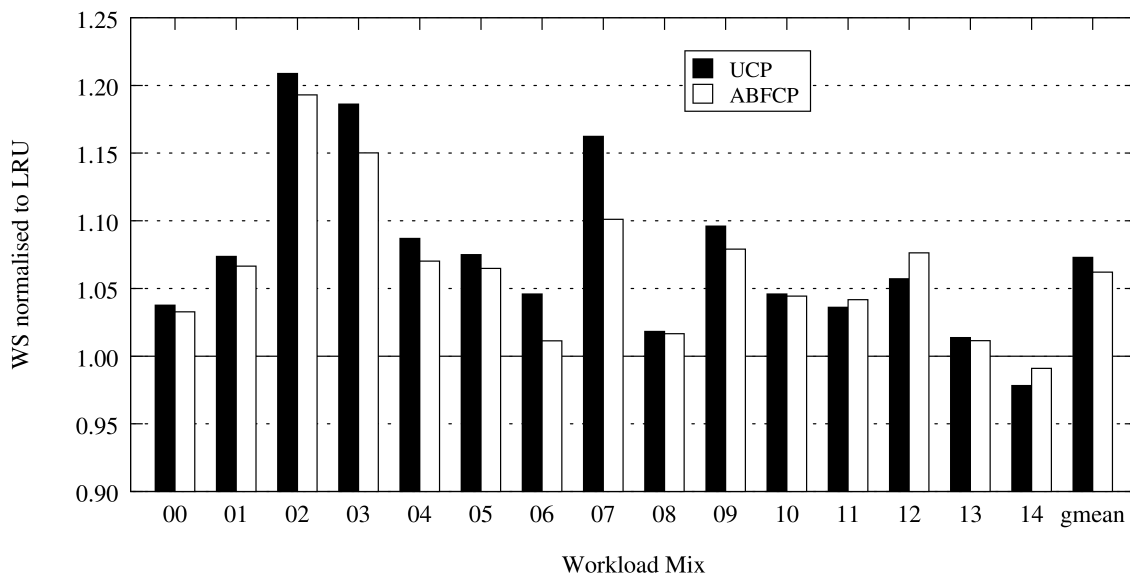


Figure 4.2: Weighted speedup of LLC partitioning policies.

Considering throughput metric, it is obvious that *UCP* has an overall better performance with 15% speedup over *LRU*, while *ABFCP* follows with a speedup of around 10.8%.

Moreover, as weighted speedup denotes, *UCP* is fairer with around 7.3% speedup over *LRU*, with *ABFCP* following with a speedup of 6.1%.

Finally, regarding harmonic mean fairness, *UCP* has also the most balanced behaviour, being both fair and well-performing, with a speedup of 8.8% over *LRU*. *ABFCP* demonstrates an approximate gain of 7.7%.

Weighted speedup metric shows a similar estimation as harmonic mean fairness metric, so hereinafter it will be omitted for brevity purposes.

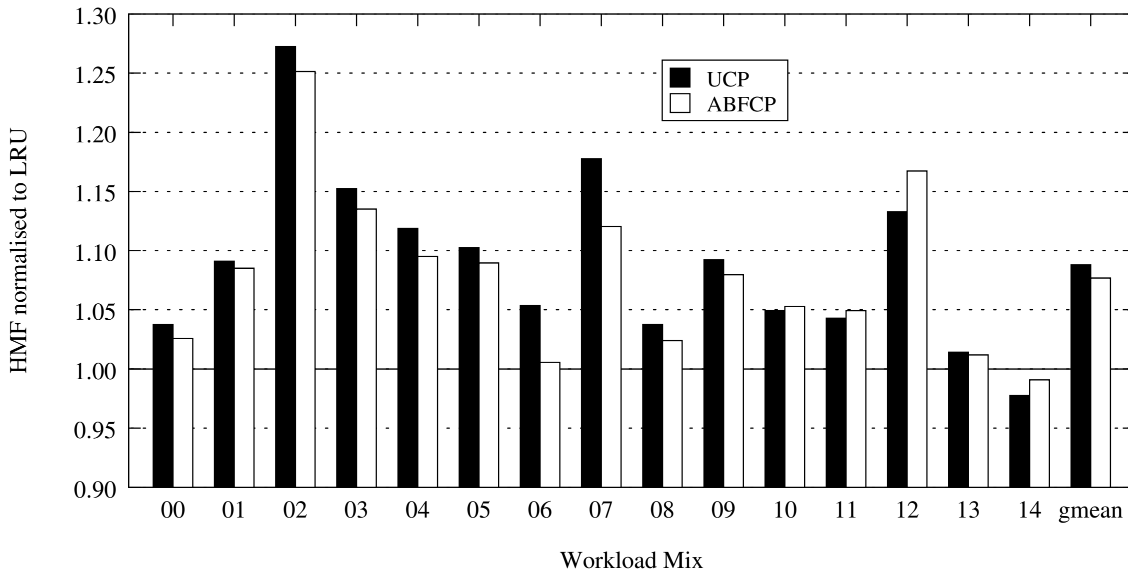


Figure 4.3: Harmonic mean fairness of LLC partitioning policies.

4.3 Optimisation of *ABFCP*

ABFCP is a scheme that possesses the flexibility of per-set but not per-way (+-1 ways) partitioning, whereas *UCP* can take per-way flexible (+-N) but uniform partitioning decisions. **It is within the scope of this diploma thesis to explore the potential optimisations that may be introduced to *ABFCP*.** For this purpose *ABFCP* is compared primarily with *UCP*, since both belong to the partitioning policies family. Table 4.1 depicts the differences between *ABFCP* and *UCP*.

Table 4.1: Differences between *ABFCP* and *UCP*.

Partitioning Properties	*ABFCP*	*UCP*
Distribution of Ways	at most +/- 1 way	all ways
Repartition Phase	all counters set to 0	all counters divided by 2
	BF reset	UMON unaltered
Set Flexibility	per-set partitioning	uniform partitioning

4.3.1 Distribution of Ways

So far *ABFCP's* partitioning decisions have differed at most 1 way from the partitioning decision of the previous repartition phase. This happens due to the nature of Bloom Filters, which do not contain per-way utility but merely probabilistic set membership. *UCP* however, redistributes all ways among competing applications on each repartition and is independent from the decision of the previous repartition phase. Thus arises the idea of making *ABFCP* more flexible on a per-way basis as well.

To achieve that, an approximate reconstruction of *UCP's* UMON has been attempted, called *BFMON (BF MONitor)*. BFMON is used just like UMON, namely *ABFCP's* partitioning algorithm becomes identical to that of *UCP*. The difference lies in structure of BFMON, which does not need an ATD but merely reconstructs the content of counters per recency position, using *LRU* stack property and 3 counters: BF counter, LRU counter and TOT counter. BF counts total Bloom Filter hits so is the same as C_{BFHIT}, LRU counts hits on LRU recency position so is the same as C_{LRUHIT} and TOT counts total hits.

The basic idea is that when a core i occupies W_i ways, then the TOT counter represents the sum of recency positions of UMON from MRU to W_i, LRU counter represents the hits due to W_ith way and BF counter represents the sum of recency positions of UMON from recency positions right after W_i to LRU. The BFMON attempts a linearly distributed reconstruction, which is shown in Figure 4.4 and is realised as follows:

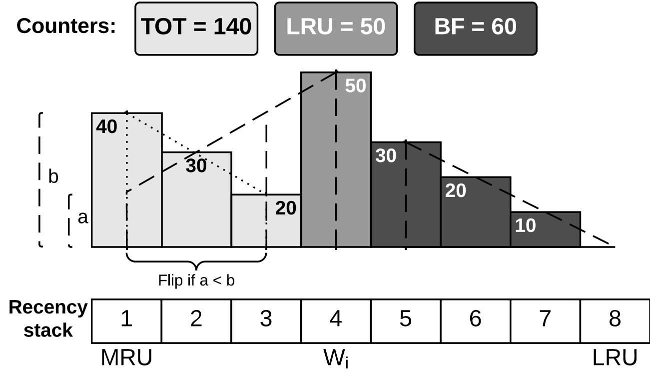


Figure 4.4: BFMON for an 8-way cache.

- $recency \in [MRU, W_i)$: values of recency positions lie on the slanted edge of a trapezoid with 2 orthogonal angles, with area equal to the TOT counter, where one side is known (LRU) and the other can be easily computed ($\frac{2\times TOT}{W_i} - LRU$). The sub-trapezoid with one side at MRU and the other at recency right before W_i is flipped if needed, so as slanted edge is decreasing towards LRU.
- $recency = W_i$: value equals to LRU counter.
- $recency \in (W_i, LRU]$: values of recency positions lie on the hypotenuse of the orthogonal triangle with area equal to BF counter, where the 'base' edge is known ($LRU - W_i$) and the height edge can be easily computed ($\frac{2\times BF}{LRU - W_i}$). The triangle's height edge is facing towards MRU.

In order to evaluate the results of ABFCP partitioning all ways rather than only +/- 1, a comparison is imperative. Figures 4.5 and 4.6 show throughput and harmonic mean fairness respectively, between *ABFCP* original (*ABFCP+-1*) and *ABFCP* new (*ABFCP+-N*).

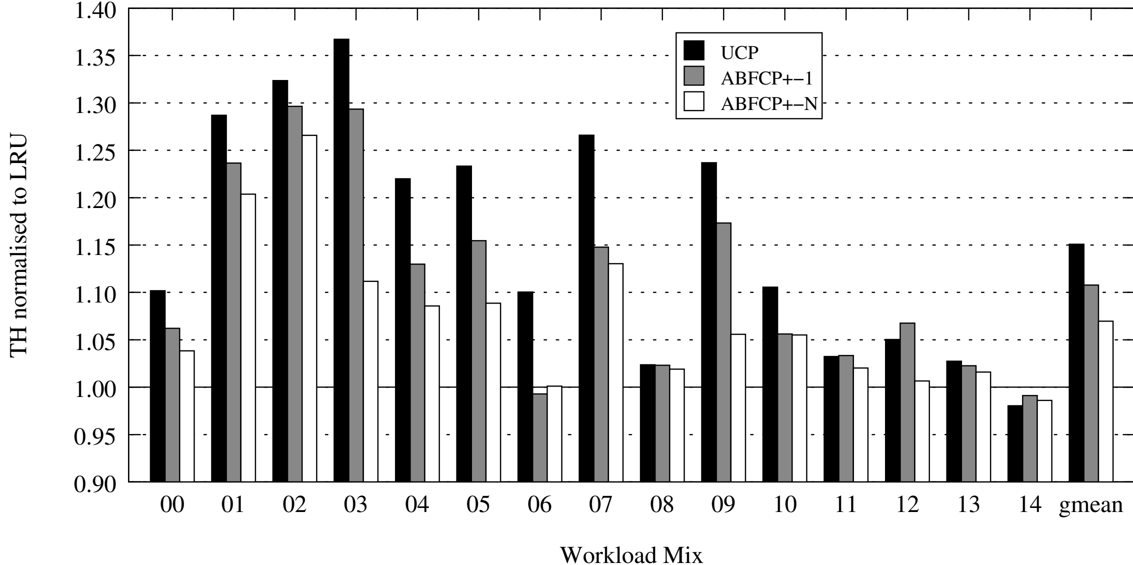


Figure 4.5: Throughput of *ABFCP* original and per-way flexible.

Considering throughput metric, new per-way flexible *ABFCP* performs worse than original, with 7% speedup over *LRU*. The same stands for harmonic mean fairness, where new *ABFCP* comes last with 5% speedup. **It seems that per-way flexible *ABFCP* doesn't affect performance significantly, if no other changes are made**.

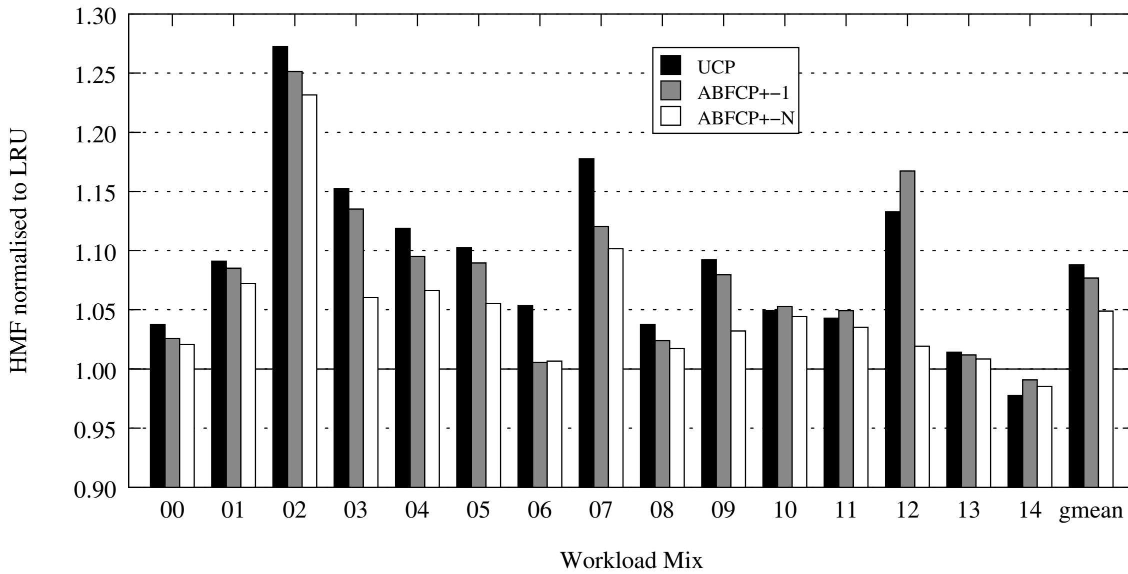


Figure 4.6: Harmonic mean fairness of *ABFCP* original and per-way flexible.

4.3.2 Repartition Phase

On each repartition phase the new partition is decided and the information holding units, like counters and monitoring structures, have to be updated accordingly. *UCP* leaves UMONs unchanged and halves its counters, while *ABFCP* resets both BFs and counters. It would be wise to check the influence of these parameters on *ABFCP* as well, since halving counters or leaving BFMONs unchanged, results in retaining of present and past information and may improve prediction. Thus Figures 4.7 and 4.8 depict throughput and harmonic mean fairness respectively, for *ABFCP* with +/-1 or +/-N distribution of ways and reset or halving of counters.

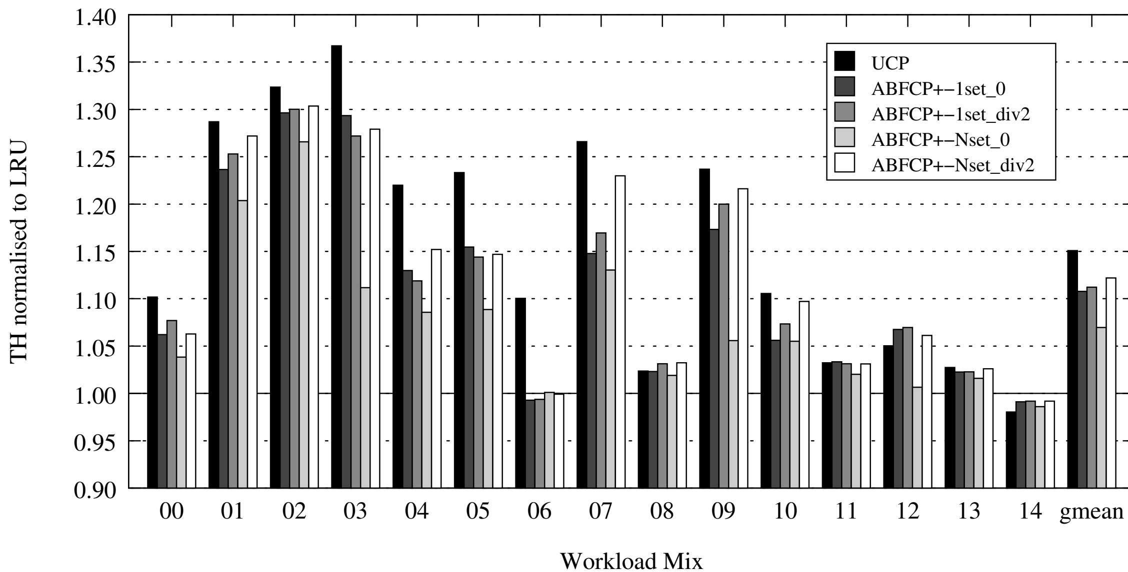


Figure 4.7: Throughput of *ABFCP* versions for both resetting and halving of counters.

Regarding both throughput and harmonic mean fairness metrics, *ABFCP* versions with halved counters perform better than those which reset counters. The biggest change is presented in the *ABFCP+-N* per-set partitioning, which had a throughput of 7% when counters were reset and ‘jumped’ to 12.2% when counters were halved. **It seems that counter halving improves performance of *ABFCP*, independently of the version**. The reset interval of BFs was also tested, with measurements for each 1 (original), 2 and 4 repartitions.

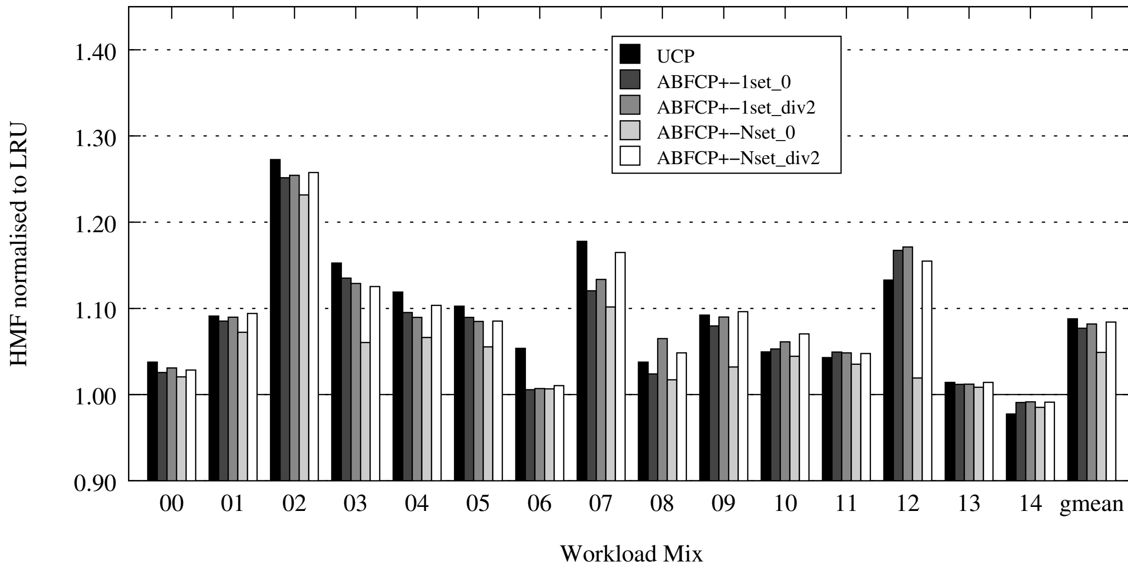


Figure 4.8: Harmonic mean fairness of *ABFCP* versions versions for both resetting and halving of counters.

Results showed that BF reset interval is either of no gain or degrades performance and for brevity reasons are neglected.

4.3.3 Set Flexibility

The main difference between *ABFCP* and *UCP* is that the former attempts to decide a partition per each set, thus sacrificing the per-way flexibility, while the latter attempts to be fully per-way flexible, enforcing uniform instead of per-set partitioning. Both policies try to find the optimal tradeoff, avoiding any prohibitive hardware overhead. *ABFCP* may not always perform like expected, namely per-set decisions might be an 'overkill'. For this reason, an alternative *ABFCP* is introduced, which enforces partition uniformly and the decision is made by taking into account the sum of counters from all sets. Figures 4.9 and 4.10 show throughput and harmonic mean fairness respectively, for *ABFCP+-1* and *ABFCP+-N* both for per-set and uniform partitioning as well as halving or resetting of counters.

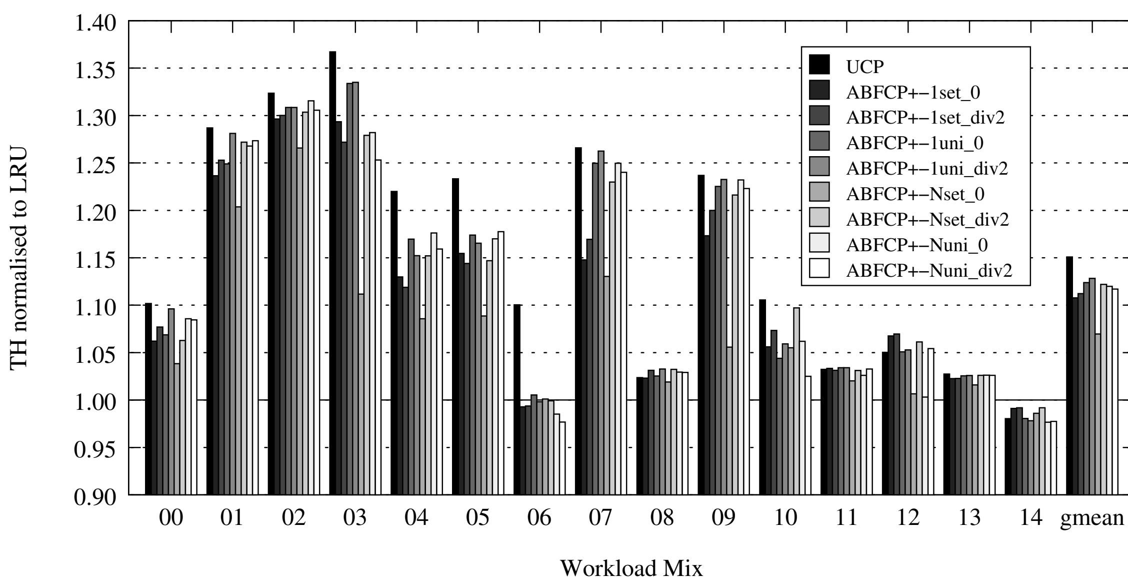


Figure 4.9: Throughput of *ABFCP* versions for both per-set and uniform partitioning.

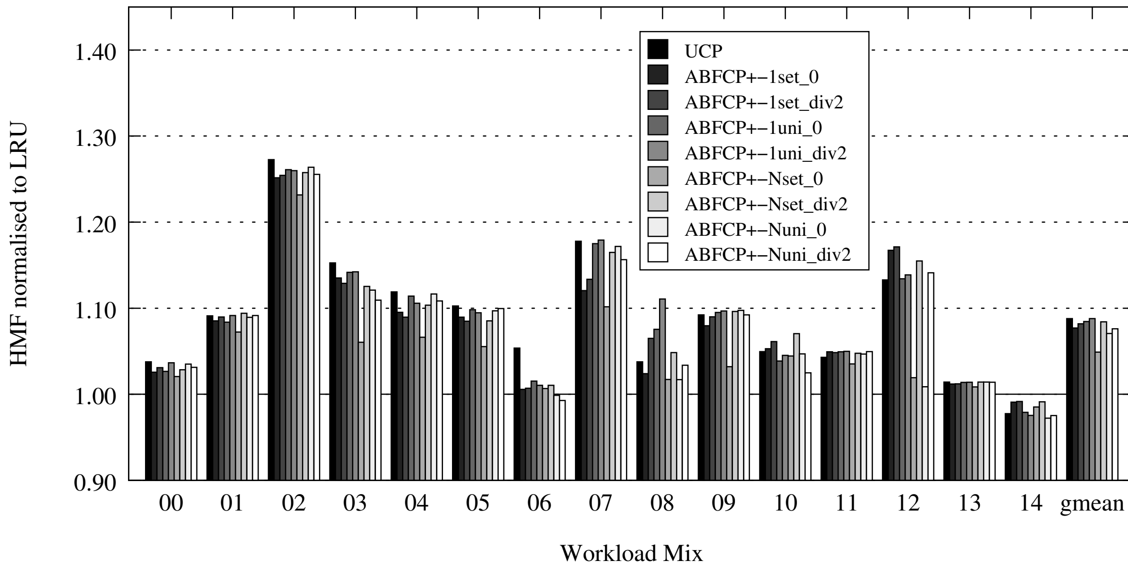


Figure 4.10: Harmonic mean fairness of *ABFCP* versions for both per-set and uniform partitioning.

Regarding either throughput or harmonic mean fairness metric, uniform *ABFCP* perform slightly better than the equivalent per-set *ABFCP* versions for +/-1 partitioning. For +/-N partitioning with resetting counters uniform *ABFCP* performs better than per-set *ABFCP*, whereas it performs slightly worse for the +/-N partitioning with halving counters. **It seems that uniform partitioning either improves or doesn't significantly affect performance compared to the per-set versions of *ABFCP*, showing that the per-set flexibility of *ABFCP*, at least for the measured workloads, may be an 'overkill', thus opening the way for further reduction of hardware overhead**.

4.4 New Estimator for *ABFCP*

Original *ABFCP* made use of two counters C_{BFHIT} and C_{LRUHIT} to count the hits in Bloom Filter and LRU recency position respectively. The partitioning algorithm (presented in Appendix A.2) created two sorted lists, one for each counter, with the first showing the gain of taking 1 more way and the second showing the cost of losing 1 way. *ABFCP* may not handle some situations well, when for instance the BF falsely estimates a lot of gain, due to false-positives or thrashing applications, that would require a lot more ways than available associativity. Applications could potentially deceive *ABFCP* into 'believing' that their gain is more than others' loss, thus getting assigned a lot more resources, that they do not utilise.

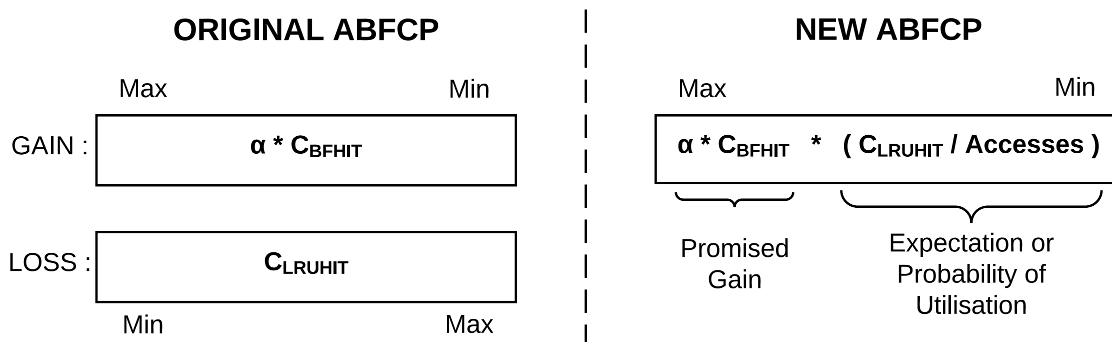


Figure 4.11: Original vs New *ABFCP*.

To address this shortcoming a new *ABFCP* estimator is proposed, as shown in Figure 4.11, which correlates the information contained in the two counters. If $\alpha \times C_{BFHIT}$ represents an estimation of per-way gain, C_{LRUHIT} represents utilisation of 1 way (LRU) and $C_{LRUHIT}/Accesses$ denotes the probability that an access will result in a hit, then the new estimator is:

$$estimator = \alpha \times C_{BFHIT} \times C_{LRUHIT}/Accesses \qquad (4.4)$$

New *ABFCP* needs only one sorted list, since it compares values of same nature (see Appendix A.2). The new estimator is predicted to filter applications that tend to be deceptive, by showing a lot of BF hits but very low LRU hit rate. There have been created all new *ABFCP* scheme's versions, in continuation of the previous analysis. Thus new *ABFCP* can partition +/-1 or +/-N ways, per-set or uniformly, while resetting or halving counters. For the +/-N ways partitioning, the value of TOT ($\alpha \times C_{BFHIT}$) counter, which reconstructed a portion of BFMON, is replaced with the new estimator in equation 4.4. Figure 4.12 shows for brevity only the geometric mean of the 15 workloads for the throughput and harmonic mean fairness of both original and new *ABFCP* schemes.

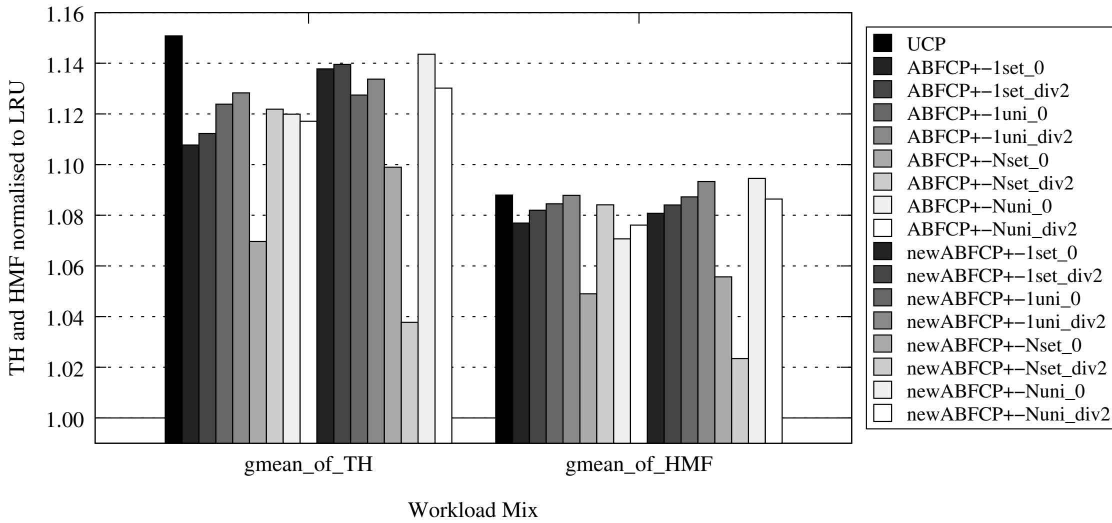
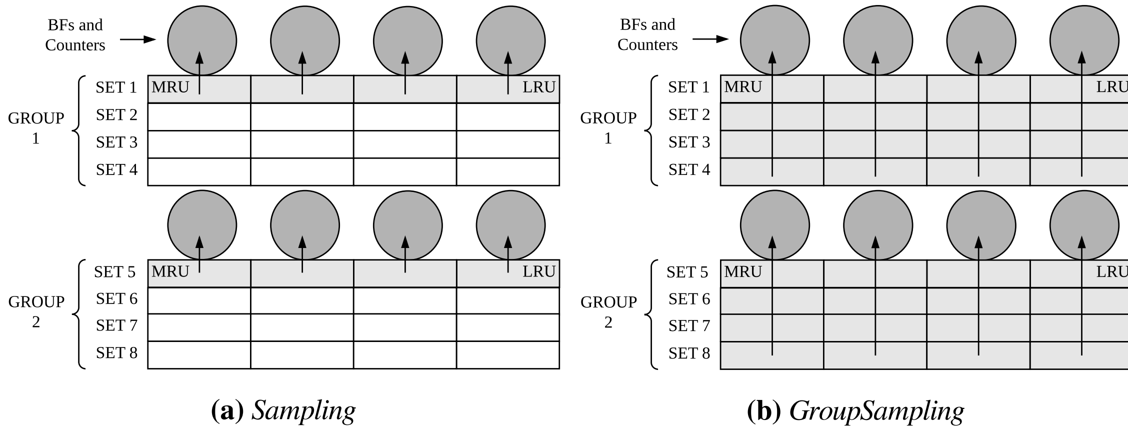


Figure 4.12: Throughput and Harmonic mean fairness of original and new *ABFCP* versions.

New *ABFCP* scheme performs better for all the corresponding versions compared to the original, except for the *ABFCP* with +/-N, per-set and halved counters partitioning (newABFCP+-Nset_div2). This may occur because of the nature of the new estimator, which multiplies the counters, resulting in a cummulative erratic prediction due to halving, if counters contain deceptive information, which then may be distributed among more ways of BFMONs for +/-N version instead of just 1 for +/-1 version. However, some versions of new *ABFCP* even surpass *UCP* for the harmonic mean fairness metric. These are new *ABFCP* with +/-N, uniform and reset counters partitioning (newABFCP+-Nuni_0) and new *ABFCP* with +/-1, uniform and halved counters partitioning (newABFCP+-1uni_div2), showing a speedup of 9.5% and 9.3% respectively over *LRU*, while *UCP* had a 8.8% speedup. **It seems that new estimator for *ABFCP* shows an overall improvement of *ABFCP* behaviour and challenges best-performing LLC policies.**

4.5 Effect of Sampling for *ABFCP*

Since per-set partitioning may be redundant, the effect of *per-set sampling* is examined, to realise how much reduction in hardware overhead may be achieved, without significant loss of performance. Set sampling is a technique like *UCP*'s DSS presented in subsection 2.4.1, where few sets decide for the rest. Two alternatives are proposed: *Sampling (S)* and *GroupSampling (GS)*, which can be seen in Figure 4.13.



(a) *Sampling* **(b)** *GroupSampling*

Figure 4.13: *ABFCP* sampling versions.

Sampling simply stores information, namely BFs and counters, only for the sampled sets, which in turn decide for all sets of their group. For k samples out of n sets, there are k groups containing n/k sets each. *GroupSampling* differs from *Sampling* in that BFs and counters of each sampled set are updated by all sets of its group, instead of the sampled set only.

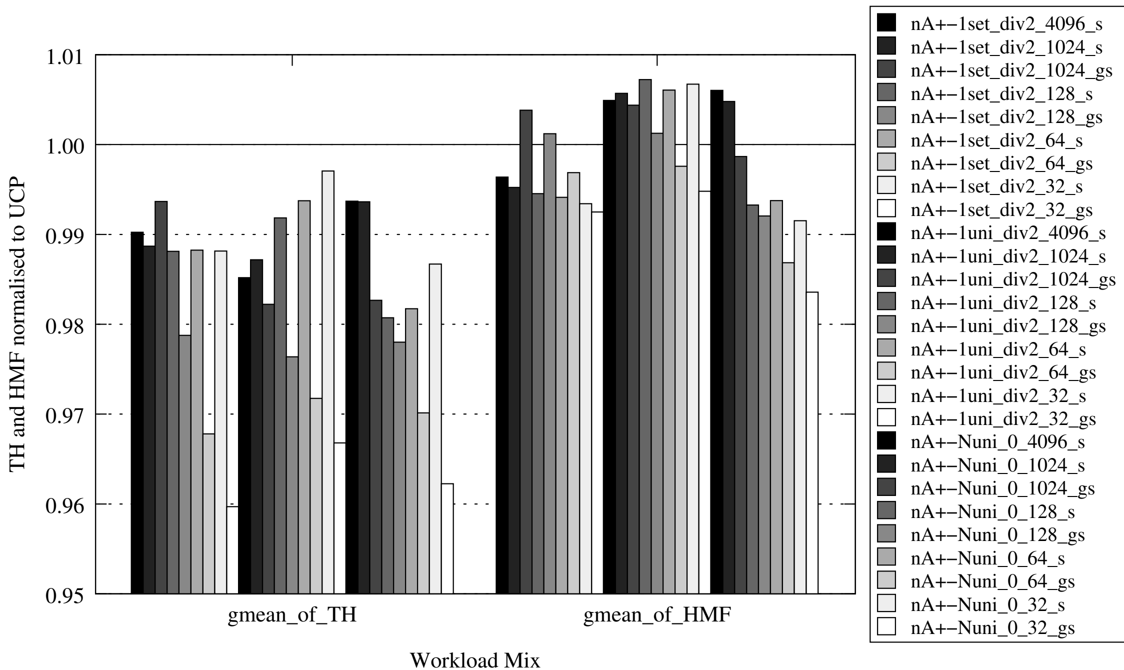


Figure 4.14: Throughput and Harmonic mean fairness of best 3 *ABFCP* versions for *Sampling* and *GroupSampling*.

Figure 4.14 shows the geometric mean of throughput and harmonic mean fairness for 3 among the best performing *ABFCP* schemes presented previously with both

Sampling and *GroupSampling* versions. The 3 selected *ABFCP* schemes are newABFCP+-1set_div2, newABFCP+-1uni_div2 and newABFCP+-Nuni_0. Moreover $samples \in \{32, 64, 128, 1024\}$. For each one of the 3 examined *ABFCP* versions, the initial unsampled version is also showed, which is equivalent to 4096 samples with *Sampling*. The results are normalised to *UCP* instead of *LRU*. Since all examined versions use new *ABFCP* the "nA" abbreviation is used.

Observing throughput metric, *Sampling* is in almost all cases a more reliable choice, while *GroupSampling* degrades performance compared to *Sampling* and only for nA+-1set_div2 version with 1024 samples performs even better than unsampled version (4096_s). The best performing version seems to be nA+-1uni_div2 with *Sampling* of 32 sets, with 14.7% compared to *UCP*'s 15% speedup over *LRU*. For this version (nA+-1uni_div2) *Sampling* appears to increase performance despite that the information collected is reduced. Observing harmonic mean fairness metric, *Sampling* seems to be better than *GroupSampling* except for the per-set version of nA+-1set_div2. ***Sampling* was expected to behave equally or slightly worse in some cases. However this is not always true, considering overall performance of nA+-1uni_div2, whose version of 32 samples is the best choice, combining fairness, performance and reduced hardware overhead, while achieving even better speedup than the initial unsampled version**.

4.6 Effect of Reduction of Bloom Filter's Tag

So far, simulations have been run with a close-to-ideal tag, which in this case is 14 bits. Since 14 bits tag for BFs is an impractical design choice due to hardware cost, the effect of reduction of BF's tag on overall performance is examined. Figure 4.15 shows the geometric mean of throughput and harmonic mean fairness for the best-performing *ABFCP* scheme, namely new *ABFCP* with +-1, uniform and halving counters partitioning, with *Sampling* of 32 sets (nA+-1uni_div2_32_s). Moreover $tag \in \{3, 4, 5, 6, 7, 8, 10\}$.

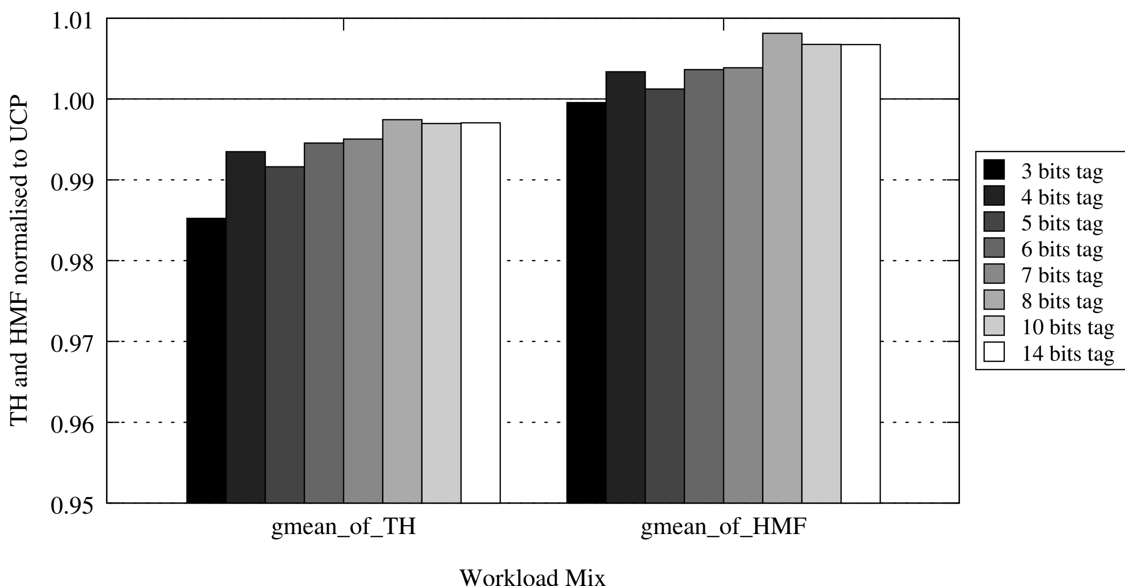


Figure 4.15: Throughput and Harmonic mean fairness of best *ABFCP* version, namely nA+-1uni_div2_32_s, with varying size of BF's tag.

Regarding both throughput and harmonic mean fairness metrics, it seems that performance is reduced as the size of BF's tag is reduced. However performance seems to have a

'peak' for 8 bits tag, instead of at 14 bits. **Thus the best *ABFCP* version which combines performance, fairness and reduced hardware overhead is nA+-1uni_div2_32_s with 8 bits tag**.

4.7 Comparison of all presented LLC schemes

Finally all presented in section 2.4 LLC replacement schemes are compared, namely *UCP*, *ABFCP*, *TADIP*, *TADRRIP*, *PIPP*, alongside the best version of new *ABFCP*, that is nA+-1uni_div2_32_s with 8 bits tag. Figures 4.16 and 4.17 show throughput and harmonic mean fairness respectively, of the aforementioned LLC schemes.

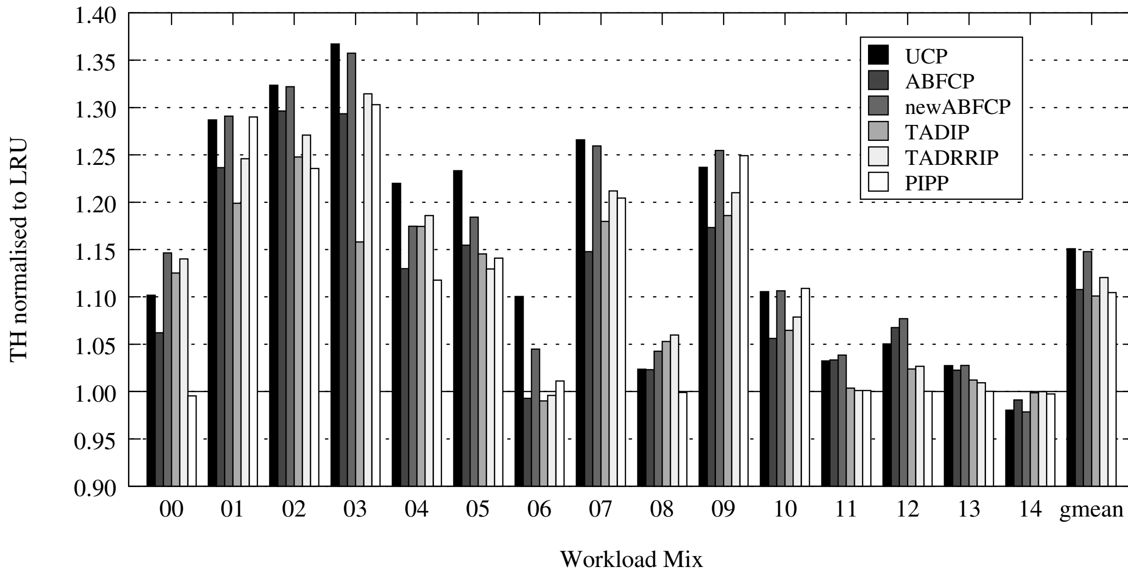


Figure 4.16: Throughput of LLC replacement schemes.

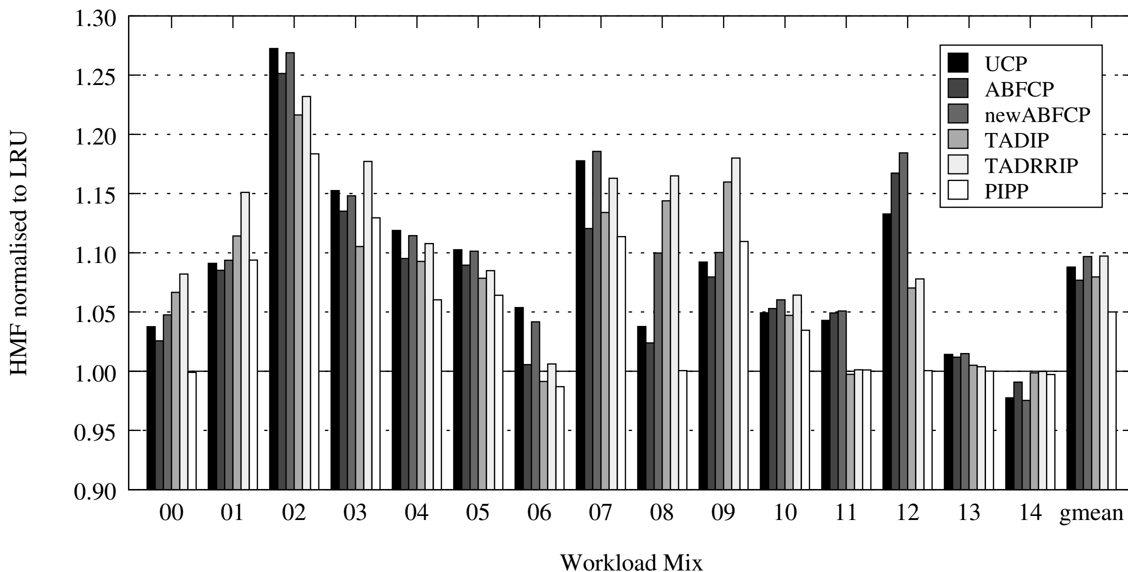


Figure 4.17: Harmonic mean fairness of of LLC replacement schemes.

Considering throughput metric, *UCP* has an overall better *performance* with 15% speedup over *LRU*, followed closely by new *ABFCP* with 14.8% speedup. *TADRRIP* comes afterwards with 12%, while original *ABFCP* and *PIPP* follow with a speedup of around 10.8% and 10.4% respectively. *TADIP* comes last performing around 10% better than *LRU*.

Considering harmonic mean fairness, *TADRRIP* and new *ABFCP* have the most balanced behaviour, being both fair and well-performing, with a speedup of 9.7% over *LRU*. They are closely followed by *UCP* which demonstrates a 8.8% gain, while *TADIP* and original *ABFCP* demonstrate an approximate gain of 7.9% and 7.7% respectively. *PIPP* comes last with 5% speedup.

Table 4.2 shows the required harware overhead of all the compared LLC schemes. *TADRRIP* has the lowest total requirements with 0.4% of cache size, followed by *TADIP* and best version of new *ABFCP* requiring 0.8% and 0.9% extra hardware respectively. *UCP* uses 1.1% more hardware, while original *ABFCP* wants 3% compared to cache. *TADRRIP* is based on *RRIP* which uses less bits to implement its priority stack, therefore requiring at least half of the hardware that *LRU* based policies need. **Even though new *ABFCP* requires 50% more hardware compared to *TADRRIP*, it performs 23% better for the throughput metric and is equally fair for the harmonic mean fairness metric. Moreover new *ABFCP* requires 15% less hardware compared to *UCP*, performing 2% worse while being 10% fairer. Finally new *ABFCP* requires 72% less hardware compared to original *ABFCP*, performing 37% better while being 23% fairer**.

Table 4.2: Required Hardware Overhead of LLC schemes

CACHE SIZE	4MB = 32Mb				
Number of Sets (Sets)	4096				
Associativity (As)	16				
Number of Threads (Th)	4				
EXTRA HARDWARE REQUIRED FOR IMPLEMENTATION OF REPLACEMENT POLICIES					
BASE POLICY (BP)	***LRU* based**				***RRIP* based**
Priority bits per Set (Pb)	$log_2 16 = 4$				2
Total bits for Cache (TBPb) = $Pb \times As \times Sets$	$4 \times 16 \times 4096$ = 256Kb				128Kb
LLC POLICIES (LP)	***ABFCP***	**new *ABFCP***	***UCP* & *PIPP***	***TADIP***	***TADRRIP***
Sampled Sets (SS)	4096	32	32	32	32
bits per Monitor (Mb)	32 bits (BF entries) for 5 bits tag	256 bits (BF entries) for 8 bits tag	16 ways (UMON ATD entries) of 44 bits (tag) each = 704bits	-	-
Total Monitor bits for Cache (TMb) = $SS \times Mb \times Th$	512Kb	32Kb	88Kb	-	-
bits per Counter (Cb)	8	8	10	10	10
Number of Counters (NC)	2 (C_{LRUHIT} C_{BFHIT}) per sample	3 (C_{LRUHIT} C_{BFHIT} $Accesses$) per sample	16 (1 counter per each UMON priority position) for all samples	1 ($PSEL$) per sample	1 ($PSEL$) per sample
Total Counter bits for Cache (TCb) = $SS \times (NC \times Cb) \times Th$	256Kb	3Kb	640bits = 0.625Kb ($SS = 1$ for counters)	1280bits = 1.25Kb	1280bits = 1.25Kb
Total LP bits for Cache (TLPb) = $TMb + TCb$	**768Kb**	**35Kb**	**88.625Kb**	**1.25Kb**	**1.25Kb**
PERCENTAGE of fewer required LP bits with respect to best LLC policy's LP bits (*UCP*)	-766% or $\times 7.66$	60.5% or $\times 0.4$	0% or $\times 1$	98.6% or $\times 0.01$	98.6% or $\times 0.01$
TOTAL EXTRA bits (Tb) = $TBPb + TLPb$	**1Mb**	**291Kb**	**344.626Kb**	**257.25Kb**	**129.25Kb**
PERCENTAGE of EXTRA bits with respect to CACHE SIZE	**+3%**	**+0.9%**	**+1.1%**	**+0.8%**	**+0.4%**

It seems that the proposed *ABFCP* optimisations improve the behaviour of original *ABFCP*, so much that in some cases it ties in both performance and fairness the top-performing examined LLC schemes, like *UCP* and *TADRRIP*, while requiring reduced hardware overhead.

Finally, Figure 4.18 shows throughput of *LRU* managed LLC for different cache sizes, where the number of sets is kept constant (4096 sets) and the number of ways is variated accordingly. The results are normalised to best new *ABFCP*, so as to estimate how much more cache size would *LRU* require in order to reach or surpass its performance.

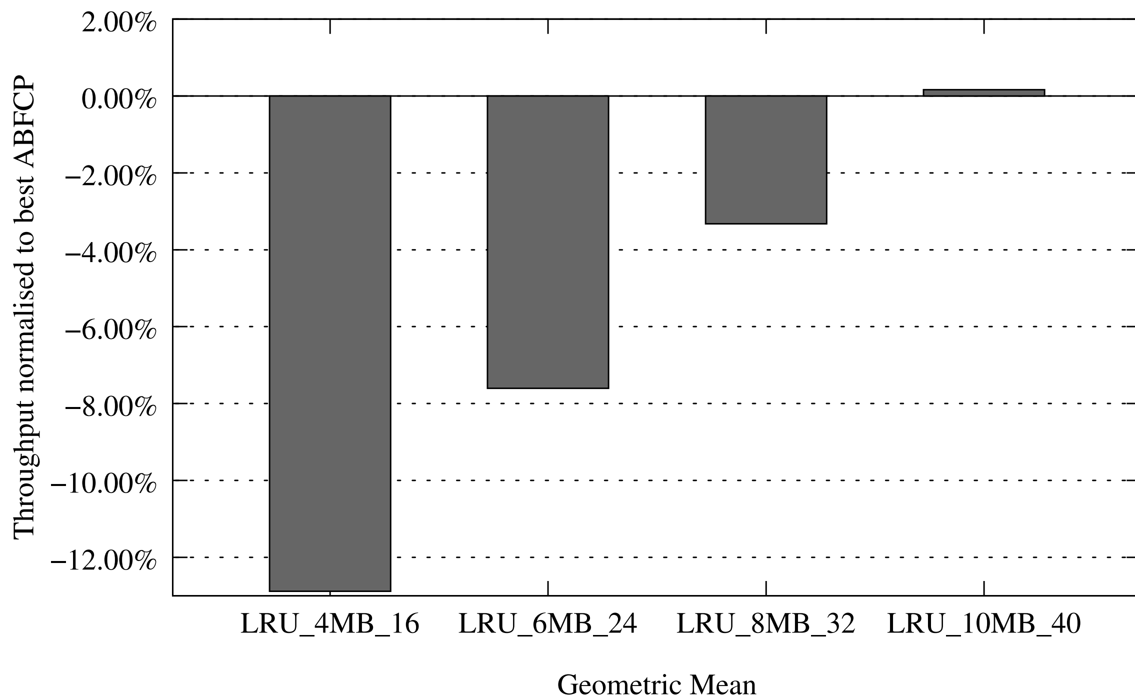


Figure 4.18: Throughput of *LRU* managed LLC for different cache configurations, normalised to best new *ABFCP*.

Best new *ABFCP* on a 4MB 16-way cache requires 0.9% extra hardware and performs equally to *LRU* on a 10MB 40-way cache, which requires 150% extra hardware compared to the 4MB 16-way one.

"The science of today is the technology of tomorrow."

Edward Teller

"If you control the code, you control the world. This is the future that awaits us."

Marc Goodman

5 Conclusions

In this diploma thesis several replacement policies have been examined, for a 4-core system equipped with a 4MB 16-way set associative last level cache. For the simulations CMP$im simulator was used, as well as 15 workload mixes of 4 out of 17 benchmarks explicitly picked from SPEC CPU 2006 benchmark suite, so as to estimate a system's performance running real-life applications that stress both CPU and memory.

5.1 Overview

ABFCP was chosen among the policies for study and potential optimisations. Several parameters of *ABFCP* were examined or differentiated, among which were:

- Introduction of +/-N (all ways) partitioning instead of just +/-1 partitioning.
- Halving of counters instead of just resetting on each repartition phase.
- Uniform (same for all cache) instead of per-set partitioning.
- New Estimator for *ABFCP*, which correlates the information of the existing counters, in order to take partitioning decisions.
- Sampling of cache sets, having either one set or all sets of each group decide for the whole group.
- Reduction of BF's tag bits.

The best among these *ABFCP* versions used new estimator, +/-1, uniform partitioning, with halving counters, 32 sets sampling (nA+-1uni_div2_32_s) and 8 bits BF tag. Best new *ABFCP* requires 72% less hardware than original *ABFCP*, while improving performance by 37% and fairness by 23%. Even though best new *ABFCP* requires 50% more hardware compared to *TADRRIP*, it performs 23% better for the throughput metric and is equally fair for the harmonic mean fairness metric. Moreover best new *ABFCP* requires 15% less

hardware compared to *UCP*, performing 2% worse while being 10% fairer. **In the 'performance race' it manages to 'climb' from 4th place among the 5 compared schemes to 2nd, closely following the 1st *UCP*. In the 'fairness race' it manages to share the 1st place with *TADRRIP*, leaving behind its predecessor at 4th place**. **Finally best new *ABFCP* on a 4MB 16-way cache requires 0.9% extra hardware and performs equally to *LRU* on a 10MB 40-way cache, which requires 150% extra hardware compared to the 4MB 16-way one**.

5.2 Future Work

ABFCP is a partitioning scheme, therefore it was primarily compared to the other partitioning scheme *UCP*, which performs better than all compared LLC policies. *ABFCP* optimisations attempted to examine the design differences between the two schemes and choose the best-performing features. However there was another approach for replacement policies, like variation of *insertion policy*, used in *TADIP*,*TADRRIP* and *PIPP*, which was not examined for *ABFCP* in this thesis. Some thoughts for future work are:

- Usage of *ABFCP* partitioning to a hybrid scheme, like *PIPP*, where partitioning decisions aren't enforced during victim selection, but instead denote insertion priority.
- Usage of *RRIP* or *NRU* instead of *LRU* as the underlying policy of *ABFCP*. This will allow for more hardware overhead reduction, like in *TADRRIP*, but may also introduce a performance improvement, similar to that demonstrated from *TADIP* to *TADRRIP*. By using an approximation of *LRU*, like *RRIP*, one among the least recently used blocks will be evicted, instead of the least recently used, opening the way for re-design of several *ABFCP* structures.
- Evaluation of power consumption requirements of each LLC scheme.

The perfect extraction and combination of information is a difficult yet challenging and charming task to achieve, especially during run-time. Thus technology and mankind advance, for it is the never-ending need for knowledge and exploration, that motivates humans to surpass and expand the boundaries they confront each time.

Appendices

"Whether we are based on carbon or on silicon makes no fundamental difference; we should each be treated with appropriate respect."

Arthur C. Clarke, 2010: Odyssey Two

Details for LLC Replacement Policies

A.1 UCP's Partitioning Algorithms

The *exhaustive* partitioning algorithm finds the best among all possible partitions, but finding an optimal solution for more than 2 cores becomes prohibitively complex (NP-hard). In fact it is proven that there are $\binom{N-1}{N-C} = \frac{(N-1)!}{(N-C)!(C-1)!}$ different possible partitions, given that all cores take at worst 1 way. For instance if $N = [16, 32]$ and $C = [4, 8]$ then there are $[455, 2629575]$ possible partitions respectively. For this reason Qureshi and Patt [5] proposed two approximate algorithms that find a nearly optimal solution. Their initial approach was the *Greedy algorithm*, which gives a way to the application with the maximum utility for this given way, iterating until all ways are distributed and is depicted in Algorithm 1. It finds an optimal solution if utility is a convex function of ways, but may behave pathologically in case of non convex utility functions.

To confront this problem Qureshi and Patt [5] proposed *Lookahead algorithm*. Let *marginal utility* MU denote the utility U per unit cache resource (way or block). Using the notation of subsection 2.4.1:

$$MU_a^b = \frac{miss_a - miss_b}{b - a} = \frac{\sum_{way=a+1}^{b} hitcounters[way]}{b - a} = \frac{U_a^b}{b - a}. \tag{A.1}$$

This algorithm can take into account possible gains from more than one extra ways for each core, by calculating MU for all possible ways that an application can have. During each iteration maximum MU (max_mu) and minimum ways needed to achieve that are calculated for each core. The core with the biggest max_mu receives the corresponding minimum required ways. The algorithm is repeated until all ways are distributed and since at least 1 way is granted on each iteration, it requires in the worst case $N + (N - 1) + \cdots + 1 = N(N - 1) \approx N^2/2$ calculations for an N-way set associative cache. Algorithm 2 shows the above logic. Lookahead algorithm outperforms Greedy and shows little to no loss of performance compared to the optimal exhaustive one.

Algorithm 1: UCP's Greedy Algorithm

```
Input: N total ways, C total cores/applications, hitcounters for each way
Output: partitioning allocations for each application
begin
    balance ⟵ N
    for i ⟵ 0 to C − 1 do
        allocations[i] ⟵ 0
    while balance > 0 do
        for i ⟵ 0 to C − 1 do
            alloc ⟵ allocations[i]
            Unext[i] = get_util_value(i, alloc, alloc + 1)
        winner ⟵ application with maximum value of Unext
        allocations[winner] ⟵ allocations[winner] + 1
        balance ⟵ balance − 1
    return allocations
Function get_util_value(p, a, b) :
    U ⟵ Σ_{way=a+1}^{b} hitcounters[way]
    return U
```

Algorithm 2: UCP's Lookahead Algorithm

```
Input: N total ways, C total cores/applications, hitcounters for each way
Output: partitioning allocations for each application
begin
    balance ⟵ N
    for i ⟵ 0 to C − 1 do
        allocations[i] ⟵ 0
    while balance > 0 do
        for i ⟵ 0 to C − 1 do
            alloc ⟵ allocations[i]
            max_mu[i] ⟵ get_max_mu(i, alloc, balance)
            blocks_req[i] ⟵ min blocks to get max_mu[i]
        winner ⟵ application with maximum value of max_mu
        allocations[winner] ⟵ allocations[winner] + blocks_req[winner]
        balance ⟵ balance − blocks_req[winner]
    return allocations
Function get_max_mu(p, alloc, balance) :
    max_mu ⟵ 0
    for ii ⟵ 1 to balance do
        mu ⟵ get_mu_value(p, alloc, alloc + ii)
        if mu > max_mu then
            max_mu ⟵ mu
    return max_mu
Function get_mu_value(p, a, b) :
    MU ⟵ Σ_{way=a+1}^{b} hitcounters[way]
    return MU/(b − a)
```

A.2 ABFCP's Partitioning Algorithms

Rather than evaluating all possible partition changes as shown in Table A.1 and choosing the best choice that maximizes a given metric, Nikas et al. [6] proposed a *Linear algorithm* that selects the best partition or a good approximation thereof. This algorithm initially reads both counters of each core. Then, in each iteration, the maximum gain value is compared against the minimum loss value. If the former is greater, then the allocation of the core associated with that C_{BFHIT} counter is increased by one way, while the core associated with that C_{LRUHIT} counter is deprived of one way. The process is repeated until no cores are left to be considered or the maximum gain value is smaller than the minimum loss value. Algorithm 3 shows the described logic. In the worst case $C/2$ comparisons need to be performed, where C the number of cores. Therefore, the complexity of this algorithm is $O(C)$.

Table A.1: Possible Partition Changes.

Cores	Part. Ch.
2	3
4	19
8	1107
16	5196627

Algorithm 3: ABFCP's Linear Algorithm

Input: **C** total cores/applications, $\mathbf{C_{BFHIT}}$, $\mathbf{C_{LRUHIT}}$, previous **allocations** of each core
Output: new **allocations** of each core
begin
  for $i \longleftarrow 0$ **to** $C - 1$ **do**
    $gain_1[i] \longleftarrow \alpha \times C_{BFHIT}$
    $loss_1[i] \longleftarrow C_{LRUHIT}$
  order $gain_1$ and $loss_1$ from min to max value
  while $max(gain_1) > min(loss_1)$ **do**
    $i \longleftarrow$ core of $max(gain_1)$ value
    $j \longleftarrow$ core of $min(loss_1)$ value
    $allocations[i] \longleftarrow allocations[i] + 1$
    $allocations[j] \longleftarrow allocations[j] - 1$
    remove $i,\ j$ from $gain_1,\ loss_1$
    $C \longleftarrow C - 2$
    if $C \leq 1$ **then**
      return $allocations$
  return $allocations$

Optimisation of *ABFCP* included the introduction of a new estimator which correlates the information stored in counters to take partitioning decisions. Algorithm 4 shows the new estimator logic for +/-1 partitioning.

Algorithm 4: ABFCP's new Estimator Linear Algorithm

Input: **C** total cores/applications, $\mathbf{C_{BFHIT}}$, $\mathbf{C_{LRUHIT}}$, $Accesses$, previous **allocations** of each core
Output: new **allocations** of each core
begin
  for $i \longleftarrow 0$ **to** $C-1$ **do**
    $gainorloss_1[i] \longleftarrow \alpha \times C_{BFHIT} \times C_{LRUHIT}/Accesses$
  order $gainorloss_1$ from min to max value
  while $max(gainorloss_1) > min(gainorloss_1)$ **do**
    $i \longleftarrow$ core of $max(gainorloss_1)$ value
    $j \longleftarrow$ core of $min(gainorloss_1)$ value
    $allocations[i] \longleftarrow allocations[i] + 1$
    $allocations[j] \longleftarrow allocations[j] - 1$
    remove $i,\ j$ from $gainorloss_1$
    $C \longleftarrow C - 2$
    if $C \leq 1$ **then**
      return $allocations$
  return $allocations$

A.3 PIPP's stream-sensitive mechanism

Xie and Loh [10] add a *stream-sensitive* mechanism to make *PIPP* stream-resistant. Although partitioning schemes may recognize any cache-unfriendly patterns and restrict them by allocating few ways to such cores, stream-sensitive *PIPP* adds an extra mechanism for safety. By using UMONs, *PIPP* tracks the total number of accesses by $core_i$ (A_i) and the number of misses the core would experience if it had access to the entire cache for itself (m_i). If a certain threshold is exceeded either by the total number of misses ($m_i \geq \theta_m$) or by the miss rate ($\frac{m_i}{A_i} \geq \theta_{mr}$), then *PIPP* assumes the core is running a stream-like application. The intuition behind this modification is that an application with a large number of absolute misses (m_i) or miss rate ($\frac{m_i}{A_i}$) would likely cause significant thrashing and any resource investment would be wasted. When *PIPP* detects a streaming $core_s$, it makes all insertions at priority position π_{stream}, regardless of target partition π_s. Insertion position π_{stream} is set to equal the current number of stream-like applications. Promotion for hits due to $core_s$ only occur with a reduced probability $p_{stream} \ll p_{prom}$. For the corner case where all applications simultaneously exhibit streaming behaviour, *PIPP* reverts to inserting all lines at highest-priority position, since there are no cache-friendly applications to hurt. The best values chosen for *PIPP* are $p_{prom} = \frac{3}{4}$, $p_{stream} = \frac{1}{128}$, $\theta_m \geq 4095$ and $\theta_{mr} = 12.5\% = \frac{1}{8}$.

Bibliography

[1] Intel and Hewlett Packard. *Inside the Intel® Itanium® 2 Processor: an Itanium Processor Family member for balanced performance over a wide range of applications*. 2002. URL http://www.dig64.org/about/Itanium2_white_paper_public.pdf.

[2] Oracle and Sun Microsystems. *UltraSPARC T2™ Supplement to the UltraSPARC Architecture*. 2007. URL http://www.oracle.com/technetwork/systems/opensparc/t2-14-ust2-uasuppl-draft-hp-ext-1537761.html.

[3] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. S. Jr. and J. S. Emer. *Adaptive insertion policies for managing shared caches*. In A. Moshovos, D. Tarditi and K. Olukotun, eds., 17th International Conference on Parallel Architecture and Compilation Techniques (PACT 2008), Toronto, Ontario, Canada, October 25-29, 2008, pp. 208–219. ACM, 2008. URL http://doi.acm.org/10.1145/1454115.1454145.

[4] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger. *Evaluation Techniques for Storage Hierarchies*. IBM Syst. J., vol. 9(2):pp. 78–117, 1970. ISSN 0018-8670. URL http://dx.doi.org/10.1147/sj.92.0078.

[5] M. K. Qureshi and Y. N. Patt. *Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches*. In 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA, pp. 423–432. IEEE Computer Society, 2006. URL http://doi.ieeecomputersociety.org/10.1109/MICRO.2006.49.

[6] K. Nikas, M. Horsnell and J. D. Garside. *An adaptive bloom filter cache partitioning scheme for multicore architectures*. In W. A. Najjar and H. Blume, eds., Proceedings of the 2008 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2008), Samos, Greece, July 21-24, 2008, pp. 25–32. IEEE, 2008. URL http://dx.doi.org/10.1109/ICSAMOS.2008.4664843.

[7] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark and K. Lai. *Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching*. In Proceedings of the 16th International Conference on Supercomputing, ICS '02, pp. 189–198. ACM, New York, NY, USA, 2002. ISBN 1-58113-483-5. URL http://www.cise.ufl.edu/~peir/pdf2/ics02.pdf.

[8] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr. and J. S. Emer. *Adaptive insertion policies for high performance caching*. In D. M. Tullsen and B. Calder, eds., 34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007,

San Diego, California, USA, pp. 381–391. ACM, 2007. URL http://doi.acm.org/10.1145/1250662.1250709.

[9] A. Jaleel, K. B. Theobald, S. C. S. Jr. and J. S. Emer. *High performance cache replacement using re-reference interval prediction (RRIP)*. In A. Seznec, U. C. Weiser and R. Ronen, eds., 37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France, pp. 60–71. ACM, 2010. URL http://doi.acm.org/10.1145/1815961.1815971.

[10] Y. Xie and G. H. Loh. *PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches*. In S. W. Keckler and L. A. Barroso, eds., 36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA, pp. 174–183. ACM, 2009. URL http://doi.acm.org/10.1145/1555754.1555778.

[11] A. Jaleel, R. S. Cohn, C.-K. Luk and B. Jacob. *CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator*. In Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA'2008. 2008. URL http://www.jaleels.org/ajaleel/publications/cmpsim_mobs2008.pdf.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood. *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation*. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pp. 190–200. ACM, New York, NY, USA, 2005. ISBN 1-59593-056-6. URL http://doi.acm.org/10.1145/1065010.1065034.

[13] The Journal of Instruction-Level Parallelism. *1st JILP Workshop on Computer Architecture Competitions (JWAC-1): Cache Replacement Championship*, . 2010. URL http://www.jilp.org/jwac-1/.

[14] J. L. Henning. *SPEC CPU2006 Benchmark Descriptions*. SIGARCH Comput. Archit. News, vol. 34(4):pp. 1–17, 2006. ISSN 0163-5964. URL http://doi.acm.org/10.1145/1186736.1186737.

[15] A. Snavely and D. M. Tullsen. *Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor*. In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX, pp. 234–244. ACM, New York, NY, USA, 2000. ISBN 1-58113-317-0. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.7963&rep=rep1&type=pdf.

[16] L. Kun, G. Jayanth and F. Manoj. *Balancing thoughput and fairness in SMT processors*. In International Symposium on Performance Analysis of Systems and Software. 2001.