A Study of Low Power and High Performance Cache Hierarchy for Multi-Core Processor

by

Geng Tian

B.Engineering. (Electrical and Electronic), Xi’an Jiaotong University, 2006
M.Engineering. (Electrical and Electronic), University of Adelaide, 2009

Thesis submitted for the degree of

Doctor of Philosophy

in

Electrical and Electronic Engineering
University of Adelaide

2015
Contents

Chapter 1. Introduction and Background 1
   1.1 Introduction and Motivation .................................... 2
   1.2 Background: Tiled CMP Architectures and Their Cache .......... 3
      1.2.1 Tiled CMP Architecture ....................................... 3
      1.2.2 Cache Design Background .................................... 8
      1.2.3 Non-Uniform Cache Architecture ............................... 13
      1.2.4 Power Dissipation Model and Simulation Method ............. 19
   1.3 Related Work ...................................................... 22
      1.3.1 Power Saving Cache ........................................... 22
      1.3.2 Novel Cache Replacement Policy ............................... 24
      1.3.3 Cache Access Latency Reduction ............................... 26
   1.4 Thesis overview: Significance and Contributions ................. 29

Chapter 2. Utility Based Cache Resizing 31
   2.1 Introduction ..................................................... 32
   2.2 Overall Power Analyses ......................................... 33
2.3 Utility of the Cache Block .............................................. 34
2.4 Utility-Based Monitoring .............................................. 35
  2.4.1 Stack Distance ..................................................... 35
  2.4.2 Performance vs Cache Resources ......................... 36
2.5 Increasing Cache Algorithm .......................................... 38
  2.5.1 Rationale of the Size-up Approach ....................... 38
  2.5.2 Concept of BEM ................................................. 40
  2.5.3 L1 Early Eviction .............................................. 44
  2.5.4 Evaluation Platform and Benchmarks ................... 49
  2.5.5 Up-Sizing Profile .............................................. 50
  2.5.6 Evaluation of Up Sizing Only Scheme .................. 51
2.6 Bi-Directional Tuning Scheme ....................................... 53
  2.6.1 Pre-reduction Preparation ................................. 56
  2.6.2 Thrashing Protection ...................................... 58
  2.6.3 Down-sizing Evaluation .................................. 59
2.7 Tile Level Evaluation ................................................ 63
2.8 Summary ................................................................. 65

Chapter 3. An Effectiveness-Based Adaptive Cache Replacement Policy  67
3.1 Introduction ............................................................ 68
3.2 Re-definition of Cache Optimal Replacement .................. 69
3.3 Study of Different Replacement Policy ......................... 77
  3.3.1 Optimal Replacement Policy ............................. 77
  3.3.2 Least Recent Used Replacement Policy (LRU): ........ 77
  3.3.3 Least Frequently Used Replacement Policy (LFU): ...... 80
3.4 Effectiveness Based Replacement Policy (EBR): ............. 81
3.5 Evaluation Environment ............................................. 83
  3.5.1 Evaluation Platform ...................................... 83
  3.5.2 Benchmark .................................................. 84
3.6 Evaluation of EBR .................................................... 85
3.7 Dynamic EBR ........................................................ 86
## Contents

3.8 Evaluation of DEBR ......................................................... 89
3.9 Sensitivity Study for Rank Granularity ................................. 90
3.10 Hardware Implementation and Overhead ............................. 98
  3.10.1 Replacement Logic .................................................. 98
  3.10.2 Storage Overhead .................................................... 99
3.11 Summary ................................................................. 100

### Chapter 4. Cache Dynamic Resize using EBR based Cache Utility Evaluation  103

4.1 Introduction ............................................................. 104
4.2 Further Study of LRU Prediction ...................................... 105
  4.2.1 Future Access Trace ................................................ 105
  4.2.2 Rank Distance Estimation ....................................... 108
4.3 Algorithm of EBR based increasing only mechanism .............. 117
4.4 False Evicting Rank Reuse ............................................. 117
4.5 Deterministic Evaluation Analyse .................................... 120
  4.5.1 Non Deterministic Simulation .................................... 120
  4.5.2 Trace Driven Evaluation ......................................... 123
4.6 LRU-based vs EBR-based Increasing Scheme ....................... 126
4.7 EBR-based Reducing Scheme ......................................... 137
4.8 EBR Reduced Evaluation and Comparison with LRU based Reducing . 140
4.9 Hardware Overhead Analysis ........................................... 145
4.10 Dynamic EBR resizing ................................................ 150
4.11 Summary ............................................................... 153

### Chapter 5. EBR based Dynamic Private and Shared Cache Partitioning  157

5.1 Introduction and motivation ........................................... 158
5.2 Latency analysis ........................................................ 159
5.3 Private/Shared Partition ................................................ 161
  5.3.1 L1 Victim can be Useful ......................................... 162
  5.3.2 Foundation of Partition ........................................... 162
  5.3.3 Cache Query Process .............................................. 164
5.3.4 L1 Cache Eviction ............................................... 165
5.3.5 Coherency Protocol ............................................. 165
5.3.6 Replacement Policy Partitioning ........................... 166
5.4 Dynamic Partitioning .............................................. 167
5.5 Performance Evaluation .......................................... 170
  5.5.1 Fixed Partition Evaluation ................................. 170
  5.5.2 Dynamic Partitioning Evaluation ......................... 173
5.6 Dynamic Partition With Resize ............................... 178
5.7 Summary .......................................................... 179

Chapter 6. Conclusion and Future Work 181
  6.1 Conclusion ....................................................... 182
  6.2 Future Work ..................................................... 183
    6.2.1 EBR Based Bypass ......................................... 184
    6.2.2 Private Cache Evaluation ............................... 184
    6.2.3 Cache Resizing Without Set-Associativity ............. 184

Bibliography 185
Abstract

The increasing levels of transistor density have enabled integration of an increasing number of cores and cache resources on a single chip. However, power, as a first order design constraint may bring this trend to a dead end. Recently, the primary design objective has been shifted from pursuing faster speed to higher power-performance efficiency. This is also reflected by the fact that design preference has transitioned from fast super-scalar architecture to slower multi-core architecture.

Tiled chip multiprocessors (CMPs) have shown unmatched advantages in recent years, and they are very likely to be the mainstream in the future. Meanwhile, increasing number of cores will exert higher pressure on the cache system. Expanding cache storage can ease the pressure but will incur higher static power consumption. More importantly, very large caches in future multi-core systems may not be fully utilised. Under-utilised caches consume static power for no productivity. Off-line profiling of applications to determine optimal cache size and configuration is not practical.

This thesis describes dynamic cache reallocation techniques for tiled multi core architectures. We proposed the idea of Break Even number of Misses (BEM). BEM defines, for a given cache configuration and time interval, the maximum number of misses that can be tolerated without increasing the energy delay product. We use BEM as the upper bound to determine a set of thresholds that are used to periodically evaluate the utility of cache. Based on this scheme, we then propose a conservative increase-only resizing method to tune the cache size at tile-level granularity. The increasing only method can be further extended to a dynamic downsizing scheme.

In simulations, our tuning scheme can reduce the static power significantly with a very minor degradation of IPC (Instruction Per Cycle). One thing that can be improved from our resizing scheme is the replacement policy. The estimation of cache utility is based on stack-distance which relies on the recency position of a real LRU (least recently used) replacement policy. As is commonly known, LRU is not easy to implement, especially in high associativity caches, which are becoming more commonly used. LRU also suffers from “cache thrashing”, “scan” and “inter-thread interference”. To solve
these three problems, we further propose a novel replacement policy, the Effectiveness-Based Replacement policy (EBR) and a refinement, Dynamic EBR (D-EBR), which combines measures of recency and frequency to form a rank sequence inside each set and evict blocks with lowest rank. To evaluate our design, we simulated all 30 applications from SPEC CPU2006 for uni-core system and a set of combinations for 4-core systems, for different cache sizes. The results show that both of them can achieve higher performance with only half the hardware overhead of real LRU. With the help of EBR, we further extend our last level cache resize scheme. We discuss how to estimate equivalent utility of cache using EBR replacement policy rather than LRU, and introduce an EBR based resizing scheme. Since EBR replacement policy is hardware economical and cache thrashing protected, it is more suitable for the utility estimation.

Finally, to shorten average cache access latency, we propose the idea of using a private replica region to store useful data replicas. Keeping them close to the requester can significantly reduce average access latency, however, it also reduces effective storage size causing higher cache miss rates. We leverage our EBR based cache utility estimation method to dynamically change the partition based on cache access patterns to achieve a near optimal result.
Statement of Originality

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I give consent to this copy of my thesis, when deposited in the University Library, being made available for loan and photocopying, subject to the provisions of the Copyright Act 1968.

I also give permission for the digital version of my thesis to be made available on the web, via the University’s digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

Signed

Date
Acknowledgments

First and foremost, I would like to thank my supervisors A.Prof Michael Liebelt and Dr Braden Phillips sincerely. Without their guidance and encouragement, I would not be able to finish my research. It is my honour to pursue Ph.D under their supervision. They have taught me not only how to do research in this particular area but also taught me how to divide and conquer the puzzle.

I also want to acknowledge my colleagues and academic members in the computer system research group for their help and advice. I would like to thank the staff of School of Electrical and Electronics Engineering at the University of Adelaide for their assistance and technical support.

I would like to acknowledge Microsoft for giving me a job before I finish my thesis which makes my thesis writing much easier.

Finally, I would like to express my gratitude to my wife Jessica for her encouragement, support and patience throughout my entire candidature.


Publications

Journal


Conference

## List of Figures

1.1 Two commonly used multi core approaches ........................................ 5  
1.2 Topology:ring ............................................................................... 6  
1.3 Topology:symmetric ................................................................. 6  
1.4 Topology:bus ............................................................................. 7  
1.5 Topology:crossbar ...................................................................... 7  
1.6 Topology:2D mesh, packet-switched NoCs ................................. 8  
1.7 Base system of our dynamic cache design ................................... 9  
1.8 Computer memory hierarchy ..................................................... 10  
1.9 Different ways of mapping data from lower level memory to higher level 11  
1.10 Example of cache incoherency .................................................. 12  
1.11 Comparison between UCA with one bank and 64 cycle average access and NUCA with 16 banks .......................................................... 14  
1.12 Layout of private cache and shared cache .................................. 15  
1.13 Break down cache mapping on physical address ....................... 18  
1.14 Block diagram of the McPAT framework (Li et al. 2009) ............... 21  

2.1 Overall power breakdown ............................................................... 34  
2.2 The minimal cache size (represented by number of ways) to achieve 95% 99% and 99.9% of original performance during the runtime of Libquantum (Uni-Core / 128 Sets / 16 Way-Associativity / 64 KB block size) .... 35  
2.3 Hits vs Misses for different configurations .................................... 37  
2.4 Performance VS Cache size ......................................................... 38  
2.5 ferret VS blackscholes, number of hits at different stack distances .... 39  
2.6 Architecture of overall tuning framework ...................................... 41  
2.7 A demonstration of early eviction ................................................ 44  
2.8 Stack distance based estimation of total L2 misses versus measurement for different number of ways along with the total number of L1 invalidations due to L2 replacement for each size (blackscholes) ............ 46
List of Figures

2.9 Stack distance based estimation of total L2 misses versus measurement for different number of ways along with the total number of L1 invalidations due to L2 replacement for each size (x264) .......................... 47
2.10 Adjusted stack distance based estimation of total L2 misses versus measurement for different number of ways along with the total number of L1 invalidations due to L2 replacement for each size (blackscholes) .......................... 48
2.11 Adjusted stack distance based estimation of total L2 misses versus measurement for different number of ways along with the total number of L1 invalidations due to L2 replacement for each size (x264) .......................... 48
2.12 The tuning procedure of multi core framework .......................... 52
2.13 The average performance loss versus power saving .......................... 53
2.14 Temporary cache utility variation over time .......................... 54
2.15 Temporary cache utility variation over time .......................... 55
2.16 Example snapshot of cache content before way shut off .......................... 58
2.17 Utility estimation using different time intervals .......................... 59
2.18 Temporary cache utility variation over time .......................... 61
2.19 Temporary cache utility variation over time .......................... 62
2.20 Comparison between the increase-only and increase/reduce tuning schemes 63
2.21 L2 demand of each tile for different applications, the sixteen tiles are represented as a grid in the horizontal plane of each graph .......................... 64
2.22 Actual L2 resource allocation after applying resize tuning .......................... 65

3.1 Illustration of effectiveness example 1: Future access sequence of block A and block B .......................... 71
3.2 Illustration of effectiveness example 1: Block B remains in the cache all the time .......................... 72
3.3 Illustration of effectiveness example 1: Swap block B with block A between T2 to T5 .......................... 72
3.4 Example 2 to illustrate block effectiveness: Future access sequence of Block A and Block B .......................... 73
3.5 Example 2 to illustrate block effectiveness: Swap block B with block A upon each miss .......................... 74
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Example 2 to illustrate block effectiveness: Block B remains in the cache all the time</td>
<td>74</td>
</tr>
<tr>
<td>3.7</td>
<td>Illustration of effectiveness example 3</td>
<td>76</td>
</tr>
<tr>
<td>3.8</td>
<td>Three different cache access patterns</td>
<td>79</td>
</tr>
<tr>
<td>3.9</td>
<td>MITT Demonstration</td>
<td>83</td>
</tr>
<tr>
<td>3.10</td>
<td>Rank Sequence</td>
<td>84</td>
</tr>
<tr>
<td>3.11</td>
<td>Normalized total number of misses using EBR with 1MB cache</td>
<td>87</td>
</tr>
<tr>
<td>3.12</td>
<td>Normalized total number of misses using EBR with 2MB cache</td>
<td>87</td>
</tr>
<tr>
<td>3.13</td>
<td>Normalized total number of misses using EBR with 4MB cache</td>
<td>87</td>
</tr>
<tr>
<td>3.14</td>
<td>MITT 1</td>
<td>89</td>
</tr>
<tr>
<td>3.15</td>
<td>Normalized total number of misses achieved by EBR with different MITT compared to dynamic EBR</td>
<td>91</td>
</tr>
<tr>
<td>3.16</td>
<td>Normalized total number of misses achieved by EBR with different MITT compared to dynamic EBR</td>
<td>92</td>
</tr>
<tr>
<td>3.17</td>
<td>Normalized total number of misses achieved by EBR with different MITT compared to dynamic EBR</td>
<td>93</td>
</tr>
<tr>
<td>3.18</td>
<td>Normalized Total Number of Misses using Dynamic EBR with 1MB cache</td>
<td>94</td>
</tr>
<tr>
<td>3.19</td>
<td>Normalized total number of misses using dynamic EBR with 2MB cache</td>
<td>94</td>
</tr>
<tr>
<td>3.20</td>
<td>Normalized total number of misses using dynamic EBR with 4MB cache</td>
<td>94</td>
</tr>
<tr>
<td>3.21</td>
<td>Normalized total number of misses compared among different D-EBR complexities for 1M 16way cache</td>
<td>96</td>
</tr>
<tr>
<td>3.22</td>
<td>Normalized total number of misses compared among different D-EBR complexities for 2M 16way cache</td>
<td>96</td>
</tr>
<tr>
<td>3.23</td>
<td>Normalized total number of misses compared among different D-EBR complexities for 4M 16way cache</td>
<td>97</td>
</tr>
<tr>
<td>3.24</td>
<td>Normalized total number of misses compared among different D-EBR complexities for 4M 64way cache</td>
<td>97</td>
</tr>
<tr>
<td>3.25</td>
<td>Simulation result of D-EBR for multi-application</td>
<td>98</td>
</tr>
<tr>
<td>3.26</td>
<td>Hardware requirements for EBR</td>
<td>99</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of future access trace</td>
<td>106</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>The number of extra misses that would be incurred by reducing cache associativity by 1 from different cache size. Test-bench: SPEC CPU2006: <em>gamess</em>, 1 billion instruction, uni-core, 4 MByte L2Cache.</td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>The number of extra misses that would be incurred when reducing cache associativity by 1 from different cache sizes. Test-bench: SPEC CPU2006: <em>namd</em>, 1 billion instruction, Uni-core, 4 MByte L2Cache.</td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>The number of extra misses would incur when reduce cache associativity by 1 from different cache size. Compare among actual measurement, estimation with knowledge of evicting rank width and estimation without knowledge of evicting rank width. Test-bench: SPEC CPU2006: <em>libquantum</em>, 1 billion instruction, uni-core, 4 MByte L2Cache.</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Runtime utility (the minimum run-time cache size to ensure a certain percentage of performance compared with a fully activated cache) of <em>bodytrack</em> and timing of way allocation events at each tile. Test-bench: PARSEC2: <em>bodytrack</em>, 1 billion instruction, 16-core, 1 MByte x 16 Shared L2 cache, core 1 to core 4 first row from left to right, core 5 to core 8 second row from left to right, core 9 to core 12 third row from left to right, core 13 to core 16 last row from left to right.</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>Runtime utility (the minimum run-time cache size to ensure a certain percentage of performance compared with a fully activated cache) of <em>bodytrack</em> and timing of way allocation events at each tile. Test-bench: PARSEC2: <em>bodytrack</em>, 1 billion instruction, 16-core, 1 MByte x 16 Shared L2 cache, core 1 to core 4 first row from left to right, core 5 to core 8 second row from left to right, core 9 to core 12 third row from left to right, core 13 to core 16 last row from left to right.</td>
<td></td>
</tr>
<tr>
<td>4.7</td>
<td>Increased only mechanism, LRU and EBR based comparison in performance, L2 leakage power, total power, EDP.</td>
<td></td>
</tr>
<tr>
<td>4.8</td>
<td>Runtime heatmap of cache utility during LRU based increase-only resize when running <em>Blackscholes</em>, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.</td>
<td></td>
</tr>
</tbody>
</table>
4.9  Runtime heatmap of cache utility during EBR based increase-only resize when running *Blackscholes*, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................ 130

4.10  Runtime heatmap of cache utility during LRU based increase-only resize when running *Bodytrack*, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................ 131

4.11  Runtime heatmap of Cache Utility during EBR based increase-only resize when running *Bodytrack*, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................ 132

4.12  Runtime heatmap of Cache Utility during LRU based increase-only resize when running *Canneal*, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................ 133

4.13  Runtime heatmap of Cache Utility during EBR based increase-only resize when running *Canneal*, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................ 134
List of Figures

4.14 Runtime heatmap of Cache Utility during LRU based increase-only resize when running X264, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................... 135

4.15 Runtime heatmap of Cache Utility during EBR based increase-only resize when running X264, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................... 136

4.16 State machine of reducing state bit ................................................................. 139

4.17 Increase/reduce mechanism, LRU and EBR based comparison in performance, L2 leakage power, total power, EDP ................................................................. 141

4.18 Comparison of performance, L2 leakage power, total power, EDP between increasing only mechanism and increase/reduce mechanism ........................................ 144

4.19 Runtime heatmap of cache utility during EBR based increase/reduce resize when running Blackscholes, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................... 146

4.20 Runtime heatmap of cache utility during EBR based increase/reduce resize when running bodytrack, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................... 147

4.21 Runtime heatmap of cache utility during EBR based increase/reduce resize when running canneal, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ........................................... 148
4.22 Runtime heatmap of cache utility during EBR based increase/reduce resize when running x264, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime ................................. 149

4.23 Hardware implementation of EBR based cache resizing and overhead at each set ................................................................. 151

4.24 Comparison between dynamic-EBR based tuning scheme and static-EBR based tuning scheme, in performance, L2 leakage power, total power, EDP when threshold T=8 .................................................... 154

5.1 Breakdown of total memory access latency of 16 Core system ........ 160

5.2 Breakdown of total memory access latency of 64 Core system .......... 160

5.3 Base system of our dynamic cache design ................................. 161

5.4 Different ways to interpret a data address when the cache size is different ................................................................. 163

5.5 An illustration of way partitioning ............................................ 164

5.6 A demonstration of how P/S controller works ............................ 167

5.7 P/S resizeable EBR replacement policy hardware ........................ 168

5.8 Exploring all possible static partitions - 1 ............................... 171

5.9 Exploring all possible static partitions - 2 ............................... 172

5.10 Dynamic private shared cache partition performance normalized to the fixed partition (2P:14S) ..................................................... 174

5.11 Partition achieved by dynamic partitioning vs optimal partitioning ... 175

5.12 Dynamic private shared cache partitioning performance normalized to the optimal scheme .................................................. 177

5.13 Tuning procedure in time .................................................... 177

5.14 Dynamic partitioning with cache resizing integrated (Number of misses and access latency are normalized to running under fully activated shared cache) .................................................. 179
# List of Tables

1.1 Flynn’s taxonomy .................................................. 4
1.2 Example of directory state entry ................................. 13

2.1 Recency position distribution upon hits ......................... 36
2.2 A set of example parameters based on our simulation framework .................. 42
2.3 Parameters for the simulation framework: multi core platform and single core platform ........................................ 49
2.4 Proportion of different block status upon way shut off .......... 57
2.5 Average performance (CPI), leakage power, total power and EDP of different tunings scheme and different threshold value relative to a full size cache ........................................... 60

3.1 Parameters for the simulation framework ........................ 84
3.2 Simulation workloads .............................................. 85
3.3 Parameters for the simulation framework ........................ 100

4.1 Replacement policy related status ............................... 106
4.2 Count reuse at evicting rank .................................... 109
4.3 Evicting rank size upon reuse SPEC CPU 2006 game associativity=16 . 111
4.4 Evicting rank size upon reuse SPEC CPU 2006 game associativity=5 . 113
4.5 PARSEC2 16MB cache 16 core: MRU re-reference/total cache re-reference 113
4.6 SPEC CPU2006 4MB cache uni core: MRU re-reference/total cache re-reference .................................................. 114
4.7 PARSEC2: streamcluster 16MB cache 16 core: run 1 and run 2 ......... 122
4.8 PARSEC2: blackscholes: performance estimation verification .......... 124
4.9 PARSEC2: streamcluster performance estimation verification .......... 124
4.10 PARSEC2: ferret: performance estimation verification ............. 126
List of Tables

4.11 Average EDP of all 8 applications from PARSEC2 achieved by different threshold with EBR based increasing only tuning mechanism and LRU based increasing only tuning mechanism, both normalized to LRU running with full size cache ........................................ 137
4.12 Average EDP of all 8 applications from PARSEC2 achieved by different threshold with EBR based increasing only tuning mechanism and LRU based increasing only tuning mechanism, both normalized to their respective full size cache ........................................ 138
4.13 Trade off between power and performance of bodytrack with different threshold ........................................ 138
4.14 Average total power of all 8 applications from PARSEC2 relative to LRU with a fully active cache, increasing only versus increase/reduce ........................................ 140
4.15 Threshold T=8, the total power comparison between EBR based increasing only and EBR based increase/reduce ........................................ 142
4.16 Average L2 usage using EBR based increasing only method and the average activated L2 at the end of simulation ........................................ 143
4.17 Average L2 usage using EBR based Increasing/Reduce method and the Average activated L2 at the end of simulation ........................................ 145
4.18 MITT interval at different associativity for static EBR policy ........................................ 152
4.19 Normalized performance of streamcluster with different MITT ........................................ 153
5.1 Average L2 hit latency for different applications (cycles) ........................................ 160
5.2 Proportion of reused L1 victims amongst L2 access requests ........................................ 162
5.3 Parameters for the simulation framework ........................................ 170
5.4 Dynamic scheme performance and optimal scheme performance normalized to fixed scheme ........................................ 175
This chapter presents the motivation for our research before introducing tiled structures and some necessary background about cache systems which sets the foundation for later chapters in which different power-efficient ideas are discussed and compared. CMPs combined with a logically shared but physically distributed Last level Cache is a promising technology for future general purpose CPU design. It has a good scalability and more importantly it can leverage identical tiles to dramatically ease the burden of design and hardware verification. In this chapter, we provide a brief introduction and overview of tiled architectures and their cache systems, also the reason why we choose it as the baseline of our research. We explain how different cache related challenges can be when the number of cores is scaled up. Finally, this chapter will provide an overview of the organization of this thesis and its contributions.
1.1 Introduction and Motivation

Thanks to improving technology and shrinking transistor sizes it is now possible to integrate billions of transistors into a single core. This gives us the possibility to place more and more cores on a single die (Gorder 2007) (Muralimanohar and Balasubramonian 2007). Power, as the most significant constraint for today’s microprocessor design, prevents us from further exploiting thread level parallelism by implementing more cores. In mobile and embedded computing, power is directly related to battery life. In desktop systems, higher power does not simply mean higher energy consumption but also represents the thermal issue and unbalanced power density. In servers or data centres, huge power consumption is the cause of sky-high electric bills. In the future, chips would fail to work eventually because of the increasing power (Huh et al. 2001). All of these compelling reasons motivate the trend toward the research in low power design.

Before the emergence of large scale CMP’s, many circuit level low power designs have been developed for uni-core processors, such as dynamic frequency scaling, sleep transistors (Brodersen et al. 2002) (Yao et al. 1995) (Flautner et al. 2001) (Wu et al. 2005) and most of them are quite mature technologies and can be ported to the CMP design space. Although these techniques can reduce power, we know that the higher the design level at which we target reducing power, the better outcomes we can achieve. That’s why we should explore new characteristics that come with chip multi-processor systems and to propose changes at architecture level in order to gain more power-savings. And one of the most worthwhile components to research is the cache, not just because it contributes a significant portion of total power consumption but also because it has huge potential in increasing performance and saving power. Esmaeilzadeh (Esmaeilzadeh et al. 2011) has proved again that future scaling will come to a dead end, only novel architectures which can deliver performance and efficiency gains can bring us new hope.

The trend of deploying tens or hundreds of cores into one piece of silicon also exerts tremendous pressure on cache systems for many reasons. First of all, the memory wall, caused by the speed gap between cache and main memory will enlarge. Cache has to be larger since we have multiple working sets which contend with each other, leading to a waste of limited on-chip memory resource. Cache has to be big enough to avoid off-chip access as much as possible since the contention between cores would make the access to the off-chip memory even more expensive than before, however a large cache also means higher static power dissipation. Secondly, the ever-increasing latency
will force aggregated on-chip memory to be replaced by distributed memory because of the scalability requirement for future processor design. Accessing on-chip memory on the remote side of the chip may cost tens of cycles which will become an issue that we cannot overlook. Finally, the interference between different working sets can pollute the very limited on-chip resources causing inefficient usage of cache resources. In this case, designing an elegant reconfigurable cache hierarchy for future CMPs to fully realise their computing capacity as well as saving unnecessary energy waste is especially important.

1.2 Background: Tiled CMP Architectures and Their Cache

1.2.1 Tiled CMP Architecture

Why use tiles?

Moore’s Law continues to apply to today’s semiconductor industry. It is routine to see billions of transistors being integrated on a 300 mm die, and that number will keep on growing. According to the forecast from the ITRS roadmap (ITRS:2013 Executive Summary 2013), the integration capacity of a 300 mm die could exceed 100 billion transistors in year 2028. How should we employ so many transistors? Higher transistor capacity means more complex architectures and higher operating frequency (Asanovic et al. 2009) (Borkar 2007) but it is widely believed that the fundamental barrier for future MPU design is power. Higher power consumption not only means higher energy consumption but also presents thermal dissipation and unbalanced power density issues. Eventually chips will fail to work because of increased power consumption (Brodersen et al. 2002) (Huh et al. 2001).

Pollack’s Rule asserts that performance increase is proportional to the square root of complexity, whereas power increases linearly with complexity. It is an unfortunate fact that the step of aggressive use of ILP (instruction level parallelism) has to slow down before we hit the power wall. Thread level parallelism and many-core systems will be a solution (Azimi et al. 2007). The idea of many-core systems is using multiple simple cores to take the place of the very complex and large superscalar core. It raises the level of parallelism from instruction level to thread level. By doing so, future designs can avoid the diminishing returns in exploiting ILP. It can distribute threads to different
1.2 Background: Tiled CMP Architectures and Their Cache

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multi Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multi Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

Table 1.1. Flynn’s taxonomy

cores instead of throwing all of them into a single but large multiple-issue core which carefully arranges the instructions to exploit instruction level parallelism. If, for example, we break a complex core into 16 smaller and simpler cores (with the same total area), we would have 16 cores each delivering $\frac{1}{4}$ of the single core performance without increasing area or power according to Pollack’s rule. Within the same area and power, we end up with a fourfold increase in performance.

Why tiled structure?

To realise thread level parallelism, a tiled structure is not the only option. CMPs or many-core systems are quite a big category. Table 1.1 shows the four classifications defined by Flynn (Flynn 1972).

**SISD**: The traditional Von Neumann architecture: a single instruction stream executed on single data.

**MISD**: Multiple instruction streams operating on the same data, a very rarely implemented architecture. This architecture can provide fault tolerance when accuracy and safety is the first priority.

**SIMD**: Single instruction operating on multiple data, GPU like architecture, good for vectorized computation.

**MIMD**: A number of processors that can function asynchronously and independently. Different processors may be executing different instructions on different pieces of data.

The concept of many-core system can only apply to **SIMD** or **MIMD**. **SIMD** is a very popular architecture and has been widely used by GPU and array processors. Its extraordinary ability in exploiting data parallelism makes it effective for graphic processing, but when talking about general purpose processors, it lacks the ability to handle heavy flow control tasks.
Chapter 1  Introduction and Background

Figure 1.1. Two commonly used multi core approaches

(a) Homogeneous CMPs  (b) Heterogeneous CMPs

So, in this thesis, we mainly focus on the last type of architecture, MIMD. However, MIMD is still a very large category. Unlike the traditional Von-Neumann architecture which has dominated the market for decades, many core architecture is still at the evolutionary stage and presents a huge design space. Two important factors characterise a particular design.

**Heterogeneous or Homogeneous**: whether or not the multi-core processor uses identical cores.

**Network On Chip**: How the cores are connected together.

Figure 1.1 shows the two options of how to use the available area. A heterogeneous chip may contain complex multi-issue super-scalar core and group of simple in-ordered RISC cores. It is more flexible, and especially good at multi-application environment or those applications that are not easily split into even parallel tasks.

However, a homogeneous design is easy to implement, places fewer requirements on the operating system (OS), and most importantly, is easy to expand by just replicating the identical cores. For the long term evolution goal, we believe here, homogeneous architecture will dominate the market in the future because of the growing cost of chip design and verification which will make heterogeneous architecture very expensive to design and manufacture.
There are quite a few researchers now proposing dynamic adaptable architectures which can dynamically reform architecture depending on the run-time workload (Gibson and Wood 2010) (Kim et al. 2007). This does look like a future trend.

Figures 1.2 1.3 1.4 and 1.5 show some of the well known multi-computer network topologies. A Network topology is defined as the form of how each node connects with other nodes or other endpoints.

**Ring** shown at Figure 1.2: Each node in ring network is connected with two neighbours and the first node connected with the last node. It is easy to implement but the average latency (node hop) is proportional to the number of cores. Beside that, traffic path overlap can lead to very serious path contention.

**Symmetric** shown at Figure 1.3: A Symmetric network looks like a binary tree. The top node acts as root and the bottom node acts as leaves. Each leave can also be the root of their children. Each sub-root node can behave like a cluster, which is effective for small-scale memory-sharing multi-thread applications. However the lack of flexibility in hardware can make it very difficult to balance workload (Cybenko 1989).

**Bus** shown at Figure 1.4: Each node is connected to a centralised bus. Any memory transaction has to issue to the bus. The broadcast method of data transmission
can make it easier to implement cache coherency protocols, however the drawback about bus based topology is that it is hard to scale up due to the characteristic of centralised component. Each transaction has to be serialised on the bus and be processed one after another.

**Crossbar** shown at Figure 1.5: Each node is connected to a crossbar switch, an arbitrator will work out which two endpoints communicate. It can reduce the serialisation problem caused by centralised component by creating separate channels between endpoints. However, with the continually increasing number of cores, the size of the crossbar will increase quadratically and will become impractical to implement.

These topologies were first introduced for printed circuit board level interconnection of system components. When porting these topologies from board level to NoC level many new challenges arise. For CMPs, two things are particularly important, latency and scalability. For small-scale CMPs, ring, symmetric, crossbar and bus can work with reasonable performance, but they are not scalable due to their centralised components. A well accepted topology for network on chip nowadays is packet-switched NoCs. Each node is connected with a router and each router is connected to others in a 2D mesh fashion. There are many advantages of mesh topology shown in Figure 1.6. Simplicity is one of them. They are easy to implement and can operate with a simple x-y routing algorithm. More importantly the ability to scale up and lower average latency due to the higher degree of parallel communication are the main reasons why we use it as our base architecture.
1.2 Background: Tiled CMP Architectures and Their Cache

Based on the above analysis, a homogeneous and 2D mesh based network on chip topology will be used as our base architecture since for the reasons stated above we believe it will represent a mainstream of future many-core architecture. Figure 1.7 shows the structure of our base architecture. Each node is made up of a RISC core, a private L1 cache and a slice of L2 cache which are logically shared with all other slices on the CMP.

### 1.2.2 Cache Design Background

To cope with the rapid change in CMPs architecture, the cache needs some innovation. Actually, ever since the first generation of computers, there has been a mismatch between processor and memory. From the view of the programmer, an unlimited amount of fast memory is always desirable. However, fast and large presents a contradiction. The larger the amount of memory, the bigger the cell array. The bigger array means longer interconnection and longer delays caused by capacitance.

Fortunately, we can take advantage of locality of memory accesses. Temporal locality and spatial locality describe access patterns evident in the way most of programs access memory. Memory blocks are not accessed by the processor uniformly. Temporal
locality occurs in time, which means the block being used recently has a high probability of being used again. Spatial locality refers to adjacency of memory accesses, which means blocks resided near a recently used block have a higher possibility of being used again soon.

By leveraging memory locality, researchers proposed the use of memory hierarchy to overcome the tension between fast access speed and large storage volume. Shown in Figure 1.8, memory hierarchy is composed of different layers of memory, each layer represents a certain level of speed and capacity trade off. Data blocks with the higher possibility to be used will be stored at higher level. Level 1 cache is the highest level in the cache hierarchy and is closest to the core. Level 2 and subsequent levels of cache are progressively further from cores. Research about how to obtain lower memory latency and higher cache hit rate has been ongoing ever since the concept of cache hierarchy has been proposed. This thesis continues that research.

**Miss Types**

Traditional cache misses can be categorised into three types, compulsory cache misses, capacity cache misses and conflict cache misses.

**Compulsory cache misses**: Cache misses occur the first time a block is brought into cache. These cache misses are called compulsory cache miss because we can not
avoid them. The total count of compulsory cache misses are only related to line size (size of cache block) and application set size (working set size) but some techniques can pre-fetch those compulsory blocks into the cache earlier. The cache miss count may be even larger but the cache miss penalty can be hidden.

**Capacity cache misses**: If the application’s working set size is larger than the cache size, capacity misses will happen. Previously fetched blocks will be evicted to make room for the new incoming one. This type of cache miss is mainly related to the application set size and the physical cache size. Replacement policy also plays a significant role here.

**Conflict cache misses**: Each level of the cache hierarchy should be able to map the entire memory space. The capacity of higher level is usually much smaller than that of the lower level memory because of the need to compromise size for lower access latency. Therefore multiple addresses on the lower level map to the same
location on higher level. This method is called direct map. If the processor requests more than one block mapped at different locations at lower level but these map to the same location at higher level, a conflict cache miss will occur. Set associative caches were proposed to reduce these type of misses. Figure 1.9 shows that compared with direct mapped cache, set associative cache can map a group of cache blocks from lower level to a small set rather than one location. Set associativity size represents the width of the set.

**Cache coherency and consistency**

In multi-core processors, when more than one processor is trying to read or write to the same location, coherency problems will arise. Figure 1.10 shows an example. Assume Core 1 and Core 2 are addressing Line 1 at the same time. Core 1 is an assignment and Core 2 is a conditional branch statement, but both of them need to load the value of X into their register first. Let’s say initially the value of X is 0. Both of the two processors would load the value of X as 0 into their private L1 cache and register. In Core 1, the next instruction would update the value of the register which hold X to 1 and store in back to memory. Unfortunately, in Core 2 the value of X remains 0 and the branch leads to the wrong direction.

In other words, cache incoherency arises when two cores have inconsistent copies of a variable stored in their caches.

The main reason for coherency issue is because of the existence of replica data. The modern CPU architecture is based on load-store. The way a processor operates is to load the data from memory and work on that data. Not until the data is stored back to the memory, can this process be considered to be finished. The same data can be
1.2 Background: Tiled CMP Architectures and Their Cache

![Diagram of Core A and Core B with Load X, Assign X with 1, Write X back to memory; Load X, Compare X with 0, Branch if True;]

**Figure 1.10.** Example of cache incoherency

loaded into several processors creating several replicas. If one of them updates to a new value, but the others do not, the processors will be using inconsistent values.

**Coherency Protocol**

To maintain a coherent system, the most important task is to ensure exclusive data upon write because only write to a replica causes incoherency. Migration or Invalidation are the two common ways to ensure this. Migration means moving the data from exclusive owner to the write initiator rather than making a copy. Invalidation means if more than one replica exists due to shared read, all of them need be invalidated prior to any data write.

A cache coherency protocol can be very different based on the type of CMPs and their interconnection manner. We will introduce the two most important protocols here.

**Snooping Protocol**: This protocol is commonly used with a centralised communication topology such as a bus. Each private cache is connected to the bus and a snooping request is sent to the bus. Upon data read, any other private cache holding the block will send a replica to the requestor. If no private cache holds that block, then the shared level 2 cache will send the replica. If any processor sends a write request to the bus, this request will be snooped by all of the private cache controllers. If the requested block exists at any private cache as a replica, it will be invalidated to ensure exclusive use by the time the write happened. The snooping protocol is easy to implement since it only involves few states.

**Directory Based Protocol**: Instead of snooping the network all the time, the directory based protocol uses a directory to record the state of every single block in the cache. For each block residing in cache, there is a state entry associated with it from the moment it is brought into the cache to the moment it is evicted by the
Table 1.2. Example of directory state entry

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Modified in one of the private caches</td>
</tr>
<tr>
<td>Invalided</td>
<td>Was cached as shared read but invalidated due to write by other core</td>
</tr>
<tr>
<td>Shared</td>
<td>Replica exists in several level 1 caches</td>
</tr>
<tr>
<td>NP</td>
<td>Not present in the cache</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Block exclusively owned by L2 cache</td>
</tr>
</tbody>
</table>

The state entry records information like whether the block is shared or exclusive, and if it is shared which cores hold replicas. Table 1.2 list a few of the commonly used states. Directory based protocols can cope with a non-centralised network (network without centralised hardware component) very well.

However, as we mentioned before, a bus based multi-processor can not scale well due to the characteristics of the bus. Memory transactions can not be transmitted in parallel which makes the memory access latency unacceptable. A bus has the advantage of keeping the cache coherent but will dramatically slow down the processing speed.

Without a bus, cache writing becomes complicated, one needs to make sure the block is not cached by any other processors before writing to it. Enquiring one by one is not feasible. A directory, which holds all the information, although residing far from the core, need only be interrogated once in order to learn everything about the cache.

Cache coherency protocols are a very broad topic and many researchers are working on improved protocols. We will not discuss the trade-off in detail here. However, coherency protocols introduce another type of cache miss which is the miss caused when a block gets flushed from the cache due to coherency issue initiated by other cores and gets retrieved again.

*Coherency Cache Miss* together with the other three “C”, *Complusory Cache Miss* Capacity *Cache Miss* and *Conflict Cache Miss*, represent the cache misses we trying to eliminate through this thesis.

### 1.2.3 Non-Uniform Cache Architecture

The cache is a critical component determining the performance of a system, presenting a vast array of design options. Over the period of a decade the size of the cache increased from less than 1 Mega Byte (256KB L2Cache, first generation Pentium 4 -
Figure 1.11. Comparison between UCA with one bank and 64 cycle average access and NUCA with 16 banks

Willamette in year 2000) to more than 10 Mega Byte for the latest Intel i7 CPU. The traditional monolithic large on-chip uniform access latency cache architecture is no longer desirable. Shown in Figure 1.11(a), a traditional uniform cache uses identical access latency regardless of where the data is stored. In Figure 1.11(a), a transaction accessing a near location may only take 5 cycles, but a transaction accessing the farthest location may take 50 cycles. To ensure correct timing, a worst case which is 64 cycles is used as the uniform cache access latency in this example and hence the average access latency would become 64 cycles. The non-uniform cache architecture was proposed by (Kim et al. 2003). It was shown that NUCA shown as Figure 1.11(b) can dramatically reduce average on-chip access latency. Each small bank has an identical access latency and the data access latency is the latency to access that bank depending on its location plus the identical bank access latency. There are two main advantages with NUCA. First of all, NUCA uses different access latencies when accessing different banks rather than always using the worst case uniform latency. As a result, the average latency is lower. Secondly, it can provide a parallel access with the help of a virtual channel of the router and reducing traffic contention. This design not only reduces the average cache access latency but also ensures a lower cache access cost, since the dynamic power consumption per access is largely dependent on block size.

NUCA can work well with a tiled architecture. A tiled structure is a homogeneous block and so is NUCA. Each node contains a core, a cache bank and a router. Thus far
we have introduced the physical outlook of our tiled structure base architecture and why we using it.

**Private Cache or Shared Cache**

NUCA divides the monolithic uniform access last level cache into pieces (called cache slice) to achieve lower average access latency as we explained before. It is obvious that those pieces are physically separated, but what about their memory space? Should they share the same memory space or they should be logically separated as well? This question is the private or shared question.

Shown at Figure 1.12(a), a private cache treats each last level cache (LLC) slice as an independent map of the entire memory space. Each cache slice is logically separated with others. If a cache miss occurs, the request will be directed to main memory and a directory is kept there to ensure everything is coherent. Shared cache treats each last level cache slice as a segment of the cache, a block is mapped to one slice depending on the physical address. The choice between private and shared LLC is the choice between lower average access latency and higher hit rate. We will elaborate the advantages and the disadvantages of each of them.

![Diagram of Private Cache and Shared Cache](image-url)
1.2 Background: Tiled CMP Architectures and Their Cache

Private Cache Advantages:
- Lower average access latency
- Lower network congestion
- Lower inter-thread interference

Private Cache Disadvantages:
- Higher cache miss rate
- More complex coherency protocol
- Unbalanced workload
- Slower inter-core communication

In a private cache, data blocks are kept close to the core. That is unlike shared cache in which the requested block may reside in the cache slice on the farthest tile from the requesting core, since data is mapped based on the address. A private cache will either find the data on local tile or have to issue a cache request to the memory. For small scale CMPs, this advantage is not very significant, but when scaling up to large systems like 64 cores or 128 cores, the difference of average access latency is very dramatic. The probability that data is mapped to a particular tile is uniformly distributed. It is very likely that most of the data a core requested is mapped at a remote tile in a shared cache. The bigger the scale of the CMPs, the larger the number of hops to reach the data hence the larger the average last level cache access latency will be. However, average access latency is calculated by equation 1.1 depending on miss rate.

\[
\text{Average access latency} = \text{Average Hit latency} \times \text{Hit Rate} + \text{Average Miss Penalty} \times \text{Miss Rate} \tag{1.1}
\]

The statement that private LLC has lower latency is based on the assumption of a similar miss rate.

For a private cache, the average hit latency equals the latency to access the local cache slice. It is a constant value only depending on slice size and implementation technology. Whereas for shared cache, the average hit latency depends on the size of the network and the average latency per hop among the nodes of the network on chip. Certainly it is much larger than the private access latency, but since private cache treats each slice as a mapping to the entire memory space, from the view of each core the capacity of the last level cache is only \(\frac{1}{\text{Number of Cores}}\) compared to shared cache. If one
core is running an application with a huge working set size and the cache on the local tile is too small, there is no way to take advantage of the free memory on adjacent tiles. This is why the miss rate is higher. Also, multiple replicas will be created and stored on the last level cache when multiple cores are sharing a block simultaneously. As a result, the effective cache capacity will be decreased and incur an even higher cache miss rate.

The reason there is a more complex coherent protocol and slower inter-core communication than shared cache is because a private cache needs to interrogate the directory each time before writing to cache. For a shared cache, if it gets a L2 cache miss then the incoming block from memory is exclusive. If that exclusive block is hit by another processor, then that processor will know which private L1 holds a replica, hence invalidation and block forwarding can be achieved fairly easily. For a private cache, it is a different story, if it gets a miss at L2 cache then the request shall be redirected to the directory to check if the block resides at any other L2 cache first. The data block will be sent to the requestor if no one else holds it, otherwise invalidations will be sent to other holders from the directory and the transaction has to wait until all invalidation confirmations have been received or necessary write backs finished.

**Shared Cache Advantages:**
- Lower miss rate due to resource sharing and balanced workload
- Less complex coherence protocol and cache level inter-core communication

**Shared Cache Disadvantages:**
- More inter-thread interference
- Higher average L2 access latency

Inter-thread interference is the most troublesome issue for shared cache. Figure 1.13 shows how a physical address is decomposed to access data in a cache block. The last section is the block offset, for example in this thesis the block size is always set to be 64 Bytes, so the last 6 bits from the address are used to index a particular byte from a block. The following two sections determine how an address is hashed to a set. The width is decided by the number of sets per L2 slice and the total number of L2 slices. In this thesis, most of the simulations are based on the configuration of 16 L2 slices and
1024 sets per L2 slice. The width of set index and core index is 10 and 4 according to equations 1.2 and 1.3.

\[
\text{Width of Set Index} = \log_2 \text{Total number of Sets per L2}
\]

\[
\text{Width of Core Index} = \log_2 \text{Total number of Cores}
\]

If multiple threads are running on different cores but sharing the same L2 cache, as long as two addresses are identical in the above mentioned 14 bits then those addresses will be mapped to the same cache set. Inter-thread interference occurs when a thread reads and stores data to a cache set and replaces the data brought by other threads when that data is still needed in future. Many studies have focused on how to eliminate this issue and we will introduce some of them later in the literature review section.

There is no conclusive research to show that shared cache is better than private cache or vice versa. My research can actually be based on either of them. However, I choose shared cache as the baseline because I believe shared cache will be the future trend especially in terms of large scale many core systems due to the following reasons. Interconnection between processors and main memory is based on metal wire which restricts the potential of having too many memory controllers due to limit of number of pins. So the bottleneck between processor and memory will always be the limitation due to this conventional electrical interconnection. Cache miss penalties will increase because of the contention raised at processor-memory interconnection. Inter-core communication will play a more and more important role in future CMPs. Difficulty in
parallelising a program in such a way as to spread the computational load evenly between cores will cause unbalanced workloads. Coherence protocols are too expensive to scale up. All these compelling reasons lead us to advocate shared cache.

### 1.2.4 Power Dissipation Model and Simulation Method

In this section, we will briefly introduce some metrics which are used to compare different architecture designs, the power model we are using and some common power simulation methods.

**Power Dissipation**

Power dissipation was dramatically reduced in the last electronics industry revolution with the introduction and development of CMOS transistors. For many years power dissipation was the last design constraint to be considered. Area, and delay were the leading constraints at that time.

Averaged power dissipation of a circuit is the total energy over time shown in equation 1.4.

\[
P_{avg} = \frac{1}{T} \int_{0}^{T} i(t)V_{DD}dt
\]  

Total energy is the integration of instantaneous power. For any moment, the instantaneous power of a circuit is the product of instantaneous current and voltage. For CMOS circuits, \(V_{DD}\) is the supply voltage which is assumed to be a constant value while the current flow from \(V_{DD}\) to ground is variable, \(i(t)\) is the transient current and \(T\) is the total time.

Current flow through a transistor only has two components and we split power dissipation into: static power (or leakage power) and dynamic power (operating power).

**Static dissipation components:**
- Leakage reverse-biased diodes or sub-threshold conduction
- Leakage through gate oxide
1.2 Background: Tiled CMP Architectures and Their Cache

Dynamic dissipation components:

- When switching, both N and P transistors are partially on
- Charging and discharging of load capacitances

Power Metric

In power studies, different metrics are used depending on the goals of the research. We will discuss energy, power and energy-delay-product here as we use these to evaluate our design later.

Power in the unit of Watts, is the rate of consumption of energy. Average power over a given period of execution is given by equation 1.4, but at any time instant there is also peak power which is used to describe the maximum power during execution. Power and Peak power are often used to evaluate whether a design is affordable and is often related to thermal design constraints.

Energy in the unit of Joules, is often used as the measure of a computing platform’s performance because it reflects the total cost of performing a given computation, as reflected for example, in electricity charges in a data center, or the size of a battery that is required to support the computation in a portable platform.

Energy-delay-product in the unit of Joules × Second is the combination of energy saving and performance loss. It was proposed in (Horowitz et al. 1994) for the purpose of low power design evaluation. It is defined as the product of the energy consumed in performing a computation and the time taken to perform the computation. It is understandable that if we slow down the circuit, the number of switching actions in the circuit will be reduced in the same time interval hence consuming less power. Either a cool and slow circuit or hot and fast circuit is not what we desire. Power-delay product (PDP) in the unit of Joules, is essentially the same as energy. It reflects the total energy expended on a computation but does not reflect the time taken for the execution. On the other hand EDP can help us to verify whether it is worthwhile to consume more power to achieve a faster computation or sacrifice computation speed in order to save power.
Power simulation

The most obvious way of doing power simulation is to synthesise the design to gate level and work out total capacitance and then run a circuit level simulation to accumulate energy consumption using the integral of voltage-current product. However this method is nearly impossible when exploring the early design space, because the simulation time would be prohibitive and it is usually not feasible to synthesise to gate level in early design stages.

From the late 1990s, architecture-level power estimation began to dominate architecture research. Cacti (Muralimanohar et al. 2009), Wattch (Brooks et al. 2000), and the SimpleScalar tool set (Burger and Austin 1997) were proposed and are all being widely used. These tools use the philosophy of divide and conquer: they have provided a power model for different components and different activities. By monitoring and counting transactions between different subcomponents, an overall power estimation can be generated. Cacti was developed to study memory hierarchies initially, but network on chip latency and power models were added later. Wattch and Simplepower are more focused on out-of-order super-scalar pipelines.

The power-tool we use in this thesis is McPat (Li et al. 2009) which is the first integrated power, area, and timing modelling framework for multithreaded and multicore/manycore processors. It is designed to work with a variety of processor performance simulators, GEMS (Martin et al. 2005), which act as a performance simulator and can pass dynamic activity statistics generated during run time to McPat, as shown.
1.3 Related Work

in Figure 1.14. It works in a similar way to Wattch but with better interface reconfigurability. McPat can not provide us an accurate power consumption since it is an architecture level simulator, but as an architecture level simulator, McPat can provide a power estimate that is sufficiently accurate for our research, which is aimed at early design space exploration.

1.3 Related Work

There have been many prior studies on cache hierarchy design for high performance and low power. In this section, we will briefly introduce some of the most closely related studies and provide a literature review in the area of our research.

1.3.1 Power Saving Cache

Drowsy Cache and Cache Decay

In (Albonesi 1999), Albonesi et al. proposed to shut down ways in a set associative cache in order to save power until the application reached cache-intensive periods. The application would need to be profiled first and during the execution special instructions would change the value of a cache way select register to tune the size of cache. However this technique is applied to a uni-processor system and only focuses on L1 cache. Another popular technique is called the drowsy cache (Flautner et al. 2002) (Kim et al. 2004) (Hanson et al. 2001). The idea here is to apply multiple supply voltages. When the cache is not frequently used, the supply voltage will drop. The advantage of this technique is that it can save a large amount of leakage power without losing the data. Many policies for placing a cache set into drowsy mode have been studied (Meng et al. 2005). However, the future development of silicon circuits will show continuing reduction in $V_{dd}$ which will cause both a diminishing return from the drowsy cache and a higher soft error rate (Li et al. 2004) (Yoon and Erez 2009).

Part of our work in this thesis should be classified as a type of cache decay (Kaxiras et al. 2001a). The concept of cache decay is to turn off cache lines whose content is not likely to be reused. (Abella et al. 2005) (Khan et al. 2010) (Liu et al. 2008) present other approaches of this type. However other cache decay approaches involve complicated inactivity prediction and significant hardware overhead. They either need an off-line profile or complicated on-line information collection.
Chapter 1  
Introduction and Background

Cache Way Partition For Power Saving

In (Suh et al. 2002), (Suh et al. 2004), Suh et al. first proposed to investigate the recency position of a set to evaluate the utility of a cache. Qureshi et al (Qureshi and Patt 2006), extended this idea by separating the monitoring circuit in order to provide utility information that is independent of other concurrently executing tasks. In (Reddy and Petrov 2010), Reddy et al. proposed to partition the cache according to a profile-based, off-line algorithm. The operating system will configure the date cache configuration during a context switch. Unnecessary cache resources can be shut down at this time to save power. However all of these methods focus on multi-task execution. Isolating the cache in a multi-threading environment will incur difficulty in sharing and communication and the benefits of implementing synchronization at the last level cache will not be as great.

In (Iyer et al. 2007) and (Chang and Sohi 2007), QoS (Quality of Service) can provide a thread with guaranteed baseline performance regardless of its individual workload, hence the performance was improved, however, these two designs involve complicated QoS calculation which need the help of software. Compared with these methods, our design can tune the size in a very simple autonomous fashion. Kotera et al. in (Kotera et al. 2011) leverage stack distance to evaluate the utility of two programs sharing a single L2 cache and generate a virtual partition between them. Under-utilized ways can be shut down to save power. However their method uses the ratio between the number of LRU (Least Recently Used) hits and MRU (Most Recently Used) hits to trigger a way allocation, which requires the use of complicated dividers and comparators. More importantly, the rationale behind their utility evaluation was not elaborated. There is no analytical support for their estimation method. The ratio between LRU and MRU maybe good enough for partitioning cache between two applications running on a shared cache, but for the purpose of improving energy-delay product dynamically for general purpose CMPs it was not enough. Our work in Chapter 2 on the other hand focuses on a different view, multi-threading applications with at least 16 cores sharing a distributed tiled cache.

Another paper (Sundararajan et al. 2012) proposed a cooperative partitioning method to lower both dynamic and static power, where each cache way is assigned to a particular core depending on their cooperative partition algorithm. Unused ways can be power-gated for static power saving which is similar with our approach, however, again the rationale behind their static power saving was not clear. Way takeover by
1.3 Related Work

A core was decided by a predefined takeover threshold which can vary significantly from application to application. Besides that, their approach needs much more complicated monitoring and partition logic to work out how the cache resources should be reallocated among different applications.

1.3.2 Novel Cache Replacement Policy

Replacement Policy: Dead Block Prediction

The idea of dead block prediction is to mark a block as Dead if it will not be re-referenced again before it is evicted from the set (Kaxiras et al. 2001b) (Liu et al. 2008) (Kharbutli and Solihin 2008) (Khan et al. 2010) (Abella et al. 2005) (Wu and Martonosi 2011). Dead block based replacement policies use access history patterns or other information to predict whether a block is dead or not and evict those that are already dead to make room for those that are alive. The method used to predict when a block is dead can vary, but the basic idea is similar in each case: a threshold is dynamically determined and if a block has been staying in the cache longer than the threshold without being re-referenced, it will be marked as Dead. These policies require significant hardware overhead, particularly for threshold tuning.

Replacement Policy in Page-Domain

The idea of combining LRU and LFU (Least Frequently Used) was suggested a long time ago (O’Neil et al. 1993) (Lee et al. 2001) (Robinson and Devarakonda 1990), but this work was aimed at page level replacement in virtual memory systems and involved hardware overhead and power consumption that is unaffordable in the context of a cache system. Dynamic adaptive hybrid caches (Subramanian et al. 2006) (Megiddo and Modha 2004) (Bansal and Modha 2004) can adapt themselves during simulation by maintaining extra information and tuning while running. Page level replacement policies have a much larger design space than block level policies, because they work at the level of main memory, which is much larger than a CPU cache. Moreover, a page miss incurs a much higher penalty than a block miss. Sophisticated page-level replacement policies are not feasible at the block level because of low levels of improvement for relatively high cost.
Chapter 1 Introduction and Background

Replacement Policy: Re-reference Distance Prediction

Distance Prediction (DIP) is a policy that combines LRU and LFU at the block level with feasible hardware overhead (Qureshi *et al.* 2007). DIP divides the replacement policy into two fields: victim selection policy and insertion policy. It is based on LRU, which predicts the victim, and uses set duelling to decide the insertion strategy, whether to insert a new incoming block into the Least Recent Position or the Most Recent Position. It can provide thrashing resistance but requires real LRU block identification as foundation, which is very expensive to implement.

Re-reference Interval Prediction (RRIP) (Jaleel *et al.* 2010) uses less hardware than real LRU and provides thrashing and scan resistance by placing an incoming line at a position where it might be evicted easily. If a line is not re-referenced within a certain time, it will be moved to the eviction region. While this work is the most similar to our EBR, our design can dynamically change the time threshold before blocks are moved to the eviction region by tuning the extent to which the policy relies on LRU or LFU. Re-use Distance Prediction (Keramidas *et al.* 2007) (Manikantan *et al.* 2011), uses a similar idea. The distance of reuse is worked out and a PC-indexed predictor is used to optimise replacement. The aim is to find a small number of instructions that are responsible for most of the cache misses and try to give them preferential treatment.

Replacement Policy: Others

MLP (Memory Level Parallelism)-Aware Cache replacement (Qureshi *et al.* 2006) identifies the different cost of cache misses. It is postulated that this cost can vary because memory level parallelism varies, hence they use a run-time miss-cost calculator to implement a cost-sensitive cache replacement policy. Pseudo-LIFO (Chaudhuri 2009) abandons the idea of the recency stack (a hypothesis stack keeps the recency of each block reference) and chooses LIFO as the substitute. Re-reference probabilities are dynamically learned during the running of a program and victims are chosen from the upper portion of the LIFO stack. They claim that LIFO is a basis for a new set of replacement policies, however LIFO requires a complex dynamic learning algorithm and additional hardware.

Many other policies exist, however they either require significantly large hardware overhead or dramatically complicate the replacement procedure. In contrast, our D-EBR policy to be described in Chapter 3 provides an easily implementable hybrid LRU/LFU scheme.
1.3 Related Work

1.3.3 Cache Access Latency Reduction

In their paper (Beckmann and Wood 2004), Bradford et al demonstrated the new challenges presented by CMPs and analysed the challenges of incorporating three latency control methods, popular in uniprocessor designs: block migration (migrate a block towards the requester depending on its usage), transmission line (use transmission line technology to access large cache) and stride-based hardware pre-fetching (pre-fetching with an offset decided by memory access pattern). They presented a hybrid design which combined all three techniques and achieved an additional 2-19% average performance increase over CMPs without latency control technologies. One of the most useful contributions of this paper was that they verified and demonstrated that the block migration proposed in (Kim et al. 2003) or (Kim et al. 2002) is less effective for CMPs with many cores because most of the cache access blocks in commercial workloads are shared blocks, and those blocks will end up located at the central slices, which are equally far from all processors.

With the common use of NUCA in CMP design and the failure of traditional cache access latency management technology, the question of how to organise the last level cache became more and more compelling. Liu et al comprehensively elaborated two obvious alternatives (Liu et al. 2004), Private and Shared. The advantages and disadvantages of each method and the conflicts between them were clarified. Finally, a novel architecture that captures the benefits of both organizations was proposed, Shared Processor-Based Split L2. The idea is to maintain a table for every core to indicate certain slices that it can use. For cores running heavy workloads, more slices will be assigned. Upon a L1 miss, these assigned slices will be searched in parallel. But this design needs a bus based network so it can broadcast requests and keep coherence. Moreover the parallel lookup is very power consuming.

With tiled structures being more and more popular, many researchers started to turn their focus from bus structure to tiled structure and how to effectively organise their last level cache. Jaehyuk Huh et al defined a concept called sharing degree (SD) (Huh et al. 2005), which specified the number of processors that share a pool of L2 cache slices. An SD equal to 1 means each core maps the whole memory space to its own L2 slices, and it is the same as a private cache scheme. An SD equal to the number of total cores means all the cores share the entire L2 cache system, and it is the same as a shared cache scheme. The idea of cache clustering is to set the degree of sharing at L2 in between private and shared. For different applications, an optimal sharing degree can
be decided through static evaluation. Mohammad Hammoud et al extended this work with the idea of dynamic re-clustering (Hammoud et al. 2009b). They believe that even the same application would encounter different phases, so the cache demand of each core is varying. DCC (Dynamic re-clustering) proposed a novel mapping strategy, a location strategy and an algorithm to realise re-clustering at a certain time intervals. Unfortunately, DCC relys on very expensive lookup and coherence mechanisms which incurs not only hardware overhead but also high power and traffic cost upon searching.

The idea of splitting the cache slice into private and shared regions was advocated by Zhao et al (Zhao et al. 2008). Upon a memory access, the fetched block will be placed at the private part of the requesting tile to assure fast local access. If the block is accessed by another processor later, it will be marked as shared and pulled out from the private part and evicted back to the shared region of its home tile. A novel search method was proposed: upon a L1 miss, the local private region was searched, if it is a miss, the shared region and the directory of the blocks home tile was searched. The directory would show the information on which tile keeps a copy of the requesting block. Different implementation options were given: some take use of the local shared region as victim cache (which is used to store the recently evicted blocks) while some swap the block if the access is to its own shared region. The evaluation showed the split cache can decrease the average access latency noticeably while keeping the miss rate close to that of a shared cache. Similarly, Merino et al proposed SP-NUCA in (Merino et al. 2008). They apply the same concept to partition the last level cache slice into private and shared. But they proposed a more detailed dynamic partitioning policy and replacement policy. Their evaluation was conducted through an event driven real performance simulation, and the result showed a 16% and 6% improvement over a static NUCA and idealized dynamic NUCA respectively. (Dybdahl and Stenstrom 2007) proposed a similar idea. Two years later, the same group of researchers further proposed ESP-NUCA (Enhanced SP) (Merino et al. 2010). To the private and shared regions, they added an extra group labelled as the helping blocks region. Replicas and victims were placed to help blocking regions to speed up the average access time.

A shortcoming of these various splitting approaches is the lack of power consideration. None of these studies took power into consideration. We are very inspired by
1.3 Related Work

their work, however, they all lack an explicit dynamic tuning scheme. Their evaluation bases are expensive (shadow tags etc) and their replacement policies cannot be partitioned.

Since the cache hit latency heavily depends on the physical distance between the requesting core and the cache slice where the data is placed, all of the schemes aimed at placing the blocks close to the requesting core. As we know the location to which a block should map is decided by its physical address. Most of the technologies we mentioned above try to remap or reroute the block from its original location to a location close to the requestor. Chishti et al proposed NuRAPID to decouple Tag and Data by using a pointer to link them so as to add flexibility (Chishti et al. 2005). However the decoupling depends on very complex and expensive lookup mechanisms. Hardavellas et al observed that the cache access patterns can be classified into only 3 types: private data, shared data, and instructions (Hardavellas et al. 2009). They carefully analysed each of the characteristics of each of the three types and proposed different placement policies for each of them. They proposed R-NUCA, which can dynamically identify the type of the block and place them according to the corresponding placement policy. They do the classification at page level. The page is labelled as either private or shared upon a TLB (translation lookaside table) miss. Once the page is labelled, all the blocks in the same page are classified as the labelled type.

With the failure to take advantage of data migration in non-uniform cache architecture of CMPs, many alternative methods were proposed to arrange the last level cache. Zhang et al proposed victim replication (Zhang and Asanovic 2005). They believe that those victims evicted from the local cache slice should be kept as replicas locally. They place L1 victims as replicas at local L2 with a replacement policy in the priority as follows:

1. Do not replace a shared L2 in favour of a L1 replica
2. Replace invalid L2 block
3. Replace a globe block with no share
4. Replace an existing replica

The problem of this design is that keeping L2 at a higher priority today can not always guarantee better performance. Kandemir et al. continuously modify the migration
mechanism by trying to select the most appropriate locations to migrate (Kandemir et al. 2008). Hammoud et al. further explore migration by predicting the optimal location by monitoring the behaviour of the program (Hammoud et al. 2009a). Muralimanohar et al. proposed a novel NUCA based architecture (Muralimanohar and Balasubramonian 2007). They advocate building a 2D mesh using two different interconnects, one to achieve lower latency, one to achieve higher bandwidth.

(Beckmann et al. 2006) proposed an adaptive selective replication method which is similar to our research, in that it can evaluate the cost and gain of replication, however, their evaluation method involves a complicated Tag buffer and lack of adjustable replacement policy.

In (Kandemir et al. 2004), the authors proposed a method to dynamically configure a storage system based on demand by analysing the application at compile time. In (Jin et al. 2006), a flexible cache mapping method was proposed to improve performance and lower power. These designs either need special profiling at compile time or need special OS level activity. Hammed (Hameed et al. 2012) proposed an adaptive cache management scheme allowing cores to steal cache capacity from remote partitions to achieve better resource utilisation. This type of reconfigurable cache was mainly focused on cache partitioning to isolate interference or lower average access latency. None of the work was aiming at lowering energy-delay product or providing a flexible scheme which allows the user to choose between performance oriented or energy oriented optimisation within the same energy-delay product.

### 1.4 Thesis overview: Significance and Contributions

This section provides an overview of the thesis. The significance and main contributions are listed as follows:

**Utility Based Cache Resizing**: Chapter 2 introduces the importance of leakage power consumption and the disadvantage of a fixed cache system used for different applications. To address that problem, we propose a cache resize scheme that can tune the size of cache based on cache utility of the running application. Different from any prior research, our scheme focuses on reducing static power while
maintaining as much performance as possible. Reduction of energy delay product is our main objective. Our advantages also include easy to implement hardware, energy saving orientation and no need for additional software or OS activities.

**Effectiveness Based Adaptive Cache Replacement Policy:** Next, in Chapter 3, in order to mitigate the disadvantage that real LRU is almost impossible to implement, we further propose an alternative cache replacement policy called EBR. EBR is realistic in terms of hardware overhead which makes it possible for utility based cache resizing. Also, EBR can alleviate cache interference from different cores and cache thrashing caused by a large working set size. Our simulation shows that applications suffering from those two phenomena can significantly improve their performance.

**Cache Dynamical Resize using EBR based Cache Utility Evaluation:** In Chapter 4, we further propose how to estimate cache utility of the application when it is running with EBR replacement policy. We show that substituting LRU with EBR is feasible. The simulation results show similar leakage power saving with even higher performance.

**EBR Based Dynamic Private and Shared Cache Partition:** Following that, in Chapter 5, we propose to set up a private replica region on each tile. Its primary intention is to reduce the effects of increasing average cache access latency caused by the ever increasing global wire delay when scaling up the number of cores. While data replicas stored near the requestor can significantly reduce average access latency, they also reduce effective storage size and cause higher cache miss rate. We leverage our EBR based cache utility estimation method to dynamically tune the partition based on cache access patterns to achieve a near optimal result.

Finally, in Chapter 6, we conclude the thesis and provide several recommendations for future study.
In this chapter, we focus on exploring methods to reduce the size of a running cache when it is not being used effectively and try to keep the penalty as low as possible. In the tiled structure we briefly introduced in the last chapter, cache consumes a large fraction of on-die transistors and will consume more when the chip scales up. Leakage power will become dominant in the near future. We propose a cache resizing mechanism which can automatically tune the cache during run time by shutting down under-utilised cache regions to save power.
2.1 Introduction

In future Chip Multi-Processors (CMPs) more than half of the transistors will be used as on-die memory cells because of both the huge cache demand and the fact that lower power density of memory (compared to logic) provides an easy way to control total power (ITRS:2013 Executive Summary 2013) (Asanovic et al. 2009). Unfortunately, when we scale down supply voltage to reduce dynamic power, the threshold voltage scales with it, leading to a much higher leakage power dissipation. Hence, the technology trend is towards CMPs with more than half of their transistors dedicated to cache and with much higher leakage power dissipation (Borkar 2007) (Brodersen et al. 2002) (Huh et al. 2001). Therefore, it is necessary to develop power management strategies to shut down cache blocks that are not being used.

Different applications have different cache demands because of different working set sizes and data structures. If we adopt a single cache structure, it will be hard to achieve both low power and high performance across a wide range of applications. However if we can dynamically identify when the cache is not being used productively and shut it down we can save static power. This means not only shutting down those cache blocks that are not being used but also those that are in use but are not significantly beneficial to performance. In (Qureshi and Patt 2006), Qureshi et al proposed that the benefit that an application obtains from cache resources might not correspond to its cache demand. In other words, an application may have a high cache demand but devoting a large amount of cache to it will not necessarily improve its performance because the application might never reuse many of the cache blocks it has occupied. Streaming applications behave like this.

In this chapter, we propose a utility-based bi-directional cache resize mechanism to lower cache leakage power with only minor performance degradation. Firstly, we propose a concept that we have called the Break Even number of Misses (BEM). BEM is a quantitative estimated trade off between increasing the number of last level cache misses and reducing leakage power by shutting down cache ways. It can be understood either as the maximum number of extra misses that would occur if we shut off one cache way or as the minimum number of misses that can be reduced if an extra way were allocated. BEM gives an indication of how aggressively we can save power by shutting down ways without increasing the overall EDP due to a larger number of misses. BEM is highly dependent on the configuration and characteristics of applications. In this chapter we propose a method for estimating BEM.
Secondly, we propose a cache increasing-only method. We tested the method with 8 applications from PARSEC 2 (Bienia et al. 2008) and all of the applications from SPEC CPU 2006. Our simulation results show significant static power and overall energy delay product (EDP) reduction. We proceed to discuss the overhead of down-sizing the cache and demonstrate a general level of workload that this may incur. We then propose how to efficiently reduce the size of the cache during execution. With the help of this bi-directional cache resizing, we can further cut the static power with similar performance loss and produce EDP reduction for single task applications and further lower the average EDP for multi-threaded applications.

This chapter is organized as follows. Section 2.2 discusses the power breakdown CMPs. Section 2.3 explains what is the utility of a cache block and verifies its relationship with execution performance. Section 2.4 explains how stack distance can be used to evaluate the utility of the cache. We detail in Section 2.5 the methodology, algorithm and results of the increase-only tuning scheme. Section 2.6 talks about the algorithm and evaluation of bi-directional cache resizing. Section 2.7 explores how this scheme works at a tile level tuning granularity. Finally we will summarize in Section 2.8.

2.2 Overall Power Analyses

To better understanding how energy is allotted for future CMP architectures, we created a model of an Opal-based 16-core system (Martin et al. 2005) which operates at 3800MHz on a 22 nm technology and inherits a similar logic and memory ratio to that of a widely used commercial CPU, Niagara2. Figure 2.1 shows the overall power breakdown of our virtual CPU and its comparison against the McPAT modelled data of Niagara2 (Li et al. 2009). In each case the level 2 (L2) cache accounts for nearly a quarter of the power consumption. Most importantly, cache probably has the highest potential in terms of power saving if compared with the other major components in the system. We also note that even though the above two devices share the similar logic to memory ratios and similar L2 leakage power ratio, since the over all power envelope increased from 76 Watt at 65 nm to 113 Watt at 22 nm, the absolute power that can be saved by making the L2 cache more efficient is likely to increase as time goes on.
2.3 Utility of the Cache Block

The function of a cache is to keep recently used data close to processors and to take advantage of the data block’s temporal or spatial locality. When these blocks are re-referenced, efforts to re-fetch from memory can be saved. If the application has a large memory footprint, correspondingly it will consume a large proportion of the cache.

Are those consumed cache resources working productively? It depends on how often cached blocks are re-referenced. For example, streaming applications can write to a large number of unique cache blocks but these blocks are hardly used again. We say the cache blocks for streaming application have a low utility and there is no benefit in keeping those blocks. As a result we consume more power to store them, but the miss rate remains the same. When we decide not to keep those blocks, we can shut down or reduce the size of the cache because it will make no difference to the overall performance but will lower static power.

Figure 2.2 shows cache utility during the run time of Libquantum (a test application from SPEC CPU2006) under the uni-core configuration defined at Table 2.3. For the entire execution, this application has committed 1 billion instructions and encountered 32 million L2 Cache misses. In other words, 32 million blocks were brought on chip during the execution. However only 20 thousand of them ever got a chance to be reused. The graphic shows that 99.9% performance can be guaranteed with only 1/8 of the original cache size (two ways out of 16 ways).
2.4 Utility-Based Monitoring

2.4.1 Stack Distance

In a traditional memory system, the size of a cache is determined by its set size, line size and associativity. Conflict misses can be reduced by using higher associativity. However if an application’s working set can fit entirely into the cache, or if there are few conflict misses, large amounts of on-die memory resources will be wasted. For a traditional L2 cache, four way associative caches are enough for most single thread applications (Sun Microsystems 2007). In the future, more multi-threading support will be required for parallel processing. Anticipating future systems with more than 100 cores, it will be necessary for the last level cache to have adequate associativity to minimise conflict misses.

The method of utility-based monitoring was first proposed by Qureshi et al (Qureshi and Patt 2006). They use this method to partition the cache when different applications are sharing the cache on the same die. In (Cascaval and Padua 2003) the authors proposed the stack distance metric, which can be used to evaluate cache utility. The idea of
2.4 Utility-Based Monitoring

<table>
<thead>
<tr>
<th>Recency Position</th>
<th>Number of Hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRU</td>
<td>300</td>
</tr>
<tr>
<td>2nd MRU</td>
<td>150</td>
</tr>
<tr>
<td>3rd MRU</td>
<td>100</td>
</tr>
<tr>
<td>4th</td>
<td>100</td>
</tr>
<tr>
<td>5th</td>
<td>50</td>
</tr>
<tr>
<td>6th</td>
<td>50</td>
</tr>
<tr>
<td>7th</td>
<td>50</td>
</tr>
<tr>
<td>8th</td>
<td>50</td>
</tr>
<tr>
<td>Misses</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 2.1. Recency position distribution upon hits

stack distance is to consider each set of a hypothetical cache with an infinite associativity as a recency stack. When a data block in a cache set is re-used, the LRU scheme will return a value representing its position (called the “recency position”) in a hypothetical list of all of the blocks in the set, sorted by their last time of use. That value is defined as the stack distance. If we set the associativity of the cache to A, then any references to blocks whose stack distance is less than A would result in a “hit” and any references to blocks whose stack distance is greater than A would result in a “miss”.

For the purpose of illustration, Table 2.1 shows hypothetical results after running a simulation with eight ways allocated. The number of cache hits at each recency position is recorded. The sum of these plus the number of misses would be exactly the total number of cache accesses to this tile. Figure 2.3 shows what would happen if fewer ways were allocated. For example, if we only release four ways at the beginning of the simulation then the total number of cache misses would be the original 150 misses plus the extra misses caused by shutting down 4 ways. Any original hit at stack distance 5, 6, 7 and 8 would become miss instead of hit. The same simulation would end up with 650 hits and 350 misses.

2.4.2 Performance vs Cache Resources

Different applications have different cache demands, however high cache demand is not equivalent to high utility. Figure 2.4 further shows the performance of various applications with different amounts of cache resource. In total there are 16 ways per
Most of these applications can lose at least a few ways without a significant performance loss, the exceptions being *canneal* and *ferret*. An asterisk and a diamond are placed to indicate where the performance loss is respectively 1% and 0.5%. In our simulation, *blackscholes*, *ferret* and *swaptions* have a very similar L2 demands since they share a similar L2 miss rate, but they have very different L2 utility. Shutting off one cache way would not cause any noticeable difference to *swaptions* and *blackscholes* but will slow down *ferret* by 3.5%.

Here we define *cache utility* as the number of reuses at the cache’s LRU recency position. We will use this definition to explain why we should not simply resize the cache according to cache demand. Cache demand is defined as the number of unique cache blocks accessed in a given interval (Denning 1968). Figure 2.5 shows the difference of the applications *ferret* and *blackscholes* in terms of cache utility. Both applications run with 16 ways, but the utility of *ferret* is more than a hundred times that of *blackscholes*. This difference explains the above phenomenon: why shutting off one way from *blackscholes* won’t affect anything whereas from *ferret* it will. By using cache utility rather than cache demand, we can easily determine whether we can produce a performance improvement if we allocate more cache resource. Also, we can determine whether we can shut down more cache resource without lowering the performance.
If we can sample the execution at small intervals then we can use current utility to predict the future utility.

## 2.5 Increasing Cache Algorithm

In this section, we will introduce a method to tune the size of cache based on the cache utility of running applications. In order not to change the current industry preferred cache system dramatically, we propose an algorithm that only involves cache size increasing. Before that, we will talk about how to estimate cache utility and the rationale behind it first.

### 2.5.1 Rationale of the Size-up Approach

Let $T_{\text{interval}}$ denote our sample interval, and $N$ denote the number of hits at the LR (least recency) position. We plan for $T_{\text{interval}}$ to be of the order of millions of cycles. If we shut down one way in the past $T_{\text{interval}}$, we would then have $N$ extra misses. We
use this as a basis for estimating future performance. We predict that if we allocate an extra way in the next $T_{interval}$, the number of misses we could reduce is similar to $N$.

The idea of our resizing algorithm is that we start execution with only a small part of the cache (that is, a small number of ways) activated. During execution, we frequently collect the utility of the cache and work out if it is worthwhile to allocate more cache resources. If the application shows low utility, no extra cache resources will be allocated since there is apparently little performance to be gained. The cache will remain at a small size and hence save leakage power. If the application shows a good utility, the number of cache ways will be increased, up to a maximum of 16 in our simulations, until the point when there is no performance benefit due to the extra cache resource allocated. We denote this position as the “stable point”. Figure 2.6 shows the overall architecture of our tuning framework.

Our resize algorithm is based on the following conditions. L2 cache is logically shared by all the cores but physically distributed amongst the 16 slices. When a program is pre-empted, the scheduler from the OS will assign a thread for each physical core. At each tile, the supply voltage to each cache way group is isolated using power-gating circuits. At the beginning of the application, the operating system will initialise the

![Figure 2.5. ferret VS blackscholes, number of hits at different stack distances](image-url)
2.5 Increasing Cache Algorithm

cache. It will guarantee that the cache is configured correctly and initialised, by which we mean that all the blocks in the cache will be written back if necessary due to the coherence issue, that only two ways are activated per set and all other ways are shut down using power gating.

A way-allocate event will be triggered under certain circumstances. A simple method is proposed to ascertain whether to trigger a way-allocation. We propose to simply use a binary counter to count the number of accesses to the LRU position. As we discussed before, the value of this counter can represent the utility of a particular cache. Once the value of that counter exceeds the threshold that we set, we consider that the application has poorly localised cache references. A bad localised cache reference means the ratio between reuses at LRU recency position and the total number of reuses is high. The system needs more resources. A way-allocate event will be triggered: a new empty way will be released, the current least recently used position then becomes the second least recently used position and the newly allocated way becomes the new LRU recency position.

As shown in Figure 2.6, for each tile one counter is used to record the number of accesses to the LRU position. The value of this LRU counter will then be compared with the threshold and a decision is made as to whether the current workload has a larger working set than the pre-set value. Once the counter’s value exceeds the threshold, a trigger signal is generated by the on/off controller. The allocate signal will inform the power gating system on the tile to release another way of the cache and also inform the replacement scheme to operate under the new configuration.

2.5.2 Concept of BEM

We borrow the concept of “Break Even Point” (BEP) from the prior work of Hu et al (Hu et al. 2004). BEP is the minimum duration (in cycles) for which an execution unit should be powered off in order to guarantee a net positive power saving. If a unit was powered off for less than BEP cycles, the power overhead caused by turning a circuit on and off would be even larger than the power that can be saved, leading to a negative total power saving. Since we are targeting a much higher level than Hu and the time interval between two tuning activities is multiple orders of magnitude higher, the circuit level power gating overhead can be considered to be negligible. We propose instead to use the Break Even number of Misses (BEM), which we define to mean the
minimum number of misses that should be saved by an extra cache way in order to produce an unchanged EDP (Energy Delay Product).

In the general case, if we use $N$, as previously defined, to predict the number of misses that can be saved in the next $T_{interval}$, it can only serve as a broad estimate. The reasons are twofold. Firstly, we use the number of re-uses of the current LRU recency position to predict the possible number of hits on the new LRU recency position if one more way was added. This will normally lead to a slight over-estimation since in most of the cases, the number of reuses at $N$ is higher than that at $N + 1$, which means we actually achieve less performance improvement than we estimate. Secondly, even if we can estimate $T_{interval}$ well enough to avoid execution phase variation, we still have to accept the fact that the same execution phase will still cause small utility variations of cache as well. So the following BEM evaluation must be treated as an estimate not an absolute measure. We only use BEM as the upper bound of our tuning threshold set. Our first priority is to guarantee EDP reduction, so those assumptions which lead
2.5 Increasing Cache Algorithm

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{interval}}$</td>
<td>10 Million cycles</td>
</tr>
<tr>
<td>$P_{ls}$</td>
<td>0.81775 (watt)</td>
</tr>
<tr>
<td>Access$_{L2}$</td>
<td>1.44 Million</td>
</tr>
<tr>
<td>$E_{ds}$</td>
<td>0.0137 (nJ)</td>
</tr>
<tr>
<td>$P_{\text{total}}$</td>
<td>56.089(watt)</td>
</tr>
<tr>
<td>$T_{\text{misspenalty}}$</td>
<td>700 (cycles)</td>
</tr>
<tr>
<td>$E_{\text{miss}}$</td>
<td>0.144 (mJ)</td>
</tr>
</tbody>
</table>

**Table 2.2.** A set of example parameters based on our simulation framework

to over-estimations on performance loss are reasonable. A tuning scheme based on over-estimated performance loss will make the tuning procedure more conservative as no aggressive cache downsizing decision will be made.

**Definitions:**

BEM : The Break Even number of Misses (BEM) is the number of misses that will cause the same energy consumption as the total static power that can be saved by shutting off one additional cache way for one $T_{\text{interval}}$.

Energy Saved: $E_{\text{saved}} = P_{ls} \times T_{\text{interval}} + \text{Access}_{L2} \times E_{ds}$

Energy Wasted: $E_{\text{wasted}} = P_{\text{peaktotal}} \times T_{\text{misspenalty}} \times BEM + BEM \times E_{\text{miss}}$

Here $E_{\text{saved}}$ is the energy saved over the next $T_{\text{interval}}$ by shutting down one cache way. It comprises two parts. $P_{ls}$ is the leakage power saved by powering off one cache way which, when multiplied by the time interval gives the energy saving. This leakage energy saving dominates the entire saving. We can also save a small amount of dynamic power since there will no longer be the need to compare the request tag with the powered-off way upon each L2 access. Access$_{L2}$ is the total number of L2 transactions during $T_{\text{interval}}$ and $E_{ds}$ is the dynamic energy that can be saved during each L2 transaction.

$E_{\text{wasted}}$ is the extra energy used over the next $T_{\text{interval}}$ due to extra misses incurred because the way is shut down. $P_{\text{peaktotal}}$ is the run time peak overall power for the entire processor. Peak power is used because we try to estimate an upper bound. $T_{\text{misspenalty}}$ is the average penalty time for which a single L2 miss may stall the system and $E_{\text{miss}}$ is the energy cost of fetching that block back to cache from main memory. Table 2.2 gives a set of example parameters base on our simulation framework.
Chapter 2 Utility Based Cache Resizing

Setting the Energy Wasted less than Energy Saved will give us a BEM less than 200 in this case.

\[ P_{ls} \times T_{interval} + Access_{L2} \times E_{ds} \geq P_{peaktotal} \times T_{misspenalty} \times BEM + BEM \times E_{miss}. \]

Substitute example value:

\[ 0.81775 \text{Watt} \times \frac{10 \times 10^6 \text{Cycles}}{3.8 \times 10^9 \text{Hz}} + 1.44 \times 10^6 \times 1.37 \times 10^{-11} \geq 56.089 \text{Watt} \times \frac{700 \text{Cycles}}{3.8 \times 10^9 \text{Hz}} \times BEM + BEM \times 1.44 \times 10^{-7}. \]

we have \( BEM \leq 207. \)

It is obvious that the value of BEM is heavily dependent on the configuration of each individual system and the length of \( T_{interval} \) also affects the value of BEM. We will discuss how to select an appropriate length later in this chapter. Here, with a fixed sample interval length, the chip configuration is the only parameter we need to estimate BEM. This can be done at the design level very easily. With an estimated BEM, we can design a set of thresholds within that upper bound to demonstrate a set of trade-offs between power and performance.

Figure 2.6 shows how we select different thresholds. In this case, the thresholds were pre-defined at values ranging from 8 to 128. The upper limit was chosen to be 128 because it provides a reasonable safety margin. Since BEM is only an estimation, it is very likely that the real BEM is smaller than 200 in this case. If we choose to stop our largest threshold at 128, we almost can guarantee that we are operating under the real BEM even if we over estimated. If we set the largest threshold larger than 128, it would be 256 since we only using a saturating counter and check the MSB in order to simplify this tuning process. A threshold 256 is already over our BEM estimation, which means it can increase the overall EDP. A multiplexer is used to select an appropriate threshold as specified by the user, either performance-oriented (low threshold) or power-oriented (high threshold). The user-specified threshold value would be stored by the operating system before running the program and would be written into the “Threshold Reg” later, before the task pre-emption.
2.5 Increasing Cache Algorithm

2.5.3 L1 Early Eviction

Before we evaluate our tuning scheme, there is a special issue to address. We name it "L1 early eviction". It is a phenomenon introduced when we use a multi-level inclusive cache. In an inclusive cache, each block residing in L1 also has a replica residing in L2. The concept of stack distance works well if there is only one single level of cache. When expended to multi-level cache, things can become a little tricky. Higher level cache behaves as a filter which shuffles the locality of cache blocks when accessing lower level cache. If the size of the higher level of cache is relatively small when compared with the working set size, this phenomenon can be ignored. In other words, if the current working set size is big enough, blocks will be evicted from higher level cache.
easily before they have a chance to be reused. After higher level eviction, that reuse-access request will be routed to the lower level cache and generate a lower level cache hit and make a contribution to the cache utility estimation of that L2 tile.

If the current working set size is too small, blocks will stay at L1 for a long time and be reused repeatedly inside L1. No L1 cache miss means no reuse access would be made to lower level cache hence no information about utility would be recorded by lower level cache (L2 in this chapter).

Figure 2.7 shows an example. Block “A” and Block “B” are two blocks requested by Core 1 and Core 2 respectively. “A” is mapped at set 2 of the L2 bank on Core 1. “B” is mapped at set 5 of the L2 bank on Core 6. Currently, both L2 banks have a 6 out of 16 way associativity. The remaining 5 block-spaces on both sets are occupied by other blocks.

As shown on the figure, “A” will be re-referenced 2 times and “B” will be re-referenced 4 times, but the way they are re-referenced makes very different contribution to the L2 utility. After “A” is brought to private L1, 5 other blocks mapped to the same L1 set are requested. As a result, “A” is evicted since private L1 only has a 4 way associativity. When block “A” is evicted from L1, it will be discarded by the system. However, as we mentioned before, a replica of block “A” still resides at L2 cache so by the second time “A” is requested, private L1 will send a request to the L2 bank where the replica “A” resided. This request would contribute 1 to the utility counter on Core 1 if “A” happens to reside at LRU recency position. The same applies to the third reference because A was evicted from private L1 again before that. If “A” was not at LRU recency position then reducing the associativity of set 2 on Core 1 by 1 would not affect the two hits regarding retrieving ”A”. Hence there would be no utility counter increments since there would be no extra misses. If “A” was at LRU position, then the utility counter of tile 1 would be increment by 2.

On the other hand, the utility counter of L2 bank on Core 6 is not increased regardless of the recency position of replica block “B” because the original block “B” is retained inside private L1 and is never retrieved from L2 bank on Core 6. If we reduce the associativity of L2 on Core 6 by 1 and “B” happened to be located at the LRU position, then “B” is very likely to be evicted from L2 in the early stage. An inclusive cache requires that any block existing in L1 should also exist on L2, so evicting ”B” from L2 will require that it first be evicted from the private L1 cache on Core 2. Upon the second time ”B” is requested by Core 2, it would be a miss on L1 and the retrieval request from
2.5 Increasing Cache Algorithm

Figure 2.8. Stack distance based estimation of total L2 misses versus measurement for different number of ways along with the total number of L1 invalidations due to L2 replacement for each size (blackscholes)

L1 to L2 would also get a miss on L2 cache. If “B” were to be demoted quickly to LRU position again and evicted before requested the third time, it would cause another L2 miss. It will be another L2 miss if the same thing happens before the fourth reference.

As a result, Block “B” did not increase the utility counter of L2 bank on tile 6 but reducing the size of that L2 bank may cause a maximum 4 extra L2 misses. Evicting a block from L2 will also invalidate that block from private L1 where if the processor is still frequently reusing it, extra misses would occur then. Our tuning scheme, which relies on LRU reuse counting based utility estimation will underestimate the utility of L2 and make the wrong decision as not to release extra cache resource.

To solve this issue, we should consider early eviction as a reflection of lower level cache exhibiting good utility. Whenever a L2 slice invalidates an active block from any private L1 caused by L2 replacement, we increment the utility counter on that L2 slice by 1. Invalidation caused by the coherency protocol when writing to a shared block should not be included.

To investigate the L1 early eviction issue, we executed blackscholes for 1 billion instructions and recorded the stack distance upon each L2 access and also recorded the entire sequence of memory requests. Then we applied that memory request trace to exactly the same configuration but lower the associativity from 16 to 6, 5 and 4 (with same set
size). We compared the total number of L2 misses for different size of cache with the estimated value based on stack distance recorded with full cache size.

As shown in Figure 2.8, estimated values are consistent with measurement when associativity is 5 and 6. When further reducing cache size, the estimation starts to deviate away from measurement. This deviation was caused by early eviction. Why? Because when L2 associativity is greater than 4, the L2 replacement was not frequent enough to early evict blocks still being reused in L1. Figure 2.8 also shows the number of L1 invalidations caused by L2 replacement. When the associativity is greater than 4, the number of L1 invalidations are similar. We can see from the figure that the difference between estimation and measurement at associativity 4 is approximately the extra number of L1 invalidations.

We re-ran the same experiment with application x264. Results showed that x264 did not show any estimation deviation because x264 did not suffer from early eviction. As shown in Figure 2.9, L1 invalidations are similar at different cache sizes.

To verify that our adjustment can solve the L1 early eviction issue, we re-generated the estimated data with the addition of the number of L1 invalidations. Figures 2.10 and 2.11 show that after this adjustment, the variation between actual measurement and estimation is much less.
2.5 Increasing Cache Algorithm

**Figure 2.10.** Adjusted stack distance based estimation of total L2 misses versus measurement for different number of ways along with the total number of L1 invalidations due to L2 replacement for each size (*blackscholes*)

**Figure 2.11.** Adjusted stack distance based estimation of total L2 misses versus measurement for different number of ways along with the total number of L1 invalidations due to L2 replacement for each size (*x264*)
Table 2.3. Parameters for the simulation framework: multi core platform and single core platform

### 2.5.4 Evaluation Platform and Benchmarks

To evaluate the relationship between performance loss and power saving, we used a full system simulator, SIMICS 3 and GEMS (Ruby and Opal) (Martin et al. 2005) with power model McPAT (Li et al. 2009) and network model GARNET (Agarwal et al. 2009) to build our tiled CMP. Ruby is an accurate memory timing model and Opal is an out-of-order execution model. At the beginning of the simulation, we only apply SIMICS without the timing model and fast-forward the execution to reach the parallelism phase to avoid serial kernel activities. From there we extend the model with Ruby to capture the detailed memory timing then warm up the cache with approximately 20 million instructions on each tile. Tuning activity and performance measurement only start after the cache warm up.

The warm up period of “20 million instructions” is only an approximation because each tile runs at different IPC, so it is only possible to stop the execution when a particular tile reaches the target of 20 million instructions. Other tiles will have a similar number of instructions executed by that time if they have a similar IPC. After the warm-up phase we checkpoint the system, make sure the cache is warmed up which means no compulsory misses will be counted during later simulation, and start our simulation for another 2 billion instructions in total. We run each configuration several times and use an average result to deal with the non-deterministic behaviour of multi-threading. Primary parameters of the model are listed in Table 2.3.

For our 16-core studies, we use 8 workloads from PARSEC 2 (Bienia et al. 2008). All of them were running with native size of input, and specified to schedule to at least 15
2.5 Increasing Cache Algorithm

threads. Each core will be assigned 1 thread and the OS activity will occupy 1 core. For our single-core studies, we use all 30 applications from SPEC CPU 2006.

To prepare for the analysis of our tuning scheme, we first classify the 8 benchmarks into three different types, following Qureshi et al (Qureshi and Patt 2006).

The first type includes *bodytrack*, *blackscholes* and *swaptions* which represent applications with a working set size that can fit into a state-of-the-art CMP’s cache, so there is no need to allocate more cache resource than that. We refer to this type as Small-Demand-Poor-Utility.

The second type contains *streamcluster*, *facesim* and *x264*, which are applications with large working sets that are almost impossible to fit into a practical cache. For example, the working set size for *facesim* and *streamcluster* can reach hundreds of megabytes (Bienia et al. 2008). However, these applications do not benefit from allocating more cache resource because their cache utility is fairly low. It is true that for these applications the need for cache is almost insatiable and grows as the process executes, but the data stored in the cache will get replaced before it has a chance of being re-used. We refer to this type of application as Large-Demand-Poor-Utility.

The last group includes *canneal* and *ferret*, which contain applications with a large working set size (exceeding total on-chip memory size) but with good cache utility. These applications could make productive use of a very large cache. We refer to this type as Large-Demand-Good-Utility. Note that these classifications are made with reference to the corresponding framework. The demand/utility property is closely related to the configuration of the hardware. For example if we halve the size of our cache, then *ferret* will become a Large-Demand-Poor-Utility application whereas *facesim* will become a Large-Demand-Good-Utility application.

2.5.5 Up-Sizing Profile

In this study, we profiled the moment when a way allocation signal was triggered. We chose the most conservative tuning threshold, a value of 8 hits. Figure 2.12 shows the amount of cache allocated to each application as execution progressed. Each asterisk on the curve represents a way allocation signal activation, at which point one single way was allocated.
An arrow marks, for each workload, the end of the intensive allocation period observed in the early stages of each application. For Large-Demand-Good-Utility applications such as canneal, allocation events were triggered frequently and continually until the cache was fully activated in the early stages of execution. These type of applications would ideally need an unbounded cache. For Poor-Utility applications, whether small or large demand, allocation events were triggered frequently at the beginning of the simulation, but slowed down afterwards.

In the early stages of execution, allocation events were triggered frequently due to an initial under-estimation of the cache size. Frequent allocations continued until the point at which the cache reached the size satisfying the current workload’s cache utility. After that, only a few allocation events were triggered due to the variations in the execution phase, which we will discuss later. We denote this position as the stable point, and we say that the cache is stabilised after the stable point. Our goal in this chapter is to tune the cache size to the stable point as quickly as possible. The sooner we tune the cache size to the stable point the less performance penalty due to initial under-estimation of cache size.

2.5.6 Evaluation of Up Sizing Only Scheme

For good-utility applications, the performance degradation caused by the initial under-allocation of cache resource is very minor since they reach a stable point very quickly, before incurring large numbers of cache misses. They may however require a large number of allocated ways to be triggered before reaching the stable point compared to the other two application types.

It is true that the longer the duration of the establishment period, the more opportunity there is for a data block to be evicted, and since it is a good utility application, these data blocks are very likely to be used again soon. In other words, while good-utility applications are in the establishment period they are running with less cache space allocated than they could productively use. However, as these applications have good-utility, their L2 is accessed much more frequently and hence their establishment period is fairly short. Also we need to note that for these applications the cache will be 100% activated eventually, after which they will suffer no relative performance loss at all.
Figure 2.12. The tuning procedure of multi core framework

Figure 2.13 shows the performance, relative to a fully activated cache, of our tuning scheme with different threshold values. The figure shows that the average CPI (performance) overhead incurred by tuning is only 0.6% with a threshold setting of 8 (representing a performance-orientated setting) and 1.4% with a threshold of 64 (representing a power-saving orientated setting). However, static power is dramatically reduced to only 73.3% and 61.7% respectively according to the McPAT figures, while the total power consumption is reduced by 4.5% and 6.7%. Among all the applications, Poor-Utility applications benefit the most, with static power reduced to 58.8% and 43.1% of the fully activated case. In terms of the energy delay product, a performance-oriented threshold can provide a 4% saving on average and 9.9% maximum. The power-oriented threshold reduced the EDP by 5.9% on average and 11.1% maximum.
Chapter 2 Utility Based Cache Resizing

2.6 Bi-Directional Tuning Scheme

Our increase-only tuning scheme is relatively easy to implement with low hardware overhead. However, this simple algorithm can lead to a problem caused by two aspects of application behaviour. Firstly, during execution an application may exhibit multiple phases that result in a range of different memory access patterns. Secondly, threads are regularly swapped between tiles which will cause the local L1 cache to be invalidated and temporarily boost the number of L2 misses. Over time, both of these two facts will inevitably lead to all of the ways being activated and hence no power savings until the next task pre-emption by the operating system, at which point the cache will be reset to the initial size and then start increasing again.

Figure 2.13. The average performance loss versus power saving
2.6 Bi-Directional Tuning Scheme

![Graph](image)

(a) GemsFDTD

(b) bwaves

Figure 2.14. Temporary cache utility variation over time
Figure 2.15. Temporary cache utility variation over time
2.6 Bi-Directional Tuning Scheme

Figure 2.14 and 2.15 show, for four applications, the minimum size of cache that is required to ensure different levels of performance (95%, 99% and 99.9%) relative to a fully activated cache, as execution progresses. These four applications show a more distinct temporal phase variation compared with other applications which have more stable cache requirements.

In Figure 2.14 and 2.15, it is clear that the increase-only tuning scheme will easily saturate the cache. For example in Figure 2.14 (b), after relatively high cache utilisation in the early stages of execution, between 130 million cycles to 340 million cycles, only 3 out of 16 ways are needed to maintain 99.9% performance. A peak, which may be caused by a scan (a one-time memory footprint walk-through), can rapidly activate all of the cache ways again, which is undesirable. Figure 2.15(b) is quite similar; once every 200 million cycles a pulse of cache miss rate increases can fully activate the cache which will remain activated until execution completes.

The principal idea in our design is to try to achieve significant power savings without noticeable performance reduction. So when we come to consider a scheme for reducing the amount of cache space allocated to an application we adopt a more conservative approach than when allocating extra space. With a given threshold used to trigger an increase in cache size, we use half this threshold value to trigger a cache way deallocation.

It is not as easy to reduce the size of activated cache as it is to increase it. There are several complicating issues. The first is the requirement to maintain cache coherency. In modern CMP systems, when a cache block is fetched from main memory to the chip, it will be documented by the L2 directory cache, including information such as shared or exclusive and dirty or clean status. This will be tracked until it is evicted. If we simply shut off one of the cache ways without attending to the status of the blocks stored in this way we may create a functionally incorrect execution. A second issue is how to avoid thrashing. It is not acceptable to turn the cache on and off frequently just because of any small cache utility turbulence such as might be caused by, for example, an algorithm that occasionally reuses a small amount of data.

2.6.1 Pre-reduction Preparation

When the decision is made to reduce the cache size, the last cache way will be shut off and all the data blocks residing in that victim way should be taken care of in order to
Table 2.4. Proportion of different block status upon way shut off

<table>
<thead>
<tr>
<th>Test-bench</th>
<th>Clean</th>
<th>Dirty</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>78.9%</td>
<td>10.8%</td>
<td>10.3%</td>
</tr>
<tr>
<td>bodytrack</td>
<td>62.2%</td>
<td>8%</td>
<td>29.8%</td>
</tr>
<tr>
<td>facesim</td>
<td>33.3%</td>
<td>10.8%</td>
<td>55.9%</td>
</tr>
<tr>
<td>swaptions</td>
<td>30.6%</td>
<td>1%</td>
<td>68.4%</td>
</tr>
<tr>
<td>x264</td>
<td>58.8%</td>
<td>8.4%</td>
<td>32.8%</td>
</tr>
</tbody>
</table>

maintain correct cache coherency. Invalidation signals should be sent to the L2 directory to update the coherency information. If a block is in a modified state, a write back action must be completed prior to the actual shut off.

These processes sound arduous and expensive in terms of power consumption, and make cache reduction look less attractive. However, we can simplify those tasks and lower the total cost of pre-reduction preparation within a reasonable range.

A simplistic approach would be to shut off the last way of each set regardless of the state of the resident data blocks. As an illustration, Figure 2.16 depicts an example of what a cache might look like before a shut down. Only 5 out of all 16 ways are active. Upon receiving another shut down signal, the L2 controller will turn off the way at the farthest recency position, way 5.

Before powering off way 5 we have to guarantee that all of the blocks stored in way 5 go through pre-reduction preparation. For Set 1, the victim block is in ‘Share’ state which means more than one L1 cache holds a replica of this data block. To invalidate a ‘Share’ block, it is necessary to send multiple invalidating signals to each of the sharing hosts and wait until all unblock signals are received back. For Set 2, the victim block is in ‘Exclusive’ state which means the block is exclusively owned by this L2 so there is no need to be concerned about coherency issues before deleting it. Set 3 is the most difficult case. For a ‘Modified’ state, we can only shut down the way after the data from this block is properly written back to main memory.

In contemporary computer architectures, a main memory write-back normally takes hundreds of cycles, and this does not take into account possible inter-thread contention. In the case of our simulation framework, the number of sets is 2048 which means when we decide to shut down one way, 2048 blocks must be emptied. If only
2.6 Bi-Directional Tuning Scheme

Figure 2.16. Example snapshot of cache content before way shut off

10% of these spaces hold ‘Modified’ blocks, then over 200 cache write-backs are required before powering down the way, which is a huge latency. Table 2.4 shows an average proportion of dirty blocks for a sample of different PARSEC2 test-bench applications upon way shut off.

Here we propose a solution called “internal substitution” to dramatically lower the workload prior to shut off. During the pre-reduction preparation, if there is a dirty block, instead of writing it back, we swap it with a clean cache entry within the same set. The priority of swapping would be: Invalid Block (random pick if more than one exists) > LRU clean > LRU dirty. This internal substitution will not help to reduce the number of misses but it will help to prevent causing extra misses. A cache read and cache write consumes much less power and time than a memory write. Our simulation shows that by applying internal substitution we can avoid almost all of the write-backs before completely shutting down one cache way. In the rest of the paper, all simulations assume that internal substitution is in effect.

2.6.2 Thrashing Protection

Because size reduction brings such a high overhead, the situation that we try to avoid most is shutting off the cache too aggressively. When we discussed BEM, we worked out the break-even point assuming a way can be powered off for at least one sample interval. However, the fact is we are predicting the future, so there will always be mispredictions. When that happens, a cache way that was shut off needs to be reactivated within the same interval. This violates the BEM assumption which assumes that the
way should be off for at least one interval so the power saving could at least compensate for the overhead of the reduction process. This may lead to a negative power saving. We could change the BEM threshold to reduce the risk of a negative power saving. The other effective method to lower this risk is to enlarge the $T_{\text{interval}}$ to lower the possibility of a false reduction. However, $T_{\text{interval}}$ must be small enough to provide enough speculative accuracy.

Figure 2.17 shows the resizing behaviour of milc with different sample intervals. When using a 1 million cycle interval, false reductions lead to a comb shape by thrashing - turning the cache way on and off frequently. When raising the interval to 100 million cycles, the false reductions can be entirely removed hence giving the least loss of performance. However this very large interval leads to a very conservative down sizing and results in an unsatisfactory power saving. A 10 million cycle interval, on the other hand, can provide both enough thrashing protection and a good level of power saving. Of course a dynamically tuned time interval will better fit with different applications, but in order to not overly complicate the system, we have used a fixed time interval. We verified various $T_{\text{interval}}$ values with all of the applications in SPEC CPU2006, and selected a 10 million cycles sample interval as the fixed sample interval. By doing this, we are leaving only one freedom degree in the system, which is the configurable BEM threshold.

### 2.6.3 Down-sizing Evaluation

To evaluate cache size reduction, all of the SPEC CPU2006 and PARSEC2 benchmark applications were simulated. Figures 2.18 and 2.19 show that saturation caused by
### 2.6 Bi-Directional Tuning Scheme

<table>
<thead>
<tr>
<th>Tuning Scheme</th>
<th>Normalized Avg CPI</th>
<th>Normalized Avg Leakage Power</th>
<th>Normalized Avg Total Power</th>
<th>Normalized Avg EDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increase Only T=4</td>
<td>1.001</td>
<td>0.873</td>
<td>0.968</td>
<td>0.968</td>
</tr>
<tr>
<td>Increase/Reduce T=4</td>
<td>1.002</td>
<td>0.676</td>
<td>0.918</td>
<td>0.920</td>
</tr>
<tr>
<td>Increase Only T=32</td>
<td>1.001</td>
<td>0.825</td>
<td>0.956</td>
<td>0.956</td>
</tr>
<tr>
<td>Increase/Reduce T=32</td>
<td>1.005</td>
<td>0.592</td>
<td>0.896</td>
<td>0.900</td>
</tr>
<tr>
<td>Increase Only T=128</td>
<td>1.002</td>
<td>0.765</td>
<td>0.940</td>
<td>0.942</td>
</tr>
<tr>
<td>Increase/Reduce T=128</td>
<td>1.015</td>
<td>0.454</td>
<td>0.859</td>
<td>0.871</td>
</tr>
</tbody>
</table>

Table 2.5. Average performance (CPI), leakage power, total power and EDP of different tunings scheme and different threshold value relative to a full size cache.

scan, phase changing or thread swapping can now be mitigated. Cache size was gradually reduced when the utility of the cache decreased. Table 2.5 gives the normalized average performance (CPI), leakage power, total power and energy delay product from simulating all applications of the SPEC CPU2006 suite on a uni-core system with the different tuning schemes. The results are based on comparison with a fully activated cache. Those figures reveal that down-sizing can dramatically lower the leakage power even further. For example, in the case when BEM threshold=128, average static power was cut to 45.4% from 76.5% leading to a 14.1% total power saving and 12.9% EDP reduction relative to a fully activated cache.

Figure 2.20 shows the comparison between increase-only and bi-directional resizing for the 16 core system. In most cases down-sizing achieves a lower energy delay product. The exception is the case where BEM threshold=64. It did not work well because of over-aggressive reduction. It can save more power than the other settings but it also incurs a larger performance penalty, which results in a higher EDP.

For a multi-core system, down sizing can further reduce leakage power by roughly 10%, reducing the EDP by 6.5%. The reason why multi-threaded applications do not benefit as much as single-task applications is because they have a relatively more stable execution phase variation pattern. Another important reason is that the multi-core system and uni-core systems have different chip configurations, so they have quite different BEM values. For the uni-core system, which exclusively owns a 4M Byte
Figure 2.18. Temporary cache utility variation over time
2.6 Bi-Directional Tuning Scheme

Figure 2.19. Temporary cache utility variation over time

(a) h264ref

(b) milc
Figure 2.20. Comparison between the increase-only and increase/reduce tuning schemes

cache, shutting off one cache way can result in a higher energy saving than in the multi-core system. On other hand, the uni-core system also consumes less peak dynamic power. As a result, the uni-core system behaves much better with same threshold.

2.7 Tile Level Evaluation

One of the other important objectives of our research is to lower the tuning granularity down to the tile level. We believe that even if tiles are running the same application this cannot guarantee a balanced utilisation on each tile. First of all, many multi-threaded applications can not be scheduled evenly. More importantly, we employ a shared cache as our last level cache and the total number of L2 accesses for each tile is decided by
2.7 Tile Level Evaluation

![Bar charts showing L2 demand for different applications: (a) blackscholes, (b) bodytrack, (c) canneal.](image)

**Figure 2.21.** L2 demand of each tile for different applications, the sixteen tiles are represented as a grid in the horizontal plane of each graph

how often the data mapped on itself can be re-referenced. The locality of this data is determined by the thread that accesses it.

At compile time it is difficult to identify or predict the future utility of each tile. Figure 2.21 shows the total number of L2 accesses for different tiles running the three applications _canneal_ (Large Demand Good Utility), _bodytrack_ and _blackscholes_ (Small Demand Poor Utility). For _bodytrack_, and _blackscholes_, some tiles were accessed much more frequently than others, in some cases by more than ten times. The reason for this unbalanced tile access may be because some frequently shared data was mapped at those tiles. On the other hand, _canneal_ as a Large Demand Good Utility application behaved much more consistently than the other two, in that the rates of access for all tiles are similar.

Should we adjust the size of each tile according to the access rates shown in Figure 2.21? The answer is “no”. That only indicates how often each tile was accessed. It is true that more accesses may represent more cache misses but it does not imply that allocating more resource will bring benefit.

Figure 2.22 shows the actual number of ways allocated to each tile. Some tiles have a very small number of L2 accesses, but since these accesses have very good utility, or the data stored on these tiles can be re-referenced frequently, more cache resources were allocated to them. Some tiles have a very large number of L2 accesses, but since the data stored on them was seldom re-referenced, fewer cache resources were allocated to them. The non-uniform utility for each tile is the reason why the performance of an averaging 4 way L2 system can execute faster than a normal 4 way L2 system.
2.8 Summary

The model of tile-structured CMPs cooperating through the use of a shared last level cache is a promising technology because of the good scalability and simplicity of design achieved by leveraging identical tiles. However, different applications have different requirements for cache resources. In this chapter, we have shown that allocating full cache resources to applications that don’t need them is wasteful of power. Moreover, we verified that the cache demand of an application is not the same as its cache utility.

We have proposed BEM as the upper bound for the machine to determine how aggressively it can resize. We also proposed a method, involving modest performance overhead, to dynamically analyse the cache utility of a running application and re-allocate cache resource. Experimental results have shown that, compared with a static fully opened cache system, our tuning scheme can reach a stable configuration in a short time, with a performance reduction of only 0.6% and 1.4% for performance-oriented and power-oriented settings, respectively. On the other hand, our scheme consumed an average of only 73.3% and 61.8% of the original cache static power. The cache performance penalty is only due to the under-estimation of the required cache size during the establishment period. After that, each tile will be tuned to its optimized size and there will be no further performance loss.

Only increasing and never decreasing the size of cache is safe in terms of performance guarantee, however, it can rapidly saturate the size of the cache due to execution behaviours like scan, phase changing or thread-swapping. To address this, we enhanced our method so that it can reduce the size when the cache becomes saturated. We showed that down-sizing can further reduce the leakage power by roughly 10%.
average across several applications it can reduce the EDP by 6.5% for multi-threaded applications and 12.9% for single-task applications.

We also note that even for the same application, the utility of each tile varies depending on how the data was mapped to the memory. The mapping of applications and threads to tiles is a non-deterministic operating system activity, therefore it is hard to predict at compile-time. Our tuning scheme lowers the granularity of cache adaptation to the tile level, meaning that performance adapts dynamically to different thread allocation outcomes.

However, there remains an important issue. Real LRU replacement policy is very expensive both in terms of hardware overhead and operating dynamic power consumption. Because of that, it is only used for low-set-associativity cache (normally associativity lower than 4). We conjecture that tiled-structure CMPs needs much higher associativity to ease the contention coming from the different working sets. What the industry does today is to use pseudo LRU for large associativity caches. Pseudo LRU policies like NRU (none recently used) and Binary Tree based pseudo LRU are commonly used in off-the-shelf products (Sun Microsystems 2007). However, they can not provide us information like stack distance which is the key component to predicting cache utility. More importantly, we believe even with real LRU policy, several fatal issues may arise with future core size scaling up. In the next chapter, we will introduce a substitution for LRU policy with light hardware overhead and higher performance. It will be more suitable for future scalable tiled structures. We can use that new replacement policy to estimate cache utility in later chapter.
In this chapter, we focus on exploring a substitute for true LRU replacement policy. Belady’s optimal cache replacement policy is an algorithm to work out the theoretical minimum number of cache misses, but the rationale behind it is too simple. In this chapter, we reconsider the essential function of caches to develop an underlying analytical model. We argue that frequency and recency are the only two affordable attributes of cache history that can be leveraged to predict a good replacement. Based on those two properties, we propose a novel replacement policy, the Effectiveness-Based Replacement policy (EBR) and a refinement, Dynamic EBR (D-EBR).
3.1 Introduction

The purpose of a cache is to temporarily hold data blocks accessed by the processor, keeping them close to the processor in case they are going to be used again in the future. In today’s multi-core systems the latency of accesses to the main memory can reach as many as hundreds of cycles. Any hit in the LLC means a reduction in the overall running time since an extremely long main memory fetch latency is eliminated. Unfortunately this benefit is not achieved for free. Practical limitations on the area and power consumption of a cache constrain its size. This means the cache will be fully occupied within a short period of time and after that, if a new data block must be accommodated, a replacement decision has to be made. Many replacement algorithms have been proposed over time, but when we apply the power, area and hardware overhead constraints when designing a CPU, not a lot of them are practical. In this paper we will re-examine the essential function of a cache and find the underlying analytical model of cache replacement. Based on that model we will propose a new replacement policy, Effectiveness-Based Replacement policy (EBR).

The purpose of a good replacement policy is to make sure that the data stored in the cache can be re-used as many times as possible. We propose a new definition of an optimal cache replacement policy which can be used to explain many widely accepted replacement policies. Replacement policies involving PC (program counter) indexes or past access intervals have been investigated by many researchers, but many of them dramatically complicate the control logic for the cache. We propose to use the readily-obtainable measures of recency and frequency to develop a replacement scheme that is not significantly more complicated than the industry-preferred pseudo LRU. The ideal Least Recently Used (LRU) policy uses recency information only, while Least Frequently Used (LFU) uses frequency only. However in the absence of the other information, LRU and LFU can produce problems, such as thrashing and scan in the case of LRU and ageing in the case of LFU (Jaleel et al. 2010). There has been previous research which proposed to integrate LFU with LRU (O’Neil et al. 1993) (Lee et al. 2001) (Robinson and Devarakonda 1990) (Subramanian et al. 2006) (Megiddo and Modha 2004) (Bansal and Modha 2004), but these works either aim at the page level with unaffordable hardware complexity for caches or use set duelling to select between LRU and LFU, which needs to retain both systems and switch between them depending on the duel result. We believe it is not an efficient method of combining.
EBR combines the measures of recency and frequency to form a rank sequence inside each set and, upon each miss, the block with lowest rank is evicted. EBR assumes that the weight of recency should be much higher than frequency. It divides the recency stack into several subgroups, each representing a recency level. Inside each group, frequency determines the ranking. However, there are cases when the assumption that recency is more important than frequency does not work effectively and to get best performance we should rely more on LFU than LRU. Dynamic-EBR uses a modified set duelling (Qureshi et al. 2007) technique to find the most suitable way to divide recency subgroups. With the help of D-EBR, we can typically double the improvement made by EBR alone with a small amount of overhead.

The rest of this chapter is organised as follows. In Section 2 we discuss the objective of cache replacement and propose a new definition of optimal replacement. Section 3 discusses the deficiencies of commonly used replacement policies and their causes. Sections 4 and 7 introduce EBR and D-EBR, respectively. Sections 5, 6 and 8 present the results of our simulations and Section 9 focuses on the complexity sensitivity of D-EBR. Section 10 talks about the hardware overhead, and the section is at Section 11.

### 3.2 Re-definition of Cache Optimal Replacement

The essential function of a cache is to hold blocks that will be accessed again in the future. If a block is not going to be re-referenced in the future then there is no purpose served in keeping it in the cache. To do so wastes power and the potential chance for other blocks to achieve hits during that period of time.

Let $R_i(t, t + \Delta t)$ denote the number of re-references to block $i$ if it can reside in cache during the future period between current time $t$ and $t + \Delta t$. Assume for the moment that there is only one way in the cache, and that at time $T_1$, two blocks A and B are candidates for eviction, whose future re-reference pattern follows Figure 3.1. One of these is a new block and the other is the block already present. In the future, $R_A(T_1, T_{10})$ would be 4 and $R_B(T_1, T_{10})$ would be 5. In other words, if we leave A in the cache in this example we will achieve 4 hits and 5 misses whereas if we retain B, we will achieve 5 hits and 4 misses.

To evaluate the above situation more precisely, we introduce a new metric. We define **Effectiveness** $E_i(t, t + \Delta t)$ of a block $i$ as the rate of re-use of the block over future time period $\Delta t$ if this block was allocated in the cache. $E_i(t, t + \Delta t) = \frac{R_i(t, t + \Delta t)}{\Delta t}$. The higher
3.2 Re-definition of Cache Optimal Replacement

the Effectiveness of a block, the more benefit we would gain if we let this block reside in cache during that period of time. If we always allocate blocks with maximum $E$ to the cache, we will maximise the number of hits. Equation 3.1 gives Total Effectiveness, $\xi$, of all of the on-chip blocks during the period of execution.

$$\xi = \sum_{j=0}^{last-1} E_i(t_j, t_{j+1})$$

$t_0$ is the start of the execution period
$t_{last}$ is the end of the execution period

A maximised $\xi$ will guarantee a maximised number of cache hits and minimum number of cache misses, so any policy that maximises $\xi$ can be considered optimal. Hence we now propose a new definition for the ultimate goal of a cache replacement policy:

**Definition 1.** The Optimal Replacement Policy is the one that maximises Total Effectiveness $\xi$.

Considering that the chance to rearrange in-cache block combinations only occurs when there is a replacement, and a replacement only occurs when a miss has occurred, it will be too late to include the effect of the miss when deciding the following replacement. (We do not consider early replacement here, since bringing the block into the cache at a time before it is to be used may actually be a waste of efficiency through occupying the cache earlier without any contribution to performance.)

Hence, we propose the following replacement strategy: upon a replacement, those blocks that have the largest Effectiveness during the time interval, from the moment when the current replacement finishes to the moment when the next replacement finishes, can remain in the cache. We denote the time interval from the moment when the current replacement finishes until the moment when the next replacement finishes as a time slice. We can then define the optimal effectiveness based replacement strategy as:

**Definition 2.** Optimal Effectiveness Based Replacement Strategy: Upon a replacement, retain those blocks that will maximise Effectiveness in the following time slice.

With these definitions, it is clear that in the example of Figures 3.1, 3.2 and 3.3, always keeping block B in the cache is not the optimal replacement strategy, because during
the time period $T1$ to $T5$ $E_A(T1, T5) = \frac{3}{4}$ whereas $E_B(T1, T5) = \frac{1}{4}$. This violates the definition of the Optimal Effectiveness Based Replacement Strategy, to always retain blocks with the highest Effectiveness. In other words, between times $T1$ and $T5$, if we retain block A in the cache we would save 2 more misses compared with the case if we had left B in the cache all the time. The cost of replacing A with B after the first 4 time slices is only 1 miss.

Up to this point we have not considered the cost of stalls caused by misses. So in this example, the optimal replacement and non-optimal replacement will look like Figures 3.2 and 3.3. At the very beginning, data block B resides in the cache, being re-referenced at time $T1$. At $T2$, block A is referenced by the processor, which is when the first miss occurs. The first miss, which is inevitable, would finish at $T3$. Between $T2$ to $T3$, we have to make a choice between the two candidates, A and B, as only one of them can remain in the cache. If we keep B, then the next miss time would start just at $T3$ and finish at $T4$, so the effectiveness for each block during this time unit would be $E_A(T3, T4) = \frac{1}{\text{misstime}}$ and $E_B(T3, T4) = \frac{0}{\text{misstime}}$. As $E_B < E_A$, it is incorrect to leave B in the cache. The other option is to swap A for B at the first miss. In this case, the next miss would be at $T5$ when block B will be re-referenced and finish at $T6$. Thus $E_A(T3, T6) = \frac{2}{\text{hittime} + 2\text{misstime}}$ and $E_B(T3, T6) = \frac{1}{\text{hittime} + 2\text{misstime}}$, also noting that $\text{misstime} \gg \text{hittime}$, so $E_A > E_B$, and retaining A is the right choice. As a result, we
3.2 Re-definition of Cache Optimal Replacement

Figure 3.2. Illustration of effectiveness example 1: Block B remains in the cache all the time

Figure 3.3. Illustration of effectiveness example 1: Swap block B with block A between T2 to T5
can guarantee that during any time unit within the execution time, the block on-chip has maximum effectiveness, hence we can expect the maximum number of hits.

As another example, consider the reference pattern depicted in Figure 3.4, in which references to blocks A and B are interleaved. We assume that at the beginning, B was stored in the cache, and when A is requested, the first miss occurs. Again, there are two candidates, A and B for residency in the cache during the next time unit. If we were to swap B for A, as in Figure 3.5 then the next miss would be just the next request and 

\[ E_A(T3, T4) = \frac{0}{\text{misstime}} \] and 

\[ E_B(T3, T4) = \frac{1}{\text{misstime}} , \] \( E_B > E_A \), so according to our definition this would not satisfy the Optimal Effectiveness Based Replacement Strategy.

The other option, leaving B in the cache, is depicted in Figure 3.6. In this case the next miss would occur when A is re-referenced the second time, 

\[ E_A(T3, T5) = \frac{1}{\text{hit} + 1\text{miss}} \] and 

\[ E_B(T3, T5) = \frac{1}{\text{hit} + 1\text{miss}} , \] so \( E_B = E_A \). Even though these two have identical effectiveness this option can avoid evicting a block with higher effectiveness, which means this is the right choice according to our definition.

The above examples illustrate the definition of Optimal Effectiveness Based Replacement Strategy using the case of a cache with only one way associativity, so there are only two candidates upon each replacement including the block already present. If the
3.2 Re-definition of Cache Optimal Replacement

Figure 3.5. Example 2 to illustrate block effectiveness: Swap block B with block A upon each miss

Figure 3.6. Example 2 to illustrate block effectiveness: Block B remains in the cache all the time
associativity were higher than one we would have more than two candidates when making a replacement decision and a tied result could occur.

In Figure 3.7(a), illustrating a 2-way associative cache, we assume that blocks A and B are originally stored in the cache. A and B are re-referenced at T1 and T2 respectively, block C is referenced at T3 and causes the first miss, which finishes at T4. To determine the two blocks that should stay in the cache, the Effectiveness of A, B and C during the next time slice, T4 to T5, are evaluated. We obtain \( E_A(T4, T5) = 0 \), \( E_B(T4, T5) = 0 \) and \( E_C(T4, T5) = 0 \). This means no matter which two blocks we choose to keep in the cache during time interval T4 to T5 it will make no difference to the overall \( \xi \), so we conclude that we can evict any one of them. However, that conclusion is not valid if we take into account more time slices. Consider the following time slice, T5 to T6. As C is re-referenced at T5, regardless of the status of C (whether it was kept in the cache after T4 or not), \( E_C(T5, Tx) \geq \frac{1}{Tx-T5} \), for all \( Tx > T5 \). The Effectiveness of C is certainly not the smallest of all of the candidates, since no way we can achieve at least one hit to each of these three blocks inside a two way-associativity cache without a miss. This means that differences in \( \xi \) will show up depending on which of A, B and C we decide to keep in the cache for the following two time slices. For now it is clear that C should stay, but what about A and B?

Let’s consider further time periods. As A is reused at T6, \( E_A(T5, Tx) \geq \frac{1}{Tx-T5} \) which is certainly greater than or equal to \( E_B(T5, Tx) \leq \frac{1}{Tx-T5} \). The worst case for Block A is that it won’t get any more references before the next cache miss, and the best case for Block B is that the next cache miss is caused by B itself. This example is the best case for B and the worst case for A. \( Tx \) will not be greater than the moment when all the blocks get at least one re-reference because this is a two way associative cache, and block C occupies one way already. For the other way if it does not store block A, \( Tx \) should be T6, meaning no block gets hit during this time unit. If it does store block A, \( Tx \) should be T7 (when block B is re-used).

Now we can determine that the correct decision for the first miss is to evict block B, which is the case shown in in Figure 3.7(b). To fix the problem of drawn Effectiveness upon replacement, we further constrain our optimised replacement definition as below:

**Definition 2 Refined.** Refined Optimal Effectiveness Based Replacement Strategy: upon a replacement, evict the block that has the smallest Effectiveness at the time from the
3.2 Re-definition of Cache Optimal Replacement

(a) Future access sequence of blocks A, B, C and D

(b) Optimal replacement

Figure 3.7. 2-way associativity example to illustrate block effectiveness
replacement moment to the future when we are first able to identify a block with the smallest Effectiveness among all the replacement candidates.

With the help of this new definition, most of the conventional replacement policies can be readily explained and analysed. In the next section, Belady’s Optimal Replacement Policy (Belady 1966) and the most widely accepted policies, LRU and LFU will be reviewed. The Effectiveness based replacement policy (EBR) will be introduced after that.

### 3.3 Study of Different Replacement Policy

#### 3.3.1 Optimal Replacement Policy

Belady introduced the concept of optimal replacement. He proposed that the only way to make an optimal replacement is to use the entire future sequence and work backward (Belady 1966). Based on his algorithm, when a replacement must occur the candidate that will be re-referenced in the farthest future should be evicted. This can be easily explained in terms of our definition: the moment that the second farthest distant block gets hit is the first moment we can distinguish the smallest Effectiveness among all the candidates. By that time the future victim is the only one whose Effectiveness is \( \frac{0}{T} \), which is the smallest value, and shall be evicted.

#### 3.3.2 Least Recent Used Replacement Policy (LRU):

LRU tends to preserve those blocks with high Effectiveness by assuming that a block is very likely to be used again if it was recently used. Let \( \Delta T_i \) be the time period from the moment when block \( i \) was used in the past, to present.

One way to view LRU is that it assumes a block \( i \) that was used \( \Delta T_i \) ago will be likely to be used again within the future time \( \Delta T_i \). Thus LRU assumes that within next \( \Delta T_i \), block \( i \) will have a relatively high probability of being used again, say \( P \) which is approximately equal to 1. The same applies to all blocks, so in LRU the Effectiveness of each block \( i \) may be approximated as \( E_i = 1 \times P/\Delta T_i \). The victim with a minimum \( E \) is the block with largest \( \Delta T_i \), which in other words, is the least recently used block.

LRU is the most widely accepted standard replacement policy so far and has been shown to work well with many applications. However, to implement the ideal LRU
policy is not realistic, as it requires that the recency sequence be re-ordered at each access, which is too expensive in terms of hardware overhead and power consumption. In contemporary commercial processors various pseudo-LRU schemes are used as a substitute for ideal LRU, such as none recently used replacement policy (NRU) (Sun Microsystems 2007) and binary tree based pseudo LRU described in patent (Wen-Tzer Thomas Chen 2006).

However, even with pseudo-LRU schemes there are still three issues that are known to cause problems, namely 'Thrashing', 'Scan' and 'Multi-Application Interference'. 'Thrashing' is a term that describes the situation when the running program has a working set that is larger than the size of the cache. Data blocks will be evicted from the cache due to capacity misses before they get a chance to be re-referenced. 'Scan' describes the situation in which there is a burst of references to a group of data that will only be used once. These data may contribute no re-reference counts in the future but they will pollute the current cache content by evicting blocks that might be re-referenced soon. 'Multi-Application Interference', occurs when the last level cache (LLC) is shared by processors running different applications. Data blocks belonging to LRU-friendly applications can be polluted by others that behave like streaming applications. Figure 3.8 illustrates all three cases.

Let us reconsider LRU’s basic assumption. LRU assumes that a recently used block will be used again soon. We believe the reason that LRU sometimes performs poorly is that there is no distinction between the future re-reference probability among blocks that have similar recency. Consider for example two blocks, A and B, that were both requested approximately $\Delta T$ ago, but B was brought into the cache just one cycle after A was re-referenced. So B is a new block and A is a re-referenced block that may have been used several times.

Let AvgT denote the average lifetime of recent evictions of the set to which A and B are mapped. If $\Delta T$ is relatively small, both A and B share the same probability, $P$, of being re-used in the future $\Delta T$. But as $\Delta T$ gets larger, if neither of them are re-used then when $\Delta T$ approaches AvgT, there will come a time to make a choice between A and B, one to be evicted and the other one to stay. Although both A and B were accessed at around the same period of time ago, according to LRU, B is one cycle after A, so B, the new coming block has a Effectiveness $E_B = P/(\Delta T - 1)$ which is greater than $E_A = P/\Delta T$. Hence, A will be evicted according to LRU.
Figure 3.8. Three different cache access patterns
3.3 Study of Different Replacement Policy

However, we contend that A and B should not share the same possibility of future re-use. Arguably A should have a higher possibility than B in terms of getting re-used since A was able to survive till now only because it was re-referenced within each of the last few AvgTs. That is, A has a history of re-use of time intervals of the order of AvgT and should be considered a more likely candidate than B for re-use soon.

3.3.3 Least Frequently Used Replacement Policy (LFU):

The Least Frequently Used replacement policy assumes that a block that has been frequently re-referenced will have a higher likelihood of being re-referenced again in the future. LFU can distinguish the re-use possibility of two blocks that have different usage history, assuming that $P_A > P_B$ if the number of uses of A has been greater than the number of uses of B. Let $\Delta T$ denote an arbitrary future 'time unit', so we have:

$$E_A(T, T + \Delta T) = \frac{P_A}{\Delta T} > E_B(T, T + \Delta T) = \frac{P_B}{\Delta T}$$

LFU can provide good protection against Thrashing, Scan and Multi-Application Interference, but it cannot distinguish between different recency histories. The most significant weakness is that LFU cannot adapt when there is a context change. A block that was frequently used in the far past is considered, under LFU, to have a higher possibility of future re-use than the block that was less frequently used in the near past.

Even if a block, A, has a much higher frequency of reuse than block B, which causes $P_A > P_B$, if block A was last reused in the far past, but block B was reused in the near past we should give preference to block B. In other words, if $\Delta T_A >> \Delta T_B$ then we expect block B to be used again sooner and so we should keep B in the cache in preference to A. In terms of our Effectiveness model, even if $P_A > P_B$, if $\Delta T_A >> \Delta T_B$, then $E_A(T, T + \Delta T_A) = \frac{P_A}{\Delta T_A}$ will be smaller than $E_B(T, T + \Delta T_B) = \frac{P_B}{\Delta T_B}$ and so we should retain B.

Many researchers have proposed to combine LFU and LRU, but their methods either require significant hardware overhead and require pre-profiling before running or they use set duelling to make a choice between LRU and LFU, which keeps two sets of replacement status. This is not really combining two into one to improve the estimation of the likelihood of block re-use. In the following section we introduce an Effectiveness-based replacement policy, which interleaves both recency and frequency information and predicts the best victim based on estimated Block Effectiveness.
3.4 Effectiveness Based Replacement Policy (EBR):  

Using previous information to predict future access sequences is the nature of any replacement policy, but with EBR, we no longer predict the future re-reference interval based on recency information only nor predict future re-reference numbers based on previous re-use numbers only. We use both of those metrics in combination to make the right choice of victim block.

For a 16 way set associative cache, we use a 2 bit counter per block, called the R-counter, to record frequency information. Each time a block is re-referenced, its R-counter will incremented by 1. The R counter saturates (when it reaches 3 it stays at 3). The R-counter will be reset to 0 when that block gets evicted and is replaced by another block.

We also use a 2 bit counter per block, called the E-counter, to represent how much time has elapsed since the block's reference. When the block is first allocated to the cache, its E-counter is set to 0 (Algorithm 1 line 21, 22). As time passes the E-counter is incremented as explained below, until a re-reference happens and resets it back to 0 (Algorithm 1 line 8).

The E-counter is a saturating counter; when it reaches its maximum value it stays at that value until it is reset by a re-reference. The entire set is thus divided into 4 groups of blocks with different recency values.

To ensure that blocks are distributed as evenly as possible into the four groups, we increment the E-counter of all the blocks in a set when 4 misses are encountered in that set, in the case of a 16 way associative cache (Algorithm 1 line 13 to line 19). Imagine an access sequence within a set in which no reuse is involved. After 16 misses to the set, blocks in that set would be grouped into 4 subgroups, each containing 4 blocks that have the same E-counter value. Even considering reuse, this is still a good approximation to subgroup blocks based on recency. Figure 3.9 shows an example.

Upon a replacement in a set, each block has an R-counter value and an E-counter value. Using those two, the estimated $E_i$ can be evaluated and sorted, the block with smallest Effectiveness being evicted. We use recency groups to estimate future reuse time interval and use the number of reuses to predict future reuse probability.

Based on empirical experience and also in order to simplify the sorting procedure, we assume that the recency weight $r$ is much greater than the frequency weight $f$, $r/f \gg 1$. So the estimated $E(i) = \frac{r^{R_{counter}(i)}}{f^{E_{counter}(i)}}$. The assumption here means that in most cases...
3.4 Effectiveness Based Replacement Policy (EBR):

Algorithm 1 PSEUDO code for EBR:

1: Upon a Request of Block with an address: Addx
2: Set_id=Findset(Addx) { //find out the set of the block}
3: for i = 1 → Associativity do
4:   if Set_id[i]==Addx then
5:     if R[Setid][i] <Rmax then
6:       R[Setid][i]+=1 { //Increment Rcounter of Block i of Set_id} { //Rmax = 4}
7:     end if
8:     E[Setid][i]=0 { //Reset Ecounter to 0 because of reuse}
9:     return Hit
10: end if
11: end for
12: //If can not find Addx in Setid, means a miss
13: if CounterMiss[Setid]==4 then
14:   //it has been 4 misses since last Ecounter increment
15:   for i = 1 → Associativity do
16:     if E[Setid][i]<Emax then
17:       E[Setid][i]+=1 { //Increment E-counter of every block in Setid by 1}
   { //Emax = 4}
18:   end if
19: end for
20: else
21:   CounterMiss[Setid]+=1
22: end if
23: Smallest_Rank= Search(Setid) { //Search smallest Rank in Setid}
24: Victim=Random(Smallest_Rank) { //Randomly Chose Victim From all Smallest_Rank}
25: R[Setid][Victim]=0
26: E[Setid][Victim]=0
27: return Victim
frequency information has a much lower weight in the overall effectiveness estimation than recency information. The sorting sequence will look like Figure 3.10.

Once the replacement decision has to be made, we search the whole set for combination Rank1. If more than one Rank1 blocks can be found, we randomly chose one to evict (Algorithm1 line 24). If there are no Rank1 blocks in the set, we move on to search Rank2 until we find the block with the smallest rank in that set and evict it.

The worst case here is we search 15 times and find that the smallest Rank is 16. Fortunately, in modern CPU system, a last level cache miss will cost hundreds of cycles before the new block arrives, so the search latency can be completely overlapped by the memory access latency, with no overall performance impact.

### 3.5 Evaluation Environment

#### 3.5.1 Evaluation Platform

To evaluate our proposed replacement strategy, we use a similar simulation platform to that described in the previous chapter. A slight difference is that in this study, a four
3.5 Evaluation Environment

![Figure 3.10. Rank sequence](image)

Table 3.1. Parameters for the simulation framework

<table>
<thead>
<tr>
<th></th>
<th>1 Out of Order Ultra-SPARC III</th>
<th>4 Out of Order Ultra-SPARC III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 ICache</td>
<td>Private 16KB/core 4-Way LRU 2cycle access latency</td>
<td></td>
</tr>
<tr>
<td>L1 DCache</td>
<td>Private 16KB/core 4-Way LRU 2cycle access latency</td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td>16-Way 1M/2M/4M Byte 15/16/18 cycles access latency</td>
<td>Shared 4-Slices 4MB 16-Way 18 cycles access latency</td>
</tr>
<tr>
<td>MSHR</td>
<td>128-entry each tile</td>
<td></td>
</tr>
<tr>
<td>NoC</td>
<td>2D-mesh 1cycle pre hop size of buffer:4</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>4GB, 300cycle access latency DDR-400</td>
<td></td>
</tr>
</tbody>
</table>

core CMP is sufficient to explore how the data sharing behaviour will affect our replacement policy. For this study, we build two platforms, one is a single core platform which can be configured as 1M, 2M or 4M L2 Cache. The other is a four core platform with a fixed 4MB cache. All the cache access latencies are calculated by Cacti6.0 (Muralimanohar et al. 2009). Table 3.1 lists the parameters of our experimental platform.

3.5.2 Benchmark

All applications from the SPEC CPU2006_INT and SPEC CPU2006_FP benchmark suites (Henning 2006) were used to evaluate our proposed replacement policy. We used the native input set, and warmed up the cache with 10 million instructions, after which 1 billion instructions were simulated. To simulate multiple applications with our four-core system, we classify these applications into three groups (light load(L), medium
Table 3.2. Simulation workloads

<table>
<thead>
<tr>
<th>Multi-Programmed Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comb 1</td>
</tr>
<tr>
<td>Comb 2</td>
</tr>
<tr>
<td>Comb 3</td>
</tr>
<tr>
<td>Comb 4</td>
</tr>
<tr>
<td>Comb 5</td>
</tr>
<tr>
<td>Comb 6</td>
</tr>
<tr>
<td>Comb 7</td>
</tr>
<tr>
<td>Comb 8</td>
</tr>
</tbody>
</table>

load(M) and heavy load(H) based on the number of total misses when simulated with the true LRU policy. Eight different combination groups were covered: 1L3H 2L2H 3L1H 3L1M 2L2M 2M2H 4L 4H. Table 3.2 lists all of the combinations of multi-application benchmarks.

### 3.6 Evaluation of EBR

Figure 3.12, 3.11 and 3.13 shows the total number of misses for each of the applications from SPEC CPU 2006 with each of the NRU, LRU and EBR replacement policies, simulated on our single core platform. The total number of misses is normalized to the baseline of the LRU policy (the total number of misses incurred with LRU). The green bar shows the total number of misses, relative to LRU, incurred with EBR (generally less than 1). The red bar shows the additional, relative misses incurred with NRU, generally giving a value greater than 1. The blue bar fills in the gap between EBR and the number of LRU misses (always 1).

Figure 3.12 shows the results for a 2M-byte cache. It is noticeable from this figure that in the best case, EBR achieved a maximum 28.7% miss reduction and in the worst case, EBR produced a 0.97% increase in misses. Out of all applications, there are only five applications that do not benefit from EBR; those applications are all recency dominated. In these cases, including frequency information when making replacement decisions does not help but instead may degrade performance. However, we made the assumption that the priority of recency is much higher than that of frequency, so the penalty due to including frequency information is small.
The geometric mean of the total misses of these five recency-dominated applications when using EBR is only 0.49% higher than real LRU. For all of the 30 applications, EBR reduced the average total number of misses by 3.75%. It is not large reduction since not all applications suffer from thrashing and scan. However, if we consider the top five applications that suffer from thrashing, hammer, xalancbmk, sphinx, soplex and perlbench, a 12.4% reduction in cache misses is achieved, compared to real LRU.

We also considered the effect of various cache sizes by simulating a 1M-byte and a 4M-byte L2 cache. Figure 3.11 shows that when the size of cache is decreased from 2M-byte to 1M-byte, the average miss reduction made by EBR was decreased to 1.63% and when the size of cache is increased from 2M-byte to 4M-byte, as in Figure 3.13, the miss rate improvement for EBR is increased to 4.41%.

Contrary to our expectation, the smallest cache, which suffered most from thrashing, benefits from EBR the least. The reason behind this is that during the simulation, EBR only works if a block can be reused at least once before it is evicted. If the cache is much smaller than the working set size, the cache would enter into a heavy thrashing mode where EBR can not take effect. To fix this problem, we introduce Dynamic EBR, as described in the next section.

### 3.7 Dynamic EBR

When we introduced EBR for the convenience of rank sorting we made that assumption $r/f \gg 1$, so that the priority of frequency is less than that of recency. The simulation results for EBR show that, in most cases, this assumption works well. However, there are two situations in which EBR does not work well.

1. During the simulation using EBR, a block can be promoted to a higher rank only if it was used at least once. In the case of really bad thrashing, upon eviction all the blocks with E-value 3 have the same R-value (0), at which point EBR degrades to a normal pseudo-LRU and is subject to the same thrashing problems.

2. With the intention not to complicate the policy too much and to minimise hardware overhead, we used a two bit saturating counter to record recency. This means we cannot tell if a block with an E-value of 3 was used 12 misses ago or 120 misses ago. In other words, once a block is marked with E-value 3, the ageing system no
Chapter 3  An Effectiveness-Based Adaptive Cache Replacement Policy

Figure 3.11. Normalized total number of misses using EBR with 1MB cache

Figure 3.12. Normalized total number of misses using EBR with 2MB cache

Figure 3.13. Normalized total number of misses using EBR with 4MB cache
3.7 Dynamic EBR

longer makes any difference. A block that was re-used frequently in the past, but because of a change in phase of the application is not required any more, because it has an R-value of 3 will always stay at the top of group 4 and be precluded from replacement if a lower ranked block exists.

As a solution, we introduce Dynamic EBR, (D-EBR) which can tune the ratio between $r$ and $f$ dynamically.

We define the Miss Interval Trigger Threshold (MITT) to be the number of misses between E-counter increments. For EBR MITT was 4.

In the first of the cases described above, when an application is in a heavily thrashing phase, blocks can not stay in the cache long enough to upgrade to a higher rank. In this situation we should adjust the policy to rely more on frequency information. By decreasing MITT, we will increase the number of blocks in group 4 - that is, that have an R-value of 3. For example, if MITT is 1, any block that has not been reused within the time interval of 3 successive misses will fall into group 4. Figure 3.14 shows an example of using MITT = 1. Thirteen out of sixteen blocks will be in group 4, which forms a LFU dominated replacement policy.

The second case normally happens when a LFU-friendly phase is finished, and we would like to remove those dead blocks that have a high R-value. To do that, we need to increase MITT, which will make EBR tend to behave more like LRU. Increasing MITT will make blocks stay longer before they fall into group 4, thus increasing the chance of evicting a dead block with high R-value.

In EBR, MITT is statically set to be 4. When an application shows the above behaviours, it means the assumption $r/f \gg 1$ fails. If we decrease MITT D-EBR will rely more on frequency. When LRU starts to work well again, we can increase MITT. To implement dynamic adjustment of MITT we adopt modified set duelling based on the design proposed by Qureshi (Qureshi et al. 2007). The entire cache will be divided into two interleaved parts. Part A uses MITT = 3 and part B uses MITT = 4. A 10 bit saturating counter called the MITT indicator is initially set to a value of 512. Any miss that occurs in part A will decrease the MITT indicator and any miss in part B will increase the MITT indicator. Once the MITT indicator reaches either 0 or 1024, the losing part will change its MITT value towards that of the winning part by 2, until it reaches the maximum or minimum value. For example, if $MITT_A$ is 3 and $MITT_B$ is 2, if part B wins, $MITT_A$ will change to 1.
Figure 3.14. Incrementing the E-counter with MITT = 1

The range of values that MITT can adopt is from 1 to 4. A MITT greater than 4 would shrink the size of group 4 too much. For example if MITT=5, and the associativity is 16, then the maximum size of group 4 is just $16 - 3 \times 5 = 1$. We set a minimum MITT value of 1 because we are using an inclusive LLC, which means evicting a most recently used block may lead to early eviction at L1, and if we the set minimum MITT to 1, it will take at least three L2 misses before a block falls into group 4 which is long enough to exclude L1 early eviction.

### 3.8 Evaluation of DEBR

Although we set the range of MITT to be between 1 to 4, in order to comprehensively explore how different MITT values will affect the performance, we conducted simulations covering different MITT values from 1 to 8. Figures 3.15, 3.16 and 3.17 show, with EBR, the effect of varying MITT on eight applications falling into three types. The first type resembles 3.15(a) and 3.15(b) which suffer from heavy thrashing. The smaller that MITT is, the more EBR will rely on LFU, and as a result, the better performance it can achieve. For these type of applications, it will be optimal to set MITT to 1.
second type behaves like 3.16(a) and 3.16(b). They are very LRU friendly and do not benefit from including frequency information. For this type, leaving MITT large will ensure that EBR resembles LRU as much as possible, so the performance will not be reduced significantly. The last type is like 3.17(a) and 3.17(b), which are applications that suffer from thrashing, but are also quite LRU friendly. For this type of application, the optimal MITT value will be a mid-range value. In Figures 3.15, 3.16 and 3.17, the red line represents the total number of misses achieved using Dynamic EBR when the simulation is re-run with the same conditions, which in each case is close to the optimal option.

Figure 3.18, 3.19 and 3.20 show that with the help of Dynamic EBR, the total reduction in miss rate compared with real LRU can improve from 1.63% (for EBR) to 3.54% (for D-EBR) for 1M-byte cache. Similarly there is an improvement from 3.75% to 4.81% for a 2M-byte cache and 4.41% to 6.39% for a 4M-byte cache.

3.9 Sensitivity Study for Rank Granularity

Figure 3.21, 3.22, 3.23 and 3.24 show the sensitivity of D-EBR to different levels of complexity.

The X-axis represents different D-EBR configurations, using different sizes of R-counter and E-counter. The Y-axis shows the mean average execution time of all applications from the SPEC CPU 2006 and the upper bound and lower bound representing the best case and worst case that can be achieved by the corresponding configuration.

Two groups were investigated; the first group is fixed associativity with different sizes while the second group is fixed size but with various associativities.

From Figures 3.21, 3.22 and 3.23, we can see that no matter how many bits are used for the E-counter, it is always beneficial to use more R-bits. However, when using more bits for the E counter than necessary, D-EBR will lose its flexibility to be tuned. In the cases when using 4 bit E counter, EBR behaves like a “pure” LRU since it divides each set into 16 subgroups. Hence if we were to use a large E-bit counter than necessary, performance would actually decline due to lack of flexibility.

The only exception here is if we select a R counter with 0 bit, in that case EBR becomes “pure” LFU, with the more bits the better. Again, the overall average performance improvement seems small, because not all applications suffer from the three problems of thrashing, scan and multi-application interference as discussed previously.
Figure 3.15. Normalized total number of misses achieved by EBR with different MITT compared to dynamic EBR (Type 1 applications)
3.9 Sensitivity Study for Rank Granularity

Figure 3.16. Normalized total number of misses achieved by EBR with different MITT compared to dynamic EBR (Type 2 applications)
Figure 3.17. Normalized total number of misses achieved by EBR with different MITT compared to dynamic EBR (Type 3 applications)
3.9 Sensitivity Study for Rank Granularity

**Figure 3.18.** Normalized Total Number of Misses using Dynamic EBR with 1MB cache

**Figure 3.19.** Normalized total number of misses using dynamic EBR with 2MB cache

**Figure 3.20.** Normalized total number of misses using dynamic EBR with 4MB cache
Chapter 3  An Effectiveness-Based Adaptive Cache Replacement Policy

The main objective of D-EBR is to improve the performance of those applications that do suffer from these problems while not undermining those that do not. So, rather than just focusing on average performance, the lower bound and upper bound which represent how much D-EBR can benefit and how much D-EBR can harm should also be considered when evaluating its performance.

In Figures 3.21, 3.22 and 3.23, the average performance when using a 1 bit E-counter is very close to that when using a 2 bit E-counter, but has the disadvantage that it can cause worse performance for LRU-friendly applications.

Overall it appears that using a 2 bit E-counter is the most optimal configuration for a 16 way associative cache. For different associativity, the optimal size of the E-counter varies. Figure 3.24 shows the simulation results for a 4M-byte cache with 64 way associativity. We find that EBR with a 4 bit E-counter no longer behaves like "pure" LRU, and a 2 bit E-counter does not look like the most optimal choice any more. This is because when using a higher associativity, a 2 bit E-counter only divides a set into 4 subgroups, which may be insufficient.

For a 64 way cache, the diversity of recency is much higher than for a 16 way cache, which is why we need more subgroups representing a finer granularity of recency. A 4 bit E-counter which divides a set into 16 subgroups seems more reasonable. Based on our experiments we conclude that the optimal number of E-bits can be estimated as $\frac{1}{2} \times \log_2(\text{Associativity})$ which can provide a useful level of granularity of recency inside each cache set at modest hardware cost.

When we increase the complexity from 4 bits to 5 bits by adding one more R-counter bit, there is not much performance improvement – only 0.18%. If we decrease the complexity from 4 bits to 2 bits (from E2R2 to E1R1), the performance does not degrade significantly either, only 0.7% miss rate increase. However doing that will greatly simplify the hardware complexity. Figure 3.25 shows the total miss rate reduction when simulating multiple applications. Four-bit EBR can achieve a maximum reduction ranging from 1.97% to 11.24%, with an average of 5.08%. With the simpler 2-bit EBR, a similar reduction can achieved (1.24% to 10.1%, average 4.68%).
3.9 Sensitivity Study for Rank Granularity

**Figure 3.21.** Normalized total number of misses compared among different D-EBR complexities for 1M 16way cache

**Figure 3.22.** Normalized total number of misses compared among different D-EBR complexities for 2M 16way cache
Figure 3.23. Normalized total number of misses compared among different D-EBR complexities for 4M 16way cache

Figure 3.24. Normalized total number of misses compared among different D-EBR complexities for 4M 64way cache
3.10 Hardware Implementation and Overhead

3.10.1 Replacement Logic

Figure 3.26 shows the replacement logic for a 4 way associativity cache. The additional hardware is inside the dashed box, comprising a 2 bit comparator for each associativity and a 2 bit rank-counter to traverse the different rank values from low to high. During each iteration, if none of the blocks has the same rank value as the rank-counter (detected by the Exclusive-OR and AND/NAND network), the all-zero-detector will increment the rank-counter and start the next iteration. For an N-way associative cache, the extra replacement logic will be N 2-bit comparators and a 2 bit counter. Fortunately, only one set of replacement logic is required per cache. For example, to build
a 4M-Byte, 16-way associative cache for a 4 core CPU, the hardware overhead we estimate is $4 \times 16 = 64$ 2-bit comparators in the cache plus an additional four 2-bit rank counters. This overhead is negligible compared with the size of a 4 MB SRAM. Extra latency caused by the comparator and rank traversal does not affect read and write speed because it is not on the critical path. The replacement logic will only be activated upon misses, however it takes several hundred cycles to load a new line, so the latency of the replacement logic can be totally overlapped and hence cause no harm to the overall performance.

### 3.10.2 Storage Overhead

To store the E-counters and the R-counters we need two bits of memory per block. For an N-way associative cache, $2N$ bits of memory are required. Table 3.3 shows the comparison between different replacement policies in terms of storage overhead. D-EBR has linear complexity and is comparable to the most efficient of the other replacement policies.
### 3.11 Summary

<table>
<thead>
<tr>
<th>Replacement Policy</th>
<th>Storage Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>$N \times \log_2 N$</td>
</tr>
<tr>
<td>NRU</td>
<td>$N$</td>
</tr>
<tr>
<td>Binary Tree PLRU</td>
<td>$N - 1$</td>
</tr>
<tr>
<td>DIP</td>
<td>$N \times \log_2 N$</td>
</tr>
<tr>
<td>SRRIP</td>
<td>$2 \times N$</td>
</tr>
<tr>
<td>DRRIP</td>
<td>$2 \times N$</td>
</tr>
<tr>
<td>4Bit D-EBR</td>
<td>$4 \times N$</td>
</tr>
<tr>
<td>2Bit D-EBR</td>
<td>$2 \times N$</td>
</tr>
</tbody>
</table>

#### 3.11 Summary

In this chapter, we proposed the idea of future Effectiveness of candidate blocks upon replacement, and redefined the ultimate goal of cache replacement as to maximise the Effectiveness of retained blocks. Effectiveness can be used to explain Belady’s optimal replacement policy. We further considered the two most widely accepted replacement policies, LRU and LFU by using our model and explained why they may perform poorly under different access patterns.

Based on these analyses, we proposed the Effectiveness Based Replacement policy which subdivides each set into several small groups, each of which represents a specific level of recency status, and exploits the local reuse history of the group with the worst recency level used to pick the victim.

We demonstrated that EBR outperforms a baseline real LRU replacement policy by a maximum of 28.71% and an average of 12.41% for all applications in SPEC CPU2006. However in EBR, we assume that the weight of recency information of a block is far more important than the weight of its reuse history information. This assumption fails in certain circumstances, such as when the code is suffering heavy thrashing.

To address this we proposed Dynamic EBR, which can dynamically tune the ratio between the weight of recency information and the weight of reuse history information. From our simulation results, D-EBR can further increase the improvement by 117% for a 1M-byte cache, 28% for a 2M-byte cache and 44% for 4M-byte cache.
We also studied the sensitivity of D-EBR complexity and concluded that a better number of E-bits is $1/2 \times \log_2(\text{Associativity})$ and the more R-bits the better. However, in order to minimise hardware complexity and because adding R-bits produces diminishing performance improvements, we decided to use a 1-bit R-counter and a 1-bit E-counter. From the multi core platform simulation, a 4-bit EBR can achieve a maximum miss rate reduction ranging from 1.97% to 11.24% with an average 5.08%. With a simplified 2-bit EBR, a similar reduction of between 1.24% to 10.1% with an average of 4.68% was achieved. Hardware overhead is discussed by comparing it with different replacement policies. A 2-bit D-EBR has approximately the same hardware overhead as RIPP (Jaleel et al. 2010), which is half that of real LRU.

In the following chapters we will explore using the EBR based replacement policy to predict and estimate cache utility and eventually study how to partition the cache based on each thread’s cache utility. We believe that EBR is more suitable for future scalable tiled structures, not only because of its better performance but also because of the affordable complexity in hardware. Hence we will explore how to estimate cache utility by using this new replacement policy in the following chapter.
In this chapter, we will discuss how to estimate equivalent utility of a cache using EBR replacement policy rather than LRU, and introduce an EBR based resizing scheme. In previous chapters, we successfully verified how significantly a resizing scheme can save power when cache resources are not under good utilisation. However, issues related to how to implement LRU efficiently for high associativity cache made our scheme less attractive. To fix that, we will propose an alternative resizing method which uses EBR replacement policy instead of LRU to dynamically evaluate runtime utilisation of the cache and resize the cache in a similar way as previous chapters. Since EBR replacement policy is hardware economical and cache thrashing protected, it is more suitable for the utility estimation.
4.1 Introduction

The intention of utility based cache resizing is to reallocate cache resource depending on how well these resources can be utilised in the future. Assume we allocated a “virtual cache resource”, for each future cache access, it can have two status pairs: “Real Hit” - “Real Miss” and “Virtual Hit” - “Virtual Miss”. Real hit and miss means this cache access will hit or miss regardless of extra cache allocation, virtual hit and miss means this future cache access will be a hit or miss if the cache was allocated the virtual cache resource.

The job for cache resizing is to evaluate whether it is worthwhile to trade between the power consumption of those virtual cache resources with the number of Virtual Hits in the future. Before the trade-off, one obstacle is how to measure or estimate the number of future “Virtual Hits” with a given amount of virtual cache resource. The conventional way based on LRU stack-distance was discussed in the last chapter.

We can easily work out how many hits can be achieved when given the number of cache associativity ways. However, as we mentioned in chapter 2 and 3, LRU has a very expensive hardware overhead. The hardware complexity is $N \times \log_2 N$.

The effectiveness based replacement policy (EBR) tends to keep blocks with higher effectiveness. Effectiveness represents the relative utility of a block compared with other blocks in the same set. To evaluate run time utility of the cache, we need the absolute utility rather than a relative utility. Unfortunately, EBR can only evaluate which block has the lowest effectiveness compared with other candidates among the set.

Relative effectiveness is sufficient to avoid any bad decision in cache victim selection. However, when talking about utility evaluation for resizing purpose, that is not enough. For example, consider two cases. Case one: The block evicted has an EBR status R:1 E:1. Case two: the block evicted has an EBR R:1 E:3. Can we say the evicted block in case one has higher effectiveness or utility than the evicted block in case two?

The answer is ‘No’. In case one, R:1 E:1 is evicted only because the rest of the blocks have higher rank of EBR status than it. Since these two cases have very different access histories, we can not compare them. The E counter of each set is decided by the number of misses to its own set, which can be very different between sets.
Chapter 4  Cache Dynamic Resize using EBR based Cache Utility Evaluation

To better explore how to estimate cache utility using EBR policy and be able to find a method to replace LRU, we will go back to revisit the underlying philosophy of LRU based utility prediction.

4.2 Further Study of LRU Prediction

In previous chapters, we discussed how to predict future cache misses using LRU stack distance. Here we are going to elaborate in detail why it works, so ideally we can expand it to different types of replacement policy including EBR.

4.2.1 Future Access Trace

The key factor for predicting how many extra hits can be achieved if more cache resources were allocated is to speculate what would be the future access trace. More precisely speaking, what would the access trace look like during the moment when a "block" is referenced the first time to the moment it is re-referenced. With this trace and other information like set-associative size, current replacement policy related status and which particular replacement policy that is being used, we could work out whether or not this block would encounter a cache miss or hit. More importantly we could work out whether or not this block would encounter an extra hit for this block if we re-size the cache.

Figure 4.1 is an example of a set of future access traces. The highlighted part is named the Future-Access-Trace-Interval which is defined as follows: For each cache access, there is a corresponding future access trace interval which starts at the first time the accessed block is referenced and finishes at the time when this accessed block is re-referenced. If the block will never be used again, then we consider this trace has an infinite length.

Trace$_i$ represents the $i_{th}$ future access trace interval and Status$_i$ is snapshot of all the replacement policy related status (listed in table 4.1) at the moment when Trace$_i$ is triggered.

With these two inputs, we would be able to know two things. For each trace, the same block will appear at both the head and tail. So we would be able to know:

1. Is the first reference going to be a hit or miss, let’s call it “current hit/miss status”.

Page 105
4.2 Further Study of LRU Prediction

Figure 4.1. Example of future access trace

Table 4.1. Replacement policy related status

<table>
<thead>
<tr>
<th>Policy</th>
<th>Recency Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td></td>
</tr>
<tr>
<td>Pseudo LRU</td>
<td>Value of each Binary Tree node</td>
</tr>
<tr>
<td>NRU</td>
<td>Value of each None-recently-used bit</td>
</tr>
<tr>
<td>EBR</td>
<td>Value of each E bit and R bit</td>
</tr>
</tbody>
</table>
2. What is going to happen in the future (hit or miss at the end of $Trace_i$). Let’s call this “future hit/miss status”.

For each $Trace_i$, if the referenced block has never appeared before, it will certainly cause an immediate cache miss and future status will depend on $Status_i$ and $Trace_i$.

If the referenced block was accessed before, eg (block C trace 6 in Figure 4.1) , then the “current hit/miss status” of “Trace 6” is actually the “future hit/miss status” of the last access to block C which in this case is “Trace 3”. So, we can ignore the current status and only count future status unless $Trace_i$ was introducing a new block. It is not hard to see that new block misses are actually compulsory cache misses. As we discussed in the previous chapter, compulsory cache misses are dependent on the application and can not be avoided. (In other words, cache replacement policy has no control of compulsory cache misses, cache pre-fetching can alleviate it but it is not considered in this work). Hence, observing the future status of each future access trace is sufficient to evaluate cache performance in the future.

The tricky part of this process is that in most cases, $Status_i$ is related to the size of the cache. In other words, changing the size of the cache will change $Status_i$ when $Trace_i$ arrives. In order to find $Status_i$ if the cache size were different, we have to redo the calculation from the beginning of the trace.

In terms of this model why does LRU with stack distance work? The answer is, $Status_i$ of each trace when using LRU is irrelevant to cache size. The $Status_i$ is also irrelevant to the recency status of the rest of the blocks in the set. The related block of $Trace_i$ will always be promoted to the most recently used position. And whether $Trace_i$ will cause a hit or not at the end of $Trace_i$ is only dependent on the how many different blocks will be accessed within $Trace_i$. We denote this as the length of a trace. Accessing to a different block will only downgrade the recency position of the related block by 1.

If the length of $Trace_i$ is larger than the size of the current set-associativity, the recency position of the related block will be demoted to the lowest position and evicted before being re-referenced. The future status will be a miss in this case. Repeated blocks within $Trace_i$ are not counted because they will not further downgrade the recency position of the related block (e.g: Trace 4, length is 5, “FDCEG ”).

Since $Status_i$ is irrelevant and the length of $Trace_i$ is the only thing we need to predict future “Hit/Miss status”. Knowing the stack distance of each $Trace_i$ is equivalent to recording the length of past traces. A hit at stack distance 3 means three different
blocks were accessed in the last Trace\textsubscript{i}. For example, Trace\textsubscript{3} and Trace\textsubscript{6}. Based on the stack distance from the past, we know what the past traces look like, and we can predict future trace length using heuristics.

For other replacement policies, things can become more or less complicated. \textit{Status}\textsubscript{i} which is such an important factor to predict future status is dependent on history. For example, NRU, current None Recent Used bit status is accumulated from the past and it can affect how future traces change the status. The same applies to other policies like RRIP and EBR.

### 4.2.2 Rank Distance Estimation

\textit{EBR} is a hybrid policy between LRU and LFU. \textit{EBR} creates \(2^{\#E + \#R}\) different ranks, where \#E, \#R represent the number of bits in the E and R counters respectively. A block is promoted upon reference and demoted if it has not been used for a period of time, the length of which depends on the size of associativity. Even given the exact future Trace\textsubscript{i}, future status is still uncertain due to the unknown value of Status\textsubscript{i} (e.g: which status shall we promote the coming block, E:0-R:3 or E:0-R:1? What is the rank of other blocks?). It seems impossible to predict future status due to lack of all of this information.

Fortunately, we know the exact rank of the victim during each eviction. This rank is denoted as evicting-rank and is recorded in the rank counter until the next cache miss.

Whenever a block residing at evicting-rank is re-used, we consider that as a combination. Past Trace\textsubscript{i} and past Status\textsubscript{i} push the related block to the boundary of the cache set but finally it gets re-used before it gets evicted. These combinations are very likely to become a miss, if we decrease the set-associativity by 1 at the early phase. In other word, this cache hit, could be a cache miss if the cache size were smaller. If we can count how many these combinations happened in the past, then we can predict how often these might happen again in the future.

However, this counting is not that straight forward. Upon each re-use, the \textit{EBR} status inside a set can be very different, so the evicting-rank of each set is dynamic and can be very different upon each access. Table 4.2 is an demonstration of how internal \textit{EBR} status was updated during a hypothetical simulation. Table 4.2 emulates a cache with 8-way associativity. For each line in this table, the \textit{EBR} status of every cache way before the transaction is displayed, followed by the number of the way which was going to
**Table 4.2.** Count reuse at evicting rank

<table>
<thead>
<tr>
<th>Way idx</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>Hit at Evicting Rank Hit Counter C</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBR Status before Trans 1:</td>
<td>E:2</td>
<td>E:1</td>
<td>E:3</td>
<td>E:3</td>
<td>E:3</td>
<td>E:2</td>
<td>E:3</td>
<td>E:2</td>
<td>way:3 N, C=0</td>
</tr>
<tr>
<td>EBR Status before Trans 2:</td>
<td>E:2</td>
<td>E:1</td>
<td>E:0</td>
<td>E:3</td>
<td>E:3</td>
<td>E:2</td>
<td>E:3</td>
<td>E:2</td>
<td>way:7 Y, C=1</td>
</tr>
<tr>
<td>EBR Status before Trans 3:</td>
<td>E:2</td>
<td>E:1</td>
<td>E:0</td>
<td>E:3</td>
<td>E:3</td>
<td>E:2</td>
<td>E:0</td>
<td>E:2</td>
<td>way:4 Y, C=2</td>
</tr>
<tr>
<td>EBR Status before Trans 4:</td>
<td>E:2</td>
<td>E:1</td>
<td>E:0</td>
<td>E:3</td>
<td>E:3</td>
<td>E:2</td>
<td>E:0</td>
<td>E:2</td>
<td>Miss N, C=2</td>
</tr>
</tbody>
</table>

be accessed in that transaction, and finally, the indication of whether this transaction is hitting at a block with evicting-rank and the value of a counter which records the total number of re-references to the block with evicting-rank.

During transaction 1, the cache is re-referencing the block at way 3 whose EBR status is E:3 R:2, while at this moment, Way 7 has an EBR status E:3 R:0 which is supposed to be the evicting-rank since it is lower than way 3. So transaction 1 should not be counted as an access to evicting rank and the counter should remain zero.

During transaction 2, the EBR status of way 3 was updated to E:0 R:0 since it was accessed at the previous transaction. Way 7 was going to be accessed this time. Since it is at the evicting-rank, the counter should be incremented to 1.

This transaction can be used to explain our “Rank Distance Estimation”. If we reduce the cache associativity by 1 at the beginning of the simulation, then one of the eight
4.2 Further Study of LRU Prediction

current blocks would vanished. The one that will not survive must be the block at way 7 since it sit at evicting rank which means it has lowest effectiveness as we defined at last chapter, and should have the highest priority to be evicted. Now, this "vanished" block is re-referenced during transaction 2 which means, this is going to be a miss if we reduce the cache associativity by 1 at the beginning of the simulation.

During transaction 3, way 4 was going to be re-referenced. Way 4 has exactly the same EBR status as way 3 at transaction 1. However, since way 7 was updated to E:0 R:0 at the previous transaction, way 4 and way 5 turned out to be the lowest rank at this set, hence this transaction should be counted as an access to the evicting-rank and the counter should be incremented to 2.

It seems that if we can count the number of hits to the evicting-rank, then we can use that count to approximately estimate the number of extra misses if we reduce the cache size by 1 way. However, this assumption is not entirely true. Taking transaction 3 as an example, both way 4 and way 5 sit at evicting-rank, which means, if the size were reduced, we would not be sure which one is going to vanish. Hence there is only a 50% chance that this hit would be a miss if the size were reduced.

This probability depends on the width of the evicting-rank which we define here as the number of blocks concurrently sitting in the cache with evicting-rank at the moment when one of these blocks is re-referenced. All the blocks with evicting-rank have an equal probability of being evicted at the previous cache replacement. Then the probability of incurring an extra miss if reducing 1 way is proportional to the number of total blocks with evicting-rank at the moment when blocks with evicting-rank are re-referenced.

Table 4.3 shows an example of the breakdown of total evicting-rank re-reference. This data was collected after simulating games for 1 Billion instructions with cache associativity 16. It shows how many evicting-rank re-references occurred with different evicting-rank widths. There are no particular reasons why we use this test bench, this is just a demonstration, and any application would do the job.

If we ignore the width of evicting-rank, the estimation of the extra number of misses with associativity 15 would be 7739. However, when we reduced the size of cache by 1 way and re-ran the simulation, there were only 3867 extra misses.

This overestimation was due to the disregard of the width of evicting-rank upon each re-reference. To adjust the estimation, we need to apply a probability coefficient to
different evicting-rank widths. If we use equation 4.1, the estimation would be 3731 which is very close to actual measurement (within 5% margin).

\[
\text{Estimated Extra Misses} = \sum_{1 \leq i \leq 16} \frac{1}{i} \times \text{Number of Hits @ i} \quad (4.1)
\]

Table 4.4 shows another example of the breakdown of total evicting rank reuses. Data was collect from simulating the same test bench but with associativity 5. The total number of reuses at evicting rank is 346342 which is much larger than the previous example. This is understandable because down sizing from associativity 16 to 15 will incur less penalty than resizing from associativity 5 to 4 where the cache size was smaller and hence being used more productively. What looks quite suspicious is that 89% of the reuse happened when the evicting rank size is 5. As expected, the actual measurement
of extra cache misses when downsizing from associativity 5 to associativity 4 varies from this estimation a lot. It was only 28112, 11 times smaller than our estimation.

What is causing such a huge overestimation? We know most recently used blocks have very high probability of being reused immediately compared with other blocks. Table 4.5 and 4.6 show that in most cases, the proportion of MRU re-references among all cache hit is between 50% to 99%.

In our EBR replacement policy, an extreme case exists. When all the blocks are at a high rank, for example E=0 R=3, any following cache hit can no longer promote any block’s rank, then the evicting-rank at that moment was also E=0 R=3, the same as with MRU’s EBR status. In this case, re-referencing the MRU block would be counted as re-referencing the evicting-block. This was happening when, and only when the evicting size is equal to the current associativity. MRU blocks will have the same EBR status as evicting-rank blocks.

When the entire set is updated to the highest rank, any hit to this set would be counted as evicting-rank access. This explains the data in table 4.4. If we remove all the number of hits when evicting-rank width is 5, total counts would be: 346342-309841 = 36501. Compared with the actual measurement, it was still 30% overestimated, but much better than the un-filtered scenario which would cause an estimation 11 times greater. If we further apply equation 4.1, the estimation would be 24136 which was only 15% underestimated.

To get rid of this phenomenon we will have to filter out those fake evicting-rank accesses that are actually MRU block access. The hardware to achieve this is fairly simple and we will discuss that later.

If we can precisely count the number of re-use accesses to the block with evicting-rank and also record the width of evicting rank, we can work out an estimation that is very close to real world. However, recording the size of evicting rank during each access can be quite difficult and expensive both in terms of hardware and power consumption. This leaves us no option but to use the total number of reuses at evicting rank. This overestimation implies a performance orientated preference, meaning that we will not aggressively shut down the cache. In other words, this means when we shut down the cache to save power, it will always end up with less performance loss than we estimated.
Table 4.4. Evicting rank size upon reuse SPEC CPU 2006 gameSS associativity=5

<table>
<thead>
<tr>
<th>Total Number of Hits at Evicting Rank</th>
<th>346342</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Evicting Rank</td>
<td>Number of Hits</td>
</tr>
<tr>
<td>1</td>
<td>16207</td>
</tr>
<tr>
<td>2</td>
<td>9229</td>
</tr>
<tr>
<td>3</td>
<td>6589</td>
</tr>
<tr>
<td>4</td>
<td>4476</td>
</tr>
<tr>
<td>5</td>
<td>309841</td>
</tr>
</tbody>
</table>

Table 4.5. PARSEC2 16MB cache 16 core : MRU re-reference/total cache re-reference

<table>
<thead>
<tr>
<th>Test Bench</th>
<th>MRU/Total Cache re-reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Bench from PARSEC2</td>
<td></td>
</tr>
<tr>
<td>Blackscholes</td>
<td>94%</td>
</tr>
<tr>
<td>Bodytrack</td>
<td>98%</td>
</tr>
<tr>
<td>Canneal</td>
<td>78%</td>
</tr>
<tr>
<td>Facesim</td>
<td>99%</td>
</tr>
<tr>
<td>Ferret</td>
<td>59%</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>84%</td>
</tr>
<tr>
<td>Swaptions</td>
<td>95%</td>
</tr>
<tr>
<td>X264</td>
<td>97%</td>
</tr>
</tbody>
</table>

Figure 4.2 compares the actual measurement of extra cache misses with the estimation with and without the knowledge of evicting rank width when reducing 1 way at different associativity (ranging from 16 to 2). For example, if we reduce the associativity from 16 by 1, we will incur 52961 extra misses, if we reduce the associativity from 2 by 1, we will incur 438282 extra misses. As we can see, the estimation is very close to the measured values, except when associativity is equal to 2. Estimation without knowledge of rank width is always an overestimate and estimation with knowledge of rank width may sometimes be an underestimate.

It is quite surprising, that the estimation with knowledge of rank width is not the best option. It is true that this type of estimation has a smaller absolute difference compared
4.2 Further Study of LRU Prediction

Table 4.6. SPEC CPU2006 4MB cache uni core: MRU re-reference/total cache re-reference

| Test Bench from SPEC CPU2006 | 68 % | 61 % | 59 % | 75 % | 80 % | 97 % | 83 % | 36 % | 61 % | 87 % | 60 % | 77 % | 75 % | 38 % | 69 % | 85 % | 46 % | 98 % | 90 % | 09 % | 85 % | 93 % | 94 % | 70 % | 82 % | 52 % | 89 % | 56 % | 78 % | 98 % |
|-------------------------------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|------|
| astart                        | 68 % |
| bzip2                         | 59 % | 75 % |
| calculix                      | 80 % | 97 % |
| gamed                        | 83 % | 36 % |
| GemsFDTD                      | 61 % | 87 % |
| gromacs                       | 60 % | 77 % |
| hmmr                          | 75 % | 38 % |
| leslie3d                      | 69 % | 85 % |
| mcf                           | 46 % | 98 % |
| namd                          | 90 % | 09 % |
| perlbench                     | 85 % | 93 % |
| sjeng                         | 94 % | 70 % |
| specrand                      | 82 % | 52 % |
| tonto                         | 89 % | 56 % |
| xalancbmk                     | 78 % | 98 % |

Figure 4.2. The number of extra misses that would be incurred by reducing cache associativity by 1 from different cache size. Test-bench: SPEC CPU2006: *gamess*, 1 billion instruction, uni-core, 4 MByte L2Cache
with actual measurement, but underestimation means we may shut off the cache and cause a slightly higher performance penalty than expected or we do not release additional cache resource because we under estimated the potential performance improvement.

Generally speaking, if we can shut down 1 way of cache by 1 cycle, we can save \( \frac{1}{\text{Associativity}} \times \text{Total L2 leakage power} \times \text{cycle time} \) joules of energy, but if we delay the performance by 1 cycle, we will stall the entire chip by 1 cycle and waste \( \text{Total Chip leakage power} \times \text{cycle time} \) joules of energy. An under estimation may lead to a worse overall EDP which is against our research objective. This conclusion saves us the trouble of recording the width of evicting-rank upon each access. We can simply use the estimation without evicting-rank width information.

The only exception, when this method will cause under-estimation, is when associativity equals 2. This is because we filtered out all the evicting-rank accesses when their width equals the current associativity which is 2 in this case. A lot of real evicting-rank accesses were also being filtered since there were only two ways per set and it is very likely that there would be identical EBR status values simultaneously.

The good news is, this underestimation only happens when the cache is under very high utilisation and also has a cache associativity equal to 2, the cache size would be increased long before this underestimation actually happened.
Figure 4.4. The number of extra misses would incur when reduce cache associativity by 1 from different cache size. Compare among actual measurement, estimation with knowledge of evicting rank width and estimation without knowledge of evicting rank width. Test-bench: SPEC CPU2006: *libquantum*, 1 billion instruction, uni-core, 4 MByte L2Cache

Figure 4.3 represents the type of applications that have much lower miss rate. Again, the estimation is quite accurate until associativity equals 2. Although we believe this situation will not happen because the cache size should be increased long before the underestimation happened, for safety purposes, we restrict our tuning scheme to start with associativity 2 rather than 1. There is another important reason why we should set this minimum associativity. If we really start with associativity 1, any L2 hit would be counted as evicting rank access, and the cache size will increase immediately, which would not make any sense.

Figure 4.4 shows another example. Application *libquantum* represents the type of application which has very poor L2 cache utility. The number of L2 misses of *libquantum* is almost 2 orders of magnitude higher than *gamess*, but reducing the size of cache from 4 Mega Byte (16 way associativity) to 512 KByte (2 way associativity) would not cause any extra L2 misses. And as we expected, the estimation at associativity 2 did not show any huge underestimation because the cache was not under good utilisation even when the size was reduced to 512 KB.

With this simple estimation method, we can find out how many misses would probably occur if down-sizing the cache by 1 way, and we also use this estimation to predict how many misses can be saved if we increase the cache size by 1 way.
4.3 Algorithm of EBR based increasing only mechanism

The life cycle of executing a program with increase-only cache resizing looks as follows:

1. Task Pre-emption:
   - Cache initialised to only two activated ways. (write back any dirty blocks in the cache)
   - Set the utility counter (denote as UC) of each tile to 0.

2. Start Execution:
   - During each L2 cache “set-MRU” event which happens after any cache access, check the rank of the promoting block and compare it with the lowest rank stored at the rank counter of each set.
   - If the rank of the promoting block equals the lowest rank, increment the UC by 1.
   - Compare UC with threshold, issue a way allocation signal if UC is greater than threshold.

3. If allocation was triggered, increase the cache by 1 extra way if available.
   - Reset UC to 0.
   - continue execution.

It may be observed that this estimation method is very similar to the LRU based method. However in the next section we will discuss a unique issue only related with the EBR based mechanism.

4.4 False Evicting Rank Reuse

The cause of this problem is that when a cache slice shows good utility and has been assigned extra cache resource, the EBR status of each block in that slice will remain the same. After the new allocation event, blocks located at the lowest rank will remain at the same rank, but that rank was due to the previous cache size. So, any hit to this type of evicting-rank only represents the necessity to increase cache size from the previous cache size. As a result, false last rank reuse may cause false utility estimation and false cache resource allocation.
4.4 False Evicting Rank Reuse

Figure 4.5 shows the way-allocation signal issue timing of each of the 16 individual cores. It shows what would happen if no special treatment were applied to deal with false last rank reuse when simulating bodytrack. At the moment when around 12.5% of the entire 1 billion instructions have been simulation, core 1’s L2 cache need should be 5 out of 16 ways in order to maintain a 99.9% of original performance. Before the cache was increased to 5 ways from 4, the program is running with undersized cache, so a lot of blocks were demoted to lowest rank and are about to be reused very soon. Within that very short period of time, those lowest ranked blocks were reused frequently and each reuse will increased the utility counter by one. The allocation trigger threshold was reached and the cache size was increased easily. However, this sprint was not finished yet. After the way-allocation event, the remaining lowest ranked blocks kept being reused which saturated the utility counter again and again regardless of the fact that the cache had already been up-sized.

This sprint increased the size of cache quickly until it was fully activated. This phenomenon which also happened at core 2, core 3, core 5, core 8, core 11 and core 14 (circled with yellow color) is the so called false evicting-rank reuse.
Ideally, we should only increment the utility counter when there is a cache hit to a block that was demoted to evicting-rank within the current cache size. That is only if a block was demoted to evicting-rank after the resize event and was then re-referenced, then the utility counter should be incremented. One solution is to add an extra state bit to each block representing whether this lowest ranked block was generated with the current cache size.

After a cache size-up, the state bits of all blocks should be reset to default. Once the block was demoted to lowest rank again this bit should be set back to 1. With this state bit, we can selectively increase the utility counter only if there is a cache hit to a block at the lowest rank and also with that state bit as ‘1’.

However, this method involves too much hardware operation overhead. To simplify it, we propose an alternative method. Instead of monitoring each block, we only monitor new incoming blocks. After a way-allocation event, one brand new cache way will be allotted to each set. Any new incoming block will be placed there and have a high EBR status as E=0 R=0. If those new blocks were demoted to evicting-rank, then most of the old blocks should have been demoted to evicting-blocks again even if they have their EBR status refreshed after way-allocation.

We expected that by the time the new incoming block is demoted to lowest rank, most of the old lowest ranking blocks that hold an evicting-rank would become valid again. From that moment, any hit to the evicting-rank block should increment the UC. We add only one state bit per set, called ”New Size Bit”. If this NSB remains ”0”, then we treat all the evicting-rank blocks in this set as if they were generated before the new size hence any hit to those evicting-rank blocks should not increment UC. This will effectively reduce the hardware and operation overhead.

This theoretical behaviour was observed in practice. Figure 4.6 shows how effectively the ”NSB” state bit can prevent false hits. In this example, we re-ran the simulation of bodytrack using the ”NSB”. Even with threshold T=8, the cache size was not saturated at those points we had previously identified.
4.5 Deterministic Evaluation Analyse

4.5.1 Non Deterministic Simulation

To evaluate our design by full system simulation, the ideal way is to run each test-bench from the POI (Point of Interest) which normally is the starting point of parallel execution to the end of the application. This will make sure that we compare two configurations with the same amount of workload. However, running to the end will usually take more than a month for a single iteration using the latest Intel i7 processor computer. In this project, we have chosen to simulate eight applications with a mixture of behaviour in order to cover a sufficient variety of workload characteristics.

Clearly, we can not afford to use a full system simulation approach. An alternative way is to create a checkpoint in the middle of the execution and count the number of instructions executed from that point. In previous chapters we used this evaluation method, and we simulated 1 or 2 billion instructions in those projects. (Bienia et al. 2008) shows there are 30.46 billion instructions in total to finish a Sim-large input of facesim. So, why do we only simulate 1 or 2 billion instructions? The reason is multi-fold.
The most important reason is time consumption. Simulating 1 billion instructions only takes a couple of days. Although we could feasibly have decided to simulate longer instruction sequences, for example 10 billion instructions, non-determinism caused by thread scheduling and branch speculation means that these sequences would not necessarily be more representative of a typical range of workloads than the shorter sequences we used. In short, we decided to use the available time to run more simulations with shorter sequences as a means of understanding application behaviour.

It is well known that the latency required to finish different types of instructions can be very different. For example, the average execution latency and instruction description of UltraSparc3 instructions $\text{add}$ and $\text{smul}$ are listed below. It is obvious that executing 100 $\text{add}$ and 100 $\text{smul}$ are not same amount of work.

1. Instruction : add — avg execution latency 0.813
   Adds the contents of register $\text{regrs1}$ to the contents of a register or to an immediate number $\text{reg_pr_imm}$ and places the result in register $\text{regrd}$.

2. Instruction : smul — avg execution latency 2.476
   Signed multiply $\text{regrs1}$ with $\text{reg_pr_imm}$ and places the 32 most significant bits of the result in $\%y$ and the rest in $\text{regrd}$.

Table 4.7 is an example showing how differently the execution path can be for two identical simulations. These two tables show the number of instructions executed by different execution units on logical Core1 and their ratio to total instruction counts, collected after each simulation.

According to the table, the same checkpoint and the same configuration will make few differences. When we compare Run-1 and Run-2, we are not comparing the same amount of workload. CPI of Run-2 is 6.72 while Run-1 is 6.79, but we can not say Run-2 is 1% faster because we can not compare average over different workloads.

To compare very different architectures, this phenomenon should not matter. But we are trying to prove we can lower the static power while maintain performance loss as low as 0.5%, so this non-deterministic variation matters a lot. Studies like (Hower et al. 2011) and (Alameldeen and Wood 2003) also discussed this issue.

Our solution for above problem in previous chapters is to run each simulation ten times and then average them. That is the reason why simulating 10 billion instructions is not feasible for this research.
4.5 Deterministic Evaluation Analyse

Table 4.7. PARSEC2: streamcluster 16MB cache 16 core: run 1 and run 2

<table>
<thead>
<tr>
<th>Execution Unit</th>
<th>Ratio to Total Instruction counts Run1</th>
<th>Retired Number of Instructions Run1</th>
<th>Ratio to Total Instruction counts Run2</th>
<th>Retired Number of Instructions Run2</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>2.8%</td>
<td>3445816</td>
<td>3.0%</td>
<td>3685616</td>
</tr>
<tr>
<td>Integer ALU</td>
<td>20.8%</td>
<td>25718047</td>
<td>21.2%</td>
<td>26358902</td>
</tr>
<tr>
<td>Integer Multiplexer</td>
<td>0%</td>
<td>2869</td>
<td>0%</td>
<td>3050</td>
</tr>
<tr>
<td>Integer Divider</td>
<td>0%</td>
<td>1138</td>
<td>0%</td>
<td>1209</td>
</tr>
<tr>
<td>Branch</td>
<td>9.0%</td>
<td>11160420</td>
<td>9.4%</td>
<td>11751289</td>
</tr>
<tr>
<td>Float Add</td>
<td>20.6%</td>
<td>29230735</td>
<td>23.2%</td>
<td>28923429</td>
</tr>
<tr>
<td>Float Cmp</td>
<td>0.1%</td>
<td>113306</td>
<td>0.1%</td>
<td>112117</td>
</tr>
<tr>
<td>Float Cvt</td>
<td>0.1%</td>
<td>113311</td>
<td>0.1%</td>
<td>112123</td>
</tr>
<tr>
<td>Float Multiplexer</td>
<td>11.8%</td>
<td>14615380</td>
<td>11.6%</td>
<td>14461728</td>
</tr>
<tr>
<td>Read Port</td>
<td>29.9%</td>
<td>36970193</td>
<td>29.7%</td>
<td>36993646</td>
</tr>
<tr>
<td>Write Port</td>
<td>1.8%</td>
<td>2206094</td>
<td>1.8%</td>
<td>2223443</td>
</tr>
<tr>
<td>Total Retired Instructions</td>
<td>123577309</td>
<td></td>
<td>124626552</td>
<td></td>
</tr>
<tr>
<td>Total number of cycles</td>
<td>839073528</td>
<td></td>
<td>837821961</td>
<td></td>
</tr>
<tr>
<td>CPI</td>
<td>6.78</td>
<td>CPI</td>
<td>6.72</td>
<td></td>
</tr>
</tbody>
</table>

In this chapter, we will evaluate 8 applications with 4 different set of configurations:

1. LRU Increase_only
2. LRU Increase_reduce
3. EBR Increase_only
4. EBR Increase_reduce

That will cost roughly 50 hours each run each iteration. Overall it will cost about 16000 machine hours (equation 4.2). This method was used in our previous chapters, but in this chapter, we will use another alternative to avoid the extremely long evaluation.

\[
50(\text{Hours/Run}) \times 8(\text{Apps}) \times 4(\text{Configs}) \times 10(\text{Iteration/run}) = 16000 \text{ Machine Hours}
\] (4.2)
4.5.2 Trace Driven Evaluation

In this chapter, we propose to use trace driven method which can provide a deterministic way to evaluate our design. We generate a 2 billion instruction trace using the base configuration from each check point. When evaluating a novel cache architecture, we simply feed the trace into Ruby (memory system) and collect the total stall latency. Since it is trace based, a deterministic execution path is guaranteed so the difference of total stall latency can be used to evaluate two configurations. The smaller the total stall latency the better the performance.

Total stall latency difference is good enough to distinguish which one is better but not sufficient to quantify the performance difference. In this chapter we will only be concerned with how many L2 misses will be generated because of a smaller size of cache and how much extra execution time results from these L2 misses. Here we use an estimation to achieve that.

\[
\text{Average CPI} = \sum_{1 < i < \text{Number of Cores}} \frac{CP_{i}}{\text{Number of Cores}} \times \frac{1}{\text{Number of Cores}} \tag{4.3}
\]

Equation 4.3 is the average CPI of the entire CPU running with a base configuration. When the same trace is applied to the new configuration, Latency\_L2\_Miss\_i represents the extra L2 miss latency caused by this new configuration at tile i. These extra Latency\_L2\_Miss\_i values are not identical because L2 misses are not evenly distributed. However, even though the L2 misses are not precisely evenly distributed, they should be close to a uniform distribution because our test program does not treat mapping differently. For that reason, as an estimation, we expect the overall slowdown in average will be near \( \frac{1}{\text{Number of Cores}} \) of the total extra L2 cache stall latency depending on how heavily the threads are synchronised. We will revisit this issue later when we talk about applications with intensive internal communication.

So the estimation model we will use in the following sections is given by Equation 4.4.

\[
\text{Estimated Performance} = \text{Base Performance} + \sum_{1 < i < \text{Number of Cores}} \frac{\text{Latency}\_L2\_Miss\_i}{\text{Number of Cores}} \tag{4.4}
\]

To verify this estimate, we selected two applications: blackscholes and streamcluster. Blackscholes represents applications with fewer synchronisations. (Bienia et al. 2008)
4.5 Deterministic Evaluation Analyse

**Table 4.8. PARSEC2: blackscholes: performance estimation verification**

<table>
<thead>
<tr>
<th></th>
<th>Blackschole Full Size(Base)</th>
<th>Blackschole Half Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Total Cycles</td>
<td>411100230 cycles</td>
<td>411649707 cycles</td>
</tr>
<tr>
<td>Standard Divination Total Cycles</td>
<td>259511 cycles</td>
<td>198770 cycles</td>
</tr>
<tr>
<td>Average Extra L2 Misses</td>
<td>0</td>
<td>11993</td>
</tr>
<tr>
<td>Average L2 Miss Latency</td>
<td>700 cycles</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.9. PARSEC2: streamcluster performance estimation verification**

<table>
<thead>
<tr>
<th></th>
<th>Streamcluster Full Size(Base)</th>
<th>Streamcluster 14M Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Total Cycles</td>
<td>844833751 cycles</td>
<td>848876938 cycles</td>
</tr>
<tr>
<td>Standard Divination Total Cycles</td>
<td>4342376 cycles</td>
<td>3975264 cycles</td>
</tr>
<tr>
<td>Average Extra L2 Misses</td>
<td>0</td>
<td>114433</td>
</tr>
<tr>
<td>Average L2 Miss Latency</td>
<td>700 cycles</td>
<td></td>
</tr>
</tbody>
</table>

shows there are only 8 barriers during the entire sim-large execution. *Streamcluster* represents applications with many more synchronisations. 183 Locks, 129,584 Barriers and 115 Conditions were found when simulating sim-large of *streamcluster*.

For each application, we selected two different configurations.

1. **Blackscholes** $\iff$ 16Mbyte(16WayAssociativity, 1MBytepercore)L2Cache
2. **Blackscholes** $\iff$ 8Mbyte(8WayAssoiciativity, 512KBytepercore)L2Cache
3. **Streamcluster** $\iff$ 16Mbyte(16WayAssoicativity, 1MBytepercore)L2Cache
4. **Streamcluster** $\iff$ 14Mbyte(14WayAssoicativity, 896KBytepercore)L2Cache

We select a larger cache variance when testing *blackscholes* because it has memory-economical behaviour, thus, only by making a big cache size difference can we detect performance difference. On the other hand when testing *streamcluster*, the L2 cache was accessed intensively, so a small variation in configuration would be enough to make noticeable differences.

Again, we run each test 10 times and average them. Tables 4.8 and 4.9 show the average performance of both configurations. Applying our estimation model we will have:
Chapter 4 Cache Dynamic Resize using EBR based Cache Utility Evaluation

1. Estimated performance of Blackscholes half size is: $411100230 + 700 \times \frac{11993}{16} = 411624923.75$

2. Estimated performance of Streamcluster 14M Byte is: $844833751 + 700 \times \frac{114433}{16} = 849840194.75$
   Estimated performance / Real performance = $849840194 / 848876938 = 100.11\%$.

By now, all the above discussion are based on the assumption of an in-ordered core which means a L2 Cache miss will stall the entire core, so there are no instruction level stall latency overlap. If considering instruction-level parallelism then the estimation model should change to 4.5.

\[
\text{Estimated Performance} = \text{Base Performance} + \sum_{i=1}^{\text{Number of Cores}} \frac{\text{Latency}_{L2-Miss_i}}{\text{Number of Cores} \times R_{apo}}
\]  \hspace{1cm} (4.5)

Average pipeline overlap rate ($R_{apo}$) is the ratio that represents how much a large memory stall latency can be hidden by out of order execution. Without this rate, the estimated performance penalty will be overestimated. It means the actual speed should be faster than our estimation. As we mentioned before, that is acceptable. In this chapter, we try to prove minor performance loss with our design and if we can prove the over estimated performance lost is also acceptable, then that is even better.

Now, let’s consider situations when applications have even more synchronisations. If multi-threads are synchronised by a lock or, for example, thread A is interlocked with thread B, it is very likely that both threads will be slowed down when only one of them is stalled. It is almost impossible for us to estimate accurately how much each one will be slowed down. This characteristic is highly application related. We will have to run the simulation multiple times to estimate, which will defeat the purpose of our estimation model. The issue becomes “How can we efficiently estimate the performance loss of an application with intensive synchronisations”.

A synchronised application can benefit from thread level parallelism better. In the previous verification, we deliberately avoided thread level parallelism. We only scheduled 1 thread to each core. When there is a large L2 cache miss stall, the core can do nothing but wait. If more than one thread per core were scheduled, the scheduler could
4.6 LRU-based vs EBR-based Increasing Scheme

Table 4.10. PARSEC2: ferret: performance estimation verification

<table>
<thead>
<tr>
<th></th>
<th>Ferret Full Size(Base)</th>
<th>Ferret 14M Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Total Cycles</td>
<td>335907410 cycles</td>
<td>340175079 cycles</td>
</tr>
<tr>
<td>Standard Divination Total Cycles</td>
<td>2627693.5 cycles</td>
<td>2075347.0 cycles</td>
</tr>
<tr>
<td>Average Extra L2 Misses</td>
<td>0</td>
<td>175359</td>
</tr>
<tr>
<td>Average L2 Miss Latency</td>
<td>700 cycles</td>
<td></td>
</tr>
</tbody>
</table>

swap the waiting thread with another one. As a result, the long cache miss penalty could be hidden by leveraging thread level parallelism. To verify that, we ran ferret with 16M Byte L2 as base and 14M byte L2 10 times since this application involves heavy thread level parallelism. The application was scheduled to 256 threads.

Applying the data in table 4.10 to our model, we have:

1. Estimated performance of Ferret 14M Byte is: $335907410 + 700 \times 175359 \div 16 = 343579366$

   Estimated performance / Real performance = 101%.

Due to thread level parallelism, our estimated performance penalty will only overestimate but not underestimate. As we have discussed before, an overestimated potential performance loss implies a performance orientated preference. This means that we will not aggressively shut down the cache. So, in the following section, we will use this model to evaluate how our design can save power, ideally without degrading performance.

4.6 LRU-based vs EBR-based Increasing Scheme

Figure 4.7 shows the comparison between the LRU based increasing scheme and the EBR based increasing scheme. Different thresholds ranging from $T=8$ to $T=64$ were evaluated. In most cases EBR based increasing shows better performance. This is mainly because of the advantage of the replacement policy itself. EBR can produce a lower miss rate than LRU because it can utilise the limited storage more efficiently. On average, EBR based increasing has a slightly higher leakage cache power consumption (1% higher). This means EBR based increasing is more likely to trigger way increasing
Chapter 4 Cache Dynamic Resize using EBR based Cache Utility Evaluation

Figure 4.7 Increased only mechanism, LRU and EBR based comparison in performance, L2 leakage power, total power, EDP.
with the same threshold. That is because LRU increments the utility counter by 1 every time the LRU was re-referenced. It can be guaranteed that a re-reference would become a cache miss if we reduce the cache associativity by 1. On the other hand what EBR does is increment the utility counter by 1 every time the lowest rank is re-referenced. If more than one block resides at the lowest rank, then the probability of causing a cache miss is proportional to the number of blocks residing at that rank. In other words, this is due to the overestimation of performance penalty when considering the width of evicting-rank as we described in the previous section. It increases the saturating speed.

To better understand how our scheme can tune the cache, Figures 4.8, 4.10, 4.12, 4.14 and Figures 4.9, 4.11, 4.13, 4.15 depict the runtime heat-map of stack-distance per core with full cache size. Each sub-graph shows the local L2 slice on that core. The vertical axis represents the depth of re-referencing stack for any time during the execution. We represent the number of reuses by the colour of heat. The dimmer the colour the lower the number of reuses are. The brighter the colour the higher the number of reuses at that stack depth.

By using these heat maps, we can gain an insight into how those applications utilise cache during runtime. From first row to last row and left to right the core index increments from core 1 to core 16.

For example, in Figure 4.10, we observe that for most of the cores, cache was under utilised, especially in the early stages. Up to close to 75% of the entire simulation, only a quarter of the cache was under good utilisation for most of the cores. Even after this stage, a cache with one third of the original size is adequate (the hot area only covered up to 1/3 of the stack depth).

Then we plot the timing of each cache way-allocation event that occurred at each tile with different threshold values, on top of those heat maps. These stair-shaped curves reveal the detail of how our tuning scheme changes the size of each cache slice on its local tile.

Figures 4.8, 4.10, 4.12, 4.14 and Figures 4.9, 4.11, 4.13, 4.15 compare the tuning procedure based on LRU and EBR. They show a very similar pattern except that, as we discussed before, with the EBR based scheme it turns out to be slightly easier to trigger a way allocation. Both of them show that cache size was tuned according to run time cache utility. Schemes with different threshold demonstrated different power-performance trade-offs.
Figure 4.8. Runtime heatmap of cache utility during LRU based increase-only resize when running Blackscholes. Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.
Figure 4.9. Runtime heatmap of cache utility during EBR-based increase-only resize when running Blacksholes. Y-axis represents cache associativity which ranges from 2 to 16. The heat color represents the cache reuse would happen at that stack depth during runtime. On top of the heatmap is the timing of cache way-allocation event happened at each tile with different thresholds during runtime.
Figure 4.10. Runtime heatmap of cache utility during LRU based increase-only resize when running Bodytrack. Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.
Figure 4.11. Runtime heatmap of Cache Utility during EBR based increase-only resize when running Bodytrack. Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.
Figure 4.12. Runtime heatmap of Cache Utility during LRU based increase-only resize when running Canneal, Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.
the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime which ranging from 2 to 16. The heat color represents the cache reuse would happen at that exact depth during runtime. On top of that, Y axis represents cache associativity.

Figure 4.13. Runtime heatmap of Cache Utility during EBR based increase-only resize when running Canneal. T=64
Figure 4.14. Runtime heatmap of Cache Utility during LRU based increase-only resize when running X264. Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.
LRU-based vs EBR-based Increasing Scheme

The heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime which ranging from 2 to 16. The heat color represents the cache reuse would happen at that stack depth during runtime. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime when running X264. Y axis represents cache associativity.
Table 4.11. Average EDP of all 8 applications from PARSEC2 achieved by different threshold with EBR based increasing only tuning mechanism and LRU based increasing only tuning mechanism, both normalized to LRU running with full size cache

<table>
<thead>
<tr>
<th></th>
<th>EBR:EDP normalized to full size LRU</th>
<th>LRU:EDP normalized to full size LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>T=8</td>
<td>95.2%</td>
<td>95.8%</td>
</tr>
<tr>
<td>T=16</td>
<td>94.5%</td>
<td>95.2%</td>
</tr>
<tr>
<td>T=32</td>
<td>93.7%</td>
<td>94.5%</td>
</tr>
<tr>
<td>T=64</td>
<td>93.3%</td>
<td>93.7%</td>
</tr>
</tbody>
</table>

Table 4.11 shows the similarity of the LRU based scheme and the EBR based scheme. Since the EBR replacement policy itself is superior to LRU in terms of the application performance, that superiority will be reflected in the overall EDP. As a result, we can not compare how similar in performance the LRU based scheme and EBR based scheme can behave. To eliminate the performance improving impact gained from replacement policy itself, we compare the EDP of the EBR based tuning scheme normalized to full size EBR with the EDP of the LRU based tuning scheme normalized to full size LRU. Table 4.12 shows that for different thresholds, EBR based and LRU based schemes achieved very similar normalized EDP. The higher the threshold, the slightly better the average EDP.

However, even though higher threshold values can further lower the average EDP, this is not the main purpose. The main function of the threshold is to determine the trade-off between instantaneous power and performance. Table 4.13 shows an example of this trade-off.

When we include the performance factor achieved by EBR replacement, the average EDP is even better, shown at table 4.11 where we average the EDP of all 8 applications and normalize it to the EDP with LRU replacement policy and full cache size.

4.7 EBR-based Reducing Scheme

In a similar manner to our discussion in chapter 2, the EBR based increasing-only scheme also faces the same problem called cache saturation. Any temporary utility
4.7 EBR-based Reducing Scheme

Table 4.12. Average EDP of all 8 applications from PARSEC2 achieved by different threshold with EBR based increasing only tuning mechanism and LRU based increasing only tuning mechanism, both normalized to their respective full size cache

<table>
<thead>
<tr>
<th></th>
<th>EBR:EDP normalized to full size EBR</th>
<th>LRU:EDP normalized to full size LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>T=8</td>
<td>95.8%</td>
<td>95.8%</td>
</tr>
<tr>
<td>T=16</td>
<td>95.2%</td>
<td>95.2%</td>
</tr>
<tr>
<td>T=32</td>
<td>94.4%</td>
<td>94.5%</td>
</tr>
<tr>
<td>T=64</td>
<td>93.9%</td>
<td>93.7%</td>
</tr>
</tbody>
</table>

Table 4.13. Trade off between power and performance of bodytrack with different threshold

<table>
<thead>
<tr>
<th></th>
<th>Normalized Power</th>
<th>Normalized Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>T=8</td>
<td>95.5%</td>
<td>100.1%</td>
</tr>
<tr>
<td>T=16</td>
<td>94.1%</td>
<td>100.3%</td>
</tr>
<tr>
<td>T=32</td>
<td>92.6%</td>
<td>100.6%</td>
</tr>
<tr>
<td>T=64</td>
<td>91.4%</td>
<td>100.9%</td>
</tr>
</tbody>
</table>

boost or incorrect estimation will irreversibly activate local cache. Eventually, the entire cache would be fully activated even though the application has already passed the phase in which it needs more cache resource.

In Figures 4.8, 4.10, 4.12, 4.14 and Figures 4.9, 4.11, 4.13, 4.15, we can see how easily cache size was increased even when the heatmap shows that the cache was under low utilisation, especially when the threshold was equal to 8 or 16. To solve this, we propose an extended resizing algorithm similar to the reducing algorithm at chapter 2 to dynamically reduce cache size.

We add one state-bit per tile called the “Reducing State Bit”(RSB), to indicate that cache resource is over allocated on that tile. After each \( T_{\text{interval}} \), we compare the value of the utility counter on each tile with half the value of the threshold. If the cache utility of a certain tile can not even demonstrate that it is higher than half of the goal, we should start to consider reducing cache size. However, to make the decision of shutting down, we need more confidence. To add some assurance that we will not turn off the cache too aggressively, we use the state machine implemented as Figure 4.16.

The algorithm of reducing cache size is given by the pseudo code below.
1: PSEUDO code for Reduce:
2: Execution Start: RSB = false; { //Set RSB to be false at the beginning of the execution }
3: while true do
4:   if T mod $T_{interval}$ == 0 then
5:     Read UC from Utility Counter { //peek at every $T_{interval}$ }
6:     if UC smaller than $Threshold/2$ then
7:       Read RSB from Reducing State Bit { //Compare value of Utility Counter with half of the threshold }
8:       if RSB==true then
9:         Reduce cache size by 1 way; { //RSB remained true for the two intervals, reduce should begin }
10:       end if
11:     end if
12:   end if
13:   if RSB==false then
14:     RSB = true;
15:   end if
16:   if UC greater than $Threshold/2$ then
17:     if RSB==true then
18:       RSB = false; { //Utility of recent interval getting better, cancel the intention to reduce size }
19:     end if
20:   end if
21:   T++;
Figure 4.17 compares LRU based downsizing alongside EBR based downsizing. As we can see, performance on average is slightly higher for the decreasing methods, compared with increasing only method. Since evicting blocks which will be used in the distant future will lower the performance eventually, keeping those blocks is a trade off between large leakage consumption and small performance gain. Exceptions shown in Figure 4.17 are canneal, ferret and streamcluster. These three applications do not show any slowdown since no reducing event was triggered. These three applications shown a good cache utility through the entire execution, hence not a single chance arose to trigger cache downsizing. Noticeable static power savings for the remaining applications are due to shutting down cache when cache utility dropped.

Table 4.14 compares the average total power when executing with different configurations, normalized to a base configuration using LRU and full size cache. The results seem quit disappointing as the increase/reduce methods improve power consumption of the increase only methods by less than 1%.
Figure 4.17. Increase/reduce mechanism, LRU and EBR based comparison in performance, L2 leakage power, total power, EDP
4.8 EBR Reduced Evaluation and Comparison with LRU based Reducing

Table 4.15. Threshold T=8, the total power comparison between EBR based increasing only and EBR based increase/reduce

<table>
<thead>
<tr>
<th>Application</th>
<th>EBRbasedIncrease/Reduce ÷ EBRbasedIncreasingonly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>98.8%</td>
</tr>
<tr>
<td>Bodytrack</td>
<td>97.6%</td>
</tr>
<tr>
<td>Canneal</td>
<td>100%</td>
</tr>
<tr>
<td>Facesim</td>
<td>96.3%</td>
</tr>
<tr>
<td>Ferret</td>
<td>100%</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>100%</td>
</tr>
<tr>
<td>Swaptions</td>
<td>98.7%</td>
</tr>
<tr>
<td>X264</td>
<td>97.5%</td>
</tr>
</tbody>
</table>

However, there are three points we should be aware of. First, total cache leakage power is only around 15%-20% depending on the run time dynamic power of each application. Second, the increasing only method is a size shrinking method, which has already made the run time cache utility more compact than original, which means a small improvement is reasonable. Finally, downsizing is only helpful if the application demonstrated obvious phase changing. Within the 8 applications we averaged, 3 applications did not improve as we discussed before. Table 4.15 shows the further power saving by EBR downsizing compared with EBR increasing only when the threshold is 8.

Again, the main purpose of downsizing is not to extract even more power saving, but to prevent cache from saturating when using increasing only method, hence ensuring that the power saving already achieved can be maintained. The total L2 usage looks like equation 4.6. We define $Time_{Trigger}$ as the time when way $i$ is activated and $Time_{Finish}$ as the time when execution finishes.

\[
\text{Total L2 Usage} = \sum_{i} \text{Average L2 Usage of Core}_i \quad (4.6a)
\]

\[
\text{Average L2 Usage Core}_i = \sum_{1}^{\text{Max Associativity}} \frac{1}{\text{Execution Time}} \times (Time_{Finish} - Time_{Trigger}) \quad (4.6b)
\]
Table 4.16. Average L2 usage using EBR based increasing only method and the average activated L2 at the end of simulation

<table>
<thead>
<tr>
<th>Application</th>
<th>Average L2 Usage</th>
<th>Number of Activated Ways</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>7.96/16</td>
<td>9.75/16</td>
</tr>
<tr>
<td>Bodytrack</td>
<td>11.37/16</td>
<td>15.69/16</td>
</tr>
<tr>
<td>Canneal</td>
<td>15.86/16</td>
<td>16/16</td>
</tr>
<tr>
<td>Facesim</td>
<td>8.95/16</td>
<td>13.5/16</td>
</tr>
<tr>
<td>Ferret</td>
<td>15.32/16</td>
<td>16/16</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>15.75/16</td>
<td>16/16</td>
</tr>
<tr>
<td>Swaptions</td>
<td>6.03/16</td>
<td>9.94/16</td>
</tr>
<tr>
<td>X264</td>
<td>10.73/16</td>
<td>15.31/16</td>
</tr>
</tbody>
</table>

Table 4.16 shows, when using EBR based increasing only method, the average L2 usage for the entire execution of 2 billion instructions and the L2 status at the end of 2 billion instructions. For example facesim, the average L2 usage over entire 2 billion instruction is only 8.95 ways out of 16, but when approaching the end of the execution of 2 billion instructions, averaged active cache size is 13.5. It is very likely that in the next few billion instructions, cache will be fully activated and will remain fully activated all the way to end of the execution. Table 4.17 shows, when using EBR based downsizing method, the average L2 usage of the entire execution for 2 billion instructions and the L2 status at the end of the 2 billion instructions. As we can see, after 2 billion instructions, applications like facesim and swaptions remain at a low active rate.

Figure 4.18 compares the performance, L2 leakage power, total power and EDP of both increasing only and increase/reduce method based on EBR replacement policy. For all the thresholds we tested, the increase/reduce method always has a lower EDP. More importantly, as we explained above, if we continue the simulation further, the longer we simulate, the more significant the EDP reduction would be.

Figures 4.19, 4.20, 4.21 and 4.22 revealed how the tuning activity was conducted during runtime. Compared with Figures 4.8, 4.10, 4.12, 4.14 and Figures 4.9, 4.11, 4.13, 4.15, we find that cache size was not easily saturated. Whenever proceeding towards a hot spot which means cache will be effectively utilised, the tuning scheme can quickly adjust and increase cache size by emitting way allocation signals. When the program phase changes and the cache utility drops, the tuning scheme can also detect the change
Figure 4.18. Comparison of performance, L2 leakage power, total power, EDP between increasing only mechanism and increase/reduce mechanism.
Table 4.17. Average L2 usage using EBR based Increasing/Reduce method and the Average activated L2 at the end of simulation

<table>
<thead>
<tr>
<th></th>
<th>Average L2 Usage</th>
<th>Number of Activated Ways</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>6.56/16</td>
<td>7/16</td>
</tr>
<tr>
<td>Bodytrack</td>
<td>8.87/16</td>
<td>11.25/16</td>
</tr>
<tr>
<td>Canneal</td>
<td>15.86/16</td>
<td>16/16</td>
</tr>
<tr>
<td>Facesim</td>
<td>5.53/16</td>
<td>5.25/16</td>
</tr>
<tr>
<td>Ferret</td>
<td>15.32/16</td>
<td>16/16</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>15.75/16</td>
<td>16/16</td>
</tr>
<tr>
<td>Swaptions</td>
<td>4.82/16</td>
<td>6.94/16</td>
</tr>
<tr>
<td>X264</td>
<td>8.11/16</td>
<td>9.94/16</td>
</tr>
</tbody>
</table>

and reduce cache size gradually. More importantly, compared with the increase only scheme, the average cache size at the end of the simulation was much lower, which means if continuing to simulate from this point, the increase/reduce scheme will benefit from much more static power saving.

4.9 Hardware Overhead Analysis

In this section, we discuss the hardware implementation and its overhead compared with EBR based non-resizable replacement policy. The comparison of hardware cost between EBR and other common replacement policies was discussed in chapter 3. As shown in Figure 4.23, the structure is derived from the structure shown in Figure 3.26. The additional components are a set of NAND gates marked inside the blue dotted rectangle. The whole structure is based on an all-zero-dictator (a combination logic circuit to detect if all bits are zero).

The rank counter is initialized by minimum rank, e.g., R:0 E:3 for a 2R2E EBR system. Upon each victim selection, the “E” and “R” values of every block are compared with the value of rank counter. If any block has the same “E R” status with the rank counter, the output of the equality comparator will NAND with the enable signal from the on/off controller. If the result is 0, that means this block’s EBR status is identical to rank counter and it is a valid block from an activated way. In a similar way to a NRU
Figure 4.19. Runtime heatmap of cache utility during EBR based increase/reduce resize when running Blackcholes. Y axis represents cache associativity which ranges from 2 to 16. The heat color represents the cache reuse would happen at that stack depth during runtime. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.
Figure 4.20. Runtime heatmap of cache utility during EBR based increase/reduce resize when running bodytrack. Y-axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.
The heat map is the timing of cache way-allocation event happened at each tile with different thresholds during runtime, which ranging from 2 to 16. The heat color represents the cache reuse would happen at that stack depth during runtime. On top of that, Y axis represents cache associativity.

Figure 4.21: Runtime heatmap of cache utility during EBR based increase/reduce resize when running canneal. Y axis represents cache associativity.
Figure 4.22. Runtime heatmap of cache utility during EBR based increase/reduce resize when running x264. Y axis represents cache associativity which ranging from 2 to 16, the heat color represents the cache reuse would happen at that stack depth during run-time. On top of the heat map is the timing of cache way-allocation event happened at each tile with different threshold during runtime.
4.10 Dynamic EBR resizing

implementation (Sun Microsystems 2007), each L2 bank has a single rotating replacement pointer which could be used as a starting point. This rotating pointer will point to a random block from a, b, c, d. From this point, we search for a block whose EBR status is identical to the rank counter and in the meantime has a high enable signal. This rotating pointer is mainly used to introduce some randomness, one for each L2 bank is enough. We can use any event to increment this rotating pointer, for example L2 hit or L2 miss.

If none of the enabled blocks have the same EBR status as the rank counter, the zero detector will increment the rank counter and start to search a higher rank. The rank counter does not need to reset unless an E counter increment activity was triggered. The On/Off controller was the same one as shown at Figure 2.6. If a way was marked as off by the On/Off controller, the enable signal will remain low hence that block will always be marked as 1 after the new NAND gate.

Compared with fix sized EBR replacement policy, there is only one NAND per cache way extra. Each L2 bank requires one replacement scheme, each replacement scheme needs an associativity NAND gate. So the total overhead in terms of hardware is only: Number of Cores × Associativity × 4 transistors which is negligible. The latency introduced by the extra NAND gate is overlapped by the extremely long cache miss. As long as a victim can be selected before the new block arrives which usually takes several hundred cycles, no penalty would occur. As a conclusion, EBR replacement policy based cache resizing can be achieved with nearly no hardware and performance overhead.

To filter out any evicting-rank access when the evicting-rank width equals associativity we need to add more simple logic. The reason why the width of evicting-rank can be identical to associativity is because all the blocks were promoted to high rank. These situation often happens when the evicting-rank has a 0 E-bit. To implement such a filter simply requires the E-bit to be compared with zero before the LJC counter is incremented.

4.10 Dynamic EBR resizing

In the previous chapter, we introduced dynamic EBR replacement in order to tune the replacement policy, making it more inclined to behave like either LRU or LFU depending on the characteristic of the application. In previous sections, we have not discussed
how the MITT interval will effect the tuning process. As introduced in the previous chapter, MITT interval is used as an aging mechanism. Each set records the number of misses and increments the E counter of all the blocks if that exceeds the MITT interval. For a fixed sized 16 way associativity cache, if the E-counter is 2 bits, a MITT interval = 4 can make the replacement policy LRU biased because it can divide the entire set into 4 subsets with a fairly similar size depending on access patterns. Now the problem is what should be the MITT interval if the cache has irregular associativity size? For example, if the associativity is 5, what should be the MITT interval? A MITT value of 4 will lead to the E-counter being incremented only every 4 misses, and as a result, almost all of the blocks in the set would have an E value "0" or "1". In this chapter, we use a 4 bit EBR (E2R2). In order to generate a LRU biased policy ideally the set should be divided into 4 subsets. Table 4.18 shows how we configure the MITT value in the static EBR resizing scheme. For static EBR policy based resizing scheme, we only configure MITT according to the table after each resize activity. The configuration
4.10 Dynamic EBR resizing

Table 4.18. MITT interval at different associativity for static EBR policy

<table>
<thead>
<tr>
<th>Associativity</th>
<th>MITT interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>

activity itself can be achieved fairly easily using a combinational logic circuit. There is only one such circuit per tile so again the hardware overhead is negligible.

By now, the simulation results have already shown that the EBR based resizing scheme can be substituted for the LRU based resizing scheme and behaves even better in terms of performance and cost. Next, we will combine Dynamic EBR with the resizing algorithm.

Dynamic EBR based resizing is almost identical to static EBR based resizing. The only difference is after each resizing activity, a new MITT interval will be configured. For static EBR based resizing, that MITT will stay fixed until the next resizing activity and reconfigured according to the new size. However, for dynamic EBR based resizing, only half of the sets will have an MITT interval value according to the table, let’s call it part A. The other half sets will be configured with an MITT interval adjacent to the table value (smaller unless the current MITT interval is 1), named part B. An MITT indicator, the same as we described in the previous chapter, will be initialized. Any cache miss at part A will decrease the MITT indicator and any cache miss at part B will increase the MITT indicator. Once the MITT indicator reaches either 0 or the maximum
value, the losing part will change its MITT interval towards that of the winning part by 2.

Figure 4.24 compares the performance, L2 leakage power, total power and EDP between static EBR based resizing scheme and dynamic EBR based resizing. The resizing activity itself seems not to be affected by DEBR very much since the average L2 usages are nearly identical.

However, three applications that benefited from dynamic MITT tuning are canneal, ferret and streamcluster. All of them suffer from L2 thrashing and thread-level interference. The application that benefits the most from DEBR is streamcluster. With the same L2 usage, its total execution cycle was successfully reduced to 85.3%. To study how efficiently DEBR can tune the MITT, we simulated streamcluster with static EBR with different MITT value ranging from 1 to 4. We have discussed in the previous chapter why MITT 4 can be considered as an upper bound.

Table 4.19 shows that when MITT = 1, we achieve best performance which corresponds to 83.8% of the total execution cycles normalized to the LRU baseline. With the help of DEBR we are very close to this. The reason for the 1.5% difference (83.8% to 85.3%) loss is because of the tuning process. Since we have no idea which is better for the running application, LRU biased or LFU biased, we use the most conservative setting. We set MITT to the LRU biased value given by Table 4.18 and then tune the MITT as we described above. This tuning process needs some time and causes that extra 1.5% loss.

### 4.11 Summary

In this chapter, we have proposed a cache size tuning method extended from the previous chapter but using the EBR based cache replacement policy. We further studied
Figure 4.24: Comparison between dynamic-EBR based tuning scheme and static-EBR based tuning scheme in performance, L2 leakage power, total power, EDP when threshold $T=8$. 
how and why LRU replacement policy and stack distance can predict cache utility and extended that to develop the EBR based cache replacement policy. We proposed the concept of evicting-rank and how to use it to estimate cache utility.

With the help of evicting-rank reuse, we can predict how many extra cache misses would occur if one cache way was shut down. We proved the feasibility of both increasing only and increase/reduce schemes. Simulation results showed that the EBR based tuning scheme can fully replace the LRU based scheme. Furthermore, it will always have better performance because of the advantage of the replacement policy itself. The tuning procedure was quite similar between LRU based and EBR Based.

We further investigated the hardware overhead. Compared with the LRU based resizing scheme, the EBR based scheme is more feasible and realistic and can be implemented for high-associativity caches.
In this chapter, we will introduce a dynamic cache partitioning method which leverages our EBR based replacement policy and cache utility evaluation method to dynamically adjust cache partitioning between the shared L2 cache and the private replica region on each tile. To cope with ever increasing cache hit latency, we propose a private replica region on each tile with a flexible size to shorten the average cache access latency. Our scheme can dynamically evaluate the potential cost and gain of reallocating one cache way and if the gain versus cost is worthwhile, then the function of that way will be changed.
5.1 Introduction and motivation

As cache sizes become larger and larger, the access latency to different addresses in the cache varies significantly due to the ever increasing global wire delay. The monolithic cache with a uniform access time is no longer feasible. The idea of NUCA was introduced to break a monolithic large cache into a number of small and fast slices (Kim et al. 2003), each of which has a unique access latency and is connected with others through a packet-switch based network on chip (NOC). The overall access time for each transaction is decided by the path though the network to reach a particular slice plus the identical cache access latency. The use of NUCA not only reduced average access latency but also ensured a lower cache access cost since the dynamic power consumption per access is largely dependent on its physical size. By dividing up the big memory array, the power spent for an access became much smaller. Apart from that, the NUCA structure matches the tiled CMPs structure very well.

In a tile-structured CMPs, each replicated tile is made up of a core, a private L1 cache and a slice of L2 cache. Private cache and shared cache are the two traditional ways to organise those L2 cache slices distributed with the tiles over the chip.

Private cache and shared cache represent extreme cases of average-access latency orientated performance and off-chip miss rate orientated performance respectively. For private cache, all the data are mapped at the L2 slice on the local tile, providing low cache hit latency. On the contrary, shared cache will have to fetch data from the remote tile and undergo on-chip traffic congestion which will lead a higher hit latency. This hit latency will increase as the number of cores scales up. Eventually it will become unacceptable. However, due to the existence of sharing replicas, private cache has a smaller effective total cache size and correspondingly exhibits higher off-chip miss rate. Besides that, private cache behaves especially badly when dealing with multi-application workloads. Because of the fact that private cache can not balance workloads, tiles executing threads with heavy workloads will encounter high miss rates whereas tiles executing threads with small working set size will end up with an under utilised local cache. Unfortunately, off-chip misses will become more expensive because of the unmatched speed between processor and main memory. With the number of cores scaling up, this bottle-neck will be more significant because of the limited bandwidth.

On one hand the shared cache is facing an ever increasing cache hit latency, while on the other hand, private cache is facing a continually increasing miss penalty. The choice between these two extreme options depends on the data access pattern of individual
Chapter 5  

EBR based Dynamic Private and Shared Cache Partitioning

applications, how intensively they will access cache and their degree of data sharing. A fixed organization will never simultaneously satisfy both. To solve this dilemma, we propose a private/shared dynamic partitioning scheme in this chapter.

The dynamic partitioning scheme that we propose in this chapter is based on a shared cache scheme. In order to eliminate the ever-increasing average access latency, we will allocate a piece of storage from shared cache at each tile to temporarily store frequently used data, which will be replicated. The utility of both parts of the cache will be monitored and periodically evaluated. Based on that evaluated utility, the proportion allocated between private cache and shared cache will be dynamically adjusted.

Allocating storage for replicas will lower effective cache size without doubt and correspondingly increase off-chip misses rates, but as long as the penalty is worthwhile we are happy to make the trade-off. Our design will dynamically evaluate the potential performance loss due to missing effective cache storage and compare it with the potential performance gain due to lowered average access latency. Overall, our approach can provide a method to improve the performance of the last level cache of future CMPs.

5.2 Latency analysis

To gain understanding of how average access hit latency can affect overall performance, we simulate 8 applications from PARSEC2 (Bienia et al. 2008) with a 16 core system and a 64 core system using the same configuration as Table 2.3 (64 core system is expended using the same NOC as 16 core system). Figures 5.1 and 5.2 show the breakdown of total memory access latency in each case. As we can see, a large portion of the total cache access latency is attributed to the total L2 access latency. When scaling up the size of the processor with more cores, this proportion is increased. There are two reasons. Firstly, average L2 hit latency is increased due to the scaling up of the 2D NOC. It will require longer routes to fetch data for each transaction. Data sharing can also cause tremendous protocol overhead. The second reason is more straightforward, that is because for each simulation, we use 1 Mbyte cache slice per node, so the overall L2 cache size is increased from 16 Mbyte to 64 Mbyte. For those applications whose working set size does not scale with the number of threads, their overall L2 miss rate will decrease and will cause a relatively higher L2 access latency proportion.

Table 5.1 shows the average L2 hit latency for the 16 core and 64 core systems. Latency is greater for the 64 core systems. There are several reasons for this increase.
5.2 Latency analysis

**Figure 5.1.** Breakdown of total memory access latency of 16 Core system

**Figure 5.2.** Breakdown of total memory access latency of 64 Core system

<table>
<thead>
<tr>
<th>Application</th>
<th>16 Core</th>
<th>64 Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>82</td>
<td>119</td>
</tr>
<tr>
<td>bodytrack</td>
<td>122</td>
<td>138</td>
</tr>
<tr>
<td>canneal</td>
<td>98</td>
<td>148</td>
</tr>
<tr>
<td>facesim</td>
<td>122</td>
<td>153</td>
</tr>
<tr>
<td>ferret</td>
<td>78</td>
<td>100</td>
</tr>
<tr>
<td>streamcluster</td>
<td>94</td>
<td>183</td>
</tr>
<tr>
<td>swaptions</td>
<td>81</td>
<td>106</td>
</tr>
<tr>
<td>x264</td>
<td>92</td>
<td>108</td>
</tr>
</tbody>
</table>

**Table 5.1.** Average L2 hit latency for different applications (cycles)
One of these is because of the nature of the expanding network. The mesh is scaled up from 4 by 4 to 8 by 8, whereas data is uniformly distributed among all the tiles. Another reason is because of the network congestion: when scheduling the process to 64 threads, the probability of a block being shared among multiple cores will also increase depending on the data access pattern of the application, which will lead to the situation that multiple cores can issue data fetch requests simultaneously and block the network. That’s the reason why canneal (most data sharing) suffers more latency increase than bodytrack (least data sharing).

5.3 Private/Shared Partition

To mitigate the above mentioned cache access latency growth, we propose a dynamic cache partitioning method which can dynamically partition the local memory to store necessary replicas.

The basic organisation of the base system is shown in Figure 5.3. As with our our base system in Chapter 1, it is a shared cache system, with each node made up of a RISC core, a private L1 cache and a slice of L2 cache which are logically shared with all other slices residing along with other tiles. To speed up the execution, we partition a piece of memory from the local shared L2 slice and use it to store useful replicas from other tiles. Unlike the proposal in (Zhang and Asanovic 2005), our replica region is only for replicas but the size of the region is tunable.
5.3 Private/Shared Partition

<table>
<thead>
<tr>
<th></th>
<th>Shared Write</th>
<th>Shared Read</th>
<th>Private Write</th>
<th>Private Read</th>
<th>Shared Inst</th>
<th>Private Inst</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackscholes</td>
<td>0.6%</td>
<td>10.9%</td>
<td>6.7%</td>
<td>43.7%</td>
<td>7.5%</td>
<td>8.6%</td>
</tr>
<tr>
<td>bodytrack</td>
<td>1.1%</td>
<td>43.2%</td>
<td>1.9%</td>
<td>6.8%</td>
<td>6.2%</td>
<td>1.6%</td>
</tr>
<tr>
<td>canneal</td>
<td>0.6%</td>
<td>53.2%</td>
<td>9.1%</td>
<td>9.3%</td>
<td>2.1%</td>
<td>1.4%</td>
</tr>
<tr>
<td>facesim</td>
<td>0.0%</td>
<td>53.8%</td>
<td>0.3%</td>
<td>6.1%</td>
<td>0.1%</td>
<td>1.8%</td>
</tr>
<tr>
<td>ferret</td>
<td>0.1%</td>
<td>51.9%</td>
<td>7.3%</td>
<td>10.0%</td>
<td>0.5%</td>
<td>0.7%</td>
</tr>
<tr>
<td>streamcluster</td>
<td>0.1%</td>
<td>57.0%</td>
<td>0.7%</td>
<td>9.4%</td>
<td>0.8%</td>
<td>1.0%</td>
</tr>
<tr>
<td>swaptions</td>
<td>0.8%</td>
<td>52.4%</td>
<td>2.9%</td>
<td>4.4%</td>
<td>2.9%</td>
<td>0.7%</td>
</tr>
<tr>
<td>x264</td>
<td>0.1%</td>
<td>56.2%</td>
<td>0.6%</td>
<td>5.0%</td>
<td>3.4%</td>
<td>1.8%</td>
</tr>
</tbody>
</table>

Table 5.2. Proportion of reused L1 victims amongst L2 access requests

5.3.1 L1 Victim can be Useful

Creating a replica region from the original L2 share slice will almost certainly lower the overall L2 hit rate. If any useless blocks are stored in that region then the L2 hit rate is sacrificed without any benefit. Only blocks with a high potential of being used again should have be replicated.

We chose to replicate selected L1 victim blocks (blocks that have been evicted from L1). To verify that there is benefit in replicating L1 victims, we created a pool with unlimited size to store all the blocks evicted from L1 and upon each L2 access request, and compare the requesting block with candidates inside the pool first. Table 5.2 shows the proportion of L1 victims that were reused again and their types. For example for canneal, among all the L2 access requests, 53.2% of them are actually re-fetching shared-read blocks from the L2 cache. Those blocks resided in L1 before they were evicted. Among all the types of replicas, shared-write blocks are the only unwelcome type. This type of replica will incur a large amount of coherency overhead. Fortunately, the quantity of them is fairly small according to Table 5.2.

5.3.2 Foundation of Partition

We have introduced how a particular address is mapped to cache in previous chapters. Figure 5.4 shows that it will be a different way of interpreting an address for different sizes of cache. When changing the size of the cache during the partitioning process, we also need to change the way the address is mapped. This task is difficult to accomplish.
Fortunately, there is another method to partition a cache, which avoids this problem called way partition where both sides will have the same number of sets but different associativity.

Figure 5.5 shows an example of way partitioning. There are 16 L2 slices in this example, each local L2 slice has 1024 sets, and each set contain 16 ways. The block size is 64 Byte, so the overall L2 cache size is 16M bytes, 1M bytes per slice. In Figure 5.5(a), we allocate 4 ways from each set to store local replicas, then the total shared L2 size becomes 12M bytes and the size of the private replica region is 256K byte per node. In Figure 5.5(b), we further allocate another 4 ways from each set to store local replicas, as a result, half of the shared L2 storage is partitioned to private replica region. With way partitioning, we can change the size without worrying about mapping methods since the number of sets remains the same.

With the help of way partitioning, we no longer need to adjust the format of data mapping dynamically when tuning cache partitioning. For a certain size of cache, only two mapping formats are required: private and shared. For shared cache, the total number of sets is the sum of all sets in the chip whereas for the private part, the total

---

**Figure 5.4.** Different ways to interpret a data address when the cache size is different

---

**Figure 5.5.** Example of way partitioning.
5.3 Private/Shared Partition

The number of sets is only the number of sets on that tile. For this reason we have two mapping formats.

Unfortunately, since there are two mapping formats, we cannot search both parts without any overhead. Zhang et al (Zhang and Asanovic 2005) proposed expanding the size of the tag so that all blocks will have the same size as private blocks. We use a similar approach here. We will match the difference between the shared cache tag and the private cache tag. The difference will be the width of core selection. For a 16 core system, the overhead would be a 4 bit pre tag entry, and 6 bits for a 64 core system, which represents a 0.78% and 1.17% storage overhead respectively.

We also update the coherency protocol and carefully design the algorithm so we can guarantee mutual exclusivity between the private region and the shared region at each tile. We use the EBR replacement policy as it can estimate the utility of each part and can provide an easy method to partition replacement policy.

5.3.3 Cache Query Process

Upon private L1 cache misses, a L2 cache request will be issued to a remote tile depending on the address of the requesting block. We denote the tile to which a particular block should be mapped as the "remote-home" tile. Now in our scheme there is a
private replica region at each tile, so the L2 cache request should enquire in this local replica region before it leaves the tile. If the requesting block has a replica then the long cache access latency can be avoided. The block will be removed from local L2 after it is sent to L1. In other words, the private L1 cache and L2 private replica region are exclusive to each other. If the requesting block happens to map at the local tile then the request should go directly to the local shared cache.

5.3.4 L1 Cache Eviction

L1 victims will be the source of data blocks for the L2 private replica region. In the conventional cache, once a block is evicted from L1, it will be discarded if it is not a dirty block which needs to be written back. In our scheme, after a block is evicted, it will be sent to the private replica region. If the region is full, the block with lowest rank will be replaced. The replica region hence acts as an exclusive private cache which stores L1 victims. Since there is no difference in terms of the hardware, the private replica region can also use EBR based replacement policy. We will elaborate on how it works later. When L1 evicts a block whose remote-home tile happens to be local, the block should be discarded since there is no need to keep a replica when the data happens to be mapped locally.

5.3.5 Coherency Protocol

To maintain coherence in our CMPs system, we select the inclusive MESI-MSI protocol and extend it to work with our private/shared tuning scheme. This protocol is a token based directory protocol. In the conventional protocol, a block will be fetched from main memory to the remote-home tile first, and then it will be forwarded to the requesting L1. A directory entry corresponding to this block will be retained in the directory on the remote-home tile. If another core initiates a cache shared read, the ID of the new sharer will also be recorded in the directory entry. If a sharing core wants to write to that block then an invalidation request will be send to all of the cores sharing that block. In our protocol, this invalidation should be sent to not only L1 cache but also the private replica region. Since L1 and the private replica region are exclusive, the number of tokens that an invalidation generates should be the same.
5.3 Private/Shared Partition

5.3.6 Replacement Policy Partitioning

In this design, we will use EBR based replacement policy for our L2 cache. The problem here is, how to conduct the replacement process for different regions whose size can be dynamically tuned by using a set of fixed hardware. For example, in Figure 5.5(a), 4 out of 16 ways are private region. It is easy if this is a fixed scheme: we can use two chains as shown in Chapter 3, Figure 3.26, one with size 4 and one with size 12. The two chains operate independently so the victim of every eviction is always the lowest rank in the corresponding chain. Unfortunately, our private/shared partition is tunable so the length of the two chains is variable.

In Chapter 4, we showed how to dynamically shut down ways to save static power. Similarly, when shutting down ways, the “Off” blocks should no longer be included in the future victim selection process. We introduced a method which leverages simple combinational logic to partition the chain between “on” and “off”, as shown in Figure 4.23.

We will use a similar idea here to partition the zero detect chain. As shown in Figure 5.7, we add another rank counter per set, one counter for each region. The enable signal for switching the two counters is issued from the private/shared (P/S) controller which takes the place of the on/off controller. Each tile has one P/S controller which records the status of each way, whether it is shared, private or closed. Upon eviction from the private region, the P/S controller will enable all the ways that have been relocated to the replica region and switch to P mode to enable the P counter. For example, in Figure 5.6, we demonstrated a L2 slice which is equivalent to a static partition with 18.75% private, 68.75% shared and 12.5% were shut down. When a shared L2 cache miss happens, a request will be sent to main memory, in the meantime, the P/S controller will change to Share mode and a victim among ways 1, 2, 4, 7, 8, 9, 10, 11, 12, 13, 14 will be selected before the new block is fetched from main memory. The EBR status of way 15 and way 16 will never be compared with the rank counter since these two ways are shut down due to low cache utility.

This schematic is only an example to demonstrate the feasibility of dynamically partitioning. We believe it is not the best possible implementation but this issue is beyond the scope of this thesis. The EBR status must be updated after any access, however, there is no stringent timing requirement. Also, updating a state bit is fairly simple and fast. Updating the rank counter is expensive in term of latency (in the worst case we need $2^{Rbits+Ebits}$ cycles, e.g.: $Rbits = 2$, $Ebits = 2$ and worst case is 16 cycles.) and it is
necessary after each L2 access. It needs a strict timing deadline but luckily this deadline is only required for a local L1 miss or a local L2 shared miss. A local L1 miss means fetching a block from a remote-home tile and a local L2 shared miss means fetching a block from main memory, both of which represent a long latency compared with the rank-counter update latency which hence can be overlapped.

5.4 Dynamic Partitioning

In the last section, we have covered the fundamentals of our dynamic partitioning cache scheme: the hardware and protocols. Next, we are going to introduce the algorithm and tuning method.

We use a 16-way associativity cache to demonstrate our method but any other level of associativity will be similar. Initially, we set 2 ways for the private replica region and 14 ways for the shared region. There will be two utility counters per tile. Utility counters are incremented each time the evicting rank is re-referenced.

The life cycle of a program execution with P/S cache partitioning can be described in the following manner:

1. **Task pre-emption**:
5.4 Dynamic Partitioning

![Diagram of P/S controller and counters]

**Figure 5.7.** P/S resizeable EBR replacement policy hardware

(a) Cache initialised to 2 private ways and 14 shared ways. (Write back any dirty blocks in the cache)

(b) Set both Shared Utility Counter and Private Utility Counter (denote as SUC and PUC) of each tile to 0.

2. Start Execution:

(a) During each L2 Cache "Set-MRU" event which updates the EBR status of a re-referenced block after access, check the rank of the promoting block and compare it with the lowest rank stored at the rank counter of each Set.

(b) If the rank of the promoting block equals the lowest rank, and this is a shared access, then increment the SUC by 1. If this is a private replica region access, then increment the PUC by 1.

3. Periodically evaluate SUC and PUC

(a) Evaluate PUC versus SUC based on equation 5.1 or 5.2

4. Change Size
(a) If the shared cache should increase and there are no available resources, decrease the private cache by 1 way and wait for all necessary write backs to be completed.

(b) If the private cache should increase and there are no available resources, decrease the shared cache by 1 way and wait for all necessary write backs to be completed.

(c) If neither of these are true, then partition remains unchanged.

(d) Reset both SUC and PUC to 0.

5. Continue execution.

SUC and PUC are reset simultaneously, so their values represent the utility of different regions during the same period of time. As we discussed in the previous chapter, the value of the utility counter can be roughly regarded as an estimation of the number of virtual hits if one more cache way was allocated during the previous time period. The comparison between SUC and PUC is actually the comparison between the estimation of virtual hits and virtual misses achieved by the private replica region and shared region respectively. For example, if SUC is 50 and PUC is 80, we can estimate that if one shared way was reduced, we might incur another 50 shared cache misses and if we increased the shared cache by one way, we might save 50 shared cache misses during the previous time period.

Clearly, the cost of a shared L2 miss and that of a private replica miss is very different. Usually the latency of shared cache misses can be several times of that of a replica region miss. In our base system, the ratio is around 8 based on simulation. We define the ratio between different average access latency as $PS_{ratio}$ which is tunable depending on the base architecture. We approximate $PS_{ratio}$ as power of 2 for the purpose of easy implementation. Equations 5.1 and 5.2 show how we evaluate SUC and PUC. If 5.1 is fulfilled, we will decrease the private replica region, if 5.2 is fulfilled, we will decrease the shared region. We use a factor of 2 as an anti-thrashing margin, so no thrashing back and forth would happen when SUC and PUC are similar.

\[
SUC \geq PS_{ratio} \times PUC \times 2 \tag{5.1}
\]

\[
PUC \times PS_{ratio} \geq SUC \times 2 \tag{5.2}
\]
5.5 Performance Evaluation

<table>
<thead>
<tr>
<th>Core</th>
<th>64 In-order UltraSPARC III</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 ICache</td>
<td>Private 16KB/core 4-Way LRU 2cycle access latency</td>
</tr>
<tr>
<td>L1 DCache</td>
<td>Private 16KB/core 4-Way LRU 2cycle access latency</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared 64-Slices 64MB in total and 1MB per slice, 16-Way</td>
</tr>
<tr>
<td>L2 Cache Latency</td>
<td>15cycle access latency</td>
</tr>
<tr>
<td>Block size</td>
<td>64 Bytes</td>
</tr>
<tr>
<td>MSHR</td>
<td>128-entry each tile</td>
</tr>
<tr>
<td>NoC</td>
<td>2D-mesh 1cycle pre hop size of buffer:4</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB, 300cycle access latency DDR-400</td>
</tr>
<tr>
<td>McPAT Parameter</td>
<td>22nm f=3.8GHz T=380K</td>
</tr>
</tbody>
</table>

Table 5.3. Parameters for the simulation framework

5.5 Performance Evaluation

5.5.1 Fixed Partition Evaluation

To evaluate our dynamic private shared partitioning scheme, we use the deterministic trace driven simulation method introduced in Chapter 4. The primary parameters of the model are listed in Table 5.3. We use the same applications from PARSEC2 as we used in previous chapters. All of them were running with native size of input (meaning they are fully loaded), and each process is forced to schedule to at least 63 threads. Each core will be assigned 1 thread and the OS activity will occupy 1 core. We simulate each application with the framework shown in Table 5.3. From the point of interest (which is the parallel execution starting point) we begin to collect traces until 5 billion instructions have been successfully committed. When evaluating our design, the generated trace was fed into the testing architecture and we were able to determine total latency caused by the different approaches.

Before we evaluated our dynamic partitioning scheme, we conducted a full set of simulations to explore all possible static partitions. We chose to have a minimum of two ways for both types of cache, so the simulation configurations range from P=2, S=14 to P=14, S=2. The reason why we need a minimum size of 2 is for the purpose of utility evaluation in the future. As we explained previously, EBR based utility estimation requires a minimum of 2 ways to start the estimation. The result is shown as Figures 5.8 and 5.9. It shows the total number of L2 misses and total L2 access latency normalized to if running with shared cache. We increase the size of the replica region and decrease that of shared cache along the x-axis from left to right. The total number of misses will...
Chapter 5  
EBR based Dynamic Private and Shared Cache Partitioning

Figure 5.8. Exploring all possible static partitions - 1
Figure 5.9. Exploring all possible static partitions - 2
always increase due to more resources being used for the replica region. Accordingly, L2 miss latency will always increase. The advantage here is that with more resources being used to store L1 victims, more and more long latency accesses can be saved hence the L2 hit latency will reduce dramatically.

More importantly, this simulation also shows that a fixed partition is not sufficient. For applications like blackscholes, the major portion of the L2 latency is caused by L2 misses, and reallocating resource to store L1 victims will not increase performance. Also, according to our simulation, blackscholes did not benefit from large shared L2 capacity either, so the best scenario for this type of application is to shut down the cache to save static power. For applications like bodytrack, more than 80% of the L2 latencies are contributed by L2 hit latencies, so if we assign more cache resource for the replica region we can dramatically reduce L2 hit latency, however, L2 miss latency will increase with this action. As a result, an optimised partition will be somewhere in the middle. For applications like canneal, L2 cache miss latency dominates the performance and the shared L2 cache demonstrates good utility so all of the resource should be used for shared L2 cache. Our dynamic P/S partition scheme can solve this problem. We will leverage the cache utility estimation we introduced in the previous chapter to evaluate the cache utility of each part in real-time and adjust the partition based on that.

5.5.2 Dynamic Partitioning Evaluation

Figure 5.10 depicts the total number of L2 misses and total L2 access latency achieved by our dynamic partitioning scheme normalized to the static initial platform (2 ways for private replica region and 14 ways for shared L2 cache). All applications benefit from our scheme except blackscholes, although to varying degrees, which depend on the memory access pattern of the application. We denote the static partition which can achieve lowest total latency as an “optimal partition”. Be aware that this “optimal partition” is only valid for our platform and only valid for “the 5 billion instruction period” since it is chosen based on our experiment and only for comparison purposes. Table 5.4 shows the exact performance increase compared to the performance achieved with a non-tuned platform. For the best case, bodytrack can reduce the total L2 latency to 28.6% and overall latency to 82.4%. For the worst case, blackscholes on the other hand, increased its total latency by 0.3%. On average, our dynamic partitioning scheme lowered the total L2 latency to 70.3% and the entire cache latency to 94%. Another
5.5 Performance Evaluation

Figure 5.10. Dynamic private shared cache partition performance normalized to the fixed partition (2P:14S)

very important point we want to clarify here is that "Total Latency" is the sum of all types of latency. Many of these latencies are inevitable and may possibly be in parallel with each other. This makes our improvement in total latency trivial, however, total L2 latency is the sum of all latencies caused outside L1, which has the potential to be improved, hence total L2 latency improvements are more important than total latency improvements.

Figure 5.11 shows the partitioning result achieved by our dynamic tuning scheme compared with the optimal partition that we determined from exhaustive static simulations. Overall, these two groups are very similar, however, minor differences exist. Figure 5.12 shows the impact of those variances. In Figure 5.12, we normalized the total number of L2 misses, total L2 hit/miss latency and overall L2 access latency achieved by our dynamic scheme to those achieved by the optimal scheme.

These 8 applications can be grouped into 3 types. Type one contains *blackscholes*, *canneal* and *facesim* for which the dynamic scheme can achieve a similar latency decrease to what the optimal scheme can achieve. *blackscholes* and *canneal* represent applications that do not benefit from the private replica region, either because of their very minor portion of L2 hit accesses or very expensive L2 miss penalty. *facesim*, however, has another cause. For this type of applications, the slope of the L2 hit latency reduction is
Chapter 5  EBR based Dynamic Private and Shared Cache Partitioning

<table>
<thead>
<tr>
<th></th>
<th>L2 Latency</th>
<th>L2 Latency</th>
<th>Total Latency</th>
<th>Total Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dyn/Fix</td>
<td>Opt/Fix</td>
<td>Dyn/Fix</td>
<td>Opt/Fix</td>
</tr>
<tr>
<td>blackscholes</td>
<td>100.3%</td>
<td>99.9%</td>
<td>100.1%</td>
<td>100%</td>
</tr>
<tr>
<td>bodytrack</td>
<td>28.6%</td>
<td>23.7%</td>
<td>82.4%</td>
<td>81.2%</td>
</tr>
<tr>
<td>canneal</td>
<td>98.7%</td>
<td>99.4%</td>
<td>99.3%</td>
<td>100%</td>
</tr>
<tr>
<td>facesim</td>
<td>77.5%</td>
<td>77.7%</td>
<td>94.9%</td>
<td>95.0%</td>
</tr>
<tr>
<td>ferret</td>
<td>66.8%</td>
<td>69.9%</td>
<td>92.6%</td>
<td>93.6%</td>
</tr>
<tr>
<td>streamcluster</td>
<td>96.5%</td>
<td>98.1%</td>
<td>98.2%</td>
<td>99.1%</td>
</tr>
<tr>
<td>swaptions</td>
<td>53.7%</td>
<td>49.2%</td>
<td>98.2%</td>
<td>98.0%</td>
</tr>
<tr>
<td>x264</td>
<td>39.9%</td>
<td>36.7%</td>
<td>86.7%</td>
<td>86.0%</td>
</tr>
</tbody>
</table>

Table 5.4. Dynamic scheme performance and optimal scheme performance normalized to fixed scheme

Figure 5.11. Partition achieved by dynamic partitioning vs optimal partitioning
5.5 Performance Evaluation

similar to the L2 miss latency increase, which was shown in Figure 5.8. As a result, the
different partitionings implemented by dynamic partitioning and optimal partitioning
will not make any obvious difference to performance.

Type two contains ferret and streamcluster for which the dynamic partitioning scheme
behaved even better than the optimal scheme. The reason for this is because our dy-
namic partition has a lower granularity. If the application shows an irregular memory
access pattern among tiles, our scheme can adjust with it. For example in ferret, our
dynamic partitioning worked out an average of 10.6 ways out of 16 ways preserved
for shared L2 data and the rest were re-allocated for private replica regions. The static
optimal partition was 10 ways for shared and 6 ways for private. These two parti-
tions seem very close but our dynamic partition is the average of all tiles. Each tile has
a different partition ranging from 6S:10P to 13S:3P depending on the memory access
pattern on each tile.

The last type consists of bodytrack, swaptions and x264. For this type of application
the dynamic scheme can not achieve a level of performance as good as the static opti-
mal scheme. The worst case in our simulation example is bodytrack which can reduce
total L2 access latency to 28.6%, however, it was still 4.9% higher than the static opti-
mal scheme which can lower that latency to 23.7%. The reason for this phenomenon
is because of the cost of tuning establishment. For applications that benefit dramat-
ically from a private replica region, a large portion of the cache resource should be
reallocated which needs time in the dynamic tuning process. We mentioned that our
starting point is 2 private ways and 14 shared ways and our tuning action can only
reallocate one way at a time. Figure 5.13 shows the tuning procedure in time. bodytrack
reached its optimal partition at the middle of the entire simulation and swaptions only
reached its optimal partition near the end of simulation. The good news is that once
the optimal partition was reached, our dynamic scheme will work as well as the static
optimal scheme, sometimes even better due to the reason we explained above, for type
two applications (ferret, streamcluster). Our simulation covered only 5 billion instruc-
tions, whereas for real applications, the execution length will be much much longer
and hence the impact of dynamically tuning to the best partition will be less.
Figure 5.12. Dynamic private shared cache partitioning performance normalized to the optimal scheme.

Figure 5.13. Tuning procedure in time.
To further exploit cache power-performance efficiency, we integrated the cache resizing scheme introduced in Chapters 2 and 4. When the utility of a cache way is lower than the threshold of both shared cache and private cache, it should be powered off to reduce static power. We simply update our partitioning algorithm by checking SUC and PUC with their corresponding threshold before comparing them with each other using equation 5.1 or 5.2. As we discussed in the last chapter, we will add two Reducing State Bits per tile “Reducing State Bit”, one for shared cache and one for private cache, so we will not turn off the cache too aggressively. We use the same state machine as shown in Figure 4.16. This update can guarantee two things. Firstly, if the utility of the cache is lower than the threshold, then there will be no Private/Shared size adjustment evaluation triggered. Secondly, if the cache size was reduced before Private/Shared size adjusting then there is no need to reduce the losing side again, the winning side can increase cache size directly.

To evaluate this scheme, we ran the simulation again with this new partitioning algorithm. We chose a conservative threshold $T=16$ in this case because we are trying to demonstrate how much power we can save even if the scheme was set to be performance oriented. Referring to Equations 5.1 and 5.2, the $T$ values for the private and shared regions must be in the ratio defined by $PS_{Ratio}$. For the systems we are simulating, with $PS_{Ratio}=8$, if $T_{share}$ is 16 then $T_{private}$ would be $16 \times 8$. Figure 5.14 showed the comparison of total number of L2 misses, total L2 access latency and cache partition compositions between the dynamic partitioning scheme and dynamic partitioning with resizing integrated scheme. With the original dynamic scheme, all 16 ways were assigned to either shared or private. With the resizing scheme, some underutilized ways were shut off. Figure 5.14 shows the total number of L2 misses and total cache access latency normalized to running with shared cache, from performance point of view, there is very little performance penalty. Two applications that benefit the most are blackscholes and swaptions, as these two applications showed a flat slope in latency in Figures 5.8 and 5.9, meaning increasing cache size will bring only a small performance improvement. In other words, they have low cache utility. All in all, with the help of the dynamic resizing algorithm, we further reduced the average usage of L2 by 11.5% with almost no performance loss.
Figure 5.14. Dynamic partitioning with cache resizing integrated (Number of misses and access latency are normalized to running under fully activated shared cache)

5.7 Summary

In this chapter, we proposed a dynamic cache partitioning method which leverages our EBR based replacement policy and cache utility evaluation method to dynamically adjust cache partitions between shared L2 cache and a private replica region on each tile. We believe that future CMPs will be encumbered by the ever increasing cache hit latency when scaling up. On-chip storage resources are used to shorten the processor stall time caused by data fetching, so performance can be improved by either saving the latency caused by a cache miss or avoiding the latency caused by long distance
5.7 Summary

forwarding. Our scheme can dynamically evaluate the potential cost and gain of real-locating one cache way, and if the gain versus cost is worthwhile, then the function of that way will be changed.

Our scheme is based on our EBR replacement policy, and we have discussed the hardware implementation and coherency protocol implementation. Unlike many other dynamic partitioning schemes, we included the replacement policy and proposed a feasible method to partition the replacement logic.

We simulated our scheme with a deterministic trace driven simulation. The results show that our tuning scheme can partition the cache to achieve a partitioning similar to the static optimal partition. The total L2 latency was reduced to 70.3% compared with baseline. We further compared our dynamic scheme with the static optimal scheme and analysed reasons that cause minor differences. With the dynamic cache resizing algorithm integrated, we can further reduce the usage of L2 by 11.5% with almost no performance loss (shown in Figure 5.14).

Our scheme can be superior to the static optimal scheme if the application shows irregular cache access patterns because our partition scheme has a lower granularity which is at the tile level.

In this chapter, a private replica region is used to store L1 victims. In future studies, we could further classify those victims based on their EBR status and bypass the victims that may not likely to be reused again.
Chapter 6

Conclusion and Future Work

This chapter summaries and draws a conclusion to the thesis and provides recommendations for potential future work.
6.1 Conclusion

Power, which is now universally recognised as the first order design constraint has changed the conventional way of how micro-processors scale up. CMPs architecture is becoming the dominant approach to providing much lower power/performance efficiency. However, it is not hard to see that in the near future conventional CMPs architectures will hit the power wall again.

The conflicting requirements of large amount of on-chip storage and managing acceptable power consumption makes a CMP’s cache hierarchy design particularly challenging.

Without advanced power management technique, the rapid scaling up of CMPs will come to an end soon. Our research has been aimed at easing this problem. We proposed dynamic cache management strategies aimed at improving performance while lowering power. At the beginning of our research, we were trying to lower the static power by shutting down the under-utilized cache resources. We discussed the fundamental idea behind our performance/power trade off and proposed the concept of the Break-Even number of Misses (BEM), a parameter that should be determined at system design time. We further proposed a cache resizing mechanism to dynamically shut down part of the cache if it is under utilized. A set of threshold values build into the BEM was embedded in the scheme representing different power-performance trade-offs. Our simulation shows a 6.5% EDP reduction for multi-threaded applications (PARSEC2) and 12.9% EDP reduction for single-task applications compared to using shared L2 cache without resizing (SPEC CPU 2006). The hardware overhead is negligible. With our design, static power can be dramatically lowered without causing noticeable performance loss. This technique can be leveraged in the future when static power issues become more significant.

Then we continued our research on how to replace LRU for two purposes. One is to improve performance and the other is to make our utility based cache resizing technique realistic. As our technique typically needs a high associativity cache in order to work well and leverages the LRU replacement policy, so the cost of implementation is quite high. In the second phase of our research, we proposed an alternative cache replacement policy. It can mitigate the disadvantage of LRU when moving from unicores to CMPs, in which inter-thread interference can cause lower cache hit rates due to pollution of useful data by other threads. The new policy provides an economic
and feasible way to replace LRU, which was too expensive to implement for high associativity caches. In proposing the Effectiveness-based Replacement policy (EBR), we firstly revisited the essential function of caches to develop an underlying analytical model. We argued that frequency and recency are the only two affordable attributes of cache history that can be leveraged to predict a good replacement. Based on those two properties, we proposed a novel replacement policy, EBR and a refinement, Dynamic EBR (D-EBR). Our simulation results showed that with negligible hardware overhead we can achieve a maximum of 28.7% and an average of 12.4% performance increase compared to running with a shared L2 cache with LRU for all applications in SPEC CPU2006. To verify our initial intention which is to make utility-based cache resizing technique realistic with the help of EBR, we studied how to integrate UBC with EBR and DEBR. Our work shows that it can provide higher performance due to its anti-thrashing and inter-thread interference protective attributes. More importantly, it is hardware economical.

Finally, for the purpose of lowering average cache access latency, we proposed a method to periodically evaluate the utility of the cache on each local tile and dynamically partition it to store L1 victims, when it was under poor utility. With little additional hardware, EBR itself can support cache partitioning, and it can provide cache utility estimation in an economical fashion. These advantages make EBR an effective and efficient replacement policy for cache partitioning. Our simulation showed, compared with shared L2 cache, on average the total cache latency (hit + miss) can be reduced by up to 70.3%. When applying the cache resizing algorithm on top of dynamic partitioning, we can further reduce the L2 usage by 11.5% and hence save static power.

In conclusion we have introduced and demonstrated the effectiveness of hardware efficient dynamic cache resizing and partitioning algorithms that can have a significantly beneficial effect on performance and power consumption on most applications in the benchmark suite, without significant adverse effects on others. This technique gives an alternative way to control the rapid growth of static power of CMPs.

6.2 Future Work

In this thesis, we proposed some feasible cache improvement techniques. Inevitably, some topics and ideas arose in the study but there was not time to investigate. They are listed below.
6.2 Future Work

6.2.1 EBR Based Bypass

To decrease average L2 access latency, we store L1 victims in the private replica region on local tiles, making the data near the requester. L1 victims are very likely to be used again as we observed, however, there still exists more potential improvement. With EBR, we can easily distinguish the utility history of a block and not store those victims that are poorly behaved. By doing this, the efficiency of the local private replica region should be improved.

6.2.2 Private Cache Evaluation

In this thesis, we chose to focus on the shared last level cache mainly because of the ability of balancing workload. Recently, many studies based on private cache and leverage on Spill-Receive (where cache can be grouped and groups with extra cache resources will spill some resource and groups that need more resource will receive it) also provided us another approach (Lee et al. 2011) (Qureshi 2009) (Herrero et al. 2010). When threads have very irregular cache utility, cache resources can be re-allocated. Nevertheless, the problem of how to evaluate cache utility is always an obstacle. Counting LRU hits or using a UMON like structure (Qureshi and Patt 2006) for future many-core system seems not very practical. EBR provides an easy way to evaluate cache utility and implement a partition-able replacement policy, so we believe it is more suitable for the spill-receive algorithm. This could be investigated in future work.

6.2.3 Cache Resizing Without Set-Associativity

In this thesis, we choose to resize and partition the cache by leveraging cache set-associativity. Other cache mapping will not work for us. Cache set-associativity is the most popular cache mapping technique for large L2 or L3 cache. It will be uncommon to abandon cache set-associativity but this could be investigated in future work.
Bibliography


Bibliography


