Power Efficient Last Level Cache for Chip Multiprocessors

A Thesis
Submitted For the Degree of
Doctor of Philosophy
in the Computer Science and Engineering

by
Aparna Mandke

Computer Science and Automation
Indian Institute of Science
BANGALORE – 560 012
January 2013
TO

My Parents,

Chitra and Chandrashekhar Mandke
Acknowledgements

I would like to thank Prof. Y. N. Srikant and Prof. Bharadwaj Amrutur for providing an opportunity to work with them and their guidance throughout my PhD. Whenever I needed any help, I could always find Prof. Srikant. Immense enthusiasm, striking brilliance and positive attitude of Prof. Bharadwaj was a great source of inspiration for me. I am deeply indebted to them for their support and guidance in academics, logistics, infrastructure, to name a few. It was indeed an honour to work with both of them.

I am very fortunate that I could interact with Late Prof. Priti Shankar. She was a role model for me. Her brilliance, gentle yet firm nature and positive attitude will always give me strength in every step of my life. I am grateful to Prof. Mathew Jacob and Prof. Govindrajan for reviewing my work and providing me guidance.

The admin staff of CSA department - Ms. Lalitha, Ms. Suguna, Mr. Jagdish and Mr. Pushparaj made my stay memorable. They were always friendly and helpful. Archana, Vasanta, Arun, Meghana, Rupesh, Keshavan and BT were always there with me.

I have no words to thank my in-laws and my parents for their support and encouragement. Their help was very critical. I owe a deep gratitude to Shubhada, Pushkaraj, Asang and Meenakshi for entertaining my kids during their holidays. Last but not the least, I cannot thank enough Atharva and Shubham, my kids and Amod, my husband for being patient, for their countless sacrifices and encouragement throughout this journey. I would like to dedicate this thesis to my wonderful family.
Publications based on this Thesis


3. TCP: Thread Contention Predictor for Parallel Programs, 20th IEEE/Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2012


Abstract

The number of processor cores and on-chip cache size has been increasing on chip multiprocessors (CMPs), thanks to advances in CMOS semiconductor technology. As a result, leakage power dissipated in the on-chip cache has become a major contributing component of power dissipated in the memory subsystem. Hence, we explore various techniques to switch-off the over-allocated cache so as to reduce leakage power consumed by it. A large cache offers non-uniform access latency to different cores present on a CMP as it is implemented using multiple banks that are inter-connected with an on-chip network. Hence, such a cache is called “Non-Uniform Cache Architecture (NUCA)”. Past studies have explored techniques to reduce leakage power for uniform access latency caches and with a single application executing on a uniprocessor. Our ideas of power optimized caches are applicable to any memory technology and architecture for which the difference of leakage power in the on-state and off-state of on-chip cache bank is significant.

Switching off the last level shared cache on a CMP is a challenging problem due to concurrently executing threads/processes and large dispersed NUCA cache. Hence, to determine cache requirement on a CMP, first we propose a new highly accurate method to estimate working set size of an application, which we call “tagged working set size estimation (TWSS)” method. This method has negligible hardware storage overhead of 0.1% of the cache size.

The use of TWSS is demonstrated by adaptively adjusting cache associativity. Our ideas of adaptable associative cache is scalable with respect to the number of cores present on a CMP. It uses information available locally in a tile on a tiled CMP and thus avoids network access unlike other commonly used heuristics such as average memory access latency and cache miss ratio. Our implementation gives 25% and 19% higher EDP savings than that obtained with average memory access latency and cache miss ratio heuristics on a static NUCA platform (SNUCA), respectively.

The number of cache misses increases due to reduced cache associativity. Hence, we also propose to map some of the L2 slices onto the rest L2 slices and switch-off mapped L2 slices. The L2 slice includes all L2 banks in a tile. We call this technique the “remap policy”. Some applications execute with lesser number of threads than available cores during their execution. In such applications L2 slices which are farther to those threads are switched-off and mapped on-to L2 slices which are located nearer to those...
threads. As explained earlier, L2 slices are distributed on the chip. Hence, some slices are located near to the executing threads, whereas, some L2 slices are far from the threads. By using nearer L2 slices with the help of remapped technology, some applications show improved execution time apart from reduction in leakage power consumption in NUCA caches using the remap policy.

To estimate the maximum possible gains that can be obtained using the remap policy, we statically determine the near-optimal remap configuration using the genetic algorithms. We formulate this problem as a energy-delay product minimization problem. Our dynamic remap policy implementation gives energy-delay savings within an average of 5% than that obtained with the near-optimal remap configuration.

Energy-delay product can also be minimized by improving execution time, which depends mainly on the static and dynamic NUCA access policies (DNUCA). The suitability of cache access policy depends on data sharing properties of a multi-threaded application. Hence, we propose three indices to quantify data sharing properties of an application and use them to predict a more suitable cache access policy among SNUCA and DNUCA for an application.
Contents

Acknowledgements i
Publications based on this Thesis ii
Abstract iii
Keywords xii
Notation and Abbreviations xiii

1 Introduction 1
  1.1 Motivation ................................................. 1
  1.2 Methodology .............................................. 4
  1.3 Organization of the thesis ................................. 6

2 Background and Related Work 9
  2.1 NUCA caches .............................................. 9
  2.1.1 Private Vs Shared cache ................................. 10
  2.1.2 Line search policies .................................. 11
  2.1.3 Line placement policies ............................... 12
  2.1.4 Line migration policies ................................ 12
  2.1.5 Line replacement policies ......................... 13
  2.1.6 Cache coherence management .................. 13
  2.2 Cache contention prediction ........................... 14
  2.3 Power efficient caches .................................. 15
  2.3.1 Dynamic power reduction ......................... 16
  2.3.2 Static power reduction ............................. 17
  2.4 Working set size estimation metrics ............... 21
  2.5 Application of Genetic Algorithms for architectural studies 22
  2.6 Summary ................................................. 23

3 Sapphire: An Experimental Setup 24
  3.1 Introduction .............................................. 24
  3.2 Motivation ............................................... 25
  3.3 Integrating SESC with Ruby .............................. 26
  3.4 Modeling the interconnect .............................. 28
    3.4.1 Modelling power in interconnect .................. 30
  3.5 Modeling DRAM .......................................... 30
  3.6 Simulation procedure .................................... 31
  3.7 Applications used in experiments .................. 32
  3.8 Summary ................................................. 36
### 4 Working Set Size Estimation

4.1 Introduction ................................. 37
  4.1.1 Dhodapkar’s WSS Estimation Method (DHP) ......................... 38
4.2 Tagged Working Set Estimation Method .......................... 39
  4.2.1 Determination of monitoring interval .............................. 39
  4.2.2 Hardware implementation and overhead ........................... 42
4.3 Comparison with the Dhodapkar’s Method (DHP) ................. 42
  4.3.1 Some typically used software hash functions ...................... 42
  4.3.2 Hardware complexity of hash functions .......................... 45
  4.3.3 Linear Feedback Shift Registers ................................. 46
4.4 Comparison with AAL and CMR metrics .......................... 48
4.5 Cache line replacement distribution .............................. 49
4.6 Sensitivity to L2 slice size ..................................... 50
4.7 Evaluating with multiprogramming workload (SPEC 2006 benchmarks) 51
4.8 Summary ...................................... 52

### 5 Leakage Power Optimization Strategies

5.1 The drowsy cache technique .................................. 54
5.2 Variable way SNUCA and DNUCA ............................... 55
  5.2.1 Determination of cache associativity ......................... 55
  5.2.2 Hardware implementation details ............................... 56
5.3 Evaluation of adaptable way SNUCA .............................. 57
  5.3.1 Comparison with Dhodapkar’s metric ............................ 57
  5.3.2 Comparison against AAL and CMR metrics ...................... 58
  5.3.3 Comparison against the drowsy cache ......................... 61
  5.3.4 Increase in memory accesses ................................. 62
  5.3.5 Application to DNUCA: comparison with Bardine’s metric .... 63
  5.3.6 Scalability of TWSS.D ........................................ 64
  5.3.7 Sensitivity to size of L2 slice ............................... 65
  5.3.8 Evaluation with multiprogramming SPEC benchmarks .......... 66
  5.3.9 Sensitivity to monitoring interval ............................ 68
5.4 Summary ...................................... 70
5.5 The Remap policy ...................................... 71
  5.5.1 Architectural changes ....................................... 71
  5.5.2 Determination of the remap table ............................. 72

### 6 Offline Determination of Remap Table using Genetic Algorithms

6.1 Simple Genetic Algorithm .................................. 74
  6.1.1 Reproduction ......................................... 75
  6.1.2 Crossover ........................................... 76
  6.1.3 Mutation ........................................... 77
6.2 Application of GA to search the NORC .......................... 77
  6.2.1 Estimation of energy consumption ............................ 78
  6.2.2 Estimation of time ..................................... 80
  6.2.3 Estimation of transit time ................................ 80
  6.2.4 Estimation of cache miss penalty .......................... 80
  6.2.5 Genetic Algorithm formulation(GA) ......................... 82
6.3 Results ........................................ 83
  6.3.1 Accuracy of cache misses estimation method .................. 84
  6.3.2 Accuracy of execution time estimation method .............. 86
  6.3.3 Change in WSS ........................................ 87
  6.3.4 Accuracy of GA formulation ............................... 87
  6.3.5 Simulation speedup ...................................... 88
## CONTENTS

6.4 Summary .................................................. 88

7 Adaptive Remap Configuration ................................ 92
  7.1 Adaptive implementation of the Remap policy ............. 92
    7.1.1 Determination of metric ............................ 92
    7.1.2 Working of the adaptive remap configuration ......... 93
    7.1.3 Directory assumption ................................ 94
    7.1.4 Area overhead and scalability ...................... 96
  7.2 Results .................................................. 96
    7.2.1 Comparison with adaptable way policy ................ 96
    7.2.2 Equal cache comparison ............................. 97
    7.2.3 Comparison with drowsy cache ....................... 98
    7.2.4 Number of memory accesses .......................... 100
    7.2.5 Overhead in the remap policy ....................... 101
    7.2.6 Directory table invalidations ....................... 102
    7.2.7 Sensitivity to L2 slice size ....................... 103
    7.2.8 Sensitivity to monitoring period ................... 103
    7.2.9 Comparison with the near-optimal remap configuration 104
    7.2.10 Evaluating with Multiprogramming SPEC 2006 benchmarks 105
  7.3 Summary .................................................. 105

8 TCP: Thread Contention Predictor for Parallel Programs .... 109
  8.1 Introduction ............................................. 109
  8.2 TCP ...................................................... 110
    8.2.1 Sharing Index (SI) .................................. 110
    8.2.2 Contention Index (CI) ............................... 111
    8.2.3 Popularity Index (PI) ............................... 112
  8.3 Cache Policy Predictor (CPP) ............................ 112
    8.3.1 Estimation of overhead in DNUCA and SNUCA ........ 113
  8.4 Results .................................................. 117
    8.4.1 CPP model validation ............................... 117
    8.4.2 Sensitivity to the L2 slice size ................... 119
    8.4.3 Sensitivity to the link latency ..................... 120
    8.4.4 Sensitivity to the number of threads ............... 121
  8.5 Summary .................................................. 121

9 Conclusions And Future Work ................................ 122
  9.1 Future work ............................................. 125

Appendices ...................................................... 128

A Variation in Working Set Size with Time ..................... 128

B Sensitivity study of EDP savings ........................... 131

C Additional Results ........................................ 134

Bibliography .................................................. 142

Index .......................................................... 157
List of Tables

3.1 Tile link latencies and power estimations using Intacte for 512KB L2 slice 30
3.2 System configuration used in experiments 32
3.3 Table shows applications used for study and their WSS information (L:Large, M:Medium, S:Small) 33
3.4 Multiprogramming workloads simulated on a quad-core CMP 35

4.1 shows average ratio of estimated WSS obtained using TWSS and DHP methods to true/actual WSS (AWSS) evaluated at L1 cache. SRAND indicates hash function used in the DHP method 45
4.2 shows the number of instructions executed and the number of clock cycles required to evaluate srand and rand functions in the DHP method 46
4.3 shows correlation of WSS estimated with various metrics to true/actual WSS (AWSS) 48
4.4 shows correlation of various metrics to actual WSS (AWSS) 52

6.1 Probability of various strings used in the optimization problem depends on their fitness 76
6.2 shows % EDP savings when the number of allocated L2 slices is varied with respect to the reference, which allocates all 16 L2 slices. R_2 indicates that two L2 slices are allocated and rest of the fourteen slices are mapped to these two allocated slices. %Pf and %ED give % savings in the execution time and energy-delay product of the application, with respect to the reference 77
6.3 gives meaning of the various terms used in genetic algorithm based problem formulation. The last column describes how these terms are obtained 79
6.4 Correlation between the number of cache misses estimated using modified trace based Dinero model and simulation 84
6.5 Near optimal remap configurations determined by dividing trace according to distinct WSS changes 86
6.6 Simulation speedup obtained using our method over cycle-accurate architectural simulation 88

8.1 gives the meaning of various terms used in the CPP model 114
8.2 shows PeerSearchCt, NearSearchCt and AvgDistanceSearchCt cost components, normalized with respect to (w.r.t.) DNUCA cost. It also shows distribution of normalized CLAs with $SI = 1$, $SI > 1$, $CI < 3$ and $CI \geq 3$. 117
List of Figures

1.1 Tiled architecture used for study ........................................ 4
1.2 SNUCA and DNUCA cache access policies ............................... 5
1.3 normalized power consumed by various memory subsystem components 5

3.1 Detailed block diagram showing the integration of SESC with Ruby. ........... 27
3.2 Interaction between various modules of SESC and Ruby for processing an instruction memory request. A similar sequence of calls are executed for a data request. ........ 28
3.3 The floorplan of a tile with 512KB L2 slice. R refers to the router. ............ 29
3.4 shows the steps to be followed for performing detailed power-performance analysis on Sapphire ...................................................... 31
3.5 IPC gain with ideal 8MB L2 cache (16 L2 slices of 512KB) ..................... 34
3.6 Experimental setup used for a quad-core CMP with 4MB shared L2 cache ....... 34
3.7 IPC gain with ideal 4MB shared L2 cache ..................................... 35

4.1 Dhodapkar’s method to estimate WSS ..................................... 38
4.2 Tagged working set size estimation method ................................. 38
4.3 shows variation in reuse time on the X axis and percentage of accesses on the Y axis. Majority of accesses have reuse time less than 4M clock cycles. ...................... 40
4.4 shows variation in reuse time on the X axis and percentage of accesses on the Y axis for SPEC 2006 based multiprogramming workloads ............................................. 41
4.5 shows average ratio of WSS estimated using TWSS and DHP methods to true/actual WSS (AWSS) evaluated at the L2 cache. SRAND, SWAPTION and MODULO indicate various hash functions used in the DHP method ......................................... 44
4.6 The 7-bit Fibonesi LFSR with taps at 6th and 7th bit positions. Its configuration is expressed as $x^7 + x^6 + 1$ polynomial ........................................ 47
4.7 shows average ratio of WSS estimated using DHP method with LFSRs as hash functions. WSS estimated using TWSS is repeated here from Fig. 4.5 for convenience. .... 48
4.8 Average and standard deviation of replacements per set. The Y axis plots average replacements per set and its standard deviation against the monitoring interval on the X axis. ............................................. 49
4.9 Sensitivity of TWSS to L2 slice size ........................................... 50
4.10 The average ratio of WSS estimated using TWSS and DHP method with various hash functions. The experiments are performed using multiprogramming SPEC benchmarks on the non-tiled CMP ....................................................... 51

5.1 An SRAM bank is switched off to reduce associativity of all sets in an L2 slice by one . 57
5.2 Graphs in (a) and (b) show normalized EDP and execution time obtained with WSS estimated using Dhodapkar’s and TWSS methods .......................... 58
5.3 Graphs show normalized EDP and execution time obtained with various implementations of power efficient SNUCA ........................................... 59
5.4 Cache usage normalized w.r.t. the total cache size ............................. 60
5.5 No. of memory accesses ........................................... 62
5.6 Quantitative comparison of our variable way DNUCA implementation with Bardine’s metric. Figure compared EDP ........................................... 63
5.7 Graphs (a) and (b) show normalized EDP and execution time obtained with various implementations of power efficient DNUCA ........................................... 64
5.8 TWSS:S: Sensitivity to L2 slice size ........................................... 66
5.9 Evaluation of SPEC benchmarks with SNUCA cache access policy ........................................... 67
5.10 blackscholes does not exhibit any active line replacement during execution. (a) Variation in EDP (on the primary Y axis) with monitoring interval (on the X axis) and distribution of reuse time in % (on the secondary Y axis) (b) Variation in WSS with monitoring interval ........................................... 69
5.11 cholesky: Variation in EDP with monitoring interval ........................................... 70
5.12 Proposed remap policy ........................................... 71
5.13 Block diagram explaining working of the remap policy ........................................... 72

6.1 Roulette wheel selection method. The size of slots occupied by various strings is proportional to their fitness values ........................................... 76
6.2 plots power dissipated in memory (in mW) on the Y axis against the number of accesses made to the offchip DRAM on the X axis ........................................... 81
6.3 The remap configuration is encoded as a chromosome. Fig. shows the crossover operation 83
6.4 The flow chart explaining our problem formulation using genetic algorithms ........................................... 84
6.5 The primary Y axis plots execution time obtained on simulating the near optimal remap configurations allocating 2, 4... number of L2 slices. The secondary Y axis plots execution time estimated using Eq. 6.10 for the same remap configurations. Execution time estimated using our trace based model shows a good correlation to the trend seen with simulation ........................................... 85
6.6 Graphs show normalized execution time (on the primary Y axis) and EDP (on the secondary Y axis) obtained on simulating remap configurations allocating 2, 4... L2 slices. All values are normalized w.r.t. the reference execution on SNUCA. “GA(X)” is the near optimal remap configuration obtained with our GA method. This remap configuration allocates X L2 slices ........................................... 89
6.7 Graphs show normalized execution time (on the primary Y axis) and EDP (on the secondary Y axis) obtained on simulating remap configurations allocating 2, 4... L2 slices. All values are normalized w.r.t. the reference execution on SNUCA. “GA(X)” is the near optimal remap configuration obtained with our GA method. This remap configuration allocates X L2 slices ........................................... 90

7.1 average and maximum static average access latency offered by each L2 slice to L1 caches ........................................... 94
7.2 Comparison with way adaptable caches ........................................... 95
7.3 Comparison with caches using SNUCA and DNUCA cache access policy and having cache associativity equal to the average L2 slices allocated by the remap policy ........................................... 98
7.4 Comparison with the drowsy cache ........................................... 99
7.5 Comparison of off-chip memory accesses ........................................... 100
7.6 Remap Overhead ........................................... 101
7.7 Finite sized directory table ........................................... 102
7.8 EDP gains obtained using the remap policy show negligible variation on varying monitoring period ........................................... 104
7.9 The dynamic remap configuration gives EDP savings comparable to that obtained with the near optimal remap configurations determined using GA ........................................... 107
7.10 Configuration used to simulate multiprogramming SPEC 2006 workload (L2 slice size: 256KB) ........................................... 108

8.1 Threads $t_1$ and $t_2$ make four accesses each. However, in (a), four accesses made by these threads are not interleaved. Whereas, in (b), accesses made by these threads are interleaved.111
LIST OF FIGURES

8.2 NUCA access policies .......................................................... 112
8.3 Accuracy of CPP Model: CostModel-Ratio of $DNUCA_{cost}/SNUCA_{cost}$ estimated by
CPP. CostSim - Ratio of time spent in transit in DNUCA to that in SNUCA, both
obtained with simulation ......................................................... 118
8.4 shows simulation speed-up obtained by CPP over cycle-accurate simulations of DNUCA
and SNUCA. Overhead is ratio of simulation time with and without statistics collecting
SNUCA simulation ................................................................. 119
8.5 CPP model predicts overhead costs accurately for L2 slice of size 256K .................... 119
8.6 CPP predicts overhead costs accurately on varying link latencies .......................... 120
8.7 CPP predicts overhead costs accurately on executing with 8 threads ....................... 120

9.1 shows normalized power consumed by various memory subsystem components on using
the remap policy ................................................................. 124

A.1 shows variation in WSS with program execution. WSS is in terms of the number cache
lines (in thousands) plotted on the Y axis and the X axis shows monitoring period (4M
clock cycles). ................................................................. 129
A.2 shows variation in WSS with program execution. WSS is in terms of the number cache
lines (in thousands) plotted on the Y axis and the X axis shows monitoring period (4M
clock cycles). ................................................................. 130

B.1 dealII-dealII-dealII-dealII: Variation in EDP with monitoring period ......................... 131
B.2 soplex-bzip2-namd-dealII: Variation in EDP with monitoring period ......................... 132
B.3 milc-deal-gcc-soplex: Variation in EDP with monitoring period ............................ 132
B.4 sjeng-mcf-milc-dealII: Variation in EDP with monitoring period ............................ 132
B.5 mcf-nilc-namd-dealII: Variation in EDP with monitoring period ............................ 132
B.6 mcf-mcf-mcf-mcf: Variation in EDP with monitoring period ................................ 133
B.7 libq-libq-libq-libq: Variation in EDP with monitoring period .............................. 133
B.8 mcf-milc-quantum-sjeng: Variation in EDP with monitoring period ......................... 133

C.1 Quantitative comparison of our variable way DNUCA implementation with Bardine’s
heuristic ................................................................. 135
C.2 TWSS.D: Sensitivity to L2 slice size .............................................. 135
C.3 fluidanimate: Variation in EDP with monitoring interval ................................. 136
C.4 swaption does not exhibit any active line replacement during execution. Variation in
EDP with monitoring interval .................................................. 136
C.5 x264: Variation in EDP with monitoring interval ........................................... 136
C.6 FFT: Variation in EDP with monitoring interval ............................................. 137
C.7 Barnes does not exhibit any active line replacement during its execution. Variation in
EDP with monitoring interval .................................................. 137
C.8 Water spatial and water nsquared does not exhibit any active line replacement during its
execution. Plots show variation in EDP with monitoring interval .......................... 137
C.9 Radix: Variation in EDP with monitoring interval ........................................... 138
C.10 ocean: Variation in EDP with monitoring interval ........................................... 138
C.11 raytrace: Variation in EDP with monitoring interval ........................................ 139
C.12 mpegenc: Variation in EDP with monitoring interval ....................................... 139
C.13 mpegdec: Variation in EDP with monitoring interval .................................... 139
C.14 Sensitivity to L2 slice size: EDP normalized w.r.t. the reference SNUCA execution ... 140
C.15 Multiprogramming SPEC 2006 workload scheduled on a quad-core CMP with shared
4MB L2 cache ................................................................. 141
Keywords

Chip Multiprocessor, Cache, Leakage Power optimization, Working Set Estimation, Genetic Algorithms
Notation and Abbreviations

- CMP: Chip Multiprocessor
- NoC: Network On Chip
- NUCA: Non-Uniform Cache Architecture
- SNUCA: Static Non-Uniform Cache Architecture
- DNUCA: Dynamic Non-Uniform Cache Architecture
- WSS: Working Set Size
- TWSS: Tagged Working Set Size
- CLA: Cache Line Address
- AAL: Average memory Access Latency
- CMR: Cache Miss Ratio
- NORC: Near Optimal Remap Configuration
- TWSS.S: Adaptable associative SNUCA cache implementation using TWSS
- TWSS.D: Adaptable associative DNUCA cache implementation using TWSS
- DHP: Adaptable associative SNUCA cache implementation using Dhodapkar’s WSS estimation method
- w.r.t.: with respect to
Chapter 1

Introduction

1.1 Motivation

Complex high performance processors, such as superscalars are unable to deliver increasing performance requirements in a power efficient manner, due to limited instruction-level parallelism present in an application. Hence, chip multiprocessors (CMPs) proliferated in server and desktop platforms. In CMP, multiple smaller processor cores are integrated on the same chip. Programmers can exploit thread level parallelism present in an application using smaller processor cores. For a multiprogramming workload, higher performance is achieved by executing multiple programs concurrently on different processor cores present on a CMP. This enables CMPs to deliver higher peak throughput and greater performance-to-power ratio than the monolithic wide superscalar processors[1, 2]. CMPs are also easier to scale because of their distributed computing design. As a result, the number of cores on CMPs have been increasing. Throughout this thesis, we use the word “core” instead of “processor core” for simplicity.


Clearly, with advances in technology, the number of cores and on-chip cache size on a CMP have been increasing. Over the last two decades, on-die cache size in Intel processors has increased from 10KB to over 10MB. This has increased total percentage area occupied by the caches in those processors from 10% to 40% of total chip area [7]. Over next ten years, authors [7] expect increase in on-chip logic transistors by 3X and cache transistors by 10X. The area occupied by on-chip L3 cache on Itanium processor of size
Chapter 1. Introduction

374 \text{mm}^2 \text{ is } 175 \text{mm}^2, \text{ which is almost 50\% of the total chip area } [8, 9]. \text{ As a result, power consumed by on-chip cache has already become a major contributing component of power dissipated in the memory subsystem. Most of these processors have the flexibility of choosing cache size. But dynamic cache resizing is yet to be implemented in processors. Softcore processors such as ARM, Tensilica enable designers to choose cache configuration before synthesis [10, 11, 12]. However, cache configuration once chosen, remains fixed throughout its lifetime. We believe, providing a run-time configurable cache has now become more essential due to growing size of an on-chip cache present on the desktop and server platforms. Hence, we explore techniques to reduce leakage power consumption of caches in the context of large cache on a CMP, where cache tuning becomes more complicated due to the presence of concurrently executing threads.}

Reducing supply voltage of less frequently used cache lines [13] and completely cutting off their supply voltage [14] are two major streams of solutions to reduce leakage power consumption in caches. Powell et al.[14] proposed gated-Vdd technique, where memory cells can be disconnected from the power supply using a gated-Vdd transistor. Since a memory cell loses its data on power off, this might cause additional cache misses if the required amount of cache is not allocated. Hence, this is a data destroying technique of reducing leakage power consumption. On the contrary, K. Flautner et al.[13] proposed a data preserving technique of reducing leakage power in caches. In the drowsy cache, the supply voltage of rarely used cache lines is reduced so that these lines retain their data, while dissipating lesser static or leakage power. They call this mode “drowsy mode”. To access data from the cache lines in drowsy mode, supply voltage of a cache line is increased back to its normal supply voltage. With 70nm technology, the ratio of leakage power dissipated in the non-drowsy mode to drowsy mode is 12.5 [13]. The same ratio reduces to 4 for 45nm technology [15]. Hence, the leakage power savings obtained in 45nm technology are lesser than that obtained in 70nm technology using dynamic voltage scaling (DVS) method. This ratio will become lower for 32nm and 22nm technologies. This is because data preserving techniques rely on a fine grained adjustment of SRAM cell electrical parameters. These parameters are less controllable in smaller transistor sizes because of larger process variation effects [16, 17, 18]. The noise margin of CMOS degrades as supply voltage is reduced [16]. This makes caches operating at lower supply voltage more susceptible to transient errors caused by alpha particles [19, 18, 20]. Charge gets collected at the junction of a transistor on an alpha particle strike. A bit flip can happen if the collected charge is greater than $Q_{\text{critical}}$. $Q_{\text{critical}}$ is a minimum charge required to flip a bit stored in the memory cell. $Q_{\text{critical}}$ decreases by 5X as voltage is scaled from 1V to 0.4V in 65 nm technology [18, 19]. This increases soft-error probability at lower voltages. Hence, effectiveness of a data preserving dynamic voltage scaling technique reduces at lower transistor sizes [19, 18, 21]. This makes the complete switch-off of the cache as the only viable solution for future caches to reduce their leakage power consumption.

Another possible solution to reduce leakage power consumption in caches is the use of different
Chapter 1. Introduction

types of memories that dissipate lesser leakage power. Commonly used Static Random Access Memory (SRAM) has higher leakage power. Other memory technologies, such as embedded DRAM (eDRAM), Magnetic RAM [22] and Phase change RAM (PRAM) [23, 24] are being considered as alternatives to the DRAMs and SRAMs in memory hierarchy due to their dense implementation and lower leakage power consumption [25]. Since PRAM, MRAM and eDRAMs have much higher cache density than SRAM, larger cache can be implemented in the same chip area constraint. However, PRAM and MRAM are very slow compared to SRAM and even DRAM. These memories are non-volatile and dissipate negligible static power. However, in addition to higher access latency, PRAM also suffers from a major drawback of limited “write endurance”. Write endurance is the maximum number of memory write operations supported by each cell. PRAM cell has a write endurance of $10^6$ to $10^8$ [26], which makes PRAM unsuitable for on-chip cache implementation.

Another dense technology is embedded DRAM (eDRAM). IBM BlueGene/L [27] server has an on-chip eDRAM Level 3 or L3 cache. IBM’s latest multicore high frequency (5.2 GHz) chip [28] implements 24MB shared L3 cache using eDRAM. This chip has 4 cores, 1.5MB private SRAM L2 cache and 24MB shared L3 cache. IBM Power7 also has 32MB on-chip eDRAM L3 cache. However, eDRAM needs to be refreshed periodically to retain its data. Hence, in addition to leakage power eDRAM also consumes power to refresh its data. Considering these factors, it seems that dynamic voltage scaling is not an easy solution to reduce static power consumption in eDRAMs. But an eDRAM bank can be switched off to save leakage and refresh power consumed by it.

In the past, the data destroying switch off policy was explored in the context of uniprocessors and smaller uniform access latency caches [29, 30, 31]. In this thesis, we explore power gains obtained with the data destroying switch off policy [31] for a CMP with large on-chip cache. Large on-chip cache on a CMP is implemented using multiple banks interconnected using on-chip network (NoC) [32]. Such a cache offers variable latency to the cores on a CMP. Hence, it is referred to as “Non-Uniform Cache Architecture (NUCA)”. Estimation of cache requirement and its resizing, is a challenging problem due to non-uniform access latency of NUCA caches. Commonly used metrics such as memory access latency [33] and cache miss ratio [34, 35] give sub-optimal energy savings for NUCA caches. Cache coherence due to concurrently executing threads further complicates cache resizing. Hence, we believe, it is necessary to study leakage power saving techniques in caches, especially in the context of CMPs with nanometer transistor sizes, large non-uniform access latency caches and concurrently executing threads.

Our work is more focused on desktop, server platforms and not on embedded platforms. This is because embedded platforms use smaller scratch pad memories which are managed by the programmer or compiler [36, 37] or have comparatively smaller on-chip cache. e.g. Intel’s Atom processor used in smart-phones, netbooks and embedded products, employs single or dual cores each with two threads (hyperthreading). It has a small on-die 512KB L2 cache. ARM’s Cortex-A9 MPCore series used in
embedded platforms like set-top boxes, portable devices and PDAs, supports 128KB to 8MB of on-die L2 cache. However, its cache is configured at design time and not at run-time. For embedded platforms, designers have a prior knowledge of applications that will be executed on these devices. Hence, determination of optimum cache configuration is feasible at design time [38, 39, 40, 41, 42]. As a result, only limited over-provisioning of cache is observed on embedded platforms. e.g. Tensilica’s Xtensa processor [11] allows cache to be customized at design time. Some ARM cores [10], such as ARM 9 and ARM 11 families provide configurable cache with size between 4KB and 128KB. Cache associativity can be set statically in MOTOROLA M*CORE [43] according to the cache requirement of an application. Hence, we focus on desktop and server platforms in this thesis. Our work is independent of memory technology. The drowsy cache technique [13] operates at fine granularity and is only applicable to SRAM technology. On the contrary, our techniques can be applied to any other memory technologies such as eDRAM and SRAM, where leakage power difference between on-state and off-state of a cache bank is significant.

![Tiled architecture used for study](image)

**Figure 1.1:** Tiled architecture used for study

### 1.2 Methodology

We consider a tiled architecture similar to Tilera [5] in our study, as this architecture is advocated as a scalable CMP. [44]. Our experimental architecture is a sixteen tiled CMP shown in Fig. 1.1. Each tile contains a core, a split instruction and data L1 cache of 32KB. Each tile also contains a slice of L2 cache. The size of L2 slice in each tile is 512KB. The L2 cache is distributed and shared by all cores. To maintain cache coherence between private L1 caches, a directory is present in each tile. These tiles are interconnected via 2D-mesh NoC and per-tile router. We assume 32nm CMOS technology.

A large cache is implemented with many banks. Such a large cache offers non-uniform access latency to different cores, unlike, monolithic caches. Hence, Kim et al.,[32] proposed two policies to access NUCA cache, which are Static NUCA (SNUCA) and Dynamic NUCA (DNUCA), as shown in Fig. 1.2(a) and Fig. 1.2(b), respectively. In SNUCA, the predetermined bits of a memory address determine the tile in which data is cached. However, such a cache offers large access latency due to its distributed nature.
Chapter 1. Introduction

5

(a) SNUCA

(b) DNUCA

Figure 1.2: SNUCA and DNUCA cache access policies

Figure 1.3: normalized power consumed by various memory subsystem components

Hence, Kim et al. [32] proposed DNUCA cache to reduce access latency. In DNUCA, the whole address space is mapped onto a column. The predetermined bits of a memory address, decide the row in which data is cached. All L2 slices in a row form a bankset and data can be cached in any of these slices. On an L1 miss, the data is first searched in the nearest L2 slice and then in rest of the L2 slices in that row, before reading it into the nearest L2 slice. The data is migrated towards a nearer slice depending on the number of accesses made by the core.

We execute applications from Splash2[45] and Parsec[46] benchmark suites which spawn sixteen threads on a 4x4 tiled CMP with SNUCA access policy. We use SES C [47] to simulate a core, Ruby from GEMS [48] to simulate the cache hierarchy and interconnects. DRAMSim [49] is used to model the off-chip DRAM. DRAMSim uses MICRON power model [50] to estimate power consumed by DRAM accesses. Intacte [51] is used to estimate low level parameters of the interconnect such as the number of repeaters, wire width, wire length, degree of pipelining and power consumed by the interconnect. Power consumed by the cache components and their area is estimated using CACTI 6.0 [52]. More details of
our experimental methodology can be found in Chapter 3. Fig. 1.3 shows power consumed by various components of the memory subsystem, normalized with respect to the total memory subsystem power consumption. Following conclusions can be drawn from Fig. 1.3:

1. The leakage power dissipated in on-chip cache is a major power component for most of the applications.

2. In ocean [45], power dissipated in DRAM is almost equal to that in cache. It indicates that ocean makes a significant number of off-chip DRAM accesses and, hence, its working set size (WSS) is greater than the cache size.

3. For other applications, power dissipated in DRAM is negligible. It indicates that WSS of these applications is smaller than the cache size. In such applications, cache utilization is lesser, that is cache holds more dead lines than live lines [53].

Hence, by allocating the required amount of cache and switching off the extra cache, leakage power dissipated in cache can be reduced without affecting performance adversely. In our experiments with SNUCA cache access policy, we additionally found that, accesses made by different threads are dispersed over all L2 slices, which offer non-uniform cache access latencies to the cores.

The leakage power dissipated in the cache is proportional to size of the powered-on cache. If less than the required cache is allocated, then the number of off-chip DRAM accesses would increase, degrading execution time of an application. On the contrary, leakage power consumption of cache would increase on allocating more than the required cache. Hence, we use energy-delay product as a performance metric which indicates a trade-off between energy consumption of an application and its execution time (delay). Energy-delay product (EDP) for an application execution is calculated as:

$$ EDP = Energy \times ExecutionTime $$(1.1)

To summarize, our goal in this thesis is to reduce energy dissipated in the memory subsystem components during an application execution without significantly degrading its performance. This reduces EDP for an application execution.

### 1.3 Organization of the thesis

Our main contributions in this thesis are:

- Chapter 2 provides a survey on the prior work done in the area of power-optimized caches. It also gives detailed information on the various metrics used for optimizing cache configuration.
• A proposal for estimating a low-overhead working set size (WSS) estimation method: Optimal cache configuration depends on the characteristics of an application. Various metrics have been used in the past to determine optimal cache configuration. D. Albonesi uses offline method to determine optimal cache associativity in [31]. Other researchers have used parameters such as cache miss ratio and memory to L1 cache data traffic [54], time interval between two accesses to a cache line [30], miss rate, instructions per cycle (IPC) and branch frequency [55] to dynamically adjust the cache size. These metrics are monitored periodically. The cache optimization decision is taken by comparing the values between consecutive monitoring intervals. These parameters form an indirect mechanism for estimating WSS. This might introduce large errors in some cases due to concurrently executing threads/processes and also due to dispersed cache. Hence, we propose a new method to estimate working set size of an application. Chapter 4 describes details of our WSS estimation method.

• A scalable algorithm to adjust cache associativity according to the cache requirement of an application - With increase in the number of components on a chip, NoC has become complex, making its latency non-negligible [56, 57]. Moreover, due to distributed nature of NUCA cache, usage of various L2 slices becomes uneven. Hence, commonly used metrics such as cache miss ratio [34, 35] and average memory access latency [33], which were used in the past to reconfigure cache, deliver sub-optimal performance. We propose and implement a scalable algorithm which uses information locally present in a tile to adjust cache associativity. Apart from avoiding NoC access, it achieves higher EDP savings as cache associativity of each L2 slice is adjusted according to its usage. Chapter 5 elucidates more on this.

• The remap policy - The number of cache misses increases on reducing cache associativity. Hence, instead of switching off cache associativity, we propose to switch off some of the powered-on L2 slices. The powered-off L2 slices are mapped on-to the powered-on slices using what we call the “remap policy”. Some applications such as H.264 [46] show poor thread scalability. Hence, even after specifying sixteen threads, the application spawns fewer than sixteen threads concurrently. Such applications show improvement in their execution time by switching off farther L2 slices and mapping them on-to nearer slices. Chapter 7 gives more details of the on-line determination of the remap configuration and its implementation.

• Determination of the near optimal remap configuration - Though the remap policy achieves higher EDP savings than the variable way policy, its hardware implementation is more complex than the latter. Hence, to estimate maximum possible energy gains that the remap policy can achieve, we determine a near optimal remap configuration as minimization of a energy-delay product. We solve this problem using scalable genetic algorithms. Chapter 6 elucidates more on this.
• Though we have implemented the remap policy in SNUCA cache, its implementation with DNUCA cache is also possible. However, gains obtained with these two implementations depend more on SNUCA and DNUCA policies than the remap policy. This in turn depends on the workload characteristics. In Chapter 8, we explore the characteristics of an application that favours SNUCA or DNUCA cache access policies. Using these characteristics, we propose a trace based tool to determine suitability of a NUCA cache access policy for an application. This model will help architects to choose a suitable cache access policy at design time. It is also helpful to characterize data sharing pattern among threads in an application.

• Sapphire Simulator - To explore power and performance trade-offs on a CMP, we use open-source SESC simulator [47]. SESC enables exploration of micro-architectural parameters of a CMP. It is very fast since it does not simulate the full system. However, it only provides shared bus model for the on-chip network. Hence, we have integrated Ruby from GEMS [48], which provides the detailed model of various cache coherence protocols and user configured switched network. We refer to our simulator as “Sapphire”. Sapphire uses cycle-accurate DRAMSim simulator [49] to model off-chip DRAM. Sapphire can be downloaded from [58]. The details on our simulator and simulation methodology can be found in Chapter 3.

• Finally, Chapter 9 deals with conclusions and future work.
Chapter 2

Background and Related Work

We propose power optimized last level cache for a chip multiprocessor in this thesis. The main steps involved in solving this problem are selection of a metric to estimate working set size and use of estimated WSS to switch-off cache. Prior studies have proposed power optimization techniques for small uniform access latency caches on uniprocessor platforms. Whereas, we study the same for large NUCA cache on a chip multiprocessor (CMP). Hence in this chapter, we begin our survey with work done on NUCA cache. We also provide studies which specifically attempt to optimize power dissipated in caches. Since we have proposed a new method to estimate working set size of an application, we also provide a brief survey on various metrics used to estimate cache requirement of an application.

2.1 NUCA caches

The cache size on a CMP has increased due to the increased number of processor cores. The access latency and power consumed by caches increases with cache size. Hence, large caches are implemented using smaller banks. This reduces its per-access dynamic power at the expense of non-uniform access latency experienced by various cores. Hence, such large caches are referred to as Non-Uniform Cache Architectures (NUCA). As explained in Chapter 1, Kim et al. [32] proposed SNUCA and DNUCA cache access policies. In case of NUCA cache on a CMP, many decisions need to be taken, such as:

1. Private Vs Shared: The lower level on-chip cache can be organized as private for the core or shared among all processor cores or subset of cores.

2. Line Search: It decides how cache line is searched in the last-level shared cache on a miss in primary cache.

3. Line Placement: It decides the bank in which a cache line is placed when it is read from the off-chip memory.
4. Line Migration: It decides how and when a cache line is migrated at run-time depending on accesses made by the primary caches.

5. Line Replacement: It decides which cache line is replaced when a new cache line is read from off-chip DRAM. Replacement and block placement are two distinct policies. Former decides which cache line to evict to make a place for the incoming line, whereas latter decides where to place the incoming line. Considering locality of accesses, the incoming line is placed near to the core and some cache line present in nearer bank is moved into a place, vacated by an evicting cache line.

The NUCA studies given below attempt to make cache accesses faster and do not specifically try to optimize power consumed by them. However, survey done on the NUCA caches will help reader to understand working and complexity of the same. We also explain in the end of each subsection what assumptions we have made for the architecture that we study.

2.1.1 Private Vs Shared cache

In the private last level cache (LLC), each core creates its own copy of a data block. This reduces aggregate capacity of LLC. Hence, to increase the aggregate cache capacity, LLC is shared among all processor cores. However, this increases cache access latency as data could be in farther bank. This also increases cache misses incurred due to unrelated processes sharing LLC. Hence, researchers have proposed variants of private and shared LLCs [59, 60, 61, 62]. Victim cache [59] considers a shared cache on a tiled architecture. Victim cache saves a cache line evicted from the primary cache in local L2 slice. This makes a replica of the cache line, which is already present in its home L2 location, in the nearest L2 slice. In order to retain large aggregate cache capacity of shared L2 cache, victim cache does not evict a global cache line with remote shares in favour of a replica of the local evicted primary cache victim. Increasing replication decreases aggregate cache capacity. This in turn increases the number of L2 misses. Hence, in [63], authors selectively take replication decision for every set. They monitor the number of hits to remote blocks and also hits to the last 1K least recently used cache blocks in each set. Using these values they estimate whether replication will be beneficial or costly.

Victim cache uses shared cache to create private copies, whereas, Chang et. al [60] use private L2 caches as globally shared L2s, using a centralized coherence engine (CCE). On a miss in local L2, CCE is contacted. If the address is cached by remote L2, then the request is forwarded to the remote L2 and a copy is made in the local L2. However, the centralized CCE faces a scalability issue with increase in the number of cores present on a CMP. Hence, Hererro et. al [61] use distributed coherence engine. With distributed coherence engine, they also show reduction in the power consumption over CCE technique. In [61], the L2 cache is marked as private and cache space from the remote L2s is utilized to save the spilled blocks using coherence engines, whereas, in ElasticCC [64], Herrero et. al divide L2 cache in each
node logically into the private and shared regions. They adjust private and shared regions adaptively for each node, whereas, ASP-NUCA [65] maintains total size of private and shared regions constant. If the private region is increased for some node, it is decreased for another node. Hence, in case of all nodes executing single threaded processes, large private caches cannot be allocated for all of them. Beckmann et. al [63] proposed a cost model which decides whether to make a copy of the L2 cache line in the private region or to share it with remote L2 slice. However, this approach needs a lot of additional hardware to estimate the cost.

Above policies, divide cache space of each L2 slice among private and shared regions, whereas, [33, 66, 67] regroup L2 slices so that they are shared by varying number of cores. The number of cores sharing an L2 slice is called its sharing degree. In [66], authors experimented with sharing degree of 1, 2, 4, 8, 16 and 32. They conclude that for SNUCA policy, one to four sharing degree works best. Whereas, for DNUCA, higher sharing degree reduces hit latency but at the cost of increased complexity and power consumption. M. Hammoud et. al [33] dynamically vary sharing degree in a tiled SNUCA architecture. Cores along with L2 slices in their tiles form a cluster. L2 slices are shared by all cores in that cluster. The average memory latency is used to determine the cluster configuration. Each core runs the estimation algorithm and determines the cluster configuration suitable for that core.

Above mentioned policies partition cache space between processors dynamically using micro-architectural modifications, whereas M. Kandemir et al. [68] use compiler techniques to partition memory banks between cores on CMP platform for embedded applications.

We consider private split L1 cache and shared L2 cache which is distributed in all tiles.

2.1.2 Line search policies

On an L1 miss, data is searched in lower level cache using line search policy. As mentioned earlier, Kim et al. proposed SNUCA and DNUCA cache search and block placement policies. Hardware implementation of SNUCA policy is simple, since on an L1 miss, a single access is made to the home location of that address in L2 cache. The predetermined bits of a memory address determine home location of the address where data is cached in the L2 cache. However, in case of DNUCA, as explained earlier, the lookup logic is complex and all banks in a bankset are searched on an L1 miss. This not only increases time spent in cache lookup before reading data from off-chip DRAM but it also increases dynamic power spent in cache lookup. Hence, in [69], authors maintain a remap table per core and mark some L2 banks as preferred and rest are marked as not-preferred. When the L2 cache is searched for a hit, the preferred L2 banks are searched in parallel. On a miss in the preferred banks, the non-preferred banks are searched in parallel. The same table is also used to partition the L2 banks among cores, thereby reducing the conflict misses in the shared L2 cache incurred due to different concurrent applications.

We use basic line search implementation in DNUCA policy, where on a miss in the nearest L2 slice,
rest of L2 slices in the bankset are searched. If data is not present in any of the L2 slices in the bankset then it is read from the off-chip DRAM into the nearest L2 slice.

2.1.3 Line placement policies

When data is read from the off-chip memory, line placement policy decides which bank receives that data. In DNUCA, data can be cached in any of the banks in a bankset. Hence, access latency can be reduced by placing cache line in a nearer L2 slice. In SNUCA as well as DNUCA, tag array is kept near to the data array. Tag search and read from data array can happen concurrently or serially. As tags are stored near to the data array, it requires NoC traversal. Hence, Chisti et al. [70] disassociated tag arrays from data arrays. Tag array is kept close to the core and data arrays are dispersed on the chip. Tag is accessed followed by data access which reduces the number of data accesses by 61% compared to NUCA. Because of disassociation of tag and data arrays, hot data from any set can be located near processor. In CMP NuRAPID [71], authors extend tag-data disassociation logic for CMPs by having private tags per core and shared data arrays.

In PageNUCA [72] proposal, SNUCA is considered as a base configuration which uses physical address for cache search. In PageNUCA, the remap logic is used to map physical page to a logical page so as to reduce cache access latency. whereas, we remap the entire less utilized L2 slice to another L2 slice and the remapped L2 slice is switched-off to reduce leakage power consumed in it. In [73], authors similar in PageNUCA [72], also take a cache line placement decision at the page level. In PageNUCA decision is taken and implemented completely by hardware, whereas, in [73, 74], operating system takes a decision after a physical page is allocated to a virtual page. In Software page mapped [73] and hardware mapped PageNUCA [72], cache line placement decision is taken at the granularity of a page, whereas, RNUCA [75] classifies data into private, shared and instructions at L2 cache line level and uses a separate placement policy for each of them.

2.1.4 Line migration policies

In case of hit to a cache line, line migration policy is invoked to take a line migration decision. In [76], Beckmann and D. Wood studied block migration technique present in DNUCA for a CMP. Block migration lowers access latency in case of uniprocessors with NUCA caches. However, they show that in case of 8-core CMPs, most of the hits are from central banks which are not near to any of the cores. Instead, they achieve lower latency for farther banks using ultra fast transmission lines. In Chapter 8, we propose indices to characterize data sharing properties of an application to detect whether a line migration in DNUCA might cause more overhead and degrade its execution time when compared to that in SNUCA cache.
In our implementation of DNUCA, on the second consecutive access from the same direction, a line migrates in that direction by one hop.

### 2.1.5 Line replacement policies

As data location is decided by the predetermined bits from a memory address in case of SNUCA, traditional replacement policy based on pseudo-LRU is applicable. However, in case of DNUCA, set spans across multiple banks. Hence, traditional replacement policy cannot be used. Lira et al.\[77\] proposed a replacement policy for DNUCA cache. They assigned different priorities to

1. data evicted from L1 cache,
2. data demoted from banks nearer to cores,
3. data arrived from off-chip DRAM and
4. data promoted from central banks towards nearer banks.

In their implementation, higher priority data replaces lower priority data. This continues till either all banks are searched or evicted data becomes lowest priority data. This may trigger too many evictions. Hence in [78], the global replacement policy is implemented by conducting auction.

A. Jaleel et al. [79] adjust replacement policy of shared cache according to the application’s behaviour. They classify applications into cache friendly, cache fitting, thrashing and stream applications depending on their cache usage behaviour. They monitor cache behaviour of each application at runtime and change replacement policy between Least Recently Used (LRU) and (Bimodal Insertion Policy) BIP. BIP avoids thrashing by inserting majority of cache lines in LRU manner and some in most recently used (MRU).

We use pseudo LRU policy in both SNUCA and DNUCA implementation. The cache line replaced from L2 slice is sent to off-chip memory in both the cases.

### 2.1.6 Cache coherence management

Multiple threads which belong to the same application share data among themselves. This might cause the same data accessed and modified by different threads at the same time. To ensure correct program execution, private caches are maintained coherent. This is achieved by maintaining directories at the shared cache. The directory maintains information regarding the identification of each L1 sharer which has cached that data. Usually, a bitmap is maintained for each cache line. If the bitmap has a bit for every possible L1 cache sharer, then it is termed a full bitmap. If a single bit is maintained for each cache sharer, then in the case of a CMP with 64 cores, each cache line has to maintain 64 bits in a bitmap. For 8MB cache and 64B cache line, the directory itself would be 1MB. Hence, it is very important to
maintain directory information in a scalable manner. The storage requirement of full bitmap increases proportionately with the number of sharers and cache size. To reduce the storage requirement, a coarse bitmap is maintained, at the expense of increased invalidation traffic. While invalidating a cache line, invalidations are sent to every sharer represented by a bit in the coarse-grained bitmap. This increases invalidation traffic. [80] mitigate false invalidation problem by maintaining a bloom filter inside each router of the NoC. To reduce directory size, H. Zhao et al. [81] propose to maintain a pattern table. They observed a very few number of distinct sharing patterns in an application. Hence, they propose to maintain a pointer to an entry in the pattern table with each cache line, instead of maintaining a sharing pattern itself.

In our implementation, we maintain a full bitmap of L1 sharers along with each L2 cache line. For cache lines present exclusively in the L1 cache, a directory table is maintained in each tile. Such lines are called non-inclusive cache lines in the literature and the cache non-inclusive cache. We assume size of the directory table as 5% of size of the L2 slice. The directory table is used for non-inclusive cache lines, whereas, a full sharer bitmap is maintained along with tag bits for inclusive cache lines, i.e. for the cache lines present in both L1 and L2 cache.

2.2 Cache contention prediction

The reuse distance is a very well known technique used to estimate cache misses for a cache of any size [82, 83, 84, 85]. The reuse distance is the number of addresses accessed between two accesses to the same address. If the number of distinct addresses are considered between two consecutive accesses then it is termed a stack distance. The reuse distance of all addresses is plotted in the form of histograms. Addresses with larger reuse distance than the cache associativity result in the cache misses. Tam et al. [82] collect L2 access trace using hardware performance counters and then generate cache miss rate curve using Mattson’s stack algorithm [86]. Mattoson [86] stack algorithm makes use of the inclusion property, which states that the contents of a memory of size $K$ pages is subset of contents of a memory of size $K + 1$ pages. G. Suh et al. [87] use cache miss rate, as function of cache size and time quantum for which a process runs on a multiprocessor systems. They use this information to predict cache footprint of a process, before the process gets context switched. Using this information, they estimate the overall cache miss rate on a multiprocessing system. In [88], Merke et al. maintain a task vector which tracks resource utilization per process. This vector is populated with the help of events obtained from the performance monitor. OS uses task vector to schedule different processes so as to reduce cache evictions.

Chandra et al. [83] proposed three performance models to predict cache misses incurred in a shared L2 cache on co-scheduling two applications on a dual core CMP. They use stack distance or circular
sequence profile determined by individually scheduling an application. Chi et al. [89] predict effective cache used by the processes scheduled on a dual-core CMP by using a relationship between reuse distance and cache access frequencies. Unlike [83], [89] use performance counters on a real platform to predict cache contention. [83, 89] consider multiprogramming workload on a dual core CMP. [90, 84] and Zhong et al. [91] make reuse distance evaluation faster by using sampling techniques.

Most of above mentioned studies either enhance cache miss rate predicting techniques or demonstrate use of cache miss rate for some optimization in case of a multiprogramming workload. Whereas, in this study, we consider multi-threaded applications and quantify the sharing characteristics at the granularity of a cache line. We show its use to determine a more profitable cache access policy between SNUCA and DNUCA.

### 2.3 Power efficient caches

In this section, we survey the studies which explicitly attempt to reduce the static or dynamic power consumed in caches. Power consumed by a digital circuit is governed by the following equation:

\[
P = \alpha CV^2 f + I_{\text{leakage}} V
\]  

(2.1)

where, \( \alpha \) is switching activity of a transistor, \( C \) is capacitance, \( V \) the power supply voltage, \( F \) is the clock frequency and \( I_{\text{leakage}} \) is the leakage current. The first term in Eq. 2.1 gives the dynamic power and the second term gives the static power consumed by the circuit. The supply voltage has reduced with advances in technology. This has significantly reduced dynamic power consumption in the digital circuits. However, the switching speed of a transistor depends on the difference between the supply voltage and its threshold voltage. Hence, to achieve faster switching time and sufficient noise margins, threshold voltage should also be reduced for the reduced supply voltage. The dependence of a leakage current on transistor’s threshold voltage and temperature is given by equation:

\[
I_{\text{leakage}} = e^{\frac{-qV_T}{nkT}}
\]  

(2.2)

where, \( V_T \) is the threshold voltage, \( T \) is temperature, \( q \) is \( e^- \) charge, \( n \) is body effect factor and \( k \) is the boltzmann constant. \( q, n, k \) are constants. Eq. 2.2 implies that reduction in the threshold voltage increases leakage current. As the on-chip cache occupies a significant area on the chip, its leakage power has become a major power consuming component of the memory subsystem. Various attempts have been made to reduce dynamic and static power consumption of the caches at architecture [31, 92, 93] and circuit level [14, 94, 13]. The dynamic power consumption reduces on reducing the switching activity, whereas, the static power reduced on reducing the leakage current.
2.3.1 Dynamic power reduction

Bardine et al. [93] use a single bit per cache line in the case of DNUCA to avoid unnecessary cache line promotions/demotions. Using this technique, they show reduction in the dynamic power consumption of the DNUCA cache for a single or dual core scenario. In [31], D. Albonesi selectively disable the set ways according to the cache requirement of an application to reduce dynamic power consumption in the cache. They use offline mechanism to determine cache requirement of an application and adjust the number of ways of a set associative cache accordingly. Rest of the ways are kept in disabled state. On a miss in the primary cache ways, the disabled secondary cache ways are searched. Dropsho et al. [29] maintain hit counters along with each cache way and use the number of hits in each counter to adjust the cache associativity. They adjust cache associativity such that performance does not degrade beyond a predetermined value. Benitez et. al [95] propose the use of field programmable gate arrays to implement a reconfigurable L1 data cache. Authors detect the phase change within an application executing on a uniprocessor and change the FPGA cache layout only on phase changes. Even though the access time of the FPGA cache is slightly greater than the cache built using SRAM, authors show overall savings in the execution time by allocating the required amount of FPGA cache than the over-allocated large SRAM cache. The smaller cache has lower access latency. Hence, the smaller FPGA cache shows improvement in the execution time over the fixed larger SRAM cache. On reconfiguration, the cache has to be flushed. This overhead is reduced by configuring it at only application phase changes. Authors use offline method to determine a suitable cache configuration for each application phase. An application phase is identified at run-time and cache is configured accordingly. Implementing FPGA cache as an LLC on CMP is not feasible. This is because offline determination of a suitable cache configuration with many applications executing concurrently on CMPs is not practical.

T. Johns et al. [96] save frequently accessed instructions in the beginning of a program binary using compiler modifications. They reserve some of the cache ways of the instruction cache to save these frequently accessed instructions. With this, full tag lookup can be avoided, savings the dynamic power in case of the embedded processor. Zhang et al. [97] use offline method to determine optimal cache configuration. They first determine optimal cache size, then the best line size and finally the best associativity to obtain the most suitable cache configuration for an embedded platform, executing single application at a time. For every configuration, they evaluate energy consumed in the cache and compare against energy consumed with the previous configuration. They iterate through the four loops to determine the optimum cache size, line size and associativity. M. Rawlins et al. [98] extended the same work for dual heterogeneous core scenario. There is a lot of work done in the past to improve dynamic power dissipated in caches, however, due to the smaller transistor sizes, dynamic power has already become very negligible compared to the static power consumption.
2.3.2 Static power reduction

The circuit level, architecture level techniques and the use of different kinds of memory are various ways to improve static power consumption in caches. **Circuit level techniques**

**Gated-Vdd:** Powell et al. [14, 35] suggested the use of a gated transistor to disconnect a memory cell from the power supply. When two transistors are connected in series, leakage current is reduced by orders of magnitude. This is due to the self reverse biasing of stacked transistors. The gated transistor connected in series with the cell transistor, causes stacking effect when it is turned off. This reduces leakage current by orders of magnitude. The memory cell is powered on when the gated transistor is powered on. With gated-Vdd technique, memory cell loses its data contents. If the same data is accessed again, it results in a cache miss incurring much larger access latency. Hence, care has to be taken while keeping cache lines in switched-off mode.

**Variation in Threshold Voltage:** Leakage power of a transistor can be reduced by increasing its threshold voltage. However, this reduces its switching speed, making the cache slower. On the contrary, decrease in the threshold voltage, increases its speed at the expense of leakage power [99, 100]. Dropsho et al. [101] statically use the low $V_T$ (high speed, high leakage) transistors on the critical paths and high $V_T$ (low speed, low leakage) transistors on the non-critical paths. On the contrary, in [102, 103], the threshold voltage is dynamically changed in multi-threshold CMOS (MTCMOS) by adjusting the back bias voltage. With this technique, a memory cell can be put in low leakage sleep mode without losing its data. In the active mode, it can be accessed at the full speed. The sleep to active mode change is done by adjusting the backgate bias voltage. Kim et al.[94] dynamically adjust the threshold voltage by applying forward body biasing to the selected SRAM cells.

**Dynamic Supply Voltage (DVS):** Flautner et al. [13] proposed a drowsy cache in which supply voltage of the less frequently used cache lines is reduced to dissipate lesser leakage power without losing their data. The supply voltage of a cache line is increased to its normal operating voltage when data is accessed from the cache line. Authors assume latency of one cycle to change the state from drowsy to normal and vice verse. Hence, the drowsy cache technique achieves large energy gains with a negligible degradation in performance. However, previous studies show that the decrease in supply voltage of a cache line, increases its susceptibility to soft-error faults [16, 20, 18, 104, 105]. A soft error is a temporary error caused due to the alpha particles and neutron from cosmic rays. [106] proposed a model to estimate a soft-error rate (SER) in CMOS SRAM circuits caused due to neutrons:

\[
SER = N_{flux} \times CS \times \exp(-Q_{critical}/Q_s) 
\]

\[
Q_{critical} \propto V_{DD} 
\]

where, SER is the soft error rate, $N_{flux}$ is the intensity of neutron flux, CS is cross section area of a
node, $Q_s$ is the charge collection efficiency and $Q_{\text{critical}}$ is the critical charge. A soft error occurs if the charge collected at node is greater than $Q_{\text{critical}}$, i.e. if charge collected due to a particle strike is greater than $Q_{\text{critical}}$, then bit flip occurs. $Q_{\text{critical}}$ is proportional to the node capacitance and supply voltage. Hence, the reduced supply voltage in the drowsy cache, reduces $Q_{\text{critical}}$, thus exponentially increasing SER as seen from Eq. 2.4. This problem exacerbates with reducing transistor sizes. With increasing systematic and random process variation in the deep sub-micron technologies, the failure rate of SRAM structures rapidly increases. The amount of power savings achieved with the dynamic voltage scaling for low Vdd values ($< 651mV$) is restricted due to failure rate of the SRAM structures [104]. Different methods can be applied to overcome soft-error problem such as maintaining error correcting code (ECC) along with each cache line [107, 9, 108, 109]. Single Error Correction Double Error Detection (SECDED) is a special class of ECC which can detect two bit errors and correct a single bit error. However, ECC guards cache lines against transient errors at the heavy cost of power, complexity and area. Hence, in [108], authors propose an adaptive error correcting code to overcome soft errors in low power SRAMs. They propose to use a single error correction code for the dirty lines in the L1 cache as there is no other copy present in the lower level caches and use a single error detection code for the clean cache lines as, data can be recovered from the lower level cache line on detecting error. In [20], authors propose to use the drowsy cache to implement an instruction cache, as cache lines are not modified in the instruction caches. They propose to re-fetch instruction from the lower level caches on detecting soft-errors. They also propose the periodic scrubbing i.e. periodically checking and correcting errors in the cache lines to avoid multibit errors in the same cache line. This enables them to use a cheaper single ECC in the drowsy instruction cache. Process variation effects are more visible in the SRAM transistors as they are smaller than the logic transistors. Hence, minimum supply voltage that should be used in SoC depends on SRAM structures. TI’s OMAP4320 processor has separate supply voltage domains for the logic and SRAM structures [15]. This also indicates that though the supply voltage has reduced with smaller transistor sizes, it cannot be made equally small for the SRAM transistors.

Bhaskar et al.[110] compare effectiveness of various leakage power saving techniques in transistors such as reduction in the supply voltage, reverse body bias, stack effect and non-minimum channel length transistors. Among these techniques, reduction in the supply voltage causes highest degradation in the saturation current of a transistor. Hanson et al.[111] explore the use of various leakage reduction techniques, such as the dual threshold voltage, Gated-Vdd and MTCMOS for the L1 and L2 cache on uniprocessor platform with just one application executing at a time. They found that the low leakage transistors achieve the highest savings if used for L2 cache. The low leakage transistors have statically set high threshold voltage. Higher threshold voltage increases the access time of a transistor. Authors observed IPC degrades by 50% if the low leakage transistors are used for the L1 cache. Hence, they suggest to use the high threshold voltage transistors for the L2 cache and the low threshold transistors
Architecture Level Techniques: D. Albonesi [31] adjusts the cache associativity to reduce dynamic power consumed in cache, whereas, Bardine et. al [92] adjust the cache associativity to reduce static power consumed in DNUCA caches. In DNUCA, less frequently accessed cache line migrates to a more remote bank in a set, promoting frequently accessed cache line nearer to the core. Hence, they evaluate ratio of the number of hits in the farthest cache line in a set to the nearest cache line. If the ratio is lesser than the threshold, the farthest powered-on cache line is switched off. They study one or dual core processor in which case both the cores are on the same side of the shared bus and the DNUCA cache is on the other side of the interconnect. Hence, perspective of the nearest and farthest lines for both the cores is same. However, in a scalable tiled architecture the nearest cache line of one core is far for the other cores. Hence, their metric cannot be applied to a tiled architecture. We compare our results quantitatively against this metric.

Meng et al.[112] study variation in the leakage current in different memory cells due to variation in their physical parameters such as gate length, gate oxide thickness and dopant ion concentration. Different cache ways are sorted using the offline method as part of built in test (BIST). The cache ways leaking more leakage power are switched-off before the ways leaking lesser power. This technique will require a finer control over the cache ways to switch them off, whereas, in our adaptable associative cache, we assume a single way of all sets is implemented in a bank. Hence, switching off a bank reduces cache associativity of all sets by one. Y. Meng et al.[113] determine an ideal access interval of a cache line, when it should be kept in the active state, or changed into the data-preserving drowsy state or the data-destroying sleepy state. To determine the state changing inflection points, they consider oracle knowledge of all future accesses. This method helps to obtain limits of leakage power savings, if all future accesses are known. We apply genetic algorithms to determine the near-optimal savings obtained using the data-destroying remap policy in non-ideal and hence, more practical scenario. Yang et al. [114] dynamically estimate cache requirement by monitoring the cache miss rate and adapt the size of instruction cache accordingly. They use the circuit level gated-Vdd technique [14] to switch off the over-allocated cache. In this work, authors adjust the number of allocated cache sets. They evaluate their proposal only in the context of instruction caches. Hence, they need not handle dirty cache lines which typically occurs in the data caches. On reducing the number of sets, the size of tag bits increases. Hence, they maintain maximum possible tag bits with each cache line. On changing the cache size, location of a cache line changes. To handle this situation, either cache can be flushed or cache lines can be moved to their new location. In this work, as authors resize instruction cache, they do not implement any of these solutions. This may cause double caching of some of the addresses i.e. the same cache line read into two different sets. However, the instruction blocks are generally not modified at runtime, hence, it is acceptable to ignore copies of the same cache line. In our implementation, the clean cache
lines to be switched off are invalidated and modified cache lines are written back to the off-chip DRAM. If a cache line is in the current working set of an application then an attempt is made to keep it on-chip and some cache line is replaced.

In [115], the L2 cache lines which are also present in the L1 cache are switched off. This achieves leakage power savings by maintaining just a single copy in the multiple cache hierarchies. This scheme is inapplicable in the case of a CMP due to data shared by concurrently executing threads. Zhang et al.[116] use the compiler directed strategy to optimize the leakage power in instruction caches. The compiler technique is used at the loop level and cache lines containing those instructions in the loop are turned off after their last use. The compiler technique can be used safely only in the case of instruction caches only. The pointer analysis makes it difficult to apply for optimization of the data caches, especially in the case of high performance applications executing on the general purpose platforms. The presence of concurrently executing processes makes the use of a compiler directed strategy very hard in the case of a CMP. Hence, our power optimization strategies make use of the information available in the hardware only.

**Employing of different memory technologies:** Emerging memory technologies such as Magnetic RAM (MRAM) [22] and phase change memory (PRAM) [23, 24] are being explored for different memory hierarchies. These memories are not susceptible to soft error rates as they are non volatile and have a zero leakage current. These memories also have very high density which enables designers to implement larger storage memories in the same area. However, the main drawback of these types of memories is their limited write endurance. For example, a PRAM cell can be written approximately up-to $10^8$ times. If written more than $10^8$ number of times, then the entire memory cannot be used. It also has a very large access latency, compared to DRAM and also requires a large write energy. Various attempts have been made to use PRAM as the off-chip memory [26, 117, 118, 119, 120, 121] and as the on-chip last level cache [25, 122, 123]. The drawback of lower write endurance has been proposed to overcome by using smaller faster traditional memories such as SRAM [122] and DRAM [26] as intermediate buffers, or by reducing the number of writes done to PRAM. The number of write can be reduced by not writing the same data already present in cell [124, 119] or by saving negated data [123]. However, like other researchers [26, 117], we also believe that PRAM cannot be used to implement unified L2 cache on a large CMP. The large write energy and access latency, and limited write endurance make the use of PRAM infeasible as on-chip cache. PRAM has become more favorable over NOR/NAND flash memories due to its comparative higher write endurance [125].

The Spin Torque Transfer Magnetic RAM (STT MRAM) [22] is yet another emerging technology which has a very negligible leakage current compared to SRAM [126]. The main challenge of STT MRAM is its high write energy and write latency. The execution time of an application degrades significantly even with out-of-order processor if no optimizations are done for the increased read and
write latency. Hence, Rasquinha et al. [126] propose to implement the tag array of L2 cache and the last level cache using SRAM and data arrays using STT MRAM, as tags are accessed more frequently than the data array. Though STT MRAM consumes lesser leakage energy, increase in the dynamic energy may offset savings in the leakage energy if proper care is not taken to reduce the number of writes done to STT MRAM[127, 128]. Sun et al. [128] model 3D stacked MRAM L2 cache. Larger L2 cache can be implemented using STT MRAM in the same area when compared to the SRAM caches. Despite decrease in L2 cache misses with STT MRAM, they show degradation in performance with STT MRAM due to larger write latency. This degradation is larger in DNUCA implemented with STT MRAM due to data migrations. To mitigate this problem, they propose to use hybrid SRAM and STT MRAM cache architecture along with a write buffer to reduce write latency and intensity. Though a lot of efforts have been made from several years to improve the yield of STT MRAM [129, 130], still its commercial use is yet to happen! It is also not known whether fabrication of STT MRAM and CMOS technology on a single chip is commercially and technically feasible. Variation in its process parameters can lead to higher failure rate of STT MRAM cells, reducing memory array yield [131]. The main causes of process variations in STT MRAM cell are tunneling oxide thickness and cross-sectional area. These parameters cause read and write failures in STT MRAM and also lesser yield of memory array. This problem is exacerbated in reduced technology.

Studies mentioned above are specific to STT MRAM and PRAM memory technologies, whereas, our leakage optimization techniques are applicable to any memory technology in which the leakage power difference between power-on state and power-off state is significant.

2.4 Working set size estimation metrics

In the case of embedded platforms, cache requirement of an application is determined using an offline mechanism and cache configuration is adjusted accordingly [38, 39, 40, 41]. However, the cache requirement varies drastically in the case of applications running on the general purpose platforms. Hence, the cache configuration is adjusted at run-time using some metric on the general purpose desktops. Cache miss rate is one of such typically used metrics [35, 34]. If lesser than the required cache is allocated, the cache miss rate increases. The average memory access latency also varies accordingly. If the working set size of an application is larger than the available cache, then more number of cache misses occur, thus increasing off-chip memory accesses. This increases average memory access latency. [33] use average memory access latency to decide the number of cores sharing a cluster of L2 cache on a CMP. Dhodapkar et al. [132] probabilistically estimate working set size of an application and adjust cache associativity of L1 instruction cache accordingly. The WSS estimated using their approach depends on the hash function it uses, whereas, our WSS estimation method uses inherent cache function and
overcomes their drawback of under-estimating WSS. Kaxiras et al. [30] use the time interval between two consecutive accesses to an L1 cache line to decide whether it is dead and can be put into a sleep state. If this interval is greater than a predetermined value, then the cache line is put into a sleep state. [133] conclude through experiments that the time between two hits to a cache line is very small, whereas time before miss is very large. They also observe that the cache lines in L2 remain cached for a long time before a miss. Hence, they track time between hits, the number of hits and time elapsed after the last access for each cache line to determine when to switch-off an L2 cache line. Both [30] and [133] incur a very large hardware overhead. On the contrary, we use just an active bit for every cache line to determine whether it is working set. The number of lines accessed in a monitoring period, gives us a working set of an application. H. Zhou et al. [134] enhance decay cache by proposing the use of global control register which determines when a cache line can be put into sleep state. If a cache line is in inactive state for longer than the value contained in the global control register, then it is switched into sleep state. The global control register value is adapted by monitoring hypothetical cache miss rate and the actual cache miss rate. The hypothetical cache miss rate is a miss rate if all cache lines are in the active state. To determine the hypothetical cache miss rate, only data arrays are changed into the sleep state, whereas, tag arrays are maintained in the active state and retain their value. So after each monitoring interval, the additional cache misses caused due to data arrays being in the sleep state can be determined. These additional cache misses are used to adjust the global control register value. Even though decay interval is adjusted at run-time, just like the decay cache [30], it also incurs a high hardware overhead. To maintain a counter for every cache line in the case of large LLC present in a CMP, hardware overhead will be enormous. Moreover, tag array size is also significant, hence by keeping tag arrays into the active state, significant amount of leakage power reduction opportunity will be lost.

2.5 Application of Genetic Algorithms for architectural studies

We use the genetic algorithms to determine the near optimal remap table configuration. Genetic algorithms have been used in various fields such as in exploring parametrized SoC architecture [135], in many fields of VLSI design [136], and medical science [137]. Genetic algorithms were applied for the cache resizing problem, for the first time by Diaz Josefa in [138]. They consider simultaneous multi-threading processor and use GA to determine an optimal partition of cache among various threads. They use architectural simulation to evaluate the fitness of each chromosome which is very slow. Hence, they use only 30 chromosomes in a population. On the contrary, we estimate fitness of a chromosome, representing an instance of a remap table in a faster trace based model. Our method is up-to 217X faster than the cycle accurate simulation method. Hence, we could use population size of 100 chromosomes.
Though GA was used in other fields for a long time, recently it has gained popularity in architectural studies\cite{139, 140, 141}. Jiang et al. \cite{141} determine the optimal co-scheduling of applications on a CMP using GA. In \cite{139}, authors use GA to determine memory bandwidth as a function of cache hit rates.

### 2.6 Summary

In this chapter, we reviewed past research done to optimize cache configuration. We found that the majority of work is done for the uniprocessor platforms. Hence, we optimize power dissipated in caches on multiprocessor platforms.
Chapter 3

Sapphire: An Experimental Setup

In this chapter, we explain Sapphire, our simulation infrastructure that we use in all our experiments. We also explain various benchmarks used for experimentation. Sapphire is a multi-processor/multicore simulator where the memory hierarchy, interconnect (network on chip) and off-chip DRAM are parametrized and can be configured to model various configurations. Sapphire addresses shortcomings in SESC [47] by integrating it with Ruby from the GEMS framework [48]. Sapphire also integrates Intacte, an interconnect power model [51]. DRAMSim models off-chip DRAM in great details [49]. This has been integrated with Ruby. Power consumed by DRAM is modelled using MICRON power model [50]. Thus Sapphire allows users to explore power and performance implications of all main system components like processor, interconnect, cache hierarchy and off-chip DRAM.

3.1 Introduction

Multi-core processors are becoming ever more popular. They are penetrating in desktop as well as in embedded processors. Embedded multi-core processors are connected through simple 1-D interconnections such as buses, rings etc whereas, processors used in servers prefer to employ 2-D interconnection structures like mesh based architectures e.g. Tilera and Intel 80 tile architecture. These kind of architectures provide a large design space which an architect would want to explore better and efficient architectures. The design space includes (but not limited to) exploring performance optimization and energy efficiency in the context of different network topologies, network flit size, routing algorithms, router design, choice of each node (spanning from single compute elements to full processors), size of caches at various levels, co-scheduling algorithms for applications etc. The simulation framework for multi-cores must be endowed with sufficient configurability so as to enable exploration of some or all of these parameters for energy efficiency/performance optimization.

Multi-core simulators available today primarily fall under two categories, namely processor simulators
and system simulators. Processor simulators model the processor pipeline in detail and are suitable for microarchitectural exploration. Simulators such as SESC[47], Opal[48] are processor simulators which are popular. These simulators cannot model the system-level impact of a certain change. System simulators such as Simics [142], M5 [143] simulate the entire system (including the processor, caches, memory hierarchy, I/O devices). Most system simulators allow booting of complete operating systems and perform end-to-end simulations. Such simulators are useful for modeling the system level impact of a macro architectural change or changes to the system software. But this makes these simulators very slow.

To make the simulator faster, various techniques have been applied like simulation of an interval from every phase of the program, instead of simulating the whole program[144], abstracting various components of the system architecture [145] or selectively simulating on the native platform and on the cycle-accurate simulator. Yet another popular tool is CMP-$\text{sim}$ [146]. Unlike other simulators, CMP-$\text{sim}$ uses dynamic instrumentation to track certain events of interest. The execution happens on the host processor and instructions are added to the application binary to capture events of interest. While such a technique can be used to perform cache modeling, the scheme cannot be applied to explore diverse cache configurations like distributed caches [147], tiled caches [148] and wide range of interconnect topologies from bus-based to point-to-point and Network-on-Chip based choices like torus and meshes.

3.2 Motivation

SESC [47] is an open-source processor simulator that is capable of modeling multi-cores or chip multiprocessors(CMPs), simultaneous multi-threading and symmetric multi-processors. The simulator models the MIPS processor instruction set. The simulator consists of a functional emulator which performs the functional simulation. The trace of instructions executed by the functional emulator is passed through a timing model. It models minute details of the processor pipeline and processor components. Such a decoupled implementation lends high efficiency to the SESC processor implementation. The simulator also models the memory hierarchy and the interconnect. However, these models cannot be easily extended to perform various architectural explorations with regard to caches and network on chip. Also, the interconnection implemented in SESC is not advanced and does not model 2-D interconnections. It models shared bus configuration. Hence, this component of SESC needs substantial improvement to support configurability of caches and NoCs.

The GEMS [48] simulation framework includes Opal processor simulator and Ruby memory hierarchy simulator. Opal simulates the SPARC instruction set and performs detailed pipelining modeling. Opal too uses a functional simulator for actual application execution and the timing is obtained from the pipelining. For functional simulation, it employs Simics [142]. The GEMS simulation framework can
operate in two modes: Simics+Ruby and Simics+Opal+Ruby. The use of Simics helps GEMS to do system-level modeling while not trading off the accuracy of pipeline modeling. However, due to the use of Simics, this approach incurs the overhead of simulating the operating system and other I/O devices. Even though Simics provides an efficient implementation of all components, it increases simulation time. Also a user intending to evaluate the micro-architectural changes does not need the completeness of system-level modeling. Ruby simulates the memory hierarchy and interconnect in great details. It is highly configurable. Ruby allows specification of various coherence protocols, in terms of states, events and transitions, through a language called SLICC. This specification is transformed into a C++ implementation by the framework. This allows accurate modeling of the coherence protocol, which is an important contributor of network traffic. The network in GEMS can be modeled at various levels of details. Garnet [149], the network model included within GEMS can be configured to specify the exact interconnect topology, the link latency and bandwidth of specific links etc. It models various stages of network router pipeline, namely, route computation, virtual channel allocation, switch allocation and switch traversal. In Sapphire, we try to leverage the advantages of both simulation frameworks by integrating the SESC processor with Ruby, the memory and interconnect model.

3.3 Integrating SESC with Ruby

Ruby is designed to work independent of the Opal processor (or Simics). Due to this reason, Ruby has a well defined interface that specifies the list of functions exported by Ruby and the list of processor functions expected by Ruby. This clear interface definition integrating with Ruby is not an engineering intensive task. For integration, we created a SescInterface class which implemented all the relevant interfaces to interact with SESC. In SESC, a new memory object was created. The memory object, referred to as RubyMemObj forwards the requests to Ruby, while ensuring that two requests belonging to the same processor, do not refer to the same cache line (which is a requirement of Ruby). The detailed block diagram is shown in Figure 3.1.

As can be observed from Figure 3.1, an object of the RubyMemObj class is created for every cache (i.e. L1-I, L1-D). These objects enqueue the incoming requests that can be processed in this cycle. Other requests that cannot be processed are placed in a separate data structure. At each clock cycle, the Request Dispatcher (which is a callback function) deques as many requests as the number of ports from each processor and forwards the requests to Ruby. The request is received by the SescInterface object in Ruby. The request is forwarded to the appropriate Sequencer (one per processor) for subsequent processing by the caches. When the data is available, Ruby invokes the call back function for hit, which triggers the corresponding function in SESC.

Figure 3.2 shows the sequence of steps involved in making a request to Ruby. An IMemRequest
Figure 3.1: Detailed block diagram showing the integration of SESC with Ruby.
Figure 3.2: Interaction between various modules of SESC and Ruby for processing an instruction memory request. A similar sequence of calls are executed for a data request.

object is created by the FetchEngine of SESC. This request is forwarded to the RubyMemObj object, which in turn checks whether Ruby can accept this request. If yes, the request is stored in the respective queue for this processor. The control is returned to the FetchEngine. On incrementing the clock cycle, the callback function that forwards all requests from RubyMemObj is invoked. This issues the requests to Ruby, through SescInterface. The SescInterface forwards the request to the appropriate Sequencer object. The Sequencer object then forwards the request to the appropriate L1 Cache Controller within Ruby. The request flows through the memory hierarchy and when the data is available, the hit callback of SESC is invoked to inform SESC about the completion of the request. The RubyMemObj in turn invokes the goUp function to inform the Load/Store Queues that the data is now available.

This simulator now addresses all the shortcomings of SESC. The technique described in the above section can be used to attach any processor front end to the Ruby memory hierarchy.

3.4 Modeling the interconnect

While the simulator by itself exposes sufficient configurability to be able to model the interconnect, most architectural simulations use single cycle communication distance between two routers or between the router and end point (i.e. cache/memory in this case). With technology scaling, while the effective length of the wire decreases, the resistance of the wire increases on account of lower wire width [150]. Due to this, the interconnects need to be appropriately enhanced through increased wire pitch, use of
repeaters, appropriate selection of repeater size, use of pipe stages etc. While many of these choices
are made while transforming a design into silicon, current available high-level simulators do not capture
the impact of these parameters during architectural simulation. These parameters determine power
consumed and latency incurred by the network link. Our framework fills this gap by integrating Intacte
[51], a low level tool to explore interconnect links.

Intacte [51] is an interconnect exploration tool which given the range of input frequencies, wire
length and technology parameter, gives the least power configuration of the link by iterating over the
following set of parameters: wire width, wire spacing, repeater size and spacing, number of repeaters,
degree of pipelining, supply voltage and threshold voltage. These values are obtained using conservative
estimates of activity and coupling factors. However, the tool can take these as input to estimate the
best configuration.

In order to estimate the latency (in cycles) of a certain wire, we need to estimate the wire length. We
estimate the interconnect length between router-router and router-end point by constructing the
floor plan of each node on the network. Each network node contains a processor, L1 Cache Controller
connected to the L1 instruction(L1-I) and L1 data cache(L1-D), L2 Cache Controller and a slice of L2
cache, a router and an optional memory controller. The memory is placed off-chip and is not a part of
the node. We henceforth refer to the network node as a tile. The floorplan of a typical tile is shown
in Figure 3.3. The area of the processor is same as the size of a single core of Intel Nehalem [4]. The
area occupied by the L2 cache is obtained using an approximate estimate of 7.5mm$^2$ per MB of cache,
which is the density achieved on the AMD Shanghai processor. This estimate is more conservative than
the estimates obtained by CACTI [52]. The area of the router is estimated to be quite negligible at
32nm [151]. The L1 cache is of size 32KB whose area is very small and included in the processor area
and hence is not depicted in the figure. Similarly, directory is present in every tile to maintain cache
coherece between private L1 caches. We assume size of directory to be negligible. This is conservative
assumption for our problem.

The wire lengths are determined for the floorplan in Figure 3.3 at a frequency of 3GHz and 32nm
technology as shown in Table 3.1. It is evident from the table that some of these latencies are quite high and hence cannot be ignored during architectural simulations. In order to account for these in simulation, we modify the network configuration file and input the specific latencies as obtained from Intacte.

<table>
<thead>
<tr>
<th>Link Type</th>
<th>Length</th>
<th>PipeLineStages</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-R</td>
<td>1.3mm</td>
<td>2</td>
<td>0.562 mW</td>
</tr>
<tr>
<td>L2-R</td>
<td>3.75mm</td>
<td>7</td>
<td>1.59 mW</td>
</tr>
<tr>
<td>R-R Horizontal</td>
<td>5 mm</td>
<td>9</td>
<td>2.125 mW</td>
</tr>
<tr>
<td>R-R Vertical</td>
<td>5 mm</td>
<td>9</td>
<td>2.125 mW</td>
</tr>
<tr>
<td>M-R</td>
<td>0.2mm</td>
<td>1</td>
<td>0.127 mW</td>
</tr>
</tbody>
</table>

Table 3.1: Tile link latencies and power estimations using Intacte for 512KB L2 slice.

### 3.4.1 Modelling power in interconnect

Yet another aspect that needs to be considered in NoC based architectures is the power dissipated in the interconnect. In order to address this, we compute the link activity and coupling factors for all request and response messages on the network. One of the requirements for performing this analysis is the ability to pass correct memory data over the network. While, Ruby implements exact data transfers when simulated with Simics, the feature is not available otherwise.

In order to implement this, a detailed understanding of the SESC memory model and update procedures need to be understood. The instructions in SESC are obtained from the MIPS executable. However the data portions i.e. stack, global and heap memories are allocated by the simulator and then passed on to the running application. The simulator updates the memory upon receipt of memory writes. This memory is not accessible to Ruby. The request structure between SESC and Ruby is modified to pass the expected data along with the request (since SESC performs the write in functional simulation before it is simulated through the pipeline). The memory implementation in Ruby is modified to return this data (available along with request) instead of looking up its copy of the memory. The activity computation function appropriately accounts for the data, upon the return path and not along the forward path.

### 3.5 Modelling DRAM

DRAMSim [49] is integrated along with Ruby to model DRAM latencies and power accurately. DRAMSim implements detailed timing models for variety of existing memories, including SDRAM, DDR, DDR2 etc. DRAMSim simulator is parametrized to support various row buffer management policies, address management policies and various layouts in terms of channels, ranks, rows and columns. This simulator uses MICRON power model [50] to estimate the power consumed in memory accesses.


3.6 Simulation procedure

Flow chart in Fig. 3.4 shows the experimental procedure. The first step of the procedure is to determine the optimal floorplan based on the processor, cache and router area. This is then used to compute the link lengths. These link lengths are then passed through Intacte to obtain the link parameters such as the number of pipe stages. To obtain these parameters, we assume conservative values of activity and coupling factor. Link parameters giving minimum power are used for the assumed activity and coupling factor. The number of pipe stages determines the latency of the link in cycles. This information is used in the network configuration file. The simulation is performed on Sapphire. Sapphire determines the activity and coupling factors of all the links. These are then passed through Intacte to determine the power dissipated in the NoC. DRAMSim model estimates power consumed in off-chip memory accesses. Cache power is estimated using CACTI power model [52].

Figure 3.4: shows the steps to be followed for performing detailed power-performance analysis on Sapphire

Table 3.2 gives the system configuration which we have used in our experiments. There are sixteen tiles on a CMP. Each tile contains a core and split instruction and data L1 cache of 32KB in size. Each tile also contains a slice of L2 cache of size 512KB. L2 cache is distributed and shared by all cores. To maintain cache coherence between private L1 caches, directory information is present in each tile. These tiles are interconnected via 2D-mesh NoC and per-tile router.
Chapter 3. Sapphire: An Experimental Setup

<table>
<thead>
<tr>
<th>Core</th>
<th>out-of-order execution, 3GHz frequency, issue/fetch/retire width of 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>32KB, 2 way, 64 bytes cache line size, access latency of 2 cycles (estimated using CACTI[52]), private, cache coherence using MOESI protocol</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>512KB/tile, 16 way, 64B line size, 3 cy. latency (estimated using CACTI), noninclusive, shared and distributed across all tiles</td>
</tr>
<tr>
<td>Directory</td>
<td>Tag bits of L2 cache lines include full bitmap for L1 sharers. A separate per tile table of 3000 entries (5% of L2 slice size) maintains dir info. for cache lines caches only in L1s and not in L2</td>
</tr>
<tr>
<td>Interconnect</td>
<td>16 bits flit size, 4x4 2D MESH, deterministic routing, 4 virtual channels/port, credit based flow control, router queues with length of 10 buffers</td>
</tr>
<tr>
<td>Off-chip DRAM</td>
<td>4GB, DDR2, 667MHz freq, 2 channels of 8B in width, 8 banks</td>
</tr>
<tr>
<td>DRAM</td>
<td>16K rows, 1K columns, close page row management policy</td>
</tr>
</tbody>
</table>

Table 3.2: System configuration used in experiments

3.7 Applications used in experiments

We study applications from a wide spectrum of workloads such as Parsec [46], Splash2 [45], SPEC2006 [153] and Alpbench [152]. In [154], authors use statistical methods to analyze similarity and differences between various multithreaded programs from Splash2 and Parsec benchmark suites. We have carefully chosen applications from splash2 and Parsec benchmark suites such that they cover programs with different characteristics and domains ranging from high performance computing, financial domain, animation, media processing and signal processing. Authors conclude in their paper [154] that these two suites have a little overlap. Table 3.3 shows different benchmarks which we consider in our study. We skip initial serial portion and simulate 4 billion instructions from the parallel section. We simulate fft, mpegenc and mpegdec applications to completion.

To determine memory intensive and non-memory intensive benchmarks, we follow procedure explained in [155, 156]. Memory intensive applications generate more L2 cache misses as opposed to non-memory intensive applications. Instructions per cycle (IPC) is determined using ideal L2 cache, which does not generate any L2 cache misses. IPC is also determined using non-ideal L2 cache. Benchmarks which show IPC gain of at least 20% with ideal L2 cache are considered as memory intensive. Other benchmarks are classified as non-memory intensive. Fig. 3.5 shows the percentage IPC speedup obtained by simulating applications with ideal L2 cache over that obtained with non-ideal L2 cache. We have a good mix of both memory intensive and non-intensive benchmarks. Applications like ocean, fft, cholesky and raytrace are memory intensive as they have large L2 cache miss rate, where as, blackscholes

---

1 Some of the PARSEC benchmarks either use OpenMP APIs or libraries which are not supported by SESC compiler. Hence remaining benchmarks cannot be compiled using SESC compiler.
and swaption have negligible cache miss rate. It should be noted that many applications have small cache miss rate with 8MB of L2 cache. Event though, this cache size is much smaller than that present in current popular processors such as Intel Nehalem (24MB) and AMD Opteron (16MB), we also show readings with smaller L2 slice of 256KB as part of sensitivity study. Like past studies, we assume one-to-one mapping between threads and cores. We have skipped initial serial portion and simulate only parallel section in all the test cases. We execute 4B instructions, unless otherwise mentioned. We test all workloads with 16 threads.

Apart from Splash2 and Parsec benchmark suites, we also perform experiments with SPEC 2006 benchmarks[153]. These benchmarks allocate large memory and have larger working set size. Due to insufficient virtual memory, malloc syscalls fail if sixteen different SPEC 2006 programs are simulated simultaneously on our simulator. This is because SESC simulator generates application binary for 32 bits machine. Hence, we simulate 4 programs on a quad-core CMP with 4MB shared L2 cache. Architecture layout is shown in Fig. 3.6. This configuration is similar to Intel Xeon [157]. Another reason to choose this type of architecture is to show applicability of our proposals to a non-tiled architecture.

We have carefully selected various programs such that workloads have low, medium and high L2 cache miss rates and hence, have varying working set sizes [158, 159]. As explained earlier, we determined IPC of SPEC 2006 benchmarks with ideal and non-ideal L2 cache. The percentage IPC speedup obtained using ideal L2 cache over non-ideal L2 cache is shown in Fig. 3.7. A short description of various programs used from SPEC 2006 benchmark suite is given below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description, WSS(L/M/S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Splash2 Benchmark[45]</td>
<td></td>
</tr>
<tr>
<td>ft</td>
<td>signal processing, FFT on 1M points, till completion, M</td>
</tr>
<tr>
<td>radix</td>
<td>Radix sort on 1M keys, M</td>
</tr>
<tr>
<td>raytrace</td>
<td>Graphics, traces ray on car object, M</td>
</tr>
<tr>
<td>cholesky</td>
<td>High Performance Computing, blocked sparse matrix factorization on tk29, L</td>
</tr>
<tr>
<td>ocean (continuous)</td>
<td>High Performance Computing, 512x512 grid points, L</td>
</tr>
<tr>
<td>water_spatial</td>
<td>High Performance Computing, simulation of 512 water molecules, M</td>
</tr>
<tr>
<td>water_nsquared</td>
<td>High Performance Computing, simulation of 512 water molecules, M</td>
</tr>
<tr>
<td>barnes</td>
<td>High Performance Computing, Barnes-Hut method on 16K bodies, M</td>
</tr>
<tr>
<td>PARSEC Benchmark[46]</td>
<td></td>
</tr>
<tr>
<td>blackscholes</td>
<td>Financial Domain, SimLarge i/p, S</td>
</tr>
<tr>
<td>swaptions</td>
<td>Financial Domain, SimLarge i/p, M</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>Animation, SimMedium i/p, L</td>
</tr>
<tr>
<td>x.264 Encoder</td>
<td>Media Domain, SimLarge i/p, M</td>
</tr>
<tr>
<td>Alpbench Benchmark [152]</td>
<td></td>
</tr>
<tr>
<td>mpegenc</td>
<td>Media Processing, Encodes 15 Frames of size 640x336 till completion, M</td>
</tr>
<tr>
<td>mpegdec</td>
<td>Media Processing, Decodes 15 Frames of size 640x336 till completion, M</td>
</tr>
</tbody>
</table>

Table 3.3: Table shows applications used for study and their WSS information(L:Large, M:Medium, S:Small).
• named: This is a floating point benchmark which simulates a large biomolecular systems. It incurs low L2 cache miss rate.

• soplex: This is a linear program solver which uses simplex algorithm. It also uses floating point instructions. IPC gains are moderate with ideal L2 cache. We have used a standard ref.mps input for this application, as suggested in SPEC benchmark suite.

• gcc: This is a C compiler written in C language. It uses integer instructions. It takes pre-processed C program as an input and generates assembly language code for AMD Opteron processor. We use expr.i as input for this program, which is one of the input files specified in the SPEC reference.

• dealII: This program uses floating point instructions and it is targeted at adaptive finite elements and error estimation. It has very low cache miss rate and hence also shows lower IPC speedup.

• mcf: This program solves a fleet scheduling problem of a single depot mass transit using combinatorial optimization. This programs exclusively uses integer instructions and incurs higher L2 cache miss rate. This program also has low IPC [158].

• milc: It simulates four dimensional SU gauge theory from physics chromodynamics. It uses floating point instructions and shows high IPC gains with ideal L2 cache.
Table 3.4: Multiprogramming workloads simulated on a quad-core CMP

- bzip2: This application uses integer instructions and shows higher IPC gains with ideal L2 cache, which means that it is memory intensive program.

- sjeng: This application with integer instructions is from artificial intelligence domain. It is also memory intensive program.

- libquantum (libq): This is integer benchmark and it simulates quantum computers. Quantum computers are based on principles of quantum mechanics. This is a memory intensive benchmark.

Multiprogramming workloads which we simulate using SPEC 2006 benchmarks are given in Table 3.4. We have performed experiments with varying number of high and low memory intensive applications e.g. a combination of four instances of dealII has all applications with very low L2 cache miss rate, whereas, a combination of sjeng, milc, mcf and libquantum has all memory intensive applications. The details of the multiprogramming workloads are given in Table 3.4, where “HHHL” in the description column indicates that it is a combination of two memory intensive and non-intensive applications. We have executed 2 billion instructions in all cases, except, in mcf-mcf-mcf-mcf, we have executed 1B instructions.
3.8 Summary

Sapphire enables us to model cache, interconnect and off-chip memory with moderate accuracy and measure power consumed by these components. We have carefully chosen applications from diverse domains. We determine memory intensive characteristics of an application by executing it on an ideal and non-ideal L2 cache, as explained in [155, 156]. The multiprogramming workloads that we simulate have applications with varying memory intensive and non-intensive properties. We consider a tiled architecture similar to Tilera processors [5] in all our experiments. To demonstrate applicability of our proposals to a non-tiles architecture, we also simulate a quad-core CMP with 4MB shared L2 cache, which is similar to Intel Xeon processors [157].
Chapter 4

Working Set Size Estimation

In this chapter, we explain a new method to determine working set size of an application. We call this method as “Tagged Working Set Size (TWSS)” estimation method. We determine accuracy of TWSS by calculating average ratio of WSS estimated by TWSS to the actual WSS. We also study factors affecting accuracy of TWSS. We quantitatively compare TWSS method with Dhodapkar’s approach [132]. Previous studies use average memory access latency and cache miss ratio as metrics for various cache optimizations on uniprocessor platforms [114, 33, 29]. Hence, we determine correlation between these metrics and the actual WSS in the context of NUCA caches on chip multiprocessor (CMP).

4.1 Introduction

To switch-off over-allocated cache, working set size (WSS) of an application should be estimated accurately. Like other researchers [132], we also define WSS as the number of unique cache line addresses accessed in a given interval. Power gains obtained by switching off over-allocated cache depend on the accuracy of WSS estimation method, how fast the controller responds to changes in WSS and overhead incurred by the estimation method. Power gains are sub-optimal if WSS is estimated conservatively. On the contrary, application incurs more cache misses if WSS is under-estimated. The execution time of an application may degrade significantly, if the metric fails to quickly respond to the changes in WSS. Previous studies estimate WSS for a single threaded application by executing it on a uniprocessor system [132] or use other metrics such as average memory access latency (AAL) or cache miss rate (CMR) as an indirect mechanism to optimize cache [21, 33, 34, 35]. Hence, we propose a new method which accurately estimates WSS of an application(s) executing on a CMP.

The working set of an application is the unique L1/primary cache line addresses accessed in an interval or monitoring period/interval. The cardinality of this set is the working set size or to be more precise, cache working set size. Mathematically, if $l_1, l_2, l_3...l_n$ are unique cache line addresses (CLAs)
accessed by an application, then the working set $S$ is,

$$S = \{l_1, l_2, l_3, \ldots, l_n\} \quad WSS = |S| = n$$

(4.1)

### 4.1.1 Dhodapkar’s WSS Estimation Method (DHP)

Dhodapkar et al. [132] hash an instruction address and set the corresponding bit in a 1K bit-vector. The number of bits set in the bit-vector probabilistically determines WSS of an application. If the total number of bits in the bit-vector is $N$ and the number of addresses hashed in the bit-vector is $K$, then the fraction $f$ of bits set is:

$$f = 1 - (1 - \frac{1}{N})^K \quad K = \frac{\log(1-f)}{\log(1-\frac{1}{N})}$$

(4.2)

Hence, in DHP, if the fraction of bits set in the bit-vector is known, then WSS can be calculated using Eq. (4.2). Even though this method has negligible hardware overhead, accuracy of the estimated WSS depends on the choice of hash function. It may under-estimate WSS of an application due to an aliasing problem, i.e. multiple addresses might hash into the same bit. Hence, we propose a new method which overcomes this drawback.
4.2 Tagged Working Set Estimation Method

To determine cache lines accessed in a given interval, an active bit is maintained for each cache line along with its tag bits as shown in Fig. 4.2. When an address is searched, the cache controller compares its tag bits against tag bits of a cache line. An active bit is set on a cache hit. On a cache miss, some cache line is replaced by the cache controller. A replacement counter (activeLineReplaced) is maintained in each L2 slice to count the evicted addresses which are accessed in the same monitoring interval. This counter counts the cache lines replaced with the active bit set. We refer to this method of estimating active cache lines in a given interval as “tagged working set size (TWSS)” estimation method as tag bits of a cache line uniquely identify addresses accessed and evicted. Using TWSS method, WSS in terms of the number of cache lines, is calculated as:

\[
WSS = A + R
\]  

(4.3)

where, \(A\) is the number of cache lines with the active bit set and \(R\) is the number of cache lines replaced with the active bit set, i.e. value of the activeLineReplaced counter. TWSS method is independent of SNUCA or DNUCA cache access policies. In the case of DNUCA, when a cache line migrates, the active bit maintained along with its tag bits is also migrated with it. TWSS overcomes the DHP method’s drawback of under-estimating WSS in following ways:

- Tags maintained along with the cache line identify cache line addresses uniquely.
- The replacement counter counts lines replaced with the active bit set.

TWSS over-estimates WSS if the same CLA is accessed and replaced multiple times in the same monitoring interval. However, this scenario is rare considering large cache size and associativity (16/32) of last level caches (LLCs). The WSS of an individual thread can be obtained by implementing TWSS in the primary cache. In our case, we are interested in WSS measured at the shared cache as we want to resize it. Hence, TWSS is implemented in LLC.

4.2.1 Determination of monitoring interval

Typically monitoring interval is measured in terms of the number of instructions graduated [132, 34]. In the case of a uniprocessor, it is possible to measure the number of committed instructions without accessing NoC. Bardine et al. [21] count the number of requests made to the shared L2 cache while reconfiguring it. However, in our case, L2 cache is shared and dispersed in many tiles. On a tiled architecture, monitoring the number of cache accesses without accessing NoC is not feasible. Hence, we use the monitoring interval in terms of the number of clock cycles.
Figure 4.3: shows variation in reuse time on the X axis and percentage of accesses on the Y axis. Majority of accesses have reuse time less than 4M clock cycles.

We reset active bits at the beginning of every monitoring interval. If the monitoring interval is too small, fewer active bits would be set. This would be misinterpreted as smaller WSS. On the contrary, if the monitoring interval is too large, larger WSS would be estimated, failing to switch-off excess L2 slices. Hence, determination of accurate monitoring interval is very important. To estimate it, we measured the number of clock cycles between two consecutive accesses made to the same CLA by any thread in an application. We call the number of cycles between two consecutive accesses as reuse time. The monitoring period should be such that the majority of accesses should have reuse time less than the monitoring period. With this, if a cache line is not accessed in the previous monitoring period, then it will not be accessed in the next monitoring period as well and such an address can be evicted from cache. Hence, we profiled reuse time of all cache line addresses. Fig. 4.3 shows percentage of total accesses with reuse time less than 1M, 2M... clock cycles. Applications shown in Fig. 4.3(a) have at least 95% of accesses with reuse time less than 4M, whereas, applications shown in Fig. 4.3(b) have less than 95% of accesses with reuse time less than 4M clock cycles. Clearly, if monitoring period is set to 4M clock cycles, then applications shown in Fig. 4.3(a) will incur minimum degradation in execution time.
In the case of blackscholes, swaption and barnes, 99% of accesses have reuse time less than 1M clock cycles. This means that the monitoring period for these applications can be as small as 1M clock cycles. For other applications, if the monitoring period is increased from 1M to 2M clock cycles, percentage of accesses having reuse time less than 2M clock cycles increases. This curve tapers after 4M clock cycles. Hence, we use 4M clock cycles as the monitoring period. In the case of radix, fluidanimate, fft and mpegenc only up-to 94% of accesses have reuse time less than 4M clock cycles. These applications give significant energy savings with 4M clock cycles as the monitoring period. Among these applications, radix and mpegenc show slightly higher degradation in execution time (up-to 5% with adaptable cache associativity implementation). The details of variation in energy savings with the monitoring period can be obtained in Chapter 5.

We have also evaluated reuse time distribution of accesses in the case of multiprogramming workloads obtained with SPEC 2006 benchmarks. This is shown in Fig. 4.4. Multiprogramming workloads like mcf-milc-sjeng-libq and libq-libq-libq-libq have only up-to 80% and 60% accesses with reuse time less than 4M clock cycles, respectively. Rest of 20%-40% accesses have reuse time much larger than 200M clock cycles. These applications have very large WSS. As a result, cache lines get replaced without accessing them again. Hence, monitoring period of 4M clock cycles works well for these applications. On the contrary, mcf-dealII-gcc-soplex has smaller WSS, and only 90% of accesses have reuse time lesser than 4M clock cycles. This application shows some degradation in execution time with the monitoring period of 4M clock cycles. Please refer to Chapter 5 for more details.

After experimenting with a variety of multiprogramming and multithreaded workloads, we conclude that monitoring period should be such that at least 95% of accesses have reuse time lesser than the monitoring period. Applications may show degradation in execution time if less than 95% of accesses have reuse time less than the monitoring period.
4.2.2 Hardware implementation and overhead

For cache of 512KB in size and 64B cache line, 8K active bits are required which is just 0.1% of space overhead. Since active bits and activeLineReplaced counter are reset at the beginning of every monitoring interval, our experimental analysis shows that, a 32 bit saturating counter per L2 slice is sufficient as an activeLineReplacement counter for our monitoring interval. TWSS adds minimal overhead in terms of hardware area and power consumption.

To count the number of active bits set, a simple scan can be done at the beginning of every monitoring interval. This requires 4K clock cycles if scanning is done on both edges of the clock. This time is negligible compared to the monitoring period of 4M clock cycles.

4.3 Comparison with the Dhodapkar’s Method (DHP)

We first evaluate TWSS against the DHP method using a hash function based on srand and rand functions in “C” language, as mentioned in [132]. However, hardware implementation of these hash functions is very complex. Hence, we also evaluate the DHP method with other hash functions typically used in practice. Linear Feedback Shift Registers (LFSR) are commonly used as a “pseudo random generators” which accept a seed and return a deterministic random number. Both hardware and software implementations of LFSR are feasible. We believe, hash function based on a LFSR is a more feasible hardware implementation of a hash function used in the DHP method. Hence, finally we also evaluate TWSS against the DHP method using LFSR as a hash function.

4.3.1 Some typically used software hash functions

We experimented with a significant number of hash functions based on srand and rand. Some of them under-estimate WSS by a large margin for most of the applications. We ignored such hash functions. Here, we have given one hash function which estimates WSS accurately for majority of the applications. It is given in Listing 4.1. We also evaluated the DHP method using some other commonly used hash functions, such as prime modulo and a hash function based on that used in swaption benchmark from Parsec benchmark suite [46]. The code listings of these hash functions are given in Listing 4.2 and Listing 4.3, respectively.

```c
srand_hash_function {

    // Select bits from position x to y from the CLA
    int m = bitSelect(addr, x, y);

    // srand ensures that the same hashed value is
```
Chapter 4. Working Set Size Estimation

// generated if the address is accessed multiple times
srand(m);
int hash_value = rand();

for {i = 0; i < (y-x); i++} {
    int bit_val = m & (i << i);
    if (bit_val) {
        hash_value ^= rand();
    }
}
hash_value = hash_value % bitVectorSize;
return hash_value;

// End of Hash Function

Listing 4.2: Swaption Hash Function

swaption_hash {
    long ix, k1;
    float dRes;
    ix = addr;
    k1 = ix/127773L;
    ix = 16807L*(ix - k1*127773L) - K1*2836L;
    if (ix < 0)
        ix = ix + 2147483647L;
    dRes = (ix * 0.04656612875);
    hash_value = (int) dRes % bitVectorSize;
    return hash_value;
}

Listing 4.3: Prime Modulo

prime_modulo_hash {
    hash_value = (int) addr % 8209;
    return hash_value;
}

Since we want to resize L2 cache, we implement TWSS in the L2 cache and determine accuracy of TWSS at the L2 cache level. We determine true/actual WSS (AWSS) by counting the number of unique CLAs accessed by all L1 caches. This includes missed read/write requests and write-backs done by L1 caches. Fig. 4.5 shows the average ratio of WSS estimated using TWSS to AWSS. It also shows the average WSS ratio obtained with the DHP method using above mentioned hash functions. All values are obtained after every 4M clock cycles. In all applications, WSS estimated using TWSS is close to 1. Even though applications such as fft, raytrace and ocean show non-zero replacements per set (Fig. 4.8(a)), over-estimation is not observed in these applications. TWSS over-estimates WSS if the same address is accessed and replaced multiple times in a monitoring period. For all these applications TWSS estimates WSS accurately within accuracy of 0.001%. We used geometric mean instead of average mean
Chapter 4. Working Set Size Estimation

Figure 4.5: shows average ratio of WSS estimated using TWSS and DHP methods to true/actual WSS (AWSS) evaluated at the L2 cache. SRAND, SWAPTION and MODULO indicate various hash functions used in the DHP method.

in all our experiments and all conclusions still remained same [160]. Hence, for consistency, we use arithmetic mean throughout this thesis.

The DHP method with srand hash function shows over-estimation in case of fft and raytrace, even though it sets the same bit if the address is accessed multiple times in a monitoring period. The over-estimation in this case is due to probabilistic nature of the method, which is clear from Eq. 4.2. The srand based hash function estimates WSS accurately or over-estimates it by a small margin for all applications, except for cholesky and ocean. It under-estimates their WSS by 19% and 37%, respectively. This is due to aliasing problem. With other hash functions, the DHP method under-estimates by a large margin which is detrimental to any cache optimization. Finding a hash function which suites all benchmarks is a very hard problem.

Fig. 4.5 shows ratio of WSS estimated by TWSS to AWSS evaluated at L2 cache. This considers accesses done by L1 caches only. We also compared WSS estimated by TWSS (implemented in L2 cache) to WSS evaluated at L1 caches. This includes only reads and writes done by all threads. In this case, write-backs done by L1 caches are not considered while calculating AWSS. This comparison is given in Table 4.1. In x.264, WSS is under-estimated by both DHP and TWSS methods, as most of accesses are L1 cache hits. The L2 cache is accessed on an L1 miss, invalidations and write-backs. x.264 creates only up-to six concurrent threads even on executing with sixteen threads as a command line parameter. As a result, most of the accesses are L1 hits. Hence, TWSS implemented at L2 cache under-estimates WSS in case of x.264. On the contrary, it over-estimates in case of radix and fluidanimate. This is because, TWSS implemented in L2 cache receives L1 write-backs. As write-back traffic is not generated by threads directly, these are not considered in AWSS evaluated at L1 cache. However, for remaining applications, WSS estimated by TWSS is close to 1. Our results indicate that TWSS can be used to estimate aggregate WSS of all applications executing concurrently on a CMP. The under-estimation
Table 4.1: shows average ratio of estimated WSS obtained using TWSS and DHP methods to true/actual WSS (AWSS) evaluated at L1 cache. SRAND indicates hash function used in the DHP method

<table>
<thead>
<tr>
<th>App.</th>
<th>TWSS</th>
<th>SRAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>radix</td>
<td>1.72</td>
<td>1.72</td>
</tr>
<tr>
<td>raytrace</td>
<td>1.04</td>
<td>1.55</td>
</tr>
<tr>
<td>cholesky</td>
<td>1.02</td>
<td>0.79</td>
</tr>
<tr>
<td>water_spatial</td>
<td>0.97</td>
<td>0.98</td>
</tr>
<tr>
<td>water_nsquared</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>barnes</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>ocean (continuous)</td>
<td>1.04</td>
<td>0.63</td>
</tr>
<tr>
<td>blackscholes</td>
<td>0.97</td>
<td>0.97</td>
</tr>
<tr>
<td>swaptions</td>
<td>0.94</td>
<td>0.94</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>1.91</td>
<td>1.19</td>
</tr>
<tr>
<td>x.264</td>
<td>0.85</td>
<td>0.87</td>
</tr>
<tr>
<td>mpegenc</td>
<td>1.18</td>
<td>1.19</td>
</tr>
<tr>
<td>mpegdec</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>average</td>
<td>1.07</td>
<td>1.06</td>
</tr>
</tbody>
</table>

of WSS in x.264 can be overcome by sending aggregate periodic notification of L1 hits to L2 cache. However, we do not explore this option in this thesis as it is out of scope for our problem. As per our knowledge, this is the first approach which can estimate aggregate WSS of multiple threads/processes executing concurrently on a CMP. TWSS can estimate WSS of an individual thread if it is implemented in L1 cache.

4.3.2 Hardware complexity of hash functions

The hash function based on prime modulo is easy to implement in hardware but it under-estimates WSS considerably for majority of applications. The same applies to hash function used in swaption benchmark. The hash function based on srand and rand function gives reasonable performance for all applications, except for ocean and cholesky. We measured the approximate number of instructions executed by srand and rand functions implemented in standard “C” library. We compiled these functions using -O3 option for SESC simulator. Altogether these functions execute 13K instructions in 19K clock cycles. Table 4.2 gives the number of instructions executed to hash addresses accessed at the L2 cache. It also shows the number of clock cycles required to do the same. Clearly, srand based hash function requires significant number of instructions. This also implies that its hardware implementation is quite complex. Considering this data, implementing these hash functions in hardware is not practical. Though latency of these functions is not a bottleneck, it will consume considerable amount of power and increase latency. This will require larger queues for these modules. Hence, next we consider a more feasible hardware implementation of deterministic hash functions.
### Table 4.2

<table>
<thead>
<tr>
<th>App.</th>
<th>total instructions (in Billions)</th>
<th>clock cycles (in Billions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>545</td>
<td>811</td>
</tr>
<tr>
<td>radix</td>
<td>109</td>
<td>162</td>
</tr>
<tr>
<td>raytrace</td>
<td>451</td>
<td>672</td>
</tr>
<tr>
<td>cholesky</td>
<td>323</td>
<td>481</td>
</tr>
<tr>
<td>water_spatial</td>
<td>21</td>
<td>31</td>
</tr>
<tr>
<td>water_nsquared</td>
<td>88</td>
<td>131</td>
</tr>
<tr>
<td>barnes</td>
<td>1163</td>
<td>1732</td>
</tr>
<tr>
<td>ocean (continuous)</td>
<td>1825</td>
<td>2718</td>
</tr>
<tr>
<td>blackscholes</td>
<td>302</td>
<td>450</td>
</tr>
<tr>
<td>swaptions</td>
<td>242</td>
<td>360</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>75</td>
<td>112</td>
</tr>
<tr>
<td>x.264</td>
<td>319</td>
<td>476</td>
</tr>
<tr>
<td>mpegenc</td>
<td>217</td>
<td>324</td>
</tr>
<tr>
<td>mpegdec</td>
<td>204</td>
<td>304</td>
</tr>
</tbody>
</table>

Table 4.2: shows the number of instructions executed and the number of clock cycles required to evaluate srand and rand functions in the DHP method

#### 4.3.3 Linear Feedback Shift Registers

Linear Feedback Shift Register (LFSR) is a shift register whose input bit is a linear function of its previous state [161]. Some of the applications of LFSR include pseudo random number generators for use in stream ciphers, deterministic input sequence generators for chip testing and serial to parallel (and vice versa) bit conversion.

The initial value of LFSR is called the seed. The most commonly used linear function in LFSR is XOR. Since LFSR is deterministic, the next number generated by LFSR depends on the previous state. The bits position that affect the next state are called taps. The rightmost bit of LFSR is called output bit and the leftmost bit is called input bit. In Fibonesi LFSR, all bits at the tap positions are XORed sequentially with the output bit and resulting bit is inserted in the leftmost position of the register. Fig. 4.6 shows the Fibonesi LFSR which is 7 bits in length and uses taps at $7^{th}$ and $6^{th}$ bit positions. The $7^{th}$ bit is the rightmost bit. The correct tap position generates a maximum sequence of $2^N - 1$ numbers, where $N$ is the length of LFSR in terms of the number of bits. The 7-bit Fibonesi LFSR can generate maximum sequence of 127 random unique numbers from a seed and then the sequence repeats. This configuration is expressed with $X^7 + X^6 + 1$ polynomial.

We used the hash functions based on Fibonesi LFSR to determine WSS using DHP method. We evaluated a few hash functions with various register lengths such as 13, 16 and 19 bits. We used the tap positions for these hash functions so that these functions generate maximal possible unique numbers. This also means that the length of bit-vector in the DHP method is $2^{13} - 1$, $2^{16} - 1$ and $2^{19} - 1$ for 13, 16 and 19 bits Fibonesi LFSR, respectively. These lengths are much larger than 8K bits used in TWSS. Listing 4.4 shows the source listing for the 13-bit Fibonesi LFSR.
Chapter 4. Working Set Size Estimation

47

Figure 4.6: The 7-bit Fibonesi LFSR with taps at 6th and 7th bit positions. Its configuration is expressed as \(x^7 + X^6 + 1\) polynomial

Listing 4.4: Fibonesi 13 bits LFSR

```c
int fibonesi_13bits(physical_address_t addr)
{
    unsigned bit;
    int m = bitSelect(addr, 6, 6+12);
    uint16_t output = 0;
    uint16_t lfsr = m;
    bit = ((lfsr >> 0) \& (lfsr >> 1) \& (lfsr >> 2) \& (lfsr >> 5)) & 1;
    lfsr = (lfsr >> 1) | (bit << 12);
    return lfsr;
}
```

We also evaluated DHP with 32-bit Galios LFSR as a pseudo random number generator. \(X^{32} + X^{31} + X^{29} + 1\) describes its characteristic polynomial. The source listing of Galios LFSR is shown below:

Listing 4.5: Galios 32 bits LFSR

```c
int galios_32bits(physical_address_t addr)
{
    uint32_t lfsr = addr;
    lfsr = (lfsr >> 1) \& ~(-lfsr & 1u) & 0xD0000001u;
    // bit vector size is 64K
    lfsr = lfsr \% bitVectorSize;
    return lfsr;
}
```

Fig. 4.7 gives WSS estimated using DHP method with hash functions based on Fibonesi and Galios LFSRs. It can be seen from Fig. 4.7 that none of the hash functions in the DHP method estimate WSS accurately for all applications. The WSS estimated with 13, 16 and 19 bit Fibonesi LFSRs underestimate WSS for majority of applications. Though the DHP method with 19 bit Fibonesi LFSR can generate up-to \(2^{18} - 1\) distinct numbers, it under-estimates WSS for most of the applications. Similar behaviour can be observed with 32 bit Galios LFSR. On the contrary, WSS estimated by TWSS is equal to the AWSS for all applications.

Hence, we conclude from these experiments that even though the DHP method is simpler to implement, WSS estimated by it depends greatly on the hash function it uses. Finding a hash function which works well for all benchmarks is a hard problem.
Chapter 4. Working Set Size Estimation

Figure 4.7: shows average ratio of WSS estimated using DHP method with LFSRs as hash functions. WSS estimated using TWSS is repeated here from Fig. 4.5 for convenience.

<table>
<thead>
<tr>
<th>App.</th>
<th>TWSS</th>
<th>AAL</th>
<th>CMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>0.99</td>
<td>0.65</td>
<td>-0.88</td>
</tr>
<tr>
<td>radix</td>
<td>0.99</td>
<td>-0.9</td>
<td>-0.93</td>
</tr>
<tr>
<td>raytrace</td>
<td>0.99</td>
<td>-0.88</td>
<td>-0.88</td>
</tr>
<tr>
<td>cholesky</td>
<td>0.99</td>
<td>0.49</td>
<td>0.58</td>
</tr>
<tr>
<td>water_spatial</td>
<td>0.98</td>
<td>0.02</td>
<td>-0.2</td>
</tr>
<tr>
<td>water_nsquare</td>
<td>0.99</td>
<td>0.2</td>
<td>0.27</td>
</tr>
<tr>
<td>barnes</td>
<td>0.99</td>
<td>0.09</td>
<td>-0.13</td>
</tr>
<tr>
<td>ocean (continuous)</td>
<td>0.99</td>
<td>0.13</td>
<td>-0.22</td>
</tr>
<tr>
<td>blackscholes</td>
<td>1.0</td>
<td>-0.17</td>
<td>0.04</td>
</tr>
<tr>
<td>swaptions</td>
<td>0.99</td>
<td>0.63</td>
<td>-0.37</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>0.99</td>
<td>0.56</td>
<td>0.43</td>
</tr>
<tr>
<td>x264</td>
<td>0.99</td>
<td>0.67</td>
<td>0.1</td>
</tr>
<tr>
<td>mpegenc</td>
<td>0.98</td>
<td>0.04</td>
<td>0.21</td>
</tr>
<tr>
<td>mpegdec</td>
<td>0.98</td>
<td>-0.98</td>
<td>-0.79</td>
</tr>
</tbody>
</table>

Table 4.3: shows correlation of WSS estimated with various metrics to true/actual WSS (AWSS)

4.4 Comparison with AAL and CMR metrics

Average memory access latency is commonly used as a metric to optimize cache [33]. However, due to the distributed nature of NUCA caches, AAL is not a very suitable metric. Similarly, cache miss ratio does not change significantly if the cache is over-allocated. Due to these reasons, it is not clear how quickly these metrics can react to the changes in WSS. Hence, we determine correlation of AAL and CMR with the actual WSS determined at L2 cache. Like our prior experiments, these readings are also taken after every 4M clock cycles. Table 4.3 shows correlation between AAL and CMR with AWSS. It also shows correlation between WSS estimated with TWSS to AWSS. AAL and AWSS are poorly correlated, which is quite intuitive. We use realistic values of NoC link latencies. Hence, AAL
Figure 4.8: Average and standard deviation of replacements per set. The Y axis plots average replacements per set and its standard deviation against the monitoring interval on the X axis.

also depends on time spent in the NoC traversal in addition to off-chip memory access time. This makes AAL unsuitable for larger NUCA caches, unlike smaller caches. CMR also shows poor correlation to AWSS, which is not intuitive. Due to larger size of cache, most of the accesses are L2 cache hits. As a result, larger AWSS does not indicate larger cache miss rate. Hence, correlation between CMR and AWSS is also poor. On the contrary, TWSS sets active bit in the case of L1 replacement, changes in sharers’ list or change in access permission, which enable it to accurately estimate WSS.

4.5 Cache line replacement distribution

TWSS counts a replaced active cache line with a replacement counter. The counter counts the same active cache line address multiple times, if it gets accessed and replaced more than once in the same monitoring period. This results in over-estimation of WSS. Hence, we profiled replacements per set of all sets in all L2 slices. Fig. 4.8 shows the average replacements per set and its standard deviation for some of the applications. We have taken these readings after every 4M clock cycles, which is the
monitoring period we use in all experiments. Some of the applications do not show any cache line replacements. Hence, we have omitted their plots in Fig. 4.8. It can be seen from Fig. 4.8 that average replacements per set varies for different applications. fft shows less than 2 replacements per set in initial execution intervals. It increases significantly in last few execution intervals. On the contrary, cholesky and ocean show up-to 10 replacements per set during the entire execution. The important point to be noted here, is that the standard deviation of the average replacements per set is low for all applications. Applications with large WSS show higher replacements per set (e.g. ocean and cholesky). For higher average replacements per set, standard deviation is also higher. However, in such applications, cache resizing does not happen. Hence, over-estimation of WSS caused due to higher replacements per set, does not introduce significant error in cache resizing using TWSS.

4.6 Sensitivity to L2 slice size

TWSS over-estimates WSS if the same cache line address is accessed and replaced multiple times in the same monitoring interval. The number of active line replacements depends on the cache size. Hence, we evaluate accuracy of TWSS for 256KB L2 slice. We use the same network latency as that used for 512KB L2 slice. Fig. 4.9 plots the average ratio of WSS estimated using TWSS to AWSS evaluated at L2 cache. For applications with smaller WSS such as fft, water_spatial and water_squared, WSS is not over-estimated. However, for ocean and cholesky, TWSS over-estimates WSS by 17% and 4%, respectively. This is because, these applications exhibit more active line replacements for smaller cache. However, in our application of cache resizing, over-estimation is acceptable as cache is not switched-off in the case of applications with larger WSS.
4.7 Evaluating with multiprogramming workload (SPEC 2006 benchmarks)

As explained in chapter 3, we also evaluate TWSS method with a multiprogramming workload obtained using SPEC 2006 benchmarks. Please refer to Section 3.7 for details on the multiprogramming workloads.

To demonstrate applicability of TWSS to a non-tiled architecture, we schedule four applications on a quad-core CMP with 4MB shared L2 cache. The experimental configuration is shown in Fig. 3.6 in Chapter 3.

Fig. 4.10 shows the average ratio of WSS estimated using TWSS and DHP methods to the AWSS evaluated at L2 cache. The DHP method uses various hash functions explained previously in Section 4.3. It can be seen that modulo function over-estimates WSS for all multiprogramming SPEC workloads. On the contrary, it under-estimates WSS for most of the multithreaded applications as shown in Fig. 4.5. The DHP method with srand based hash function over-estimates WSS for all applications, except libq-libq-libq-libq in which case it under-estimates by 52%. The DHP method significantly under-estimates if swapation or LFSR based hash functions are used. This re-confirms that WSS estimated with the DHP method depends on the hash function it uses, which is in line with conclusions drawn for the multithreaded benchmarks.

![Figure 4.10: The average ratio of WSS estimated using TWSS and DHP method with various hash functions. The experiments are performed using multiprogramming SPEC benchmarks on the non-tiled CMP](image)

Table 4.4 shows correlation of WSS estimated using TWSS to the AWSS measured at L2. TWSS is very well correlated to AWSS. AAL and CMR also show good correlation to AWSS in case of mcf-mcf-mcf-mcf. However, these metrics are poorly correlated to AWSS for the rest of workloads. It should be noted that, when compared to multithreaded workloads, multiprogramming workloads show comparatively better correlation between AAL/CMR and AWSS.
Table 4.4: shows correlation of various metrics to actual WSS (AWSS)

<table>
<thead>
<tr>
<th>App.</th>
<th>TWSS</th>
<th>AAL</th>
<th>CMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>dealII-dealII-dealII-dealII</td>
<td>0.99</td>
<td>0.61</td>
<td>0.65</td>
</tr>
<tr>
<td>milc-dealII-gcc-soplex</td>
<td>0.99</td>
<td>0.34</td>
<td>0.33</td>
</tr>
<tr>
<td>soplex-bzip2-named-dealII</td>
<td>0.87</td>
<td>0.39</td>
<td>0.56</td>
</tr>
<tr>
<td>sjeng-mcf-milc-dealII</td>
<td>0.99</td>
<td>0.68</td>
<td>0.67</td>
</tr>
<tr>
<td>milc-mcf-namd-dealII</td>
<td>0.99</td>
<td>0.69</td>
<td>0.69</td>
</tr>
<tr>
<td>mcf-mcf-mcf-mcf</td>
<td>0.99</td>
<td>0.98</td>
<td>0.97</td>
</tr>
<tr>
<td>libq-libq-libq-libq</td>
<td>0.94</td>
<td>0.41</td>
<td>0.87</td>
</tr>
<tr>
<td>mcf-milc-sjeng-libq</td>
<td>0.99</td>
<td>0.29</td>
<td>0.21</td>
</tr>
</tbody>
</table>

The graphs showing average replacements per set and WSS information for multiprogramming workloads is given in Appendix B. SPEC benchmarks have larger WSS compared to the multithreaded benchmarks. Hence, a combination with lesser memory intensive programs, such as dealII-dealII-dealII-dealII shows lesser replacements per set than mcf-deal-soplex-gcc which has one application with higher cache miss rate. TWSS method estimates WSS accurately for SPEC benchmarks as well. WSS estimated with TWSS shows an average correlation of 99% with AWSS. The DHP method with srand and prime modulo hash functions over-estimates up-to 48% and 87%, respectively. On the contrary, rest of the hash functions under-estimate WSS significantly.

4.8 Summary

We propose a highly accurate WSS estimation method which we call “Tagged Working Set Size (TWSS)” method. TWSS method is implemented in L2 cache to estimate aggregate cache requirement of all threads/processes. However, it can also be implemented in L1 cache to determine WSS of an individual thread. It over-estimates WSS if the same address is accessed and replaced multiple times in a monitoring period. However, our empirical data shows that the inaccuracies due to higher active line replacements are very negligible as these caches have a large associativity of 16/32.

We quantitatively compare TWSS to WSS estimated using Dhodapkar’s approach [132]. We show that WSS estimated using DHP method depends greatly on the choice of a hash function. The hash function based on srand and rand function gives reasonable performance for majority of applications. However, hardware implementation of these functions is very complex. On the contrary, other commonly used hash functions such prime modulo and linear feedback shift registers under-estimate WSS significantly for majority of applications.

We also evaluate correlation between WSS estimated using TWSS and other commonly used metrics such as average memory access latency and cache miss ratio. These metrics are poorly correlated to WSS on a CMP. This is because of large over-allocated dispersed nature of NUCA caches. Next few
chapters demonstrate the use of TWSS method to switch-off over-allocated cache.
Chapter 5

Leakage Power Optimization Strategies

In this chapter, we propose a scalable implementation of adaptable associative cache. The TWSS method is implemented in each L2 slice. The use of an L2 slice is estimated using the TWSS method and over-allocated cache associativity is switched off accordingly. This saves the leakage power consumed by the cache.

We quantitatively compare EDP gains obtained with our implementation of adaptable associative NUCA cache to that obtained with other metrics such as average memory access latency and cache miss ratio. We also quantitatively compare our implementation to Bardine’s [21] implementation. Finally, we propose the remap policy in which farther L2 slices are switched off and are mapped on-to the nearer L2 slices.

5.1 The drowsy cache technique

K. Flautner [13] proposed to decrease supply voltage of less frequently used cache lines to reduce their leakage power consumption, which is proportional to the supply voltage. Authors refer to the state of a cache line with reduced supply voltage as a “drowsy state”. In drowsy state, the cache line retains its data. The supply voltage of a cache line is increased to its normal operating supply voltage, before accessing its data. Even though the drowsy cache technique achieves large leakage power savings at negligible performance degradation, prior studies show that the reduction in supply voltage makes caches prone to transient errors [16, 15, 20, 104, 105]. Hence, we compare our technique to the drowsy cache technique for completeness. Moreover, the drowsy cache technique is only applicable to SRAM technology, whereas, our technique can be applied to any future memory technology in which leakage
power difference between the on-state and off-state of a cache bank is significant.

5.2 Variable way SNUCA and DNUCA

David Albonesi [31] first proposed to vary associativity of a set associative cache to save its dynamic power consumption. In his implementation, the over-allocated cache ways are disabled. He suggested to access disabled secondary cache ways in the case of a miss in the enabled primary cache ways. Bardine et al. [21] used the same technique to save leakage power consumption in DNUCA caches. They proposed to switch-off over-allocated cache ways to reduce their static power consumption. However, they applied this technique to a DNUCA cache on a single and dual core platform. They used ratio of the number of hits to the farthest cache way to the nearest cache way as a metric to evaluate cache associativity. We call this metric “cacheUsageRatio”. In DNUCA, less frequently used cache lines are demoted to farther cache ways of a set. Their metric can be applied to smaller CMPs only, in which cores and cache are present on either side of the shared bus. In a more scalable architecture such as tiled architecture or a CMP with many cores, the “cacheUsageRatio” metric cannot be applied. This is because cache ways nearer to some cores can be far for other cores. Moreover, interconnect on a CMP has become more complex and adds significant latency. Hence, algorithm used to vary cache associativity should use locally available information to avoid additional delays and traffic on NoC. The NoC has to be accessed to evaluate commonly used metrics, such as average memory access latency [33] and cache miss ratio [34, 35]. This makes implementations using these metrics unscalable. Hence, we propose to use TWSS estimation method to adjust cache associativity according to cache requirement of an application. As explained in the next subsection, our implementation of variable way associativity scales with the number of tiles/cores. This is achieved by avoiding NoC access while adjusting associativity of L2 slice. Associativity of each L2 slice is adjusted according to its usage. Hence, our algorithm achieves higher energy-delay savings than that obtained with implementations using AAL and CMR metrics.

5.2.1 Determination of cache associativity

As explained in Chapter 4, we use the monitoring interval of 4M clock cycles to estimate cache requirement of an application. The L2 cache controller in each tile executes Algorithm (1) to evaluate associativity of its L2 slice independently without accessing NoC. The number of active bits set (A) and the number of active lines replaced (R) in the previous monitoring interval are used to estimate WSS (EWSS)(lines 1-3). The WSS is in terms of the number of cache lines. Line 4 evaluates size of the cache bank. We assume vertical implementation of cache, i.e. a single way of all sets in an L2 slice is implemented in one bank. Hence, one bank is switched off to reduce associativity by one. Line 5 evaluates associativity of the L2 slice. The minimum associativity of 2 is assigned to avoid frequent
Algorithm 1 evaluates associativity of each L2 slice

1: Get the number of active Lines(A) of an L2 slice
2: Get the number of activeLineReplaced(R) of an L2 slice
3: EWSS = A + R
4: bank_size = total_cache_size/max_associativity
5: assoc_1 = ceil (EWSS/bank_size) + 1
6: r = ceil (R / total_number_of_sets_in_L2_slice)
7: if r > 0 then
8: assoc_1 = assoc_1 + 1*r
9: end if
10: new_assoc = max(max_associativity, assoc_1)
11: current_assoc = getCurrentAssociativity()
12: assoc = 0.5 * new_assoc + 0.5 * current_assoc

conflict misses. Line 6 calculates the number of active cache lines replaced per set. Fig. 4.8 and Appendix B show the distribution of average replacements per set and its deviation. Our simulation data shows lesser deviation if the average replacements per set is small. Hence, for nonzero replacements per set, the cache associativity of all sets is increased and increment is proportional to the number of replacements per set.

The use of filter in line 12 avoids sudden changes and oscillations in associativity. When phase interval of an application is smaller than the monitoring interval, the filter reduces fluctuations caused due to phase changes in an application. For phases longer than the monitoring interval, associativity calculated in the previous interval is reused. Multiple threads/processes execute concurrently in different phases [162] on a CMP. Hence, associativity should be adjusted after every fixed interval of time, instead of on detecting phase changes [132].

5.2.2 Hardware implementation details

We assume that one way of cache is implemented in one bank. Hence, to reduce associativity of all sets by four, four banks are switched off. In other words, the switch-off granularity is at the bank level. The circuit level implementation is shown in Fig. 5.1. A bank is switched off (on) by switching off (on) the transistor connected in series with the SRAM bank.

To reduce cache associativity, the L2 controller holds new L1 requests arrived in its queue. After in-progress L1 requests are serviced, clean L2 cache lines from banks to be switched off are invalidated and modified cache lines are written back to off-chip memory. Then associativity is reduced and servicing of held L1 requests is resumed. In case of increase in cache associativity, L1 requests are held, associativity is increased and then L1 requests are released. This will require slightly large queues. However, it should be noted that once servicing of cache requests is slowed down, automatically instruction execution will slow down. Thus these queues need not be significantly larger than queue if selective associativity is not used. For a fair comparison, we use queues of the same size in all our implementations and the reference
Chapter 5. Leakage Power Optimization Strategies

Figure 5.1: An SRAM bank is switched off to reduce associativity of all sets in an L2 slice by one execution. Our simulator models time and power overhead caused due to held L1 requests, cache line write-backs and invalidations.

In case of DNUCA, when a cache line is promoted/demoted, its active bit is also transferred along with its tag bits. The active bit is set due to L1 misses only and not due to cache line migrations caused in DNUCA. This ensures that the over-estimation of WSS does not happen due to promotions/demotions of a cache line in DNUCA. We call our implementation of adaptable way DNUCA and SNUCA “TWSS.D” and “TWSS.S”, respectively hereafter.

5.3 Evaluation of adaptable way SNUCA

5.3.1 Comparison with Dhodapkar’s metric

DHP Implementation: As explained earlier, Dhodapkar et al.[132] maintain a bit-vector of 1K bits in the core. They hash CLA and set the corresponding bit in the bit-vector. The number of bits set in the bit-vector probabilistically determines WSS of an application. In DHP, for a fair comparison, we maintain a bit-vector of 8K bits in each L2 slice and use hash function based on srand and rand functions as mentioned in Section 4.3. The bit in the bit-vector is set by hashing CLA accessed by L1 cache. We call this implementation “DHP.srand” in our description. We have also implemented hash function based on galois linear feedback shift register. For this implementation, we have used a 64K bit-vector which is eight times compared to that used in TWSS.S. We call this implementation “DHP.galios” in our description.

Fig. 5.2(a) shows EDP obtained with TWSS.S, DHP.srand and DHP.galios. These reading are normalized with respect to (w.r.t.) that obtained with SNUCA implementation. The EDP savings obtained using TWSS.S and DHP.srand are comparable for all applications, except in case of cholesky and ocean. DHP.srand underestimates WSS by 19% and 37%, respectively (Fig. 4.5). As a result, their EDP degrades by 28% and 31%, respectively. The degradation in EDP is mainly because of around
5.3.2 Comparison against AAL and CMR metrics

Implementation details of AAL.S, CMR.S: In these configurations, AAL and CMR are used as a metric to vary cache associativity. Their values are evaluated by considering accesses made by all threads to the last level L2 cache. The cache associativity of all L2 slices is reduced (increased) if
Figure 5.3: Graphs show normalized EDP and execution time obtained with various implementations of power efficient SNUCA

AAL/CMR is lesser (greater) than the metric value obtained in the previous time slot by 10%, otherwise it remains same. Here, it should be noted that NoC access is essential to evaluate AAL or CMR. In these configurations, decision is taken by one L2 controller and then conveyed to the remaining L2 controllers. Because of this, these implementations scale poorly with the number of cores. Apart from this, all L2 slices use the uniform cache associativity, despite their non-uniform usage. We call these implementations “AAL.S” and “CMR.S” in the following description.

Fig. 5.3(a) and Fig. 5.3(b) compare EDP and execution time obtained with TWSS to that obtained with AAL and CMR metrics. TWSS.S gives 25% and 19% higher EDP savings than that obtained with AAL.S and CMR.S, respectively. Larger EDP savings in TWSS.S are due to lesser cache allocation. Fig.
Chapter 5. Leakage Power Optimization Strategies

Figure 5.4: Cache usage normalized w.r.t. the total cache size

5.4 shows the average cache allocated by each of these metrics. It is normalized w.r.t. the total cache size. TWSS.S allocates average of 22% and 12% lesser cache than that allocated by AAL.S and CMR.S, respectively. This is because AAL.S and CMR.S implementations assign uniform cache associativity to all L2 slices despite their uneven usage and also due to poor correlation to AWSS. On the contrary, TWSS estimates WSS accurately and adjusts cache associativity appropriately.

CMR and AAL show poor correlation to WSS of an application in case of large NUCA caches (Table 4.3). Hence, these implementations are unable to switch-off over-allocated cache in fft, radix and water.spatial applications even if their cache requirement is much lesser. AAL/CMR implementations compare their metric values to that obtained in the previous time slot and not to that in the reference execution allocating the entire cache. This causes significant degradation of execution time in some applications such as in ocean and cholesky. In the case of ocean, AAL.S and CMR.S incur 14% and 22% degradation in execution time. This degrades EDP significantly by 14% and 22% in AAL.S and CMR.S, respectively.

The maximum degradation in execution time caused by TWSS is of 5.4% in mpegenc with 37% savings in EDP. As explained in section 4.2.1, the degradation is due to only 93% of accesses in mpegenc have reuse time lesser than 4M clock cycles. As TWSS.S reduces cache allocation, 7% of accesses that have reuse time greater than 4M clock cycles incur additional cache misses, causing 5.4% degradation in execution time. On the contrary, applications such as blackscholes, water.spatial, swaption etc have 100% accesses with reuse time less than 4M clock cycles. Hence, these applications do not show any degradation in their execution time. Despite lower usage of L2 cache, TWSS.S shows negligible degradation in execution time (average 1.3%) (Fig. 5.3(b)).
5.3.3 Comparison against the drowsy cache

**Drowsy.2, Drowsy.4:** As explained before, the supply voltage of less frequently used cache lines is lowered so as to reduce their leakage power consumption. Cache lines operating at lower supply voltage are said to be in the drowsy mode. The cache line in drowsy mode is switched back to the normal mode (and supply voltage) before its data is accessed. We assume a penalty of 1 clock cycle for drowsy to normal mode change and vice versa. We implement the drowsy cache at subbank level and check for its mode change at every 4000 clock cycles, as mentioned in [13]. We assume size of each subbank as 128KB. This assumption is similar to PageNUCA proposal [72].

Gammie et al. [15] use the ratio of leakage power dissipated in the drowsy state to that in the normal state as 4 for 45nm technology. [13] uses the same ratio as 12.79 for 70nm technology. We consider 32nm technology in our experimental setup. Considering the trend, this ratio will be lesser than 4 for 32nm technology. Hence, we give readings with 4 and 2 as ratio of the leakage power dissipated in normal to drowsy mode. We call these implementations “Drowsy.4” and “Drowsy.2” in rest of our thesis.

Fig. 5.3(a) also compares EDP obtained with TWSS.S to that obtained with Drowsy.4 and Drowsy.2 implementations. Fig. 5.3(b) compares execution time obtained with these implementations. The drowsy technique causes very negligible degradation in execution time which is in line with results obtained in [13]. The drowsy cache technique gives higher EDP savings in case of cholesky and ocean. These applications have larger WSS. Hence, TWSS.S is unable to switch-off cache and achieve energy savings. However, in case of applications with lesser cache requirement such as in mpegdec, water_spatial, water_squared and blackscholes, TWSS.S achieves higher EDP savings than the drowsy cache technique. This is because, with the complete switch-off strategy, TWSS.S reduces leakage power consumption to zero, whereas, the drowsy reduces leakage power consumption to a lower value by reducing the supply voltage.

It should be noted that with decreasing transistor sizes and nominal supply voltage values, reliability of caches reduces [16, 15, 20, 104, 105]. The cache becomes more susceptible to transient errors. In [20], authors propose to read data from the lower level cache if soft-error is detected in the drowsy instruction cache. Error correcting code maintained with each cache line is used to detect a soft-error. They also propose to periodically verify cache data correctness so as to fix errors with error correction code (scrubbing). We have implemented the basic drowsy cache technique and do not model any such corrective measures in our simulation. With these corrective techniques, the actual gains obtained with the drowsy cache will be lower than that shown here. On an average, TWSS.S achieves 15% higher EDP savings than the Drowsy.2 implementation.
5.3.4 Increase in memory accesses

Fig. 5.5 shows the number of memory accesses made by all implementations. Since the drowsy technique is a data retaining implementation of reducing leakage power, the number of memory accesses made by the drowsy cache technique is same as that in the reference SNUCA. Hence, its memory access data is not not plotted in Fig. 5.5. Similarly, the number of memory accesses made by DHP.galios are more than that done by the DHP.srand implementation. This is because under-estimation of WSS is higher in DHP.galios than that observed in DHP.srand. Hence, we have also not plotted data for DHP.galios in Fig. 5.5. The number of memory accesses made by the DHP.srand, AAL.S and CMR.S methods are significantly higher in the case of radix, cholesky and ocean. This explains degradation in execution time of these applications. Since WSS of radix is small, increase in the number of memory accesses is compensated by reduction in the leakage power dissipated in L2 cache. However, cholesky and ocean have large cache requirement. Hence, memory accesses and execution time increase significantly on allocating lesser cache. This degrades EDP by 28%-31% with the DHP.srand method.

In the case of barnes, though the number of memory accesses increases by 7x with TWSS.S and DHP.srand methods, power consumed in off-chip DRAM increases by 1.89x. These methods save leakage energy dissipated in L2 cache by 2.13x with overall 2% degradation in execution time. With this, EDP savings of around 30% are achieved using TWSS.S and DHP.srand methods. This is because DRAM operates at lower frequency than the on-chip cache. Hence, energy dissipated in off-chip DRAM does not increase proportionately with increase in memory accesses. TWSS.S shows higher average number of memory accesses than AAL.S and CMR.S implementations. This is because of lower cache usage by TWSS.S compared to the other two implementations, which can be seen in Fig. 5.4.
As explained earlier, Bardine et al. [21] use ratio of number of hits to the farthest cache line to hits to the nearest cache line as a metric to change cache associativity in DNUCA. We refer to this ratio as “cacheUsageRatio”. This metric works in case of DNUCA as frequently used cache lines gradually move towards cores. However, in a tiled architecture, cache lines nearer to one core are far for other cores. Hence, Bardine’s method can be applied only up to four threads scheduled on cores in the first column of a tiled architecture (see Fig. 1.1.). Hence, to compare TWSS.D to Bardine’s implementation, we execute all workloads with only four threads scheduled on cores in the first column. If the cacheUsageRatio is less than the threshold \( T_1 \) then associativity is reduced by 1. If the cacheUsageRatio is greater than the threshold \( T_2 \) then associativity is increased by 1. We refer this implementation as “Bardine.D” in rest of the thesis. In this comparison of Bardine.D and TWSS.D (spawning four threads), L2 cache lines are migrated to farther L2 slices in the same bankset on replacement in its present L2 slice. In case of TWSS.D, the replacement counter is incremented only in the last L2 slice when a cache line is evicted to off-chip DRAM.

Fig. 5.6 compares EDP obtained with TWSS.D to that obtained with Bardine’s metric. TWSS.D achieves an average of 7.8% higher EDP savings than Bardine.D. This is because in Bardine.D, cache associativity is gradually adjusted according to changes in WSS, whereas, TWSS.D estimates WSS accurately and adjusts cache associativity to the correct value immediately in the next monitoring interval. Accurate estimation of WSS achieves higher EDP savings with negligible performance degradation, except in case of ocean. Ocean has smaller WSS initially and it increases suddenly after a few monitoring intervals. In Bardine.D\(^1\), associativity is reduced if the hit ratio is smaller than that obtained in the

---

\(^1\)Threshold values are used as per [21].
Figure 5.7: Graphs (a) and (b) show normalized EDP and execution time obtained with various implementations of power efficient DNUCA.

previous monitoring interval. As associativity reduces to very small value of four, hit ratio does not increase on a sudden increase in WSS. Hence, Bardine.D fails to increase cache associativity, degrading execution time by 37%. On the contrary, in TWSS.D cache lines are migrated to farther slice along with its active bit and the replacement counter counts the number of replaced active cache lines in the last L2 slice of a bankset. With this, TWSS.D detects sudden increase in WSS and increases cache associativity, giving execution time degradation of 9% in ocean which is much lesser than 37% observed in Bardine.D. For other applications, execution time degradation is less than 5% in TWSS.D, as shown in Fig. C.1 (please refer Appendix C). In addition to giving average of 7.8% higher EDP savings than that obtained with Bardine.D, TWSS.D is scalable with the number of cores on a CMP, unlike Bardine.D.

5.3.6 Scalability of TWSS.D

To demonstrate scalability of TWSS, we execute applications with sixteen threads scheduled on a sixteen tiled CMP with DNUCA. In above experiments, TWSS.D migrates cache lines to farther L2 slice on
evicting it from the current L2 slice. It is sent to offchip DRAM only when it gets evicted from the
farthest L2 slice. However, in these experiments, sixteen threads execute concurrently on sixteen cores.
Hence, cache line is sent to the off-chip DRAM on replacement and each L2 cache controller increments
its activeLineReplaced counter on replacing a cache line with active bit set.

We compare TWSS.D quantitatively against EDP savings obtained with AAL and CMR as metrics
and also with the drowsy cache technique applied to DNUCA. These implementations are same as that
described in the case of SNUCA cache with the exception that here DNUCA cache access policy is used.
These implementations are referred to as AAL.D, CMR.D, Drowsy.4 and Drowsy.2 in the following
description.

Fig. 5.7(a) plots EDP normalized with respect to that obtained with the reference DNUCA. TWSS.D
estimates cache usage of each L2 slice accurately and assigns associativity accordingly. As a result, it
achieves an average of 36% EDP savings over the reference DNUCA. On the contrary, AAL.D and
CMR.D achieve only an average of 10% and 13% EDP savings over the reference DNUCA. As shown
in Table 4.3, AAL and CMR poorly correlate with WSS estimated at the last level shared L2 NUCA
cache. Previous approaches use these metrics to reconfigure L1 cache [34, 35]. As majority of accesses
are L1 hits, these metrics do not correlate very well with WSS estimated at L2 cache. Apart from
this, EDP savings obtained depend on how quickly metric responds to changes in WSS. Unlike TWSS,
CMR and AAL metrics estimate WSS indirectly. Hence, in case of ocean, CMR.D and AAL.D show
7.7% and 18% degradation in execution time as shown in Fig. 5.7(b). This degrades EDP by 13% and
29%, respectively. Due to initial smaller WSS, cache associativity is decreased. On increase in WSS,
cache allocation is increased gradually in the following monitoring intervals. Moreover, CMR/AAL is
compared against values obtained in the previous monitoring intervals and not against the corresponding
values in the reference execution with the entire cache allocated. Hence, these metrics fail to quickly
respond to changes in cache needs of an application. Whereas, TWSS gives direct estimation of cache
requirement, because of which, it responds quickly to cache needs of an application. Hence, TWSS.D
does not degrade execution time in case of ocean. Overall, TWSS shows very negligible degradation in
execution time (an average of 1.1%).

In case of DNUCA, the drowsy cache technique is not as effective as in SNUCA because of cache
line promotions and demotions performed in DNUCA. Hence, TWSS.D achieves 24% and 16% higher
EDP savings than Drowsy.2 and Drowsy.4 implementations.

5.3.7 Sensitivity to size of L2 slice
We determine EDP savings obtained with L2 slice of size 256KB. In these experiments, we keep network
latencies same as that of 512KB L2 slice. This is to demonstrate the effect of smaller cache on TWSS.S
and TWSS.D policies. Fig. 5.8(a) plots EDP of TWSS.S normalized with respect to the reference
execution on SNUCA cache access policy allocating the entire L2 cache. For smaller 256KB L2 slice, the number of cache ways switched-off are lesser than that with 512KB L2 slice. Hence, EDP savings with 256KB L2 slice are lesser than that obtained with 512KB L2 slice. Clearly, the EDP savings will be more for larger L2 cache. These experiments also show that TWSS estimates cache requirement accurately even for smaller L2 slice and adapts cache associativity accordingly. Hence degradation in execution time for smaller L2 cache is minimal for all applications (Fig. 5.8(b)). Similar results can be observed for TWSS.D as shown in Fig. C.2 (please refer Appendix C). In this case, EDP is normalized w.r.t that of the execution with DNUCA cache access policy.

5.3.8 Evaluation with multiprogramming SPEC benchmarks

We evaluated EDP savings by scheduling four SPEC programs on a quad-core CMP with 4MB shared L2 cache. The details are given in Chapter 3.7. Fig. 5.9(a) and Fig. 5.9(b) plot EDP and execution time normalized w.r.t. the reference execution with SNUCA cache access policy. TWSS.S achieves higher EDP savings than CMR.S and AAL.S as CMR and AAL gradually adjust cache associativity by comparing metric values to that obtained in the previous monitoring interval. On the contrary, TWSS.S adjusts cache associativity according to WSS of an application. As given in Table 4.4, CMR and AAL
(a) Smaller EDP value is better

(b) Smaller execution time is better

Figure 5.9: Evaluation of SPEC benchmarks with SNUCA cache access policy

show poor correlation to WSS of a workload. As a result, TWSS.S achieves up-to 32% higher EDP savings than both AALS and CMR.S implementations.

We compare DHP method with srand and galios LFSR based hash functions. As given in Fig. 4.10, galios under-estimates WSS significantly for all workloads. Hence, execution time with DHP.galios degrades by 15% and 10% in case of dealII-dealII-dealII-dealII and milc-dealII-gcc-soplex workloads. However, leakage power savings compensate for the increased execution time. As a result, EDP does not degrade proportionately in these applications. In case of mcf-mcf-mcf and libq-libq-libq-libq, significant under-estimation of WSS does not degrade execution time proportionately. This is because, in these applications, majority of accesses have very large reuse time. So even if such addresses are kept in L2 cache, eventually these addresses get replaced without a cache hit. Hence, despite under-estimation of WSS in DHP.galios, execution time does not degrade. On the other hand, DHP.srand does not under-estimate WSS and hence cache usage is approximately equal to that in TWSS.S. Thus TWSS.S and DHP.srand achieve similar EDP savings. These experiments re-confirm that the EDP
savings obtained with the DHP method depend on the hash function it employs, whereas, TWSS uses inherent cache hash function which evenly distributes addresses in cache sets. TWSS overcomes aliasing drawback of the DHP method by using tags bits maintained along with the cache lines.

Fig. 5.9(a) also compares EDP savings obtained with the drowsy cache technique. Since TWSS.S is unable to switch-off cache for applications with larger WSS such as libq-libq-libq-libq, the drowsy cache gives higher savings than TWSS.S. On the contrary, TWSS.S gives higher EDP savings in case of applications with smaller cache requirements such as dealII-dealII-dealII-dealII and soplex-bzip2-dealII-namd. As explained earlier, leakage power savings that can be obtained with the drowsy cache technique are decreasing for smaller transistor sizes. Hence, here also we give EDP savings for both Drowsy.4 and Drowsy.2 implementations, in which the ratio of leakage power consumed in the normal mode to that in the drowsy mode is 4 and 2, respectively. As WSS of all these applications is large when compared to 4MB shared L2 cache, both drowsy and TWSS.S give on an average similar EDP savings. However, considering the growing size of L2 cache in current processors (e.g. 24MB in Nehalem and Itanium 9300 processors), EDP savings with TWSS.S will be much higher for larger L2 cache than the drowsy cache technique. It should be noted that with smaller transistor sizes, larger process variation is observed. This makes varying transistor parameters difficult [16, 17, 18]. In addition to this, the drowsy technique makes caches susceptible to soft errors [19, 18, 20]. Hence, we compare it with TWSS.S just for the completeness of our study.

5.3.9 Sensitivity to monitoring interval

To study variation in EDP savings with the monitoring interval, we performed experiments with 1M, 2M, 4M, 8M and 16M as the monitoring interval. Fig. 5.10 and Fig. 5.11 plot variation in the normalized EDP on the primary Y axis against the monitoring interval on the X axis for blackscholes and cholesky. The percentage of accesses with reuse time less than the monitoring interval is plotted on the secondary Y axis. We have also plotted variation in WSS for these applications. Similar comparison for other applications can be found in Appendix C.

The fraction of addresses with reuse time less than the monitoring interval, plays important role in achieving higher energy savings without significant degradation in execution time. If majority of accesses have reuse time less than the monitoring interval, then lesser performance degradation is observed. blackscholes, swaption, x.264, raytrace, water, spatial, water, nsquared and barnes have 98%-99% of CLAs with reuse time less than 1M clock cycle, which means that, if a cache line is not accessed in the previous 1M clock cycles then it will not be accessed in next 1M clock cycles as well. Hence, it is not a part of the working set of an application. In such applications, monitoring interval of 1M clock cycles achieves highest EDP savings. With the monitoring interval of 4M clock cycles and greater, data remains cached even if it has no future use. Hence, EDP gains reduce for the monitoring interval greater
Figure 5.10: blackscholes does not exhibit any active line replacement during execution. (a) Variation in EDP (on the primary Y axis) with monitoring interval (on the X axis) and distribution of reuse time in % (on the secondary Y axis) (b) Variation in WSS with monitoring interval

than 4M clock cycles. On the contrary, cholesky and ocean has larger WSS. Hence, with monitoring interval smaller than 4M clock cycles, execution time and EDP degrades. With 4M and larger monitoring interval, cache is not powered off and thus no EDP degradation is seen. In fft, approximately 94% of accesses have reuse time lesser than 1M clock cycles. However, fft shows 10-15 average replacements per set during execution. Our adaptive way algorithm given in Algorithm (1) increases cache associativity proportionately on nonzero replacements per set. Hence, even if only 94% of accesses have reuse time less than 1M clock cycles in fft, it gives higher EDP savings for both 1M and 2M clock cycles as the monitoring interval.

In general, majority of applications have at least 95% of CLAs with reuse time less than 4M clock cycles. Hence, 4M clock cycles is a suitable monitoring interval for all applications. Applications that have at least 95% of accesses with reuse time less than 1M/2M clock cycles (e.g. blackscholes, swaption), give slightly lesser EDP savings with 4M clock cycles as the monitoring interval.

Multiprogramming applications show higher EDP savings for smaller monitoring interval (1M/2M clock cy.), however, with higher degradation in execution time. Most of these applications also have at least 95% accesses with reuse time less than 4M clock cycles. Hence, these applications give significant EDP savings (20%-40%) with 4M clock cycles. mcf-mile-sjeng-libq and libq-libq-libq-libq have only up-to 80% and 60% accesses with reuse time less than 4M clock cycles. Rest of 20%-40% accesses have reuse time greater than 200M clock cycles. These applications have very large WSS. As a result, cache lines get replaced without accessing them again. Hence, in such cases also monitoring interval of 1M-4M clock cycles works well. Sensitivity study of EDP savings to the monitoring interval for multiprogramming workloads can be obtained in Appendix B.
In this chapter, we demonstrate use of TWSS method to switch-off over-allocated cache associativity. Past studies have used the technique of adjusting cache associativity in case of smaller uniform access latency caches \cite{132, 114}. However, presence of concurrently executing threads make task of estimating cache requirement of co-scheduled processes challenging. Our adaptable associativity implementation for SNUCA cache achieves 25% and 19% higher EDP savings than commonly used metrics such as average memory access latency and cache miss rate, respectively. In case of DNUCA, it achieves 26% and 23% higher EDP savings than that obtained with AAL and CMR metrics, respectively.

Dhodapkar et al. \cite{132} proposed to estimate WSS of a single-threaded application. However, the main drawback of this method is estimated WSS and EDP savings depend on the choice of hash function. Our experiments show that it is hard to find a deterministic hash function which suites for all applications. Hence, in case of cholesky and ocean, execution time degrades by around 20% with SNUCA cache access policy. This degrades EDP by 28% and 31%, respectively. The similar behaviour is also seen with multi-programming workloads obtained using SPEC 2006 benchmarks. The execution time degrades up-to 15% in some cases. Experiments with shared, monolithic cache in case of SPEC 2006 benchmarks also demonstrate that TWSS can be applied to tiled as well as non-tiled cache architecture. On the other hand, Bardine's \cite{21} metric is only applicable in case of DNUCA cache on a smaller CMP. TWSS.D achieves 7.8% higher EDP savings than that obtained with the Bardine's metric. Unlike Bardine's implementation, TWSS.D is also applicable to a scalable large CMP platform. TWSS.D achieves on an average 35% of EDP savings over the reference DNUCA cache implementation. It achieves 24% higher
EDP savings than the drowsy cache technique. Cache line migrations present in DNUCA reduces effectiveness of the drowsy cache technique.

## 5.5 The Remap policy

In adaptable associative cache, cache associativity of each L2 slice is selectively switched off according to its usage in order to achieve leakage power savings. Though our algorithm to adjust cache associativity is far better than the current state-of-the art, cache associativity is decreased conservatively to avoid degradation in execution time of an application. Reduction in cache associativity increases conflict misses. Hence, we propose a new approach in which the entire over-allocated L2 slices are switched off instead of switching off the over-allocated cache ways. The powered off L2 slices are mapped onto nearer powered-on L2 slices. Apart from achieving higher EDP gains, this method also reduces execution time in some applications. We achieve this with what we call the “remap policy”.

![Figure 5.12: Proposed remap policy](image.png)

Fig. 5.12 shows working of the remap policy. The L2 slice in tile 15 is remapped to L2 slice in tile 5 using a remap table/configuration. In this case, L2 slice in tile 5 caches data of both the 5\textsuperscript{th} and 15\textsuperscript{th} L2 slices. On an L1 miss, the remap table is looked up and if the original L2 destination is L2 slice in tile 15 then the request is sent to L2 controller in tile 5.

The remap policy can be applied to SNUCA or DNUCA. However, in DNUCA architecture, on a cache miss in the nearest L2 slice, all remaining L2 slices in a bankset are searched. This makes the lookup logic of DNUCA complicated. Also in case of multi-threaded workloads with threads sharing a lot of data in an interleaved manner, shared data might migrate in conflicting directions. This degrades execution time in such applications. Hence, we believe SNUCA is more likely to be used in future due to its simple lookup and replacement logic. Therefore, we implement the remap policy with SNUCA cache access policy only.

### 5.5.1 Architectural changes

We propose to remap a certain subset of L2 slices onto the remaining ones. When one L2 slice is mapped to another slice, everything else remains same including cores on which threads are executing and hence
location of their L1 caches. Only accesses made to the L2 slice in one tile are remapped to the L2 slice in another tile.

The remap configuration can be implemented as part of an L1 controller and is similar to a hardware register file. On an L1 miss, the L1 controller sets a destination L2 slice to its remapped L2 slice before sending request over NoC, as shown in Fig. 5.13. The remap policy is independent of whether L1 is physically or virtually tagged. This is also independent of position of tile id bits in the memory address. This is because, virtual to physical address translation (TLB lookup) happens before sending a request to L2 over NoC. Hence, the destination L2 slice bits are always available during the remap table lookup. The remap table access latency can be hidden while generating out-going request to the destination L2 slice. We assume that the position of tile identifier bits is between tag and set index bits. The remap policy can also be applied if tile identifier bits are between cache offset and set index bits.

5.5.2 Determination of the remap table

To evaluate the remap configuration at run-time, the metric should consider the following:

1. What is the cache requirement of an application or of a multiprogramming workload executing on a CMP?

2. Which L2 slices should be powered-off and which should be kept powered-on?

3. How powered-off L2 slices should be mapped to the powered-on L2 slices?

The “tagged working set size” estimation method implemented in the L2 cache is used to estimate cache requirement of an application. However, determination of powered-on/powered-off L2 slices is a non-trivial task. If cache requirement is \( m \) number of L2 slices and total number of L2 slices present on a CMP is \( n \), then there are \( \binom{n}{m} \) ways in which, \( m \) L2 slices can be selected. Each of the powered off \( (n - m) \) slices can be mapped to any of the \( m \) powered-on L2 slices. Hence, there are total \( \binom{n}{m} \times m^{(n-m)} \) remap configurations possible.

On the remap configuration change, cache lines belonging to L2 slices whose mapping has changed are transferred to their new location. This incurs power and time overhead. Hence, two consecutive remap
configurations should not be very different. It is evident that determination of the metric for the adaptive remap policy is a challenging task. Hence, we first determine the near-optimal remap configuration using an offline trace-based approach. We formulate this problem as an energy-delay minimization problem and solve it using scalable genetic algorithms. The near-optimal remap configuration obtained using GA will be helpful to us in two ways:

1. to obtain a metric for adaptive determination of the remap configuration.

2. simulation results obtained with the near optimal remap configuration can also be used to evaluate efficiency of our adaptive remap policy implementation.

Determination of the near optimal remap configuration using genetic algorithms is the topic of our next chapter.
Chapter 6

Offline Determination of Remap Table using Genetic Algorithms

In this chapter, first we give a brief introduction to Genetic Algorithms. We also give details of how we use GA to determine the near optimal remap configuration (NORC). Genetic Algorithms are adaptive randomized search algorithms based on the evolutionary ideas of natural and genetic selection. This is similar to evolution of population according to Darwinian principles [163]. These algorithms are useful in function optimization.

Genetic Algorithms were developed by John Holland [164] and his team. Their goal was:

1. To abstract and rigorously explain the adaptive processes of natural systems
2. To incorporate the underlying mechanisms in software systems

The viewpoint is that the process of evolution seen in natural systems is some kind of optimization with the aim of attaining the best state. The genetic algorithms are a close imitation of natural evolution mechanisms involving survival of the fittest, reproduction and inheritance of attributes by the off-springs from the parents. The genetic algorithms have been reported to be successfully used in a wide range of optimization problems. Goldberg [165] gives a survey of various areas where these techniques have been used. These areas range from optimization problems in biology, physical sciences to social sciences. The wide acceptance of GA is a result of simplicity of the algorithms both in terms of functioning as well as their computer implementation. Genetic algorithms (GA) are different from common optimization and search procedures in four ways:

- GAs work with a coding of the parameter set, not the parameters themselves
- GAs search from population of points, not a single point
Chapter 6. Offline Determination of Remap Table using Genetic Algorithms

- GAs use payoff (objective function) information, not derivatives or other auxiliary knowledge
- GAs use probabilistic transition rules, not deterministic rules

Because of above properties, GAs are more suitable for complex systems where scalability of a search algorithm is more important. GAs are more attractive for complex systems because they are very simple in functioning and their application. Here, we explain the functioning of a simple genetic algorithm.

6.1 Simple Genetic Algorithm

A simple genetic algorithm is an iterative procedure comprising of the following three steps which are often called genetic operators.

1. Reproduction
2. Crossover
3. Mutation

Each of these operators operate on a finite fixed size collection of fixed length strings of some finite alphabet set. Each string is termed a chromosome. The collection of chromosomes is called a population. The string or chromosome is an encoded representation of a point in the domain of the objective function being maximized. The cost function of each string is evaluated. The cost is termed fitness of the solution. The sequence of operators listed above operate on the population and convert it into another population of strings. The final goal of entire iterative procedure is to generate a population containing a string with the maximal objective function value.

6.1.1 Reproduction

Reproduction is a process in which individual strings are copied according to their objective function (fitness) values. The strings with higher fitness value have higher probability of contributing one or more off-springs in the next generation. The reproduction operator can be implemented in algorithmic form in a number of ways. We use roulette wheel algorithm which is explained next. In this, the fitness value of each string in the population is determined and then it is normalized with the sum of all fitness values. The normalized fitness value of a string determines size of the slot that string occupies on a roulette wheel. It also indicates probability with which it is reproduced in the next generation. The string with lower normalized fitness value has lower probability of creating one or more off-springs in the next generation. If \( f(S_i(t)) \) is the fitness of individual \( S_i \) in the population of size \( P \) at generation \( t \), then under fitness proportionate selection, the probability that individual \( S_i \) will be copied into the
Table 6.1: Probability of various strings used in the optimization problem depends on their fitness.

<table>
<thead>
<tr>
<th>No.</th>
<th>String</th>
<th>Fitness</th>
<th>Probability</th>
<th>Cumulative Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01100</td>
<td>50</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>2</td>
<td>11000</td>
<td>25</td>
<td>0.25</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>01000</td>
<td>25</td>
<td>0.25</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

next generation as a result of reproduction operation is,

\[
P_i = \frac{f(S_i(t))}{\sum_{i=1}^{n} f(S_i(t))}
\]  

(6.1)

Consider an example of population of three strings encoded as 01101, 110000, and 01000. Let us assume these strings have fitness values as shown in Table 6.1.

![Roulette wheel selection method](image)

Figure 6.1: Roulette wheel selection method. The size of slots occupied by various strings is proportional to their fitness values.

From this table, it can be seen that the string 1 has the highest probability of going into the next generation (50%) and strings 2 and 3 have lower probability of going into the next generation (25%). As shown in Fig. 6.1, each string occupies proportionate slot on the roulette wheel. Table 6.1 also shows cumulative probabilities of all strings. To determine the string going into the next generation, random number between 0 and 1 is generated. Suppose the generated number is 0.55 then the string 2 is copied as it is in the next generation. If the number generated is less than 0.5 then the string 1 is copied as it is. Similarly, if the number generated is greater than 0.75 then the string 3 is copied. The random numbers are generated till the population of the next generation again becomes equal to size of the initial population. It is evident that the number of copies created in the next generation depends on the size of slot occupied by each string on the roulette wheel (see Fig. 6.1).

### 6.1.2 Crossover

After reproduction, crossover may proceed in two steps. First, two members in the new generation are selected randomly. The two strings then swap their contents at a randomly selected crossover point.
Chapter 6. Offline Determination of Remap Table using Genetic Algorithms

Table 6.2: shows % EDP savings when the number of allocated L2 slices is varied with respect to the reference, which allocates all 16 L2 slices. R2 indicates that two L2 slices are allocated and rest of the fourteen slices are mapped to these two allocated slices. %Pf and %ED give % savings in the execution time and energy-delay product of the application, with respect to the reference.

### 6.1.3 Mutation

Reproduction and crossover play important role in GA by creating new population and thus exploring various regions in the function domain. Typically, mutation is used with a very low probability. It helps to change bits from “0” to “1” or vice-versa within the same string.

We again summarize the working of GA. The important parameters of a problem are encoded in the binary string. The fitness function of a string is defined depending on the problem whose solution is to be determined. The initial population of strings is created randomly. The fitness value of each string is determined. The new population is selected using a selection method based on the fitness of strings. The string with the maximum fitness is tracked in each generation. This procedure continues till either solution of acceptable fitness is obtained or the predetermined number of generations are explored. Next section describes how we use GA to find the NORC.

### 6.2 Application of GA to search the NORC

Previous chapter gives more details of the remap strategy. The remap strategy switches off farther L2 slices and maps them on-to the nearer L2 slices. The remapped L2 slices are switched-off to save leakage power. We varied the number of allocated L2 slices in steps of 2 on a sixteen tiled CMP. For a fixed number of powered-on L2 slices, we searched many remap configurations and evaluated them using the cycle-accurate simulator. The remap configuration giving the highest EDP savings over the reference which allocates all the sixteen L2 slices, is desirable. Table 6.2 shows variation in % EDP savings of mpegenc application [152] as the number of allocated L2 slices is varied.

In mpegenc, the execution time degrades by 1.1% over the reference SNUCA on allocating two L2 slices, however, it achieves 54% EDP savings. The degradation in execution time is due to increased number of DRAM accesses. As the number of allocated L2 slices increases, EDP savings reduce without improving execution time significantly, e.g., on allocating 12 L2 slices the execution time improves by 4.4%. However, the improvement is only 4.7% when 14 L2 slices are allocated. This is because, increase in the allocated cache reduces the number of DRAM accesses, but at the same time, it increases the time spent in transit. It also reduces EDP savings as the total leakage power consumed by L2 slices increases. Hence, one needs to allocate the optimum number of L2 slices so that EDP of an application is...
minimum. In our problem, the number of allocated L2 slices and their location are important. Clearly, as the number of tiles and hence, the number of L2 slices present on a CMP is increased, exhaustive search method does not scale. Hence, we solve this problem using more scalable “Genetic Algorithms”.

The objective of our optimization function is to determine the remap configuration so as to minimize EDP of an application which is described by Eq. 6.2.

\[
EDP = (E_{\text{cache}} + E_{\text{interconnect}} + E_{\text{DRAM}}) \times \text{Time}
\]  

(6.2)

GA is used to generate various remap configurations. EDP value in Eq. 6.2 gives fitness of the remap configuration. To estimate execution time of an application, we predict cache misses and time spent in transit for a given remap configuration using a trace based approach. The meaning of various terms used in our problem formulation is described in Table 6.3.

### 6.2.1 Estimation of energy consumption

- \( E_{\text{cache}} \): Leakage power is proportional to the number of allocated L2 slices. Leakage power per L2 slice is obtained using CACTI 6.0 [52]. The difference between total dynamic power dissipated in L2 slices for various remap configurations is negligible compared to their leakage power consumption. This is evident from the graph shown in Fig. 1.3 of chapter 1 which plots power dissipated in various memory subsystem components. Hence, we ignore dynamic energy consumption in the objective function.

- \( E_{\text{interconnect}} \): Interconnect power does not vary by large amount if the application performance is comparable to the reference. Hence, we ignore it in the objective function.

- \( E_{\text{DRAM}} \): Power consumed in off-chip DRAM depends on the number of memory accesses, row management policy and address correlation between the consecutive accesses. We performed simulation on various applications by varying the number of powered-on L2 slices and obtained off-chip DRAM power consumption. We empirically observed that the memory power consumption remains almost same if the number of DRAM accesses are comparable to the reference. We use closed row management policy, hence, correlation between the consecutive memory accesses can be ignored. We use an approximate memory power consumption model based on the linear interpolation of empirical data. Fig. 6.2 shows the models used to estimate power consumed by offchip memory for various remap configurations. If significantly higher memory accesses are predicted for a remap configuration than the reference, then the memory model discards that remap configuration. If the number of memory accesses are low due to over-allocation of L2 slices
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Method of Determination</th>
</tr>
</thead>
<tbody>
<tr>
<td>$EDP$</td>
<td>Energy-Delay Product</td>
<td>Eq. 6.3</td>
</tr>
<tr>
<td>$E_{cache}$</td>
<td>Energy consumed by the cache</td>
<td>CACTI[52]</td>
</tr>
<tr>
<td>$E_{interconnect}$</td>
<td>Energy consumed by interconnect</td>
<td>-</td>
</tr>
<tr>
<td>$E_{DRAM}$</td>
<td>Energy consumed by off-chip DRAM</td>
<td>Statistical model of data obtained using DRAMSim[49]</td>
</tr>
<tr>
<td>$Time$</td>
<td>Time taken by an application to complete its execution</td>
<td>Eq. 6.4</td>
</tr>
<tr>
<td>$C_{total}$</td>
<td>Total time spent in servicing all L1 misses</td>
<td>Eq. 6.9</td>
</tr>
<tr>
<td>$C_{remap}$</td>
<td>Total time spent in servicing all L1 misses for a given remap table</td>
<td>Eq. 6.10</td>
</tr>
<tr>
<td>$K$</td>
<td># of threads used by an application for execution</td>
<td>Specification</td>
</tr>
<tr>
<td>$N$</td>
<td># of tiles/L2 slices on the CMP</td>
<td>Specification</td>
</tr>
<tr>
<td>$S$</td>
<td># of L2 slices allocated to an application</td>
<td>GA (Section 6.2.5)</td>
</tr>
<tr>
<td>$\eta_{ij}$</td>
<td># of accesses between L1 cache in tile $i$ and L2 slice in tile $j$</td>
<td>Profiled information</td>
</tr>
<tr>
<td>$L_{ij}$</td>
<td>Network access latency between L1 cache in tile $i$ and L2 slice in tile $j$</td>
<td>Floorplanning, Intakte[51]</td>
</tr>
<tr>
<td>$\alpha_{ij}$</td>
<td># of cache misses incurred by L2 slice in tile $j$ due to traffic generated by thread executing on a core in tile $i$</td>
<td>Dinero (Section 6.2.5)</td>
</tr>
<tr>
<td>$p_j$</td>
<td>Penalty to service a single cache miss in L2 slice in tile $j$. This penalty includes latency of a memory access and network latency between L2 cache in tile $j$ and the memory controller.</td>
<td>Intakte, an average of DRAM latencies obtained using DRAMSim</td>
</tr>
<tr>
<td>$\delta$</td>
<td>remap function, $\delta(j) = i$ refers to L2 slice in the tile $j$ is mapped to L2 slice in the tile $i$. $\delta(i) = i$ refers to L2 slice from tile $i$ is mapped to itself.</td>
<td>GA (Section 6.2.5)</td>
</tr>
<tr>
<td>$K_c$</td>
<td>Constant decides priority given to the cache miss penalty</td>
<td>Empirically</td>
</tr>
<tr>
<td>$K_t$</td>
<td>Constant decides priority given to the transit time penalty</td>
<td>Empirically</td>
</tr>
</tbody>
</table>

Table 6.3: gives meaning of the various terms used in genetic algorithm based problem formulation. The last column describes how these terms are obtained
then that solution is discarded due to excessive leakage power consumption in the cache. This ensures that our model allocates the right number of the L2 slices.

Considering these assumptions, EDP can be simplified as in Eq. 6.3.

\[
EDP \approx \left( (\text{Leakage Pwr Per L2 Slice} \times \# \text{Of L2 Slices Allocated} + \text{mem Pwr}) \times \text{Time} \right) \times \text{Time} 
\]

**6.2.2 Estimation of time**

*Time* component of Eq. 6.3 which is expressed in terms of the processor clock cycles, is proportional to the time spent in servicing L1 cache misses. It is lower if the number of cache misses are lower. It also depends on time spent in traversing NoC. Thus *Time* in Eq. (6.3) is,

\[
Time \propto C_{\text{remap}} \times 1/\text{Processor Frequency} 
\]

**6.2.3 Estimation of transit time**

Total time spent by thread *i* in transit is equal to the traffic generated by the thread *i* between L1 cache in tile *i* and all L2 slices. Formally,

\[
\tau_i = \sum_{0 \leq j < N} \eta_{ij} L_{ij} 
\]

Hence, the total time spent in transit by all the threads is,

\[
\tau = \sum_{0 \leq i < K} \tau_i = \sum_{0 \leq i < K} \sum_{0 \leq j < N} \eta_{ij} L_{ij} = \sum_{0 \leq j < N} \sum_{0 \leq i < K} \eta_{ij} L_{ij} 
\]

Since all threads execute concurrently, time spent in transit by these threads overlaps. The sum of their timings indicates the maximum time an application might spent in transit. The trace between L1 caches and L2 slices also contains traffic caused due to coherence.

**6.2.4 Estimation of cache miss penalty**

Execution time of an application is greatly affected by cache miss rate, especially in the last level cache (LLC) in the cache hierarchy. Hence, the remap configuration should be chosen such that the number of L2 misses should not increase significantly. The cache miss cost *c_j* of L2 slice *j* is proportional to penalty *p_j* incurred on a miss in L2 slice in tile *j* and the number of misses generated in L2 slice *j* by all threads.
Chapter 6. Offline Determination of Remap Table using Genetic Algorithms

Figure 6.2: plots power dissipated in memory (in mW) on the Y axis against the number of accesses made to the offchip DRAM on the X axis.

(a) mpegenc: \[ Y = 0.00136X + 633.34 \]
(b) mpegdec: \[ Y = 0.00765X + 513.436 \]
(c) cholesky: \[ Y = 0.00128X + 2017.55 \]
(d) fft: \[ Y = 0.0000745X + 1595.87 \]
(e) radix: \[ Y = 0.00209X + 935.73 \]
(f) water_spatial: \[ Y = 0.0375X + 372.24 \]
(g) water_nsquared: \[ Y = 0.00684X + 495.79 \]
(h) barnes: \[ Y = 0.0322X + 509.77 \]
(i) raytrace: \[ Y = 0.0009X + 904.39 \]
(j) ocean: \[ Y = 0.000292X + 3882.54 \]
(k) blackscholes: \[ Y = 0.000347X + 1428.33 \]
(l) swaption: \[ Y = 0.02345X + 390.312 \]
(m) fluidanimate: \[ Y = 0.00273X + 975.31 \]
(n) x264: \[ Y = 0.00066X + 562.0644 \]
Chapter 6. Offline Determination of Remap Table using Genetic Algorithms

\[ c_j = \sum_{0 \leq i < K} p_{ij} \alpha_{ij} \]  \hspace{1cm} (6.7)

Total cache miss penalty, \( C_{\text{Miss}} \) incurred by all L2 slices is,

\[ C_{\text{Miss}} = \sum_{0 \leq j < N} c_j = \sum_{0 \leq j < N} \sum_{0 \leq i < K} p_{ij} \alpha_{ij} \]  \hspace{1cm} (6.8)

Hence, the cost incurred by misses in all L1 caches is sum of costs given by Equation (6.6) and Equation (6.8).

\[ C_{\text{total}} = \sum_{0 \leq j < N} \sum_{0 \leq i < K} \eta_{ij} L_{ij} + \sum_{0 \leq j < N} \sum_{0 \leq i < K} p_{ij} \alpha_{ij} \]  \hspace{1cm} (6.9)

GA chooses the remap configuration such that \( C_{\text{total}} \) is minimized. We give more weightage to the cache miss penalty than the transit time penalty since latency of one miss in LLC is much higher than its transit time penalty. We empirically found that \( K_t = 1 \) and \( K_c = 8 \) remap switched-off L2 slices more uniformly. We use these values of \( K_c \) and \( K_t \) in all our experiments. Hence, for a given remap table, the cost is

\[ C_{\text{remap}} = K_t \sum_{0 \leq j < N} \sum_{0 \leq i < K} \eta_{ij} L_{i\delta(j)} + K_c \sum_{0 \leq j < S} \sum_{0 \leq i < K} \alpha_{ij} p_j \]  \hspace{1cm} (6.10)

To determine the number of L2 cache misses for a given remap table, we use a modified trace-driven Dinero cache simulator[166]. We simulate an application with all L2 slices and capture traffic generated between all L1 and L2 caches in a trace. The modified Dinero simulator requires the trace and a remap configuration as an input. It estimates the number of cache misses incurred by each L2 slice for a given remap configuration. We consider a CMP with non-inclusive shared L2 cache. For a non-inclusive cache, conflict misses in L2 do not cause eviction in L1 cache. Most of the accesses are L1 hits. Due to these reasons, this method gives reasonably accurate estimation of L2 cache misses. The same trace is used to determine the transit time cost.

6.2.5 Genetic Algorithm formulation(GA)

As explained earlier, genetic algorithms are a class of randomized search algorithms useful in function optimization. A possible solution of the problem is encoded as a fixed size binary array, called chromosome. We use a chromosome to represent one remap configuration. As shown in Fig. 6.3, the remap configuration is represented as 2 strings. 1\textsuperscript{st} binary string gives the L2 slices allocated to an application and 2\textsuperscript{nd} gives the mapping of unallocated slices e.g. a chromosome “(1,0,0,1)(0,3,0,3)” indicates that 0\textsuperscript{th} and 3\textsuperscript{rd} L2 slices are powered on. 1\textsuperscript{st} and 2\textsuperscript{nd} L2 slices are switched off and mapped on-to the 3\textsuperscript{rd} and 0\textsuperscript{th} slice, respectively. The details of our genetic algorithm approach are described below:

- The fitness value of a chromosome is given by \( EDP \) in Eq. 6.3.
The population size is 100 chromosomes

- randomly selected single point crossover between two chromosomes with probability of 0.65. Crossover is done only on the first binary string which represents the allocated L2 slices. Unallocated L2 slices are randomly mapped to the allocated L2 slices. Each remapped slice is mutated with another allocated L2 slice with probability of 0.015. We have chosen mutation and crossover probabilities within the standard range [138].

- Generally, in the case of genetic algorithms, higher fitness value of a chromosome indicates better suitability. But in our problem, lower EDP value indicates better remap configuration. Hence, to convert this minimization problem into a maximization problem, fitness value of all chromosomes is subtracted from the largest fitness value of the population. We use roulette wheel selection method.

- The search algorithm is terminated after 80 generations and the fittest solution is used for further evaluation. Our GA implementation stabilizes after 20-30 generations for all applications.

The flow chart in Fig. 6.4 shows the steps used to determine the NORC.

## 6.3 Results

Though we simulate 4B instructions to evaluate the dynamic remap table and variable cache associativity policies in rest of the thesis, for these experiments, we skip initial serial part and simulate only 1B instructions from the parallel section for all applications. This is due to time and storage limitations of the trace-based evaluation of the GA approach. Moreover, GA is used to find out effectiveness of
Table 6.4: Correlation between the number of cache misses estimated using modified trace based Dinero model and simulation

<table>
<thead>
<tr>
<th>Application</th>
<th>Correlation</th>
<th>Application</th>
<th>Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>0.96</td>
<td>radix</td>
<td>0.98</td>
</tr>
<tr>
<td>raytrace</td>
<td>0.99</td>
<td>cholesky</td>
<td>0.99</td>
</tr>
<tr>
<td>water_spatial</td>
<td>0.99</td>
<td>water_n squared</td>
<td>0.99</td>
</tr>
<tr>
<td>barnes</td>
<td>0.99</td>
<td>ocean</td>
<td>0.99</td>
</tr>
<tr>
<td>blackscholes</td>
<td>0.99</td>
<td>swaption</td>
<td>0.98</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>0.87</td>
<td>x264</td>
<td>0.97</td>
</tr>
<tr>
<td>mpegenc</td>
<td>0.99</td>
<td>mpegdec</td>
<td>0.99</td>
</tr>
</tbody>
</table>

the dynamic remap policy implementation. Hence, if EDP savings obtained using the dynamic remap policy are close to that obtained with the GA approach on simulating 1B instructions, then it can be deduced that EDP savings between two approaches will be similar on executing 4B instructions.

6.3.1 Accuracy of cache misses estimation method

As mentioned earlier, to estimate the number of cache misses incurred for a remap configuration, we have modified Dinero cache simulator. It takes a trace between L1 and L2 caches and a remap configuration as an input and estimates the number of cache misses incurred by each online L2 slice. To check accuracy of this method, we modified GA algorithm to obtain the remap configuration which allocates 2, 4, 6 ... L2 slices. The remap configurations obtained using this method are simulated. Table 6.4
Figure 6.5: The primary Y axis plots execution time obtained on simulating the near optimal remap configurations allocating 2, 4... number of L2 slices. The secondary Y axis plots execution time estimated using Eq. 6.10 for the same remap configurations. Execution time estimated using our trace based model shows a good correlation to the trend seen with simulation.
Chapter 6. Offline Determination of Remap Table using Genetic Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Normalized EDP</th>
<th>Normalized Exe</th>
<th>average L2 slice Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With partition</td>
<td>0.46</td>
<td>1.01</td>
<td>1.42</td>
</tr>
<tr>
<td>Without partition</td>
<td>0.39</td>
<td>0.97</td>
<td>1</td>
</tr>
<tr>
<td>raytrace</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With partition</td>
<td>0.53</td>
<td>0.99</td>
<td>2.91</td>
</tr>
<tr>
<td>Without partition</td>
<td>0.56</td>
<td>1.01</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 6.5: Near optimal remap configurations determined by dividing trace according to distinct WSS changes

gives correlation between the number of cache misses estimated by our method and that obtained using simulation. All the applications show very good correlation between two values (average of 97%). This is because, typically CMPs have shared last level caches which are noninclusive. Hence, eviction of a cache line from L2 cache does not cause eviction from L1(s). Due to over-allocation of cache, interleaving of cache accesses made by different threads does not show significant increase in cache misses. In case of applications with large WSS, this might cause some inaccuracies. Despite this, ocean shows 99% correlation. Similarly, even though cholesky has fairly large WSS, it also shows 99% correlation between the cache misses predicted using these methods.

6.3.2 Accuracy of execution time estimation method

We model total time spent in transit and in cache misses with Eq. 6.10. To evaluate accuracy of this method, we use the same approach as described in subsection 6.3.1. We modified GA algorithm to obtain the remap tables allocating varying number of L2 slices. We simulated these remap configurations and obtained execution time. We also obtained \( C_{\text{remap}} \) values using Eq. 6.10 for the same remap configurations. Fig. 6.5 plots execution time estimated using these methods. These plots should not be compared by their absolute values since the execution time depends on the memory level parallelism. Time obtained using Eq. 6.10 estimates the total time spent in transit and cache misses. Execution time is much lower than this due to memory level parallelism and out of order execution on a CMP. Hence, a trend in values (on varying the number of allocated L2 slices) is more important than their absolute values. Fig 6.5 clearly shows a good correlation between the two values. In case of fft, variation in execution time is very little on varying the number of L2 slices since it needs only one L2 slice. Hence, execution time does not vary significantly on allocating more L2 slices. Due to this reason, GA method shows poor correlation to the simulation method. Similar behaviour is seen to some extent in case of water_spatial as well. However, for rest of applications, two values are very well correlated. This shows that our method can replace cycle-accurate simulation, when estimating variation in execution time is more important than its absolute value.
6.3.3 Change in WSS

Fig. A.1 and Fig. A.2 in Appendix A show variation in WSS of an application. Some applications do not show distinct variation in WSS. It varies slowly during execution such as in swaption, ocean and cholesky or it remains constant throughout the execution such as in blackscholes. In these applications, we determine the NORC considering the entire trace. On the contrary, fft shows a sudden increase in its WSS in last few monitoring intervals. Hence, we divided its trace into two parts where it shows a sudden increase in WSS. We determined the NORC separately for both the parts and simulated using these two remap configurations. The second remap configuration allocates more number of L2 slices than the first one. We switch over to the second remap configuration on detecting a phase change in WSS. We also determine the NORC by considering the entire trace for fft. To our surprise, EDP savings obtained with the second approach are higher than the first. The details are given in Table 6.5. This is because, the remap configuration change incurs overhead of invalidating cache lines and relocating active lines to their new destination. As a result, execution time degrades by 4% compared to the former case. The second WSS phase lasts only for 10-20 monitoring periods. Changing the remap configuration for small WSS phase changes is not effective. Hence, for applications such as barnes and fluidanimate which show a sudden increase in WSS for a very short period (less than 2-3 monitoring periods), we use the entire trace to determine the NORC. Similarly, in case of x.264, though WSS increase remains steady after first 188 monitoring periods, still the number of L2 slices required does not vary. Hence, even for x.264, we determine the NORC by considering the entire trace. On the contrary, for raytrace, we divide the trace into two parts as it shows a prolonged variation in WSS. The remap configurations determined by partitioning the trace into two parts give 3% higher EDP savings than that obtained without partitioning, as shown in Table 6.5.

6.3.4 Accuracy of GA formulation

Like prior experiments, we modified GA formulation to obtain remap configurations allocating different number of L2 slices. The remap configurations obtained using this method are simulated and EDP is determined. We obtain the NORC using GA method which explores all remap configurations in a single execution. EDP is also determined by simulating this remap configuration. EDP of the NORC is minimum for most of the applications and is within average of 5% of minimum EDP obtained for all applications. Fig. 6.6 and Fig. 6.7 show execution time and EDP obtained with remap configurations allocating varying number of L2 slices. As shown in Fig. 6.6(a), fft needs only one L2 slice. Therefore, EDP savings decrease on allocating more than one L2 slice. The remap table determined with GA method also allocates only one L2 slice. Similarly in the case of cholesky (Fig. 6.6(d)), if only 8 L2 slices are allocated then EDP degrades by 22% and execution time by 17% over the reference SNUCA. It needs at least 10 L2 slices to obtain maximum EDP savings. However, execution time degrades by
5% on allocating 10 L2 slices. The NORC determined using GA allocates 11 L2 slices and maps rest of L2 slices such that execution time shows degradation of 3% only. With 14 L2 slices, degradation in execution time is minimal but EDP savings are only 4%, whereas, the NORC gives EDP savings of 5%.

It should be noted that our GA approach varies the number of powered on L2 slices and eventually finds a remap configuration which gives maximum EDP savings.

### 6.3.5 Simulation speedup

Diaz et al. [138] use cycle-accurate architectural simulation to evaluate fitness of a chromosome, which is very slow. Hence, we evaluate simulation speedup obtained using our method of evaluating fitness over the cycle-accurate simulation of the same remap configuration (Table 6.6). Applications such as waterա spatial and x.264 show significant speedup (up-to 218x) as most of the accesses are L1 hits. Hence, a trace between L1 misses and L2 is smaller. Memory intensive applications such as ocean shows 14x speedup which is also higher than the cycle accurate simulation. Hence, we believe our method can help in solving other cache optimization problems such as cache partitioning and cache design space exploration on a CMP.

### 6.4 Summary

Genetic algorithms have been used in various fields such as medical, biomedical, architecture exploration and chip design. Diaz et al. [138] used GA for the first time to solve cache optimization problem. However, they use architectural simulation to find fitness of a chromosome. This makes it very slow. On the contrary, we use a trace based model which shows a simulation speedup between 12X-218X over the cycle-accurate simulation. Dinero [166] is a well known model which estimates caches misses using a memory trace of a single threaded application executing on a uniprocessor. However, our results show that it can be used for simulating shared non-inclusive cache on a CMP with multithreaded workloads. On an average, cache misses estimated using our method show 97% correlation to results obtained using simulation. We also show accuracy of our method of estimating execution time by comparing it with

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedup</th>
<th>Application</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>12</td>
<td>radix</td>
<td>35</td>
</tr>
<tr>
<td>raytrace</td>
<td>17</td>
<td>cholesky</td>
<td>26</td>
</tr>
<tr>
<td>wateratial</td>
<td>157</td>
<td>water nsquared</td>
<td>37</td>
</tr>
<tr>
<td>barnes</td>
<td>23</td>
<td>ocean</td>
<td>14</td>
</tr>
<tr>
<td>blackscholes</td>
<td>29</td>
<td>swaption</td>
<td>62</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>218</td>
<td>x264</td>
<td>83</td>
</tr>
<tr>
<td>mpegencode</td>
<td>57</td>
<td>mpegdec</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 6.6: Simulation speedup obtained using our method over cycle-accurate architectural simulation.
Figure 6.6: Graphs show normalized execution time (on the primary Y axis) and EDP (on the secondary Y axis) obtained on simulating remap configurations allocating 2, 4... L2 slices. All values are normalized w.r.t. the reference execution on SNUCA. “GA(X)” is the near optimal remap configuration obtained with our GA method. This remap configuration allocates X L2 slices.
Figure 6.7: Graphs show normalized execution time (on the primary Y axis) and EDP (on the secondary Y axis) obtained on simulating remap configurations allocating 2, 4... L2 slices. All values are normalized w.r.t. the reference execution on SNUCA. “GA(X)” is the near optimal remap configuration obtained with our GA method. This remap configuration allocates X L2 slices.
results obtained using simulation. The near-optimal remap configuration determined using GA method achieves comparable EDP savings when compared to the exhaustive search method. This helps it scale better with increase in cores on a CMP. Our GA formulation can also be used in solving other cache related problems such as cache partitioning or design space exploration.
Chapter 7

Adaptive Remap Configuration

This chapter explains online implementation of the remap policy. We quantitatively compare EDP savings obtained using the adaptive remap policy with those obtained using the drowsy cache and adaptable associative cache implementations. We also compare it with EDP savings obtained by simulating the near optimal remap configuration.

7.1 Adaptive implementation of the Remap policy

7.1.1 Determination of metric

We use TWSS method to estimate cache requirement of an application. To determine L2 slices which should be powered-on and the mapping of powered-off L2 slices to powered-on L2 slices, we analyze the near optimal remap configurations (NORC) evaluated using GA approach. On analyzing NORC we reached following conclusions:

1. Applications with smaller cache requirement prefer L2 slices from tiles 5, 6, 9 and 10 over the L2 slices in other tiles. These slices are located near the center of the tiled CMP. We determine the average network latency offered by each L2 slice to all L1 caches, assuming L1 caches make uniform number of accesses. In that case, the L2 slices in tiles 5, 6, 9 and 10 offer least average access latency. This is application agnostic and only depends on network configuration of the chip.

2. In most of the applications, number of powered-off L2 slices mapped on-to the powered-on L2 slices is nearly same.

3. We evaluated the NORC for every distinct WSS phase and then switched to the corresponding remap configuration on detecting change in the WSS phase. We also evaluated a remap table considering the entire trace. However, the remap overhead caused due to changes in the remap
Algorithm 2 evaluates the remap configuration

1: Get the number of activeBits(A) and activeLineReplaced(R) of all enabled L2 slices.
2: \( EWSS = A + R \)
3: \( numL2Slices = EWSS/sizeOfL2Slice \)
4: \( replacementsPerSet = R/totalSets \)
5: if \( replacementsPerSet > 0 \) then
6: \( numL2Slices = numL2Slices + 1 \)
7: end if
8: allocateSlices\((numL2Slices, SAAL_{List})\)
9: sortUnallocatedSlices()
10: mapUnallocatedSlices()

configuration obtained by partitioning the trace gives slightly lower EDP gains than that obtained without partitioning. This means that overhead due to the remap configuration change should be minimum to achieve least degradation in execution time.

Hence, we first evaluate the average network latency offered by an L2 slice to L1 caches. We call this “static average access latency (SAAL)” as it depends only on the network configuration. L2 slices are maintained in increasing order of SAAL in a sorted list. We refer to this list as “SAAL_{List}”. The details of dynamic implementation of the remap policy are described below:

7.1.2 Working of the adaptive remap configuration

As explained in Section 5.5, the remap configuration is maintained in L1 cache controllers. On an L1 miss, the request is sent to the remapped L2 slice. The new remap configuration is evaluated by executing Algorithm 2 after every 4M clock cycles. The number of L2 slices to be allocated is equal to the estimated WSS (EWSS) divided by the size of L2 slice (Lines 1-3). We use TWSS method to estimate WSS of an application which is described in Chapter 4. The number of cache lines replaced increases with increase in the cache requirement of an application. This happens when an application exhibits a phase change. If the number of cache lines replaced per set is nonzero then the number of powered-on L2 slices is increased by one (Lines 4-7).

Similar to NORC, the dynamic remap implementation also gives higher preference to tiles offering lower SAAL while allocating L2 slices. The average and maximum SAAL offered by all L2 slices on a sixteen tiled CMP is shown in the Fig. 7.1. Please check chapter 3 for details of the tile link length and floor planning. Higher preference is given to L2 slices in tiles 5, 6, 9 and 10 which offer lowest SAAL of 30. Intuitively, they are located in the center of the chip, so they offer lower access latency to all the cores. Since \( SAAL_{List} \) is independent of an application, it is not computed in every monitoring period. This reduces the number of remap configuration changes performed in the consecutive monitoring periods, thereby reducing the total time spent in the remap configuration changes. Our experimental results show that the overhead incurred due to remap configuration change, in terms of clock cycles, is an
Figure 7.1: average and maximum static average access latency offered by each L2 slice to L1 caches

average of 1.8% of the execution time.

If the cache requirement is estimated as \( \text{numL2Slices} \) (number of L2 slices), then first \( \text{numL2Slices} \) from the \( \text{SAAL\_List} \) are maintained in the powered-on state (Line 8). Remaining L2 slices from \( \text{SAAL\_List} \) will be powered-off in the next monitoring period. These slices are sorted in decreasing order of total number of accesses serviced by them and mapped on-to the powered-on L2 slices (\( \text{numL2Slices} \)) so that the number of accesses serviced by them is nearly same. This ensures uniform distribution of cache accesses among the powered-on L2 slices (Lines 9, 10).

The remap configuration change protocol can be implemented as a separate hardware module without any intervention of the operating system. The L2 controller from tile “0” (as shown in Fig. 7.1) acts as a master. It receives WSS data from the remaining L2 controllers over the NoC. Cache lines are transferred to a new location when mapping of their L2 slice changes. To update the remap configuration in all L1 controllers, the master uses \textit{two-phase-commit} protocol. The new L1 misses are put on-hold. After currently issued L1 misses are serviced, L2 controllers initiate cache line transfers. The remap configuration is updated in L1 controllers after cache lines are transferred. Finally, the on-hold L1 misses are serviced. The active bit used to estimate WSS serves one more purpose. Cache lines with active bit reset are not considered to be part of the working set. Such lines are invalidated and modified lines are written back to the off-chip memory. This avoids overhead of unnecessary cache line transfer. In the destination L2 slice, if a cache line is not available to receive an incoming line, then some other cache line is replaced depending on the cache replacement policy. Our simulator models power and time overhead caused due to cache line transfers and periodic monitoring events initiated by the remap master.

7.1.3 Directory assumption

Our base SNUCA architecture implements noninclusive cache. The directory information of a cache line is saved along with its tag, which includes a full bitmap of L1 sharers. For cache lines only cached in L1(s) and not in L2, a separate per tile directory table is maintained to save L1 sharer(s) information. For a sixteen tiled architecture, L2 slice of size 512KB, 64B cache line size and physical address of 48 bits, the number of bits required for tags and index bits is 42. The full bitmap requires 16 bits to store sharers’ information. We estimated from our remap implementation that a directory entry requires 6
(a) Smaller EDP value is better

(b) Smaller execution time is better

Figure 7.2: Comparison with way adaptable caches

bits to save its state information. Hence, the size of each entry of a directory table maintained for non-inclusive cache lines is 64 (16+42+6) bits. We assume size of the directory table is 5% of L2 slice. As a result, the directory table contains about 3276 entries. In the remap policy, when an inclusive cache line is transferred, its directory information is also transferred along with its tag bits to a new location. For a non-inclusive cache line, its entry in the directory table is transferred. If in the recipient L2 slice, the number of directory table entries exceeds 3000 (which is just 3.4% of size of the L2 slice), then some other entry with least number of L1 sharers is invalidated. We present results with infinite and finite sized directory tables later in this chapter. In summary, the remap policy can be applied to inclusive, non-inclusive caches or caches with centralized directories.
7.1.4 Area overhead and scalability

L2 tag size increases as it contains cache lines from the remapped L2 slice besides its own. Hence the slice/tile identifier bits need to be included in the L2 tag. This does not add any additional access latency but size of the tag array increases by $\log_2(\#\text{tiles}) \times \#\text{cachelines}$. For example, in a CMP with 16 tiles, a physical address of 48 bits, per tile L2 slice of 512KB, with 64B cache line and an associativity of 16, the tag is 29 bits if remapping is not used. With the remap policy, tag increases to 33 bits. Additional space requirement for 512KB L2 slice is $8192^1 \times (4+1) = 40$K bits or 5KB, which is just 0.9% increase in the storage requirement. An additional bit is required to track active lines in a monitoring period. Similarly, extra power required to perform the remap table lookup and additional tag comparison is minimal.

The number of tile identifier bits increases only by one when the number of tiles is doubled. In the above example, storage overhead increases from 0.9% to 1.17% and 1.3%, if the number of tiles is increased from 16 to 32 and 64, respectively. For a very large number of tiles (128 onwards), we believe, L2 cache will be shared and distributed among a subset of tiles and cache coherence will be maintained between these private L2 caches to reduce L2 cache access latency. In that case, the remap policy can still be applied within private L2 slices without large overhead.

7.2 Results

7.2.1 Comparison with adaptable way policy

We quantitatively compare the remap policy with the reference SNUCA, DNUCA, adaptable way implementation of SNUCA (TWSS.S) and DNUCA (TWSS.D). Please refer to Chapter 5 for more details of TWSS.S and TWSS.D.

Fig. 7.2(a) and Fig. 7.2(b) compare EDP and execution time obtained with the above mentioned power optimization policies. All values are normalized w.r.t. that obtained with the reference SNUCA. It should be noted that for a fair comparison, WSS is estimated using TWSS method in the remap, TWSS.S and TWSS.D. In fft, the remap policy gives 7% higher EDP savings than TWSS.S. This is mainly because of 4% savings in execution time of fft with the remap policy. The remap policy shows 6% savings in execution time of x.264. x.264 has poor thread scalability as it uses pipeline parallelism. Hence, even if it is executed with 16 threads, it spawns only up-to six concurrent threads. Its execution time improves with the remap policy as it allocates nearer L2 slices, unlike the variable way policy, achieving 6% higher EDP savings. Similar behaviour is seen in mpegdec. mpegdec uses “fork and join” type of parallelism. Hence, the remap policy shows 5% improvement in execution time over the reference.

\(^1\text{number of cache lines}\)
For all applications, the remap allocates slightly lesser cache compared to TWSS.S without causing significant degradation in their execution time (on an average of 1.2%). The maximum degradation in execution time for the remap policy is 8% in radix. This application has a very small cache requirement in the beginning and shows a sudden increase later during the execution. This degrades with execution time with the remap and variable way policies. The remap achieves an average of 6% higher EDP savings than that obtained with TWSS.S.

An application that shares more data between its threads in an interleaved manner, shows significant degradation in its execution time with DNUCA. For example, raytrace shows 25% degradation in its execution time with DNUCA over SNUCA, which causes 50% degradation in EDP. Similar behaviour is seen for fft, barnes and mpegenc. This is mainly due to conflicting migrations caused by concurrently executing threads. On the other hand, execution time improves with DNUCA in case of blackscholes and ocean. Hence, TWSS.D shows higher EDP savings than that obtained with the remap policy in blackscholes and ocean. The remap achieves an average of 9.5% higher EDP savings than that obtained with TWSS.D.

### 7.2.2 Equal cache comparison

In these experiments, we determined the average number of L2 slices allocated by the dynamic remap policy for each application. If the remap allocates 16 way associative “x” number of L2 slices, each with “y” number of sets for an application, then we simulate the same application with 16 L2 slices of “x” cache associativity and “y” number of sets. Please note that in both the scenarios, cache utilization for an application is same.

We performed above experiments with both SNUCA and DNUCA cache access policies. We refer to these configurations as “SNUCA.Assoc” and DNUCA.Assoc, respectively. The term SNUCA.Assoc indicates that SNUCA cache has associativity equal to the average number of L2 slices allocated by the dynamic remap policy for an application. DNUCA.Assoc also has a similar meaning. The aim of these experiments is to demonstrate higher degradation in execution time or EDP of an application for caches with lesser associativity when compared to the remap policy, even if all three configurations use equal amount of cache on an average. However, it should be noted that, the dynamic remap policy incurs additional overhead of adapting remap configuration according to cache requirement of an application. Whereas, SNUCA.Assoc and DNUCA.Assoc use the same cache configuration throughout the execution of an application.

Fig. 7.3(b) shows execution time normalized with respect to the base SNUCA configuration. This graph clearly shows that execution time degrades more with SNUCA.Assoc and DNUCA.Assoc policies than with the remap policy. The significant degradation in case of fft, raytrace, mpegenc and mpegdec with DNUCA.Assoc configuration, is mainly due to DNUCA cache access policy. The remap achieves
Chapter 7. Adaptive Remap Configuration

98

(a) Smaller EDP value is better

(b) Smaller execution time is better

Figure 7.3: Comparison with caches using SNUCA and DNUCA cache access policy and having cache associativity equal to the average L2 slices allocated by the remap policy

higher EDP savings than that obtained with SNUCA.Assoc and DNUCA.Assoc, as shown in Fig. 7.3(a) despite remap configuration change overhead.

7.2.3 Comparison with drowsy cache

We quantitatively compare the remap policy with the drowsy cache technique. As explained in Chapter 5, we have performed our experiments with ratio of leakage power dissipated in the non-drowsy mode to that in the drowsy mode as 4 and 2. We refer to these implementations as Drowsy.4 and Drowsy.2, respectively.

As shown in Fig. 7.4(b), the drowsy cache technique causes very negligible degradation in execution time which is in line with [13]. The remap policy achieves higher EDP savings in the case of fft, x.264, blackscholes and swaption etc. than the drowsy cache. This is due to lower cache requirement of these applications than the cache available. Hence the remap obtains higher EDP savings by switching off L2 slices. On the contrary, in the case of applications with large WSS, such as cholesky and ocean, the
Chapter 7. Adaptive Remap Configuration

(a) Smaller EDP value is better

(b) Smaller execution time is better

Figure 7.4: Comparison with the drowsy cache
Chapter 7. Adaptive Remap Configuration

7.2.4 Number of memory accesses

Since the remap allocates lower cache than the TWSS.S and TWSS.D, it makes more number of memory accesses. Fig. 7.5(a) (a) shows the number of memory accesses made by TWSS.S, TWSS.D and the remap. These are normalized w.r.t. the total accesses made by SNUCA. The remap exhibits more number of memory accesses in the case of mpegenc, radix, and barnes. However, power consumption in offchip DRAM does not increase proportionally. Please check Fig. 9.1 for the detailed graph on power consumption. This is because DRAM operates at lower frequency than the cache. However, the increased memory accesses might degrade execution time of an application. Execution time degrades
Chapter 7. Adaptive Remap Configuration

101

(a) % Remap overhead w.r.t execution time

(b) Total no. of lines transferred

Figure 7.6: Remap Overhead

by 7% in radix and by 2% in mpegenc and barnes. Since the remap allocates an average of 4.5 L2 slices out of 16 in radix, it obtains 33% EDP gains.

It should be noted that in the case of applications with large WSS, such as ocean and cholesky, the remap does not show increase in the memory accesses. It estimates its WSS accurately and allocates almost all the 16 slices. Thus execution time and EDP do not degrade for these applications. Fig. 7.5(a) (b) shows average memory access latency offered by all implementations. As execution time improves for x.264, fft and mpegdec, memory latency obtained with the remap policy is lower than other implementations.

7.2.5 Overhead in the remap policy

To change the remap configuration, cache lines are transferred to their new location, during which L1 requests are put on hold. We measure the time spent in the remap configuration change. Fig. 7.6(a) plots time spent in the line transfer, normalized w.r.t execution time of an application on SNUCA. The remap incurs an average overhead of 1.6%, in terms of the number of clock cycles required to transfer the cache lines. Fig. 7.6(b) shows the total number cache lines transferred by the remap. Cholesky
Chapter 7. Adaptive Remap Configuration

7.2.6 Directory table invalidations

For inclusive caches, directory information (a bitmap for L1 sharers), is included in the tag for each L2 cache line. In the remap, directory information is transferred along with cache line to its new location. Hence, no changes are required for inclusive caches/lines. In case of non-inclusive caches, a separate per-tile directory table is required for cache lines which are present only in L1 cache and not in L2 cache, for all cache policies including our base SNUCA architecture.

Fig. 7.7(a) shows the total number of directory table entries created by all policies. As shown in Fig. 7.7(a), all cache policies create nearly equal number of entries. However, in case of the remap, data from different L2 slices is merged in a single slice. Hence it may create more entries in a directory table
than the available entries. In that case, some other directory entry with least number of L1 sharers is replaced. To replace a directory entry, data is invalidated from all its L1 sharers. As explained in section 7.1.3, we assume size of the directory table as 5% of L2 slice size. With this assumption, the directory contains 3276 entries. However, to be on the conservative side, if the number of entries exceed 3000, then some other entry is replaced. Fig. 7.7(b) shows the total number of directory entries invalidated by the remap. As we accurately predict and conservatively allocate required number of L2 slices, total invalidated directory entries is negligible. As shown in Fig. 7.7(c), EDP savings obtained with finite and infinite sized directory table do not vary much. As mentioned earlier, the remap can be applied to inclusive or noninclusive caches with centralized or distributed directories.

7.2.7 Sensitivity to L2 slice size

We evaluate EDP savings obtained with 256KB L2 slice. We use same network latencies as used for 512KB L2 slice. Fig. C.14 (in Appendix C) plots EDP of the remap policy normalized with respect to that of the reference SNUCA. For smaller L2 slice of 256KB, more number of L2 slices are allocated. Hence, EDP savings are lesser than that obtained with L2 slice of 512KB for majority of applications. EDP savings will be higher with increased L2 slice. It can be seen from Fig. C.14(b) that execution time degradation is minimal even with smaller L2 slice. This also demonstrates that TWSS estimates cache requirement accurately for smaller L2 slice and accordingly allocates L2 slices.

7.2.8 Sensitivity to monitoring period

We use 4M clock cycles as the monitoring period in all our experiments. As mentioned in chapter 4, all applications that we study, have majority of accesses with reuse time less than 4M cycles. It implies that if the address is not used in the last 4M clock cycles then it will not be used in the next 4M clock cycles as well. Hence, we determine WSS of an application after every 4M clock cycles. To test sensitivity of the remap w.r.t. the monitoring period, we have also taken readings for 2M clock cycles as the monitoring period. The monitoring period greater than 4M clock cycles will result in lesser EDP savings. This is similar to EDP savings obtained with adaptable associative SNUCA/DNUCA cache policies. (Please check Section 5.3.9.) Fig. 7.8(a) shows normalized EDP when the monitoring period used is 2M and 4M clock cycles. On varying monitoring period, EDP savings show negligible variation. Fig. 7.8(b) shows normalized execution time on changing the monitoring period. cholesky shows significant degradation in execution time for monitoring period of 2M clock cycles due to its large WSS. However, EDP does not degrade with the monitoring period of 4M clock cycles.
Chapter 7. Adaptive Remap Configuration

104

(a) Smaller EDP value is better

(b) Smaller execution time is better

Figure 7.8: EDP gains obtained using the remap policy show negligible variation on varying monitoring period

7.2.9 Comparison with the near-optimal remap configuration

We determine the NORC using genetic algorithms as described in Chapter 6. As mentioned before, we skip initial serial portion and simulate 1B instructions from the parallel section for all applications. Though we simulate 4B instructions in all our experiments, to check efficiency of the dynamic remap policy, we have simulated only 1B instructions. This is due to time and space limitations of a trace based GA approach.

Fig. 7.9(a) shows the average number of L2 slices allocated by the dynamic remap policy and NORC. Since the dynamic remap policy allocates minimum 2 L2 slices, the average number of L2 slices it allocates is slightly greater than that allocated by the NORC.

Fig. 7.9(b) shows normalized EDP of an application with the dynamic remap policy and for simulated NORC. The EDP savings obtained with the NORC are marginally higher than that obtained with the dynamic remap policy, as it does not incur remap configuration change overhead. Hence, slightly lower EDP savings are justified. In case of ocean, the dynamic remap allocates an average of 14.5 L2 slices
and achieves 4% EDP savings. However, GA approach allocates all sixteen L2 slices. For remaining applications, EDP savings obtained with the dynamic remap policy are within an average of 5% of that obtained with NORC. This shows that the metric used for the dynamic remap policy to determine remap configuration gives near optimal EDP savings.

7.2.10 Evaluating with Multiprogramming SPEC 2006 benchmarks

All results presented in this chapter till now use multi-threaded applications. We also demonstrate applicability of the remap policy to multiprogramming workloads using SPEC 2006 benchmarks. We schedule four programs on a quad-core CMP sharing 4MB L2 cache. The L2 cache is made up of sixteen 256KB L2 slices. The configuration is shown in Fig. 7.10. We assume link latency of 1 cycle for all links connecting L2 slices.

Fig. C.15(a) and Fig. C.15(b) (shown in Appendix C) show EDP and execution time normalized w.r.t SNUCA simulated on the same architecture. The remap gives higher EDP savings than that obtained with the adaptable way policy (TWSS.S) for all applications. TWSS.S is implemented as explained in Chapter 5. The remap achieves an average of 10% higher EDP savings than TWSS.S. Both these policies show negligible degradation in execution time for most of the applications. TWSS.S and the remap policies show an average of 3.5% and 2.5% degradation in the execution time. When compared to the drowsy cache technique, the remap policy achieves higher EDP savings in all applications except in libq. This application shows higher active line replacements per set (Fig. B.7(c)). Hence, the remap policy allocates more number of L2 slices. It achieves an average of 13% higher EDP savings than the drowsy cache implementation.

7.3 Summary

To reduce leakage power consumption in on-chip caches, we propose to switch off farther over-allocated L2 slices and map those slices on-to nearer L2 slices using the remap policy. Apart from reducing leakage power consumption, it also achieves up-to 6% improvement in execution time. On an average, the remap policy achieves 40% and 45% higher EDP savings than SNUCA and DNUCA. Unlike the drowsy cache technique, it is not susceptible to the transient errors. When compared to the drowsy cache technique, it achieves 21% and 26% higher EDP savings than that obtained with the drowsy cache technique implemented for SNUCA and DNUCA policies, respectively. As associativity of L2 slices is kept constant, it achieves higher EDP gains with lesser degradation in execution time when compared to the adaptable way policies.

EDP can also be minimized by improving execution time, which depends on the static and dynamic NUCA access policies. The suitability of cache access policy depends on data sharing properties of a
multi-threaded application. Hence, in the next chapter, we explore characteristics of an application that help find a suitable cache access policy.
Figure 7.9: The dynamic remap configuration gives EDP savings comparable to that obtained with the near optimal remap configurations determined using GA.
Figure 7.10: Configuration used to simulate multiprogramming SPEC 2006 workload (L2 slice size: 256KB)
Chapter 8

TCP: Thread Contention Predictor for Parallel Programs

With proliferation of CMPs on desktops and embedded platforms, multi-threaded programs have become ubiquitous. Existence of multiple threads may cause resource contention, such as in on-chip shared cache and interconnects, depending upon how they access resources. Hence, in this chapter, we propose a tool - Thread Contention Predictor (TCP) to help quantify the number of threads sharing data and their sharing pattern. We demonstrate its use to predict a more profitable cache access policy among SNUCA and DNUCA for a CMP.

8.1 Introduction

Ever increasing demand for performance has caused proliferation of CMPs on desktop and embedded platforms. As a result, multi-threaded programs have become ubiquitous. Writing a multi-threaded program is a very complicated task due to use of synchronization variables. A lot of tools are available to debug data race conditions in multi-threaded programs [167]. But not many tools are available to identify hot data structures in a program. Such tool will help programmers identify bottlenecks in their programs. Profiling tools such as “gprof” [168] are limited to instruction profiling. Here, we propose a method to profile data accesses for a program. This will help programmers to identify hot data objects in the program. Hence, we call this model “Thread Contention Predictor (TCP)”. TCP quantifies the number of threads sharing an object and pattern in which data is shared amongst these threads. We believe, our abstraction of data sharing properties of a program has wide applications such as determining hot data structures and predicting false sharing in a data structure. It can assist in reducing aborts in a transactional memory. This can be achieved by serializing transactions which access data that is highly
popular among threads. It can also assist in data mapping decisions for a CMP with large distributed shared cache. In this work, we implement a model to predict more profitable cache access policy between SNUCA and DNUCA, as the last level shared distributed cache on CMP. Researchers have proposed various trace-based[166], analytical[169] or statistical[170, 83] models to predict cache miss rate in a program. However, according to our knowledge, this is the first attempt to predict a suitable cache access policy between SNUCA and DNUCA for a CMP. Following are our major contributions in this chapter:

1. Introducing a metric to measure the number of threads sharing an object. We call it a “sharing index (SI)”. We also introduce a metric called “contention index (CI)” to measure contention caused by threads accessing that object.

2. Use of TCP to predict a more profitable cache access policy between SNUCA and DNUCA, as the last level shared cache (LLC). Our cache policy predictor is 2.2 times faster compared to cycle-accurate simulations for SNUCA and DNUCA put together.

8.2 TCP

TCP can evaluate SI and CI at the granularity of a cache line or individual data addresses. By obtaining additional information from the binary file for an application and trapping malloc/free calls, it can be used at the data structure granularity as well. Hence, we use the term object to denote this flexibility.

8.2.1 Sharing Index (SI)

TCP keeps track of the number of times each thread accesses an object. For example, $P_1, P_2, \ldots, P_T$ are probabilities of $T$ threads accessing a single object such that $\sum_{i=1}^{T} P_i = 1$. We define the sharing index, $SI$ of an object by Eq. (8.2). This definition is inspired from entropy as used in information theory [171].

$$H(P) = - \sum_{1\leq i \leq T} P_i \log(P_i)$$

$$SI = 2^{H(P)}$$

As per this definition, $1 \leq SI \leq T$. The case $SI = 1$ arises when only one of the $P_i = 1$ and rest of all probabilities are 0. This happens when objects such as local variables declared on the stack, for whom $P_1 = 1$ and $\log(P_i) = 0$, for the accessing thread. For rest of the threads, $P_i = 0$. Hence, $SI = 1$ tallies
Chapter 8. TCP: Thread Contention Predictor for Parallel Programs

Figure 8.1: Threads $t_1$ and $t_2$ make four accesses each. However, in (a), four accesses made by these threads are not interleaved. Whereas, in (b), accesses made by these threads are interleaved.

with the fact that only one thread accesses that object. On the contrary, for an object where all threads make equal number of accesses, $P_i = 1/T$ for all threads, where $T$ is the total number of threads present in an application. In this case, $SI = T$. Hence, Eq. (8.2) captures the notion of number of threads accessing an object.

8.2.2 Contention Index (CI)

Objects with higher SI, may not cause performance bottleneck if accesses made by all threads are serialized and not interleaved. Hence, there is a need to quantify interleaving of accesses done by different threads. This is done by CI. We define a runlength as the number of consecutive accesses made by the same thread to an object. Runlength statistics are collected for each object. The weighted average of runlength and its dispersion is defined as contention index. The average runlength is smaller if accesses done by different threads are more interleaved. Fig. 8.1 shows serialized and interleaved accesses made by two threads, executing on separate cores. In Fig. 8.1(a), both the threads make four consecutive accesses. So there are two runlengths of size four. Whereas, in Fig. 8.1(b), due to interleaved accesses, there is one runlength of size 1 and 3 each and two runlengths of size 2. Access to objects with smaller value of CI (average runlength) can cause higher traffic in shared components such as cache or interconnect.

While evaluating CI, we do not consider the number of clock cycles between the accesses. This is because, typically programs have good temporal locality. Hence, if average runlength is smaller, then accesses made by different threads occur within a short span of time and mostly would cause contention by increasing cache coherence messages. We also treat reads and writes equally in our use-case. The contention index definition can be adapted appropriately with respect to the number of clock cycles between the accesses and read/write distinction depending on the use-case.
8.2.3 Popularity Index (PI)

As explained above, SI gives the number of threads sharing an object, whereas, CI indicates the sharing pattern. On improving CI of an object, program execution may improve. However, improvement depends on the actual number of times an object is accessed. Hence, we define another term, called *Popularity Index (PI)* as follows:

\[
P_I = \frac{N \times SI}{CI}
\]

(8.3)

where, \(N\) is the total number of accesses made by all threads, SI and CI are sharing and contention indices of that object. An object is *popular* in a program execution if it is accessed significant number of times, popular among many threads and has more contention causing potential.

For every object, TCP also tracks instruction address when an object is accessed. Objects can be mapped back to data structures using information found in the application binary, instruction addresses and by intercepting malloc/free calls. Depending on PI, hot data structures can be determined. This can help programmers to re-organize data structures so that their SI and mainly CI improves.

8.3 Cache Policy Predictor (CPP)

In this section, we demonstrate the application of sharing and contention indices to estimate overhead incurred on using SNUCA and DNUCA cache access policies. Here, we repeat details of SNUCA and DNUCA cache access policies from Section 1.2 for convenience.

In SNUCA, the predetermined bits of memory address determine the bank in which data is cached (Fig. 8.2(a)). It is referred to as “home location” of that address. Whereas, in DNUCA, the whole address space is mapped onto a single column and predetermined bits decide the row in which data is cached (Fig. 8.2(b)). Data can be cached in any of the banks in a row, which form a “bankset”. On an L1 miss, L1 first checks data in the nearest L2 bank and then rest of the L2 banks in that bankset are searched. The data is read in the nearest L2 bank from memory if it is not present in any of these
banks.

In DNUCA, private data is cached in the nearest bank, offering lesser latency. On the contrary, private data could be in a farther bank in SNUCA, incurring higher latency than that in DNUCA. In DNUCA, consecutive accesses cause data to migrate in nearer bank at run-time. However, data shared by many threads, might migrate in conflicting directions, incurring higher latency. Since the cache set spans across multiple banks in a row, traditional replacement logic cannot be applied in DNUCA. We assume data is sent offchip on replacement in an L2 slice.

To summarize, though DNUCA offers lower access latency, it suffers from a drawback of complex lookup and replacement logic. On the contrary, SNUCA has simple lookup logic but may suffer from higher cache access latency. Hence, at design time, architects have to make a careful choice between SNUCA and DNUCA policies. Performing cycle-accurate simulation of many workloads is time consuming. Therefore, we solve this problem by using data collected by TCP with a one-time cycle-accurate simulation on SNUCA platform. Here, SI, CI are evaluated per cache line address (CLA). As this tool predicts a more suitable cache access policy, we call it “Cache Policy Predictor (CPP)”.

8.3.1 Estimation of overhead in DNUCA and SNUCA

While determining penalties incurred by DNUCA and SNUCA, we assume that the data is already present in on-chip cache. This assumption is valid since the number of DRAM accesses for an application depends on its working set size and is independent of cache access policy. The meaning of various terms used in CPP model is explained in Table 8.1.

In DNUCA, data might migrate in conflicting directions if it is shared by many threads at the same time. Hence, interleaving of accesses made by different threads determines cache access latency. Consecutive accesses made by the same thread, as in Fig. 8.1(a), cause data to migrate gradually in nearer L2 slice, offering lower access latency. However, in SNUCA all threads make a single access to the home location of an address, though it is farther than their nearest L2 slice. To determine penalties incurred by an application with these two policies, for every CLA, we track the total number of accesses made by each thread and maintain runlength statistics per thread, using a cycle-accurate simulation with SNUCA. As an example, in Fig. 8.1(a), threads $t_1$ and $t_2$ execute on cores 0 and 1, respectively. There are four L2 slices in a bankset (see Fig. 8.2(b)). Suppose, distance between $t_1$ and four L2 slices is 1, 2, 3 and 4. We make similar assumption for $t_2$. Suppose, distance of home L2 slice from $t_1$ and $t_2$ is 3 ($D_{1\text{Home}} = 3$) and 4 ($D_{2\text{Home}} = 4$), respectively. Assume that data is already present in home L2 location. As each thread makes four accesses to home L2 slice in Fig. 8.1(a), SNUCA cost is 28 ($4 \times 3 + 4 \times 4$).

Let us assume a worst case scenario in DNUCA. At the beginning of every runlength, data is not present in the nearest L2 slice for all threads. This is because, as previous access is made by some
other thread, data might be present in the nearest tile of that thread. Hence, due to this assumption, all threads search data in all peer L2 slices in a bankset during the first access of every runlength. If $DNUCA_{cost}$, calculated using this worst-case assumption is less than $SNUCA_{cost}$, then DNUCA will definitely give better performance for that application.

As per our earlier assumption, distance from peer L2 slices in a bankset where data can be cached is 1, 2, 3 and 4. In Fig. 8.1(a), for both the threads, search cost incurred during the first access in the runlength, is 10 ($(1+2+3+4)*1$) each. To determine distance between core and L2 slice where data will be found for rest of the accesses in a runlength, we evaluate SI and CI of a CLA. From Eq. (8.2), SI is 2 in this case as both threads make equal number of accesses to the CLA. CI (weighted average runlength) is four. For rest of the accesses in a runlength, we assume that data is present in the nearest L2 slice, if either SI is 1, i.e. data is shared by only single thread or CI is greater than 2 (as there are 4 L2 slices in a row). In this case, distance between thread and its nearest L2 slice is 1. Hence, the cost for remaining three accesses of each thread is $(3*1=3)$ and total $DNUCA_{cost}=2*10+2*3=26$, which is less than that of SNUCA (28). Hence, in case of Fig. 8.1(a) we conclude that DNUCA is preferable over SNUCA.

In Fig. 8.1(b), there is one runlength of size 1 and 3 each, and 2 runlengths of size 2. In this case, $SNUCA_{cost}$ remains constant as the total number of accesses made by both the threads is same as in Fig. 8.1(a). However, for DNUCA, total cost required to search data for the first access of every runlength in peer L2s is $40 ((1+2+3+4)*4)$. As CI is 2 in this case, we consider average distance for rest

Table 8.1: gives the meaning of various terms used in the CPP model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i$</td>
<td>thread executing on core $i$</td>
</tr>
<tr>
<td>$T$</td>
<td>total # of threads present in an application</td>
</tr>
<tr>
<td>$A_{ij}$</td>
<td>total # of accesses made by thread $i$ to cache line address (CLA) $j$</td>
</tr>
<tr>
<td>$N$</td>
<td>total # of CLAs</td>
</tr>
<tr>
<td>$K$</td>
<td>runlengths of size 0, 1, $\ldots$, $K$ tracked during one-time simulation on SNUCA. Runlengths of size equal to and greater than $K$ are counted by $(K-1)^{th}$ array entry.</td>
</tr>
<tr>
<td>$r_{ijk}$</td>
<td># of times thread $i$ exhibits runlength of size $k$ for an address $j$</td>
</tr>
<tr>
<td>$D_{ip}$</td>
<td>distance between L1 in tile $i$ and L2 slice in tile $p$ where data can be cached in DNUCA</td>
</tr>
<tr>
<td>$P$</td>
<td>total # of peer L2 slices in a bankset in DNUCA</td>
</tr>
<tr>
<td>$D_{i,j\text{-Nearest}}$</td>
<td>distance between L1 in tile $i$ and its nearest L2 slice where address $j$ can be cached in DNUCA</td>
</tr>
<tr>
<td>$D_{i,j\text{-Average}}$</td>
<td>average distance between L1 in tile $i$ and all L2 slices in a bankset where address $j$ can be cached in DNUCA</td>
</tr>
<tr>
<td>$D_{i,j\text{-Home}}$</td>
<td>distance between L1 in tile $i$ and the &quot;home&quot; L2 slice of address $j$, where data is cached in SNUCA</td>
</tr>
<tr>
<td>$SNUCA_{cost}$</td>
<td>Time spent in transit for SNUCA</td>
</tr>
<tr>
<td>$DNUCA_{cost}$</td>
<td>Time spent in transit for DNUCA</td>
</tr>
</tbody>
</table>
Chapter 8. TCP: Thread Contention Predictor for Parallel Programs

Algorithm 3 Cache Policy Predictor

1: Evaluate runlength and total number of accesses made by each thread to all CLAs using a cycle accurate simulator with SNUCA
2: Evaluate SNUCA cost using Eq. (8.4)
3: for first access of all runlengths of thread-CLA pair do
4: evaluate peer search cost using Eq. (8.5)
5: end for
6: for rest of accesses in runlengths of thread-CLA pair do
7: Evaluate SI and CI per column for each CLA
8: if \((SI == 1)\)\(\|\)\((CI \geq CI_{\text{threshold}})\) then
9: Estimate cost using Eq. (8.6)
10: else
11: Estimate cost using Eq. (8.7)
12: end if
13: end for
14: Obtain total \(DNUCA_{\text{cost}}\) using Eq. (8.8)
15: \(CostRatio = DNUCA_{\text{cost}} / SNUCA_{\text{cost}}\)
16: if \(CostRatio < 1\) then
17: DNUCA is winner
18: else
19: SNUCA is winner
20: end if

of the accesses in all runlengths. This is because consecutive accesses gradually migrate data towards the nearest L2 slice. As the average distance is 2.5 \(((1+2+3+4)/4)\), cost required for rest of the accesses in runlengths is 10 \((2.5 \times (2+1+1))\). Hence, \(DNUCA_{\text{cost}}\) is 50 \((40+10)\). Therefore, for Fig. 8.1(b) we conclude that SNUCA will perform better than DNUCA. It should be noted that when average runlength is small, cost incurred for first accesses in runlengths forms a major part of the DNUCA cost.

In this application of TCP, we evaluate SI and CI by aggregating statistics for all threads executing on the cores which belong to the same column. Hence, SI and CI are evaluated on a per column per object basis. This is because, if two threads execute on different cores which belong to the same column (see Fig. 8.2(b)), then these cores have the same nearest L2 slice, which is the L2 slice present in that column. CPP procedure explained above is described in Algorithm 3.

We use cycle-accurate simulator to obtain runlength and total accesses made by each thread. We consider addresses \textit{missed} in L1 alone to evaluate SNUCA and DNUCA costs, as these accesses decide overhead incurred to access the last level SNUCA or DNUCA cache.

Total time spent in transit in SNUCA by thread \(i\) while accessing an address \(j\) is estimated using Eq. 8.4.

\[
SNUCA_{\text{cost}} = \sum_{0 \leq i < T} \sum_{0 \leq j < N} A_{ij} \cdot D_{ij,Home} \tag{8.4}
\]

For DNUCA, as explained above, we evaluate costs separately for the first access and remaining accesses in every runlength. Eq. (8.5) estimates cost for a thread in tile \(i\), accessing address \(j\), when all peer
L2 slices have to be searched (according to our assumption, this happens for the first access of every runlength of all threads).

\[ PeerSearchCt_{ij} = \sum_{0 \leq p < P} \sum_{0 \leq k < K} r_{ijk} \times D_{ip} \] (8.5)

For rest of the references made by each thread, we determine SI and CI by aggregating statistics of all threads belonging to the same column. Depending on values of SI and CI, there are two cases as listed in if-else conditions on lines 9-12 in Algorithm 3. If SI is 1 (which is true when all threads accessing that CLA belong to the same column), then we use distance between the thread and its nearest L2 slice \(D_{i,j\text{Nearest}}\). We also use \(D_{i,j\text{Nearest}}\), if CI is greater than or equal to 3 (\(CIThreshold = 3\) in Algorithm 3). This is because, in our experimental setup (Fig. 8.2(b)), there are 4 L2 slices in a bankset. If the core makes an average 3 or more consecutive accesses to CLA then it will find data in its nearest L2 slice. In summary, if accesses are private or done through a single column, then runlengths are longer in size and threads in that column will find data in their nearest L2 slice. Eq. (8.6) estimates cost for the remaining accesses made by thread \(i\) to an address \(j\), with lesser contention causing potential.

\[ NearSearchCt_{ij} = (A_{ij} - \sum_{0 \leq k < K} r_{ijk}) \times D_{i,j\text{Nearest}} \] (8.6)

However, if SI is greater than 1 or CI is less than 3, then most of the threads will have to search data in all L2 slices for the remaining accesses in a runlength. Hence, we use average distance of all L2 slices in a bankset \(D_{i,j\text{Average}}\). If average runlength is less than 3, then \((A_{ij} - \sum_{0 \leq k < K} r_{ijk})\) in Eq. (8.7) is negligible and \(PeerSearchCt_{ij}\) in Eq. (8.6) contributes majority of DNUCA cost (Please refer to Table 8.2 for validation). Eq. (8.7) evaluates time spent in transit for rest of the accesses in a runlength, for addresses causing more contention.

\[ AvgDistanceSearchCt_{ij} = (A_{ij} - \sum_{0 \leq k < K} r_{ijk}) \times D_{i,j\text{Average}} \] (8.7)

Total time spent in accessing data with DNUCA is given by Eq. (8.8).

\[ DNUCA_{cost} = \sum_{0 \leq i < T} \sum_{0 \leq j < N} (PeerSearchCt_{ij} + \newline NearSearchCt_{ij} + AvgDistanceSearchCt_{ij}) \] (8.8)

If \(DNUCA_{cost}\) is lesser than \(SNUCA_{cost}\) for an application then DNUCA is more profitable policy for that application and vice versa.
Table 8.2: shows PeerSearchCt, NearSearchCt and AvgDistanceSearchCt cost components, normalized with respect to (w.r.t.) DNUCA cost. It also shows distribution of normalized CLAs with $SI = 1$, $SI > 1$, $CI < 3$ and $CI \geq 3$.

### 8.4 Results

#### 8.4.1 CPP model validation

Table 8.2 shows the contribution of the individual components\(^1\) towards the total DNUCA cost. This is obtained by normalizing these components against the total DNUCA cost (Eq. (8.8)). We compare the normalized costs obtained by the CPP against those obtained using simulator, which are also shown in Table 8.2. The cost distribution determined by CPP closely matches to that obtained using simulator.

For applications such as ocean and blackscholes most of the accesses are private ($SI = 1$) or are done from cores belonging to the same column, hence, $CI > 2$ for them. As a result, NearSearchCt is a major component. Hence, DNUCA allocates data near to their threads. For these applications, CPP estimates $DNUCA_{cost}$ to be lesser than $SNUCA_{cost}$.

For remaining applications, majority of data is shared by more than one thread and hence our model predicts a higher cost with DNUCA than SNUCA. For such applications, PeerSearchCt contributes a majority of $DNUCA_{cost}$. For instance, 19% of addresses are private ($SI = 1$) in fft. Hence, PeerSearchCt contributes to 87% of a $DNUCA_{cost}$. fft shows 9.5% degradation in execution time with DNUCA access policy.

Fig. 8.3 shows ratio of DNUCA to SNUCA costs predicted by CPP and that obtained using a cycle accurate simulator. The ratio of execution time with DNUCA to that with SNUCA, both obtained using a cycle accurate simulator is also shown in Fig. 8.3. CPP evaluates higher DNUCA cost for applications like fft, mpegdec and raytrace. These applications show 9.5%, 8% and 25% degradation.

---

\(^1\)PeerSearchCt in Eq. (8.5), NearSearchCt in Eq. (8.6), AvgDistanceSearchCt in Eq. (8.7)
Figure 8.3: Accuracy of CPP Model: CostModel-Ratio of $DNUCA_{cost}/SNUCA_{cost}$ estimated by CPP. CostSim - Ratio of time spent in transit in DNUCA to that in SNUCA, both obtained with simulation in their execution time respectively, with DNUCA. For applications like ocean and blackscholes, CPP predicts lesser $DNUCA_{cost}$ than $SNUCA_{cost}$, which also tallies with our experimental results. These applications show 7% and 4% improvement in execution time with DNUCA over SNUCA. Most of the accesses in these applications are private.

Fig. 8.3 also shows L2 access latency obtained using DNUCA and normalized with that obtained with SNUCA for these applications. Normalized L2 latency obtained for DNUCA does not show a large degradation as seen for cost ratios. This is because, while calculating average L2 latency in simulator, offchip DRAM access latency is also considered. DRAM latency is much larger than time spent in NoC. CPP considers time spent in accessing data from L2 banks alone. It assumes data is already present in the L2 cache. The number of DRAM accesses depends on the working set size of an application, and does not depend heavily on DNUCA/SNUCA cache access policy. The cost values determined using model should not be considered in absolute sense. Their relative ratio indicates whether DNUCA would be a better choice over SNUCA or vice verse.

CPP predicts costs for DNUCA and SNUCA policies with one-time simulation using SNUCA. Hence, by saving one simulation with DNUCA, it achieves an average simulation speed-up of 2.2 times compared to SNUCA and DNUCA cycle accurate simulations put together(Fig. 8.4). Fig. 8.4 also shows overhead over SNUCA simulations. The overhead is expressed as ratio of simulation time required to simulate SNUCA which collects runlength statistics over the SNUCA execution which does not collect any statistics. It can be seen that overhead to track CLA access pattern is very negligible. This is because we only track cache accesses which miss in L1s which are much lesser than the total number of accesses done to the cache subsystem. Due to slight variation in the number of instructions executed by each core, in some cases simulation time with CPP is slightly lesser than SNUCA simulation. Statistics
Chapter 8. TCP: Thread Contention Predictor for Parallel Programs

Figure 8.4: shows simulation speed-up obtained by CPP over cycle-accurate simulations of DNUCA and SNUCA. Overhead is ratio of simulation time with and without statistics collecting SNUCA simulation collection can be done by another thread without affecting time required to simulate SNUCA. CPP can be used by system architects to make a choice between DNUCA and SNUCA cache policies while designing a system.

8.4.2 Sensitivity to the L2 slice size

We also performed experiments for L2 slice of size 256KB. Our predictions match with the simulation results. Graph in Fig. 8.5 compares ratio of costs determined by CPP to ratio of execution time required for DNUCA and SNUCA policies. When L2 slice of 256KB is used, all applications give a better performance with SNUCA, except ocean and blackscholes. This behaviour is same as with L2 slice of 512KB size. This shows that the CPP model also works for smaller L2 cache.
8.4.3 Sensitivity to the link latency

To evaluate sensitivity to link latency of CPP, we used unit latency for all links of the NoC. Fig. 8.6 shows cost predicted by CPP on the primary Y axis and DNUCA execution time normalized w.r.t. that with SNUCA on the secondary Y axis. In raytrace, the execution time with DNUCA degrades by only 7% if link latencies are set to unit value as opposed to 25% for non-unit link latencies, which is intuitive. Similar behaviour is seen with mpegdec. However, in both the cases CPP accurately predicts DNUCA cost to be higher than SNUCA cost. In case of ocean, DNUCA shows only 4% improvement in execution time as opposed to 8% with non-unit link latency case. CPP accurately predicts DNUCA as more suitable than SNUCA in both the scenarios. These experiments show that the CPP can predict accurately even on varying link latencies.
8.4.4 Sensitivity to the number of threads

We simulated applications with eight threads on a sixteen tiled CMP with SNUCA cache policy and used CPP model to predict suitability of NUCA cache access policy. Fig. 8.7 shows ratio of DNUCA to SNUCA cost determined by CPP model. It also shows ratio of execution time obtained with DNUCA to that obtained with SNUCA. blackscholes and ocean show lesser execution time with DNUCA than with SNUCA. Like our previous experiments, CPP’s predictions are in line with the experimental observations.

8.5 Summary

We present TCP, a tool to profile data accessed by a multi-threaded application. TCP quantifies the number of threads sharing data and their sharing pattern. We propose a sharing index (SI), inspired from entropy in information theory. SI quantifies the number of threads sharing an object. Higher SI does not necessarily imply higher contention for an object. Hence, we define contention index (CI) expressed in terms of the average number of consecutive accesses made by the same thread. We define a runlength as the number of consecutive accesses done by the same thread. Higher average runlength size denotes lower contention for an object. We use SI and CI to accurately predict a more profitable cache access policy between SNUCA and DNUCA for an application. This model achieves an average simulation speed-up of 2.2 times compared to SNUCA and DNUCA cycle-accurate simulations put together. We varied the number of threads, L2 cache size and link latencies to study sensitivity of CPP.
Chapter 9

Conclusions And Future Work

The number of cores and on-chip cache size have been increasing on chip multiprocessors (CMPs) due to advances in technology. As a result, leakage power consumed in caches has become a major contributor to the power dissipated in the memory subsystem. Large caches are implemented using multiple banks that are interconnected with an on-chip network. Such cache offers non-uniform access latency to different cores present on a CMP. Hence, it is called Non-Uniform access Cache Architecture (NUCA)\cite{32}.

The prior circuit and architecture level studies for optimization of leakage power in caches consider a single-threaded application executing on a uniprocessor with small cache which offers uniform access latency. Hence, heuristics such as average memory access latency (AAL) and cache miss ratio (CMR) give acceptable performance. However, these heuristics give suboptimal performance in case of threads executing concurrently on a CMP with NUCA cache. Hence, we have proposed a new cache working set size estimation method, which we call “Tagged Working Set Size (TWSS)” estimation method. We compare TWSS against WSS estimated using Dhodapkar’s approach (DHP) \cite{132}. In DHP, a bit-vector is maintained in a core. The instruction addresses are hashed and the corresponding bit is set the bit-vector. The fraction of bits set in the bit-vector probabilistically determines WSS. Accuracy of this method is highly dependent on the choice of hash function. The DHP method under-estimates WSS in some cases due to aliasing problem. To demonstrate this, we experimented with various commonly used hash functions. Our experiments show that it is very difficult to find a hash function which suites all applications.

TWSS overcomes the drawback of under-estimating WSS by using inherent hash function built in the cache. TWSS maintains an active bit along with tag bits of a cache line. The tag bits uniquely identify a cache line address, setting an active bit. A replacement counter is also maintained in each L2 slice. This counter counts the number of times cache lines with their active bit set, are replaced. However, TWSS might over-estimate WSS if the same cache line is accessed and replaced multiple times in a monitoring...
period. The over-estimation of WSS does not compromise working of the cache optimization policies by keeping more caches powered-on when higher number of active lines are replaced. The WSS estimated by TWSS accurately matches the actual WSS estimated at L2 cache. The WSS estimated by DHP method using srand based hash function over-estimates by an average of 19%. On the contrary, the DHP method under-estimates WSS significantly (up-to 50%) with linear feedback serial register based hash functions.

The WSS estimated using TWSS can be used to switch off cache implemented using any memory technology for which the difference between leakage power dissipated in the power-on state and power-off state is significant. We demonstrate its use to adjust cache associativity. Past studies [34, 132, 35] perform leakage power optimization for L1 caches, whereas, in this thesis, we have implemented it for the last level NUCA cache on a CMP. Due to distributed nature of NUCA cache and concurrently executing threads/processes, typically used heuristics, such as AAL [33] and CMR [34] give suboptimal energy savings. Moreover, due to increased number of components on a CMP, the interconnect has become more complex and adds considerable latency. Hence, decision to adjust cache associativity should be taken locally using information available in the same tile. This is not feasible if heuristics such as, AAL, CMR or instructions per cycle are used. To determine values of these heuristics, information is gathered from all the tiles and cache associativity decision is taken by one of the L2 cache controllers. Apart from accessing NoC, this sets uniform cache associativity for all L2 slices despite their non-uniform usage. Hence, we propose a scalable algorithm to adapt cache associativity. Each L2 cache controller estimates the usage of its L2 slice using TWSS method and adjusts its associativity accordingly. We compared energy-delay product (EDP) savings obtained with this method to the implementation using CMR and AAL heuristics. Our implementation gives 25% and 19% higher EDP savings than that obtained using AAL and CMR on SNUCA platform, respectively.

K. Flautner [13] proposed a drowsy cache implementation in which the supply voltage of less frequently used cache lines is lowered so as to dissipate lesser leakage energy. TWSS.S and TWSS.D implementation achieve 15% and 23% higher EDP savings than that obtained using the drowsy cache technique. The drowsy cache technique gives higher EDP savings than the adaptable associative caches for applications with larger WSS. On the contrary, for applications with smaller WSS, the adaptable associative cache gives higher EDP savings. It should be noted that reduction in supply voltage makes cache unreliable and prone to transient errors [19, 18, 20]. This problem is exacerbated by decrease in transistor size. Hence, the dynamic voltage scaling is not a reliable solution to reduce leakage power consumption in caches.

On reducing cache associativity, the number of conflict misses increases. Hence, we have also proposed the remap policy in which cache associativity remains constant. In this policy, farther L2 slices are mapped onto nearer L2 slices to reduce leakage power consumption. The remap policy obtains up-to 6%
improvement in execution time for some applications. On an average, the remap policy achieves 6% to 8.7% higher EDP gains when compared to the adaptable associative caches implemented with SNUCA and DNUCA policies. It achieves an average of 26% higher EDP savings when compared to the drowsy cache implemented using DNUCA. Fig. 9.1 shows power consumption for various memory subsystem components. Power is normalized with respect to the total power consumed in the reference execution on a tiled SNUCA CMP. Since ocean has large WSS, the TWSS method detects large cache requirement accurately and allocates all the L2 slices. Thus ocean executes without causing EDP degradation. On the other hand for applications like blackscholes, swaption, fluidanimate and x264 large EDP savings are obtained.

The remap policy obtains 6% improvement in execution time of x264. Similarly, it achieves 4%-6% improvement in execution time of fft and mpegdec applications. This is due to “fork and join” type of parallelism present in mpegdec. Whereas, x.264 shows poor thread scalability as it uses pipeline parallelism. Hence, on specifying sixteen threads, the application spawns up-to six threads concurrently. The remap policy allocates L2 slices nearer to threads and achieves improvement in execution time of these applications, even on reducing cache allocation. This is not possible in adaptable associative or drowsy caches.

Though the remap policy achieves higher EDP savings compared to the adaptable SNUCA and DNUCA associative implementations, its hardware implementation is more complex than the latter. Hence, we explore the maximum EDP savings that can be obtained using the remap policy. To do this, we use genetic algorithms and formulate determination of the near optimal remap table problem as a energy-delay minimization problem. Our GA implementation estimates fitness of each chromosome using a trace based model. The near optimal remap tables obtained using GA approach are simulated. The EDP gains obtained by the remap policy are within 5% of that obtained using GA based remap
Energy-delay product can also be minimized by improving execution time, which depends mainly on the static and dynamic NUCA access policies (DNUCA). Some applications like blackscholes and ocean have smaller execution time with the DNUCA policy, whereas, applications like fft and raytrace show significant degradation (10%-25%) with the DNUCA policy, when compared to the SNUCA policy. Hence, we also study characteristics of an application which decide a more profitable policy among SNUCA and DNUCA. The main factors that decide the suitability of cache access policy are:

1. The amount of data shared between the threads: DNUCA migrates data to nearer L2 slice. Hence, if data is not found in the nearest L2 slice, remaining L2 slices in a bankset are searched which causes additional delay. For applications in which data is not shared by many threads, DNUCA cache is preferred.

2. Interleaving of accesses made by various threads: If a cache line is shared by many threads then the interleaving of these accesses, is also important. If the accesses are not interleaved then it will cause lesser coherence traffic and data will be found in the nearer L2 slice. In such a case, DNUCA gives lesser execution time than SNUCA. On the contrary, applications in which threads share more data in interleaved manner, SNUCA is more preferable.

We believe that the data sharing indices which we have proposed in Chapter 8, have more applications such as detecting false data sharing and reducing aborts in a transaction memory. They can also be used to detect important cache line addresses for which migration should be avoided in DNUCA. This will reduce the overhead incurred in conflicting cache line promotions/demotions.

We believe that the study done in this thesis will help architects to take important decisions such as deciding between SNUCA or DNUCA and various leakage power optimization policies. It will help programmers organize their programs better so as to reduce cache contention caused due to interleaved accesses. This thesis proposes a new highly accurate method to estimate WSS of an application which can be used to partition cache across different applications. Apart from this, we also present a new method to estimate cache misses and execution time of a multi-threaded application running on a tiled CMP. This method can be used for other platforms such as non-tiled CMPs with shared cache. With this, time consuming cycle accurate architectural simulation can be partially avoided.

9.1 Future work

Traditionally SRAM is used as on-chip cache. The latest IBM’s processors such as Power 7 implement large on-chip cache using embedded DRAM (eDRAM) due to its higher density. However, eDRAM needs to refresh its data periodically. Hence, additional energy is dissipated in refreshing banks. It is
not clear whether the dynamic voltage scaling can be applied to eDRAM. However, eDRAM bank can be switched off to reduce both leakage and refresh power dissipated in it. We would like to evaluate our proposals with the on-chip cache implemented using eDRAM in future.

In our current proposal for thread contention predictor, contention and data sharing indices are determined at the granularity of cache line addresses. However, in future we want to implement the same at the granularity of data structures. This will make it more user friendly. Programmers can easily detect data structures which are more popular among threads. They can re-organize these data structures so as to reduce false data sharing that might be caused due to multiple threads accessing various elements in these structures. This will also help architects allocate popular data structures in scratch pad memories [36] in case of embedded platforms or change migration policies of such addresses in DNUCA. In transactional memories, transactions involving such addresses can be serialized to reduce aborts. “gprof” tool profiles instruction addresses in an application. However, we would like to write a tool to profile data structure accesses for an application. Apart from giving usage statistics, this tool would display contention and sharing indices as well.

Currently, we have implemented TCP in a cycle-accurate simulator which makes it slower. To use this tool, one needs to install Sapphire simulator and also compile applications using a cross-compiler. This makes it harder to use. Hence, we would like to implement TCP with widely used dynamic instrumentation tool such as PIN [172]. This will improve usability and speed of this tool significantly.

In this thesis, we use the remap policy to reduce leakage power dissipated in the on-chip cache on a CMP. It can also be used to partition L2 slices among virtual machines or applications. This will reduce cache contention caused in a shared cache due to unrelated processes executing concurrently on a server. It will also reduce transit time spent in accessing L2 cache as nearer L2 slices can be allocated using the remap policy.
Appendices
Appendix A

Variation in Working Set Size with Time

Fig. A.1 and Fig. A.2 give variation in working set size of all applications. WSS is the number of unique cache line addresses accessed in a monitoring period of 4M clock cycles. WSS is plotted on the Y axis in terms thousands of cache line addresses.
Figure A.1: shows variation in WSS with program execution. WSS is in terms of the number cache lines (in thousands) plotted on the Y axis and the X axis shows monitoring period (4M clock cycles).
Figure A.2: shows variation in WSS with program execution. WSS is in terms of the number cache lines (in thousands) plotted on the Y axis and the X axis shows monitoring period (4M clock cycles).
Appendix B

Sensitivity study of EDP savings

In this appendix, we give plot variation in EDP savings against the monitoring period for multiprogramming workloads based on SPEC benchmarks. Fig. B.1 to Fig. B.8 show variation in EDP savings obtained with the variation in the monitoring period for multiprogramming workload. For each application, part (a) plots EDP savings on the primary Y axis and monitoring period value on the X axis. The percentage of accesses with reuse time less than the monitoring period is plotted on the secondary Y axis. We also plot variation in WSS with the monitoring period in Part (b). WSS is plotted in terms of the number of cache line addresses on the Y axis for various monitoring periods on the X axis. Part (c) plots the average replacements per set and its deviation on the Y axis for various monitoring periods on the X axis.

![Plot variations](image)

(a) Reuse time (on the secondary Y axis) and EDP variation (on the primary Y axis)  
(b) WSS  
(c) Replacements/set

Figure B.1: dealII-dealII-dealII-dealII: Variation in EDP with monitoring period

libq-libq-libq-libq, mcf-mc-fl-quantum-sjeng and mcf-mc-fl-mc-fl have only up-to 60%, 85% and 94% of total accesses with reuse time less than 4M clock cycles. However, remaining accesses have a very large reuse time (greater than 200M clock cycles). Hence, Fig. B.6(a), Fig. B.7(a) and Fig. B.8(a) plot two distribution curves of the reuse time, one each when such accesses are ignored and considered.
Appendix B. Sensitivity study of EDP savings

Figure B.2: soplex-bzip2-namd-dealII: Variation in EDP with monitoring period

Figure B.3: milc-deal-gcc-soplex: Variation in EDP with monitoring period

Figure B.4: sjeng-mcf-milc-dealII: Variation in EDP with monitoring period

Figure B.5: mcf-milc-namd-dealII: Variation in EDP with monitoring period
Appendix B. Sensitivity study of EDP savings

Figure B.6: mcf-mcf-mcf-mcf: Variation in EDP with monitoring period

Figure B.7: libq-libq-libq-libq: Variation in EDP with monitoring period

Figure B.8: mcf-milc-quantum-sjeng: Variation in EDP with monitoring period
Appendix C

Additional Results

Fig. C.1 compares execution time of an application using Bardine’s heuristic and TWSS.D are used to optimize leakage power dissipated in cache. Please refer to Chapter 5 for more information.

Fig. C.2 compares EDP and execution time in the case of DNUCA on varying L2 cache size. Clearly, EDP savings are higher for larger L2 cache.

Figures C.3 - Fig. C.13 show how WSS and EDP vary on changing reuse time for various applications.

Fig C.14 show variation in EDP savings obtained using remap policy on varying size of L2 slice.

Fig. C.15(b) and Fig. C.15(a) compare execution time and EDP savings obtained with the remap policy in the case of the multiprogramming workload.
Appendix C. Additional Results

Figure C.1: Quantitative comparison of our variable way DNUCA implementation with Bardine’s heuristic

(a) Smaller EDP value is better

(b) Smaller execution time is better

Figure C.2: TWSS.D: Sensitivity to L2 slice size
Appendix C. Additional Results

Figure C.3: fluidanimate: Variation in EDP with monitoring interval

Figure C.4: swaption does not exhibit any active line replacement during execution. Variation in EDP with monitoring interval

Figure C.5: x264: Variation in EDP with monitoring interval
Appendix C. Additional Results

Figure C.6: FFT: Variation in EDP with monitoring interval

Figure C.7: Barnes does not exhibit any active line replacement during its execution. Variation in EDP with monitoring interval

Figure C.8: Water spatial and water nsquared does not exhibit any active line replacement during its execution. Plots show variation in EDP with monitoring interval
Appendix C. Additional Results

Figure C.9: Radix: Variation in EDP with monitoring interval

Figure C.10: ocean: Variation in EDP with monitoring interval
Appendix C. Additional Results

(a) Reuse time and EDP variation

(b) WSS

(c) Replacements/set

Figure C.11: raytrace: Variation in EDP with monitoring interval

(a) Reuse time and EDP variation

(b) WSS

Figure C.12: mpegenc: Variation in EDP with monitoring interval

(a) Reuse time and EDP variation

(b) WSS

Figure C.13: mpegdec: Variation in EDP with monitoring interval
Figure C.14: Sensitivity to L2 slice size: EDP normalized w.r.t. the reference SNUCA execution
Appendix C. Additional Results

141

(a) Smaller EDP value is better

(b) Smaller execution time is better

Figure C.15: Multiprogramming SPEC 2006 workload scheduled on a quad-core CMP with shared 4MB L2 cache
Bibliography


J. Holland, “Book on Genetic Algorithm.”


Index

Abstract, 5
Front matter, 4
I.I.Sc. logo, 4
Index, 8
Line spacing, 6
page headings, 9
Preface Section, 4
Title page, 3