DRAM Aware Last-Level-Cache Policies for Multi-core Systems

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik

der Karlsruher Institut für Technologie (KIT)

genehmigte

Dissertation

von

Fazal Hameed

Tag der mündlichen Prüfung: 6. Februar 2015

Referent: Prof. Dr.-Ing. Jörg Henkel, Karlsruher Institut für Technologie (KIT), Fakultät für Informatik, Lehrstuhl für Eingebettete Systeme (CES)

Korreferent: Prof. Dr. rer.nat. Wolfgang Karl, Karlsruher Institut für Technologie (KIT), Fakultät für Informatik, Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung (CAPP)

DOI: 10.5445/IR/1000049345

Fazal Hameed
Luisenstr. 22,
76137 Karlsruhe

Hiermit erkläre ich an Eides statt, dass ich die von mir vorgelegte Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen – die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

_____

F a z a l H a m e e d

Dedication

To the memory of my beloved late mother, Parveen Subhan. I miss you very much. May your soul rest in eternal peace

Acknowledgements

This thesis is dedicated to my loving beloved mother, Parveen Subhan, who passed away during the course of my PhD studies. My mother was an everlasting source of inspiration, love, and affection. She prepared me to face the challenges of life with patience and dignity.

I express my sincere gratitude to Professor Dr.-Ing. Jörg Henkel for his guidance in mentoring me during the course of my PhD studies. He taught me the true essence of academic research and motivated me to explore new research ideas and directions. He provided the proper balance of guidance and freedom required to make a smooth progress in research. It has been a more pleasant learning experience to work under his guidance.

I would like to pay my deepest gratitude to Professor Dr-Ing. Wolfgang Karl for his acceptance to become my co-examiner.

I am grateful to my co-supervisor, Dr.-Ing Lars Bauer, whose valuable suggestions and in-depth critique feedback during proof reading helped me to complete this work. He has been very friendly and kind to me and his thoughtful reviews significantly improved the quality of my several conference submissions. In addition to the technical support, I explicitly acknowledge his efforts to arrange HiWi and DFG funding to support this work. I would also like to thank my former co-supervisor Dr.-Ing. Mohammad Abdullah Al Faruque who provided me valuable feedback and suggestions while submitting my first paper in Design Automation and Test in Europe (DATE) conference.

I want to thank my colleagues in the Chair of Embedded systems (CES) for their company to provide a congenial and learning work environment for my studies. I would like to thank my office-mates Manyi Wang, Heba Khdr, and Florian Kriebel for their patience and tolerance with me and my jokes. I would like to thank Waheed Ahmed, Nabeel Iqbal and Usman Karim for joining me at the lunch breaks to have some fun. I will never forget the family gathering and sport activities (especially cricket) we have done together on weekends. I would like to thank Martin Buchty, Sammer Srouji, Hussam Amrouch, Artjom Grudnitsky, Martin Haaß, Chih-Ming Hsieh, Sebastian Kobbe, Mohammadi Abbas, Volker Wenzel, and Farzad Samie for their wonderful company during my stay at CES.

I am also grateful to Institute of Space Technology (IST) and German Research Foundation (DFG) for providing the financial support to complete this work.

Finally, I am specially thankful to my father Fazal Subhan and wife Rozina Khan for their encouragement, patience, and prayers during my PhD. studies. I especially thank my wife for her continued moral support and sacrifices when I felt dejected and depressed.

“Ideally one would desire an indefinitely large memory capacity such that any particular […] word would be immediately available […] It does not seem possible physically […]. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.”

A.W. Burks, H.H. Goldstine, and J. von Neumann - 1946

List of Publications

Conferences (double-blind peer reviewed)

[C.1] F. Hameed, L. Bauer, and J. Henkel, "Reducing Latency in an SRAM/DRAM Cache Hierarchy via a Novel Tag-Cache Architecture", In *IEEE/ACM Design Automation Conference (DAC'14)*, June, 2014.

[C.2] F. Hameed, L. Bauer, and J. Henkel, "Reducing Inter-Core Cache Contention with an Adaptive Bank Mapping Policy in DRAM Cache", In *IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*, October 2013.

Nominated for the best paper.

[C.3] F. Hameed, L. Bauer, and J. Henkel, "Simultaneously Optimizing DRAM Cache Hit Latency and Miss Rate via Novel Set Mapping Policies", In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'13)*, October 2013.

[C.4] F. Hameed, L. Bauer, and J. Henkel, "Adaptive Cache Management for a Combined SRAM and DRAM Cache Hierarchy for Multi-Cores", In *Proceedings of the 16th conference on Design, Automation and Test in Europe (DATE'13)*, pages 77–82, March 2013.

[C.5] F. Hameed, L. Bauer, and J. Henkel, "Dynamic Cache Management in Multi-Core Architectures through Run-time Adaptation", In *Proceedings of the 15th conference on Design, Automation and Test in Europe (DATE'12)*, pages 485–490, March 2012.

[C.6] F. Hameed, M.A. Al Faruque, and J. Henkel, "Dynamic Thermal Management in 3D Multi-core Architecture Through Run-Time Adaptation", In *Proceedings of the 14th conference on Design, Automation and Test in Europe (DATE'11)*, pages 299–304, March 2011.

PhD Forum

[P.1] F. Hameed, L. Bauer, and J. Henkel, "DRAM Aware Last Level Cache Policies for Multi-Core Systems", In *17th conference on Design, Automation and Test in Europe (DATE)*, March 2014.

The Big Picture

The Chair for Embedded Systems (CES) at the Karlsruhe Institute of Technology (KIT) provided me the required research platform and infrastructure to complete my thesis. The people at CES are primarily involved in research in the field of Design and Architectures for Embedded Systems. A current focus of research are multi-core systems, dependability, and low power design. I worked in the multi-core research group with a focus on cache and processor architectures leveraging the application access patterns to achieve performance and reliability goals.

The CES is involved in various projects of the DFG Transregional Collobarative Research Center (TCRC) 89 "Invasive Computing" [44, 122]. This collaboration between various departments at KIT (including the CES), Technische Universität München (TUM), and University of Erlangen (FAU), aimed to investigate a novel paradigm for the design and resource aware programming of future parallel computing systems. This scientific team includes experts in algorithms for parallel algorithm design, hardware architects for multi-core system development as well as language, compiler, application and operating system designers. The key idea of invasive computing is to develop resource aware programming support so that programs are given the capability to dynamically spread its computations over a dynamic area in the multi-core systems similar to a phase of invasion. This requires revolutionary changes in architecture, programming languages, compilers, operating systems, and design of multi-core systems to effectively enable resource aware programming for various processor and memory configurations. I primarily contributed to the "Invasive Computing" research project by introducing novel architectural concepts in the multi-core cache hierarchy [C.1][C.2][C.3][C.4][C.5]. The proposed architectural techniques aimed to develop efficient run-time hardware techniques to exploit the full potential of the multi-core cache hierarchy.

The CES also makes significant contributions to the DFG SPP 1500 "Dependable Embedded Systems" projects. The key idea of SPP 1500 research project is to address the increasing dependability and reliability issues in future nano-scale technologies [41] due to device aging, high chip power densities, and process variation. The SPP aimed to develop reliability aware mechanisms at both the hardware level (i.e., architecture and micro-architecture), and software level (i.e. compilers, operating system, and application). The SPP 1500 comprises various individual sub-projects, working synergistically to develop reliability modeling and aging mitigation techniques based on the future technology trends [42]. The major challenges addressed by SPP 1500 include dark silicon and thermal challenges [43, 110]. One of the key contributions of the CES towards SPP 1500 is the "VirTherm-3D" research project in collaboration with Technische Universität München (TUM) [23]. The goal of this project is to address dependability problems in 3D stacked many-core architectures resulting from extreme temperatures. I also contributed to the VirTherm-3D project by proposing an efficient processor architecture and designed different components of processing cores (arithmetic units, register file, instruction scheduler, cache, reorder buffer, etc.) for performance, power and thermal improvement [C.6].

Abstract

In addition to fast and small L1 and L2 caches, that are typically dedicated to a particular core in multi-core systems, the larger so-called Last-Level-Cache (LLC) is shared among all cores and used to bridge the latency gap between high speed cores and slower off-chip main memory. Traditionally, the LLC in multi-core systems consists of on-chip SRAM memory which comes at a large area overhead that limits the LLC size. As multi-core systems employ more and more cores on a single chip, the limited LLC size leads to an increasing number of off-chip memory accesses. This increases the average latency per access which reduces the overall performance. Therefore, recent research in academia and industry (e.g. IBM POWER7) has employed high capacity on-chip DRAM as LLC between L1/L2 SRAM cache and main memory. The primary reason for employing on-chip DRAM cache is that it provides greater capacity benefits for a given area compared to SRAM cache (~8×), which reduces off-chip accesses.

When the DRAM cache is shared among multiple cores, the cores might interfere with each other in the DRAM cache controller causing inter-core interference that increases DRAM cache hit latency. The problem is exacerbated, when one core evicts useful data belonging to another core causing inter-core cache eviction that increases DRAM cache miss rate. This thesis primarily focuses on reducing DRAM cache miss rate and DRAM cache hit latency via novel application-aware and DRAM-aware cache policies while addressing the above mentioned challenges. This thesis makes the following novel contributions:

1. Different applications have different cache access behavior that make them better suited to use different DRAM insertion rates (DRAM insertion rate is defined as the percentage of data insertions into DRAM cache). To choose a suitable insertion rate at runtime, this thesis proposes an adaptive DRAM insertion policy that mitigates inter-core cache eviction by adapting the DRAM insertion rate in response to the dynamic requirements of the individual applications with different cache access behaviors. The proposed policy selects a suitable insertion rate from multiple insertion rates depending on which insertion rate provides reduced off-chip memory accesses.
2. To further mitigate the miss rate, this work proposes a DRAM set balancing policy after analyzing that DRAM accesses are not evenly distributed across the sets of the DRAM cache, which leads to increased conflict misses via unbalanced set utilization. The proposed set balancing policy reduces conflict misses via reduced inter-core cache eviction that lead to a reduced miss rate, hereby improving the overall system performance.
3. DRAM row-buffer conflicts occurs, when multiple simultaneous requests are mapped to different rows of the same DRAM bank, causing high DRAM cache hit latency compared to a scenario when these requests are mapped to the same row. To reduce DRAM cache hit latency, this thesis proposes a novel DRAM row buffer mapping policy that reduces row buffer conflicts by exploiting data access locality in the row buffer.
4. To further reduce the DRAM cache hit latency, this thesis proposes a small and low latency SRAM structure namely DRAM Tag-Cache (*DTC*) that holds the tags of rows that were recently accessed in the DRAM cache. The proposed *DTC* has a high hit rate, because it exploits data access locality provided by the proposed DRAM row buffer mapping policy mentioned above. It provides fast tag lookup because for a *DTC* hit, it reads the tags from the low

latency *DTC* in two cycles. In contrast, state-of-the-art DRAM cache always reads the tags from DRAM cache that incurs high tag lookup latencies of up to 41 cycles.

In summary, high DRAM cache hit latencies, increased inter-core interference, increased inter-core cache eviction, and the large application footprint of complex applications necessitates efficient policies in order to satisfy the diverse requirements to improve the overall throughput. This thesis addresses how to design DRAM caches to reduce DRAM cache hit latency, DRAM cache miss rate and hardware cost, while taking into account both application and DRAM characteristics by presenting novel DRAM and application aware policies. The proposed policies are evaluated for various applications from SPEC2006 using a cycle accurate multi-core simulator based on SimpleScalar that is modified to incorporate DRAM in the cache hierarchy. The combination of the proposed DRAM-aware and application-aware complementary policies improve the average performance of latency-sensitive applications by 47.1% and 35% for an 8-core system compared to [102] and [77] respectively while requiring 51% less hardware overhead.

Zusammenfassung

Neben schnellen und kleinen L1- und L2-Caches, die in Mehrkernsystemen typischerweise einem bestimmten Kern fest zugeordnet sind, wird der größere sogenannte Last-Level-Cache (LLC) von allen Kernen gemeinsam genutzt und wird verwendet, um die unterschiedlichen Latenzen von Kernen mit hoher Geschwindigkeit und dem langsameren externen Hauptspeicher zu überbrücken. Traditionellerweise besteht der LLC in Mehrkernsystemen aus einem on-chip SRAM-Speicher, der einen hohen Flächenverbrauch mit sich bringt, welcher die Größe des LLC begrenzt. Da Mehrkernsysteme immer mehr Kerne auf einem einzelnen Chip integrieren, führt die begrenzte LLC-Größe zu einer steigenden Anzahl von off-chip Speicherzugriffen. Dies erhöht die durchschnittliche Latenz pro Zugriff, was die Gesamtleistungsfähigkeit reduziert. Deshalb verwenden jüngste Forschungen in Akademie und Industrie (z.B. IBM POWER7) on-chip DRAM mit hoher Kapazität als LLC zwischen L1/L2 SRAM Cache und dem Hauptspeicher. Der primäre Grund für die Nutzung eines on-chip DRAM Caches ist, dass dieser im Vergleich zu SRAM Cache höhere Kapazitätsvorteile für eine festgelegte Fläche erreicht (~8×), wodurch die off-chip Zugriffe reduziert werden.

Wenn der DRAM Cache von mehreren Kernen gemeinsam genutzt wird, könnten die Kerne sich im DRAM Cachecontroller gegenseitig beeinflussen und Inter-Kern-Interferenzen verursachen, welche die Cache-Hitlatenz erhöhen. Das Problem verschlimmert sich, wenn ein Kern nützliche Daten verdrängt die einem anderen Kern gehören und damit eine Inter-Kern-Cacheverdrängung verursacht, welche die DRAM Cache Missrate erhöht. In dieser Arbeit liegt der Schwerpunkt auf der Verringerung der DRAM Cache Missrate und der DRAM Cache Hitlatenz durch neue applikations- und DRAM-bewusste Cacherichtlinien unter Berücksichtigung der oben genannten Herausforderungen. Diese Arbeit leistet die folgenden neuen Beiträge:

1. Verschiedene Applikationen haben unterschiedliches Cachezugriffsverhalten, wodurch sich verschiedene DRAM Einfügungsraten besser eignen (die DRAM Einfügungsrate ist definiert als der Anteil der vom Hauptspeicher zu einem Kern gebrachten Daten, die auch in den DRAM Cache eingefügt wird). Um zur Laufzeit eine geeignete Einfügungsrate zu wählen, schlägt diese Arbeit eine adaptive DRAM Einfügungsrichtlinie vor, welche die Inter-Kern-Verdrängung durch Anpassung der DRAM Einfügungsrate als Antwort auf die dynamischen Anforderungen der einzelnen Applikationen mit unterschiedlichem Cachezugriffsverhalten reduziert. Die vorgeschlagene Richtlinie wählt aus mehreren Einfügungsraten eine geeignete aus – abhängig davon welche Rate eine Verringerung der off-chip Speicherzugriffe liefert.
2. Nach der Analyse, dass DRAM Zugriffe nicht gleichmäßig über die Sätze (Sets) des DRAM Caches verteilt sind, was zu einem Anstieg der durch Konflikte verursachten Cache Misses durch unbalancierte Satznutzung führt, schlägt diese Arbeit eine DRAM Satzbalancierungsrichtlinie vor, um die Missrate weiter zu verringern. Sie reduziert die durch Konflikte verursachten Cache Misses durch Verringerung der Inter-Kern-Cacheverdrängung, was zu einer geringeren Missrate führt, und verbessert damit die Gesamtleistung des Systems.
3. DRAM Zeilenpufferkonflikte treten auf, wenn mehrere gleichzeitige Anfragen auf verschiedene Zeilen derselben DRAM Bank abgebildet werden, was eine hohe DRAM Cache Hitlatenz zur Folge hat (verglichen mit dem Fall, dass diese Anfragen auf dieselbe Zeile

abgebildet worden wären). Um die DRAM Cache Hitlatenz zu verringern schlägt diese Arbeit eine neuartige Richtlinie zur DRAM Zeilenpufferabbildung vor, welche die Zeilenpufferkonflikte durch Ausnutzen von Datenzugriffslokalität im Zeilenpuffer verringert.

4. Um die DRAM Hitlatenz weiter zu verbessern schlägt diese Arbeit eine kleine SRAM Struktur mit geringer Latenz namens DRAM Tag-Cache (DTC) vor, welche die Tags der Zeilen enthält, auf die kürzlich im DRAM Cache zugegriffen wurde. Der DTC hat eine hohe Hitrate, da er die Datenzugriffslokalität nutzt, welche von der oben genannten vorgeschlagenen Richtlinie zur DRAM Zeilenpufferabbildung bereitgestellt wird. Er ermöglicht ein schnelles Nachschlagen des Tags, da für einen DTC Hit der Tag innerhalb von zwei Zyklen aus dem DTC gelesen werden kann. Im Gegensatz dazu werden bei derzeitigen DRAM Caches die Tags immer aus dem DRAM Cache gelesen, was eine hohe Tag-Nachschlagelatenz von bis zu 41 Zyklen mit sich bringt.

Zusammenfassend lässt sich feststellen, dass eine hohe DRAM Cache Hitlatenz, gestiegene Inter-Kern-Interferenz, gestiegene Inter-Kern-Cacheverdrängung und der große Bedarf von komplexen Applikationen effiziente Richtlinien notwendig machen, um die verschiedenen Anforderungen zur Verbesserung des Gesamtdurchsatzes zu erfüllen. Diese Arbeit behandelt das Design von DRAM Caches zur Reduzierung der DRAM Cache Hitlatenz, DRAM Cache Missrate und Hardwarekosten, wobei sowohl die Eigenschaften der Applikationen als auch die des DRAM durch neuartige DRAM- und applikationsbewusste Richtlinien berücksichtigt werden. Die vorgeschlagenen Richtlinien wurden für verschiedene Applikationen aus der SPEC2006 Benchmarksuite mit Hilfe eines zyklenakkuraten Mehrkernsimulators bewertet, der auf SimpleScalar basiert und modifiziert wurde, um DRAM in die Cachehierarchie zu integrieren. Die Kombination aus den vorgeschlagenen und sich ergänzenden DRAM- und applikationsbewussten Richtlinien verbessert die durchschnittliche Leistung von latenzsensitiven Applikationen um 47,1% und 35% für ein 8-Kern System verglichen mit [98] und [73], wobei ein um 51% geringerer Hardwareaufwand notwendig ist.

Contents

List of Figures

List of Tables

Abbreviations

A	Associativity
ADIP	Adaptive DRAM Insertion Policy
CRBM	Configurable Row Buffer Mapping Policy
D$	Shared DRAM cache
DRAM	Dynamic Random Access Memory
DTC	DRAM Tag-Cache
FR-FCFS	First Ready First Come First Serve
HL	Hit Latency
HMIPC	Harmonic Mean Instruction Per Cycle
KB	Kilo Byte (also KByte): 1024 Byte
ILP	Instruction Level Parallelism
IPC	Instruction Per Cycle
LLC	Last Level Cache
LRU	Least Recently Used Policy
MB	Mega Byte (also MByte): 1024 →KB
ML	Miss Latency
MR	Miss Rate
MUX	Multiplexer
OS	Operating System
RAM	Random Access Memory
RBM-A7	Row Buffer Mapping with Associativity of seven
RD	Reuse Distance
ROB	Reorder Buffer
S$	Shared SRAM cache
SB-Policy	Set Balancing Policy
SiP	System In Package
SRAM	Static Random Access Memory
STC	SRAM Tag-Cache
t_{CAS}	Column Access Strobe delay
t_{RAS}	Row Access Strobe delay

t_{RCD}	Row to Column Command delay
t_{RP}	Row Precharge delay
t_{WR}	Write Recovery time
TSV	Through Silicon Via

Chapter 1 Introduction

Moore's law [85] predicts that the number of transistors for a given chip area double every 18 months. The exponential growth in transistor density driven by Moore's law and advanced microarchitecture techniques such as pipelining [119] and out-of-order execution [125] has led to the significant increase in processor performance [12, 82] over the past several years. However, compared to processor performance, the memory performance [140] has increased at a slower pace as illustrated in Figure 1.1. This slower improvement in memory performance has led to the significant speed gap between processor and memory referred to as "Memory Bandwidth" problem [12, 60, 108, 131, 136, 144].

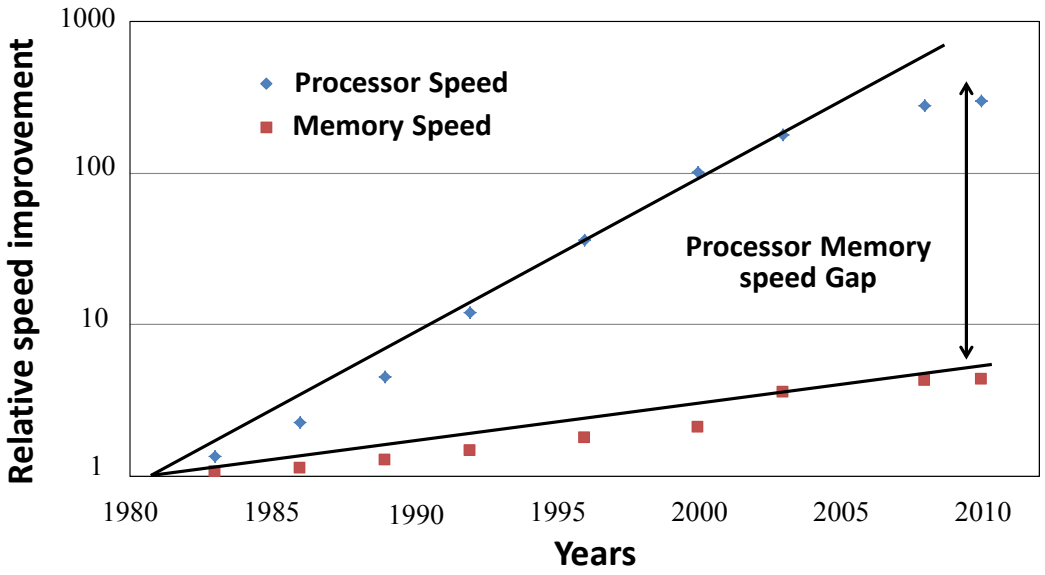


Figure 1.1: Processor memory speed gap over the past 30 years [12]

To alleviate the "Memory Bandwidth" problem, caches [29, 30] have been used to bridge the latency gap between high speed cores and slower main memory. The cache is a smaller and faster on-chip memory that stores copies of recently accessed data from frequently used memory locations to take advantage of the spatial and temporal locality of the applications. At first, processors used a single level of cache as used by Intel Pentium P5 processor in 1993. However, the widening gap between processor and memory speed has led to the evolution of multi-level cache hierarchies [12, 59, 96]. For instance, the recent Intel Xeon E5-2690 processor chip [3], introduced in 2012 employs three levels of cache hierarchy. In these hierarchies, fast and small L1 and L2 caches are dedicated to each core and provide low hit latency. The larger Last-Level-Cache (LLC) is shared among all cores and provides low miss rate.

1.1 Why On-chip DRAM cache?

A recent trend in industry towards mitigating the "Memory Bandwidth" problem is to use a large on-chip SRAM Last-Level-Cache (LLC) by dedicating a larger die area for the LLC. Table 1.1

illustrates this observation showing SRAM LLC capacity over the past several years for Intel processor chips [1]. For example, Intel P5 processor introduced in 1993 was equipped with a small 16KB SRAM cache, the recent Intel Xeon E5-2690 processor, introduced in 2012, has employed a larger 20MB SRAM LLC. Larger high-speed SRAM cache improves the performance by sending fewer requests to the low-speed off-chip memory because it can contain the working set size (i.e. the amount of memory required to execute the program) of many applications. However, for a given cache capacity, SRAM cache significantly increases the system cost compared to DRAM cache in terms of larger die area because it provides lower density compared to DRAM cache [18, 54, 55].

Year	Intel Processor Name	# Cores	On-Chip Caches	LLC size (Cache Level)
1993	Pentium P5	1	L1	16KB (L1)
1995	Pentium Pro 6	1	L1, L2	256KB (L2)
1997	Pentium II Klamath	1	L1, L2	512KB (L2)
2001	Pentium III-S Tualatin	1	L1, L2	512KB (L2)
2004	Pentium IV Prescott	1	L1,L2	2MB (L3)
2006	Core 2 Duo Conroe	2	L1, L2	4MB (L2)
2008	Xeon 7130M	2	L1,L2	8MB (L2)
2010	Xeon 7130M	6	L1, L2, L3	12MB (L3)
2012	Xeon E5-2690	8	L1, L2, L3	20MB (L3)

Table 1.1: On-Chip SRAM LLC sizes over the past years for Intel Processor Chips [1]

Despite continual increase in SRAM LLC size over the past years, the demand for cache space has always exceeded due to large working set sizes of complex applications [32]. Figure 1.2 illustrates this observation by showing LLC misses per thousand instructions for different LLC sizes and different SPEC2006 [5, 46] applications. It shows that the working set sizes of some applications (e.g. 462.libquantum, 471.omnetpp, and 473.astar.train) exceeds the available SRAM LLC size, even for the Intel Xeon E5-2690 processor chip with 20MB LLC. On the other hand, the applications with small working set sizes (e.g. 437.leslie3d.train) obtain significant benefits from a 20MB LLC, because the majority of cache requests will be satisfied in the LLC for these applications.

Single-core processors [2, 7, 9, 25, 63, 111] have shown significant performance increase during the last decades which is mainly driven by transistor speed as well as by exploiting instruction level parallelism (ILP) [12, 82, 91]. However, the diminishing transistor-speed scaling and energy limits of single-core processors have led to the evolution of multi-core systems [2, 7, 9, 14, 25, 26, 27, 31, 33, 63, 111] because it is less complex to design a chip with many small cores compared to a chip with a single larger core. For this reason, computing industry has announced multi-core processor chips that consist of several computing cores fabricated on a single chip in contrast to traditional single-core processor chips.

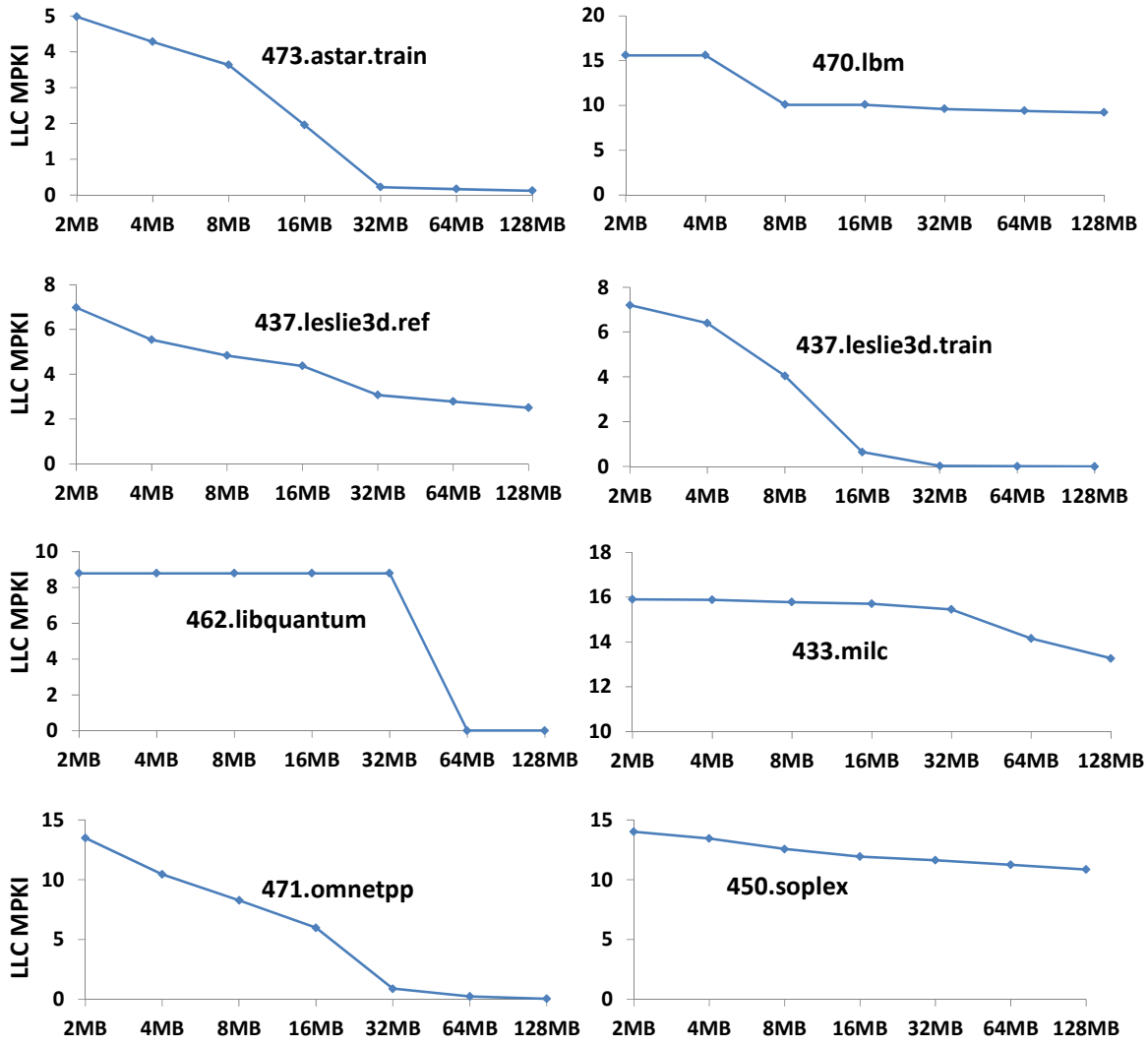


Figure 1.2: LLC misses per thousand instructions (LLC MPKI) for different SPEC2006 [5] applications

Multi-core systems provide improved performance compared to a single-core system through better resource utilization by replicating multiple cores on the chip. However, these multi-core systems place a high pressure on the SRAM LLC due to their limited cache capacity because it has to be shared among multiple applications. Recent trends likes Intel Tera-scale [127] and Tilera TILE64 [9] multi-core processor chips show that the number of cores will likely continue to increase in the future. As future multi-core systems are expected to have a large number of cores (see Table 1.1), the aggregate working set size (i.e. the amount of memory required to execute all applications) on a multi-core system will increase as well. As a result, increased number of insertions in the limited size SRAM LLC from multiple cores will cause *inter-core cache eviction* [15, 16, 35, 51, 64, 65, 73, 75, 86, 98, 99, 138] where one core could evict useful data used by another core. Increased inter-core cache eviction for traditional on-chip SRAM LLC [77, 78, 102] increases the number of off-chip memory accesses, which may degrade the performance due to limited off-chip memory bandwidth [12, 60, 108, 131, 136, 144].

The total die area dedicated for the LLC is an important design parameter for multi-core chip vendors. Increasing the amount of cache capacity for the SRAM LLC can greatly improve the performance by increasing the cache hit rate and reducing the number of high-latency off-chip accesses. However, it increases the system cost in terms of larger die area due to high cost-per-bit of the SRAM cache [123]. DRAM cache offers a lower cost-per-bit because it provides 8 to 16 times [54, 55] higher density compared to traditional SRAM caches. For a given cache capacity, DRAM cache significantly reduces the system cost in terms of smaller die area. As a result, it provides significantly higher cache capacity that leads to reduced off-chip accesses and reduced inter-core cache eviction compared to an area equivalent SRAM cache. For instance, the IBM POWER7 processor [7, 129] utilizes a 32MB on-chip DRAM as LLC between L1/L2 SRAM cache and main memory.

1.1.1 Benefits of On-Chip DRAM cache

Integrating on-chip DRAM cache in the cache hierarchy provides significant performance benefits due to the following reasons:

1. It provides eight times more bandwidth benefits compared to an off-chip memory [54, 55], because it provides wider bus widths through the use of shorter on-chip interconnects [34, 77, 78] compared to conventional off-chip memory interfaces.
2. It operates at a higher clock speed through the use of low latency on-chip interconnects [52, 61, 62, 70] compared to off-chip memory.
3. It provides more independent channels compared to off-chip memory [66] because off-chip memory cannot provide more channels due to limited pin bandwidth [60].
4. It provides 8 to 16 times capacity benefits compared to an area equivalent SRAM cache [11, 54, 55, 75] due to its small cell size per bit. Thus, it reduces contention for the off-chip main memory due to its high capacity [77, 78, 102], hereby reducing off-chip memory accesses.
5. It offers up to four times higher bandwidth compared to an SRAM cache [53] due to its capability to service multiple outstanding requests in parallel due to the large number of DRAM banks.

1.2 Challenges in DRAM Cache Hierarchy

On-chip DRAM cache is a promising alternative to SRAM cache, but its high access latency prohibits its adoption as SRAM cache replacement. Neither SRAM nor DRAM cache alone can provide both highest capacity and fastest access for multi-core system, respectively. Therefore, state-of-the-art SRAM/DRAM cache hierarchies [77, 78] exploit the latency benefits of fast SRAM cache and the capacity benefits of slower DRAM cache.

The advantages of on-chip DRAM cache come at the cost of higher latency compared to SRAM cache (but lower latency compared to off-chip memory). If designed efficiently, DRAM cache could satisfy the high capacity needs of complex applications [32] while reducing the number of high latency off-chip memory accesses. Before summarizing the thesis contributions in Section 1.3, the following subsections explain the key challenges and drawbacks that are faced by state-of-the-art and that are addressed in this thesis:

1.2.1 Inefficient resource allocation

When the DRAM cache is shared among multiple cores, the cores might interfere with each other in the DRAM cache controller causing inter-core interference that increases DRAM cache hit latency. State-of-the-art DRAM cache suffers from increased inter-core interference because it always allocates DRAM resources for both *highly-reuse* data (i.e. data that is reused in the near future) and *zero-reuse* data (i.e. data that is not reused before it gets evicted). Furthermore, they lead to inefficient DRAM cache bandwidth utilization and increased miss rate due to unnecessary resource allocation for *zero-reuse* data.

1.2.2 Limited row buffer hit rate

The DRAM sub-system is composed of DRAM banks which consist of rows and columns of memory cells called the DRAM array [68, 69, 88, 89, 141]. Each DRAM bank provides a row buffer (typically 2 to 8 KB) that consists of SRAM cells (detailed background of a DRAM bank is provided in Section 2.2.2) that operate faster than the DRAM array. Data in a DRAM bank can only be accessed after it is fetched to the row buffer. Any subsequent access to the same row (so-called row buffer hit) will bypass the DRAM array access and the data is directly read from the row buffer. Such row buffer locality reduces the access latency compared to when actually accessing the DRAM array. State-of-the-art DRAM cache architectures [77, 78] do not exploit the full potential of row buffer locality and their disadvantageous row buffer hit rate leads to high DRAM cache access latencies due to reduced spatial locality because they map consecutive memory blocks to different row buffers.

1.2.3 High tag lookup latency

In state-of-the-art DRAM cache [77, 78], each DRAM row (2048 bytes) consists of one cache set which is divided into 29 64-byte data blocks (29 × 64 = 1856 bytes) and 3 tag blocks (3 × 64 = 192 bytes). The tag lookup latency is a severe bottleneck due to the following reasons. First, it requires reading the tags (192 bytes) and data (64 bytes) for every DRAM cache access. The extraneous DRAM bandwidth required for reading this large tag information results in higher tag lookup latency. Second, the structure and access methods for DRAM subsystem (detailed background of DRAM subsystem is provided in Section 2.2) incurs high tag lookup latency compared to SRAM cache tag lookup.

1.2.4 High Hardware cost

Recent state-of-the-art DRAM cache architectures [77, 78] invest noticeable hardware overhead for auxiliary structures to circumvent some of the above-mentioned drawbacks. For instance, they require 2MB SRAM storage for managing 128MB DRAM cache, which reduces the area advantages of DRAM cache.

1.3 Thesis Contribution

The major challenges in the design of an SRAM/DRAM cache hierarchy is to reduce the on-chip latency and off-chip memory accesses that majorly depends upon efficient utilization of DRAM

cache bandwidth and capacity, tag-store mechanism (i.e. where to store the tags of the DRAM cache and how to access them), efficient utilization of off-chip memory bandwidth, and DRAM cache row buffer hit rate. This thesis investigates state-of-the-art SRAM/DRAM cache hierarchies for multi-core systems and presents novel application-aware and DRAM-aware policies for efficiently managing SRAM/DRAM cache hierarchies, while addressing the above mentioned challenges.

In particular, this thesis makes the following novel contributions:

1. This thesis proposes an application-aware **adaptive DRAM insertion policy** (an insertion policy decides whether an incoming data when brought from off-chip memory should be inserted into cache or not). It adaptively selects from multiple insertion policies at runtime on a per-core basis depending on the monitored miss rate behavior of concurrently running applications. It provides efficient utilization of DRAM cache bandwidth that leads to improved performance via reduced inter-core interference in the DRAM cache controller.
2. This thesis proposes a **DRAM set balancing policy** after analyzing that DRAM accesses are not evenly distributed across the sets of the DRAM cache, which leads to increased conflict misses via unbalanced set utilization. The proposed policy improves the DRAM capacity utilization via reduced conflict misses, which leads to a reduced miss rate.
3. To reduce the DRAM cache hit latency, this thesis proposes several DRAM **row buffer mapping policies** that improve the row buffer hit rate by exploiting data access locality in the row buffer.
4. To reduce the tag lookup latency, this thesis proposes a small and low latency SRAM structure namely **DRAM Tag-Cache** that allows most DRAM accesses to be serviced at significantly reduced access latency compared to when tags are accessed from the DRAM cache.

Altogether, this thesis develops a combined SRAM/DRAM cache organization that integrates all of the proposed policies in a single unified framework. This includes modifying existing DRAM cache controller policies to incorporate the proposed row buffer mapping policies and DRAM Tag-Cache structures. This thesis also reduces the storage overhead required for DRAM cache management with minimal impact on the overall instruction throughput for our novel row buffer mapping policies.

1.4 Thesis Outline

The thesis is organized as follows: Chapter 2 presents the background for caches (especially the DRAM cache) and the recent related work on SRAM and DRAM caches.

Chapter 3 provides a short overview of the proposed application and DRAM aware policies employed in this thesis. The detailed explanation and investigation of the proposed policies will later be presented in Chapter 5 and Chapter 6.

Chapter 4 provides the details of the experimental setup used for the extensive presentation of the evaluation results using the proposed policies presented in Chapter 5, Chapter 6, and Chapter 7. The simulation methodology for this work and state-of-the-art is presented, as well as the tools and benchmarks used in this thesis.

Chapter 5 presents two novel policies namely adaptive DRAM insertion policy and set balancing policy for miss rate reduction, detailing their operations and implementation. This chapter investigates the problems of state-of-the-art DRAM insertion policies in detail before presenting the proposed adaptive DRAM insertion policy. At first, a short overview of the different DRAM insertion probabilities is given, which is required to describe the proposed adaptive DRAM insertion policy. Afterwards, the adaptive DRAM insertion policy that can select from multiple insertion probabilities for each application is explained in detail. Subsequently, the online monitoring mechanism to select the suitable insertion probability is explained. Finally, the Set Balancing Policy (SB-Policy) is introduced to reduce conflict misses via an improved DRAM cache set utilization.

Chapter 6 presents the policies for latency reduction demonstrating novel DRAM row buffer mapping policies followed by an innovative Tag-Cache architecture. This chapter investigates the latency trade-offs in architecting the DRAM cache and analyzes the effects of different DRAM row buffer mapping policies on the overall performance. It analyzes the problems of state-of-the-art DRAM row buffer mapping policies before presenting the proposed novel row buffer mapping policies. This chapter also presents the concept of a Tag-Cache – a small and low latency structure – that improves performance by reducing the average cache hit latency. Afterwards, it presents modifications to state-of-the-art DRAM cache controllers to further reduce the DRAM cache hit latency.

The evaluation results for the adaptive DRAM insertion policy and set balancing policies are presented in Chapter 5 and the results for DRAM row mapping policy and Tag-Cache organizations are presented in Chapter 6. In addition, Chapter 7 presents an evaluation for the combined contributions.

Chapter 8 concludes this thesis and provides an outlook to potential future work.

Chapter 2 Background and Related Work

The primary focus of this thesis is to design and optimize DRAM cache based multi-core systems that cover both application and DRAM aware policies in order to improve the overall instruction throughput. This chapter presents the general background for caches followed by a short overview of SRAM caches. Afterwards, it introduces the organization of DRAM cache, detailing its implementation and access mechanisms. Finally, the most recent related work in the area of DRAM cache is reviewed.

2.1 Cache Basics and Terminology

Cache was first introduced in 1965 to bridge the latency gap between high speed cores and slower main memory [132]. The cache [45, 48, 116, 132] is a smaller and faster on-chip memory that stores copies of recently accessed data from frequently used memory locations in order to take advantage of spatial and temporal locality of the applications [6, 45]. Figure 2.1 shows a typical logical cache organization.

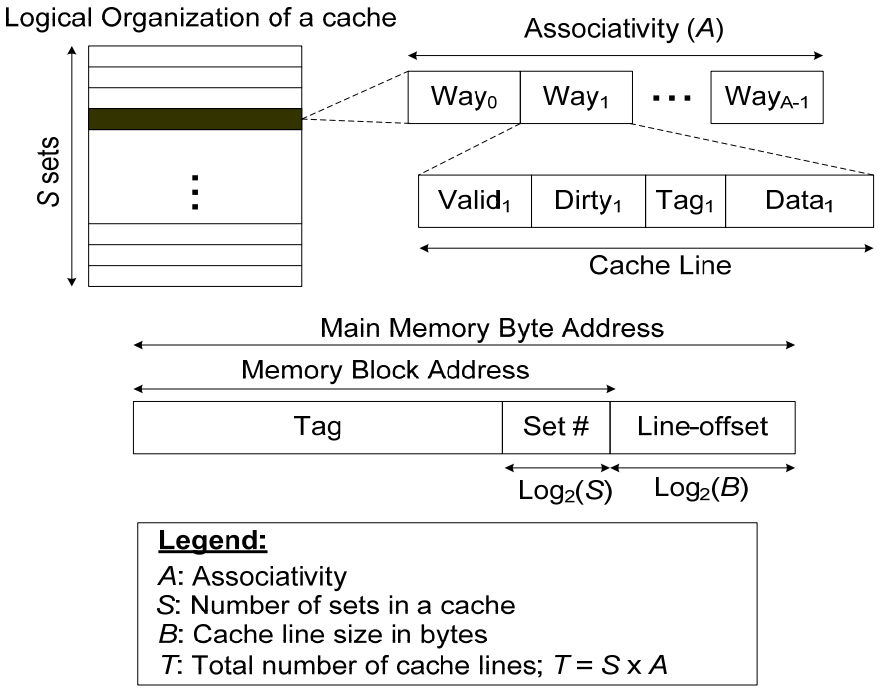


Figure 2.1: A logical cache organization

The following defines some of the basic terms required to understand caches.

Instruction Cache: An instruction cache only holds the instructions of a program and is used for issuing instructions to processor's fetch unit at a faster rate.

Data Cache: A data cache only holds the program's data and is used for fetching data to processor's execution unit at a faster rate.

Unified Cache: A unified cache contains both program's instructions and data.

Logical Cache organization: There are two basic types of logical cache organization namely private and shared cache organizations [17, 22, 35, 97] in a multi-core system.

Private Cache organization: In the private cache organization, each core is provided with its private cache that holds only the recently accessed data requested by its core (called local core).

Shared Cache organization: In a shared cache organization, the whole cache resources are shared by all of the cores providing capacity sharing among different applications.

Block: 'Block' or a 'memory block' is a group of contiguous bytes in main memory (typical block size is 64 byte). A block is identified by bits of the memory address, namely block address as shown in Figure 2.1.

Cache line: Cache line is the basic unit of cache storage [95, 115]. A cache line may contain a single block [77, 102, 112, 113] or multiple blocks [54, 55, 133]. Each cache line consists of a valid bit, dirty bit (not used in instruction caches), tag bits and the data as shown in Figure 2.1.

Set: A set is a group of cache lines. A particular cache set is determined by the 'Set #' field of the main memory address as shown in Figure 2.1.

Associativity (*A*): Typically, cache is composed of a single set or multiple sets [49, 116], where each set contains "*A*" cache lines, i.e. an associativity of *A*. Each block is mapped to a particular cache line of a particular cache set that is determined by the cache organization.

Valid bit: The valid bit of a cache line indicates whether it contains valid (valid bit is 1) or invalid (valid bit is 0) data. All the valid bits of each cache line are set to zero on power up or on a cache reset. Some systems set the valid bit to zero in some special situations. For instance, when a cache line is occupied by multiple cores in a multi-threaded environment, the valid bit of that cache line in the core's private cache is set to zero after its modification by another core. This ensures that the cache lines in the core's private cache are not stale.

Dirty bit: The dirty bit of a cache line indicates whether the cache line has been modified by the processor (dirty bit is 1) or remained unchanged (dirty bit is 0) since it was fetched from main memory.

Tag: Each cache line includes the data itself as well as the tag which is used to identify a particular block (belonging to a cache line) in a particular cache set.

Cache lookup: When the core needs to read/write data from/to a location in memory, it first needs to identify the set (determined by the 'Set #' field; Figure 2.1) followed by tag checking to identify whether a copy of that block resides in the cache set or not. A cache lookup requires searching maximum of *A* valid cache lines (i.e. invalid cache lines are excluded) in the relevant set to identify a cache hit or a miss.

Cache hit: Following a cache lookup, if the requested data is found in the cache (called cache hit), the core immediately performs a read or write operation on the data which is much faster than a memory read or write operation.

Cache miss: Following a cache lookup, if the requested data is not found in the cache (called cache miss), then the data is brought from the next-level cache or main memory and inserted in the cache.

Victim line: After a cache miss, a new cache line is allocated and a resident cache line called victim line is chosen for eviction. The victim line is the candidate for eviction to make room for an incoming cache line which is determined by the cache replacement policy (see Section 2.1.1).

Cache replacement policy: A cache replacement policy decides which cache line should be evicted from the cache set when the set does not have enough space to accommodate a new cache line. A well-known replacement policy for cache is described in Section 2.1.1.

Physical Cache organization: The physical cache organization decides how blocks are mapped to a particular set of a cache. There are three basic types of physical cache organizations [45, 49] namely direct-mapped, fully associative, and set associative cache organizations.

Direct-mapped Cache: In a direct-mapped cache [10, 47, 57, 58], a cache set consists of a single cache line (i.e. $A = 1$; see Figure 2.1). On a cache lookup, a single cache line must be searched in a set to find whether the request results in a cache hit or a miss. The replacement policy for direct-mapped cache is simple as only one cache line is checked for a lookup and the cache line residing in that particular cache set is the victim line to accommodate an incoming cache line.

Fully Associative Cache: A fully associative cache consists of a single cache set (i.e. $S = 1$; see Figure 2.1) and all T cache lines are mapped to this single set. Fully associative cache has a complex cache lookup operation because all T cache lines (see Figure 2.1) in the same set must be searched in parallel to identify a cache hit or a miss.

Set Associative Cache: In a set-associative cache [50, 114, 120, 130], a cache consists of multiple cache sets (i.e. $S \geq 2$) where each set consists of multiple cache lines (i.e. $A \geq 2$). A cache line is first mapped onto a cache set, then the cache line can be placed anywhere in the set. On a cache lookup, 'A' cache lines must be searched in a set to find whether the requested address results in a cache hit or a miss.

Hit latency: Hit latency is the time elapsed to transfer the requested data (includes the time spent to identify a hit/miss) to the core after a cache hit.

Miss latency: determines the time elapsed to transfer the requested data to the core after a cache miss. The miss latency is much higher than the hit latency due to the slower latency of the next cache level or the main memory.

Miss rate: defined as the number of cache misses divided by total number of cache accesses.

Fetch granularity: It is the unit of data transfer between cache hierarchy and main memory [95].

2.1.1 Least Recently Used (LRU) Replacement Policy

A cache replacement policy decides which cache line should be evicted from the cache set when the set does not have enough space to accommodate a new cache line. The two commonly used replacement policies are least recently used (LRU) [56, 90] and Pseudo-LRU replacement policies [111]. This section explains the traditional least recently used replacement policy [56, 90], which is employed in state-of-the-art DRAM cache [77, 78].

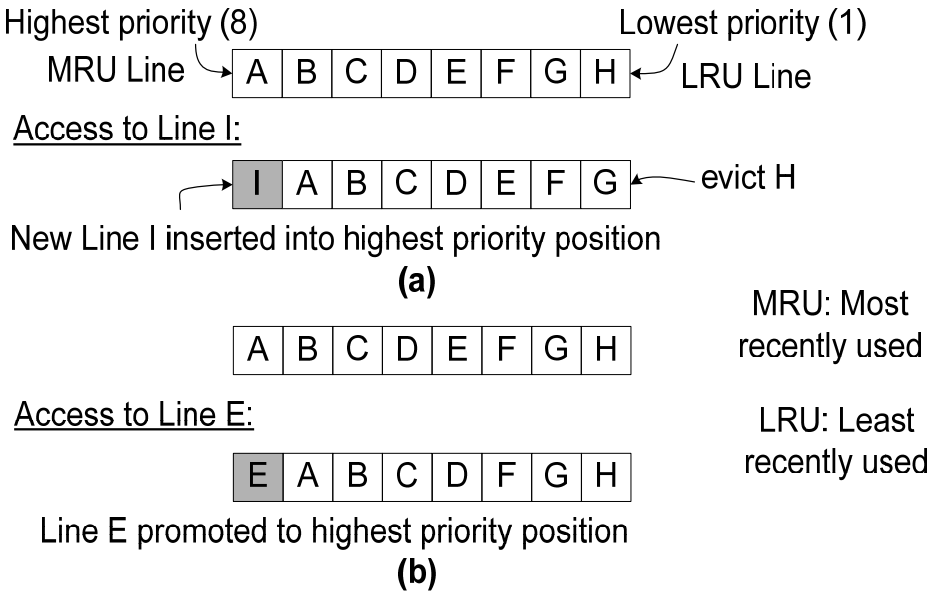


Figure 2.2: Example illustrating LRU replacement policy for an 8-way associative cache (i.e. *A* = 8) (a) Insertion and eviction Policy (b) Promotion Policy

The LRU replacement assigns priority values to each cache line in a cache set. The priority values of the cache line belonging to a particular set are modified after a cache hit or a miss. The LRU replacement policy can be divided into eviction, promotion and insertion policies described as follows:

Eviction Policy: After a cache miss, one of the cache lines in a particular cache set is selected for eviction (i.e. victim line) to make room for an incoming cache line. The *eviction policy* evicts the cache line with the least priority of the relevant set.

Insertion Policy: The insertion policy modifies the priority values of the cache lines belonging to a cache set after a cache miss to that particular set before insertion a new cache line.

Promotion Policy: The promotion policy decides how the priority values of the cache lines belonging to a cache set should be modified after a cache hit to that particular set.

To make room in a set for an incoming cache line, the traditional Least Recently Used (LRU) replacement policy evicts a cache line that is least recently used [56, 90]. Figure 2.2-(a) shows a logical organization (cache lines are shown from left to right in priority order) of a cache set with priority values assigned to each cache line. The cache line with the highest priority is called Most Recently Used (MRU) cache line, while the cache line with the least priority is called Least Re-

cently Used (LRU) cache line. The LRU cache line (cache line H with a priority value of 1) is the candidate for eviction to make room for an incoming cache line on a cache miss. After a cache miss, the insertion policy modifies the priority values of the cache lines in the priority list. In the example shown in Figure 2.2-(a), the new cache line I is assigned a highest priority value of 8 (i.e. cache line I becomes the MRU cache line) while the priority values of the remaining cache lines are decremented by one. The promotion policy enhances the priority value of a cache line on receiving a cache hit. In an LRU based cache, a hit causes the cache line to get the highest priority value. In the example shown in Figure 2.2-(b), cache line E (i.e. cache line E becomes the MRU cache line) gets the highest priority value of 8 after receiving a cache hit.

2.1.2 Multi-level Cache Hierarchies

There are two different design alternatives for organizing caches in multi-core systems: private and shared cache organizations [17, 22, 35, 97]. In the private cache organization, each core is provided with its private cache that holds only the recently accessed blocks requested by its core (called local core). The private cache organization provides **inter-core performance isolation** because the cores are not allowed to insert their requested cache lines into other core's private caches. Inter-core performance isolation means that an application running on one core cannot hurt the performance of concurrently running applications on other cores. However, in a private cache organization, some of the private caches may be under-utilized, whereas others may be severely over-utilized. In a shared cache organization, the whole cache resources are shared by all of the cores providing capacity sharing (to prevent under-utilization and over-utilization of cache resources) among different applications.

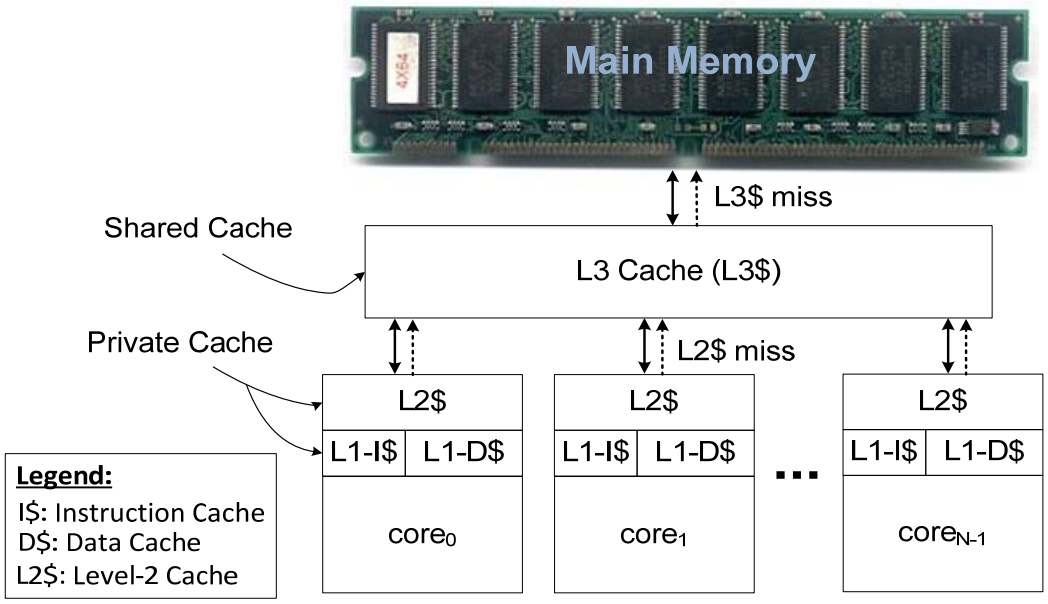


Figure 2.3: A typical three-level multi-core cache hierarchy

The performance of a cache depends on the miss rate and hit latency and there is a tradeoff between them when varying the cache size. On one hand, a larger cache has a reduced miss rate at the cost of increased hit latency (a larger SRAM structure has longer access latency). On the other hand, a smaller cache has a reduced hit latency at the cost of increased miss rate due to lim-

ited cache capacity. To address this, multi-core systems employ multiple levels of cache [8, 45, 96, 128] with small low-latency private caches backed up by high-capacity shared caches.

Figure 2.3 illustrates a typical three-level multi-core cache hierarchy, where fast and small private L1 and L2 caches are employed to satisfy the core's needs in terms of low latency. The larger L3 cache with its relatively high hit latency is shared among all cores to satisfy the cores' needs in terms of reduced miss rate. When a core issues a read or write request to a particular address, the cache hierarchy first check the L1 cache for a hit. If the data is found in the L1 cache (i.e. L1 cache hit), then it is forwarded to the core. However, if the data is not found in the L1 cache (i.e. L1 cache miss), then the L2 cache is checked for a hit or a miss, and so on, before the request is forwarded to main memory.

2.2 DRAM Cache

2.2.1 Physical Realization

As compared to the transistor delay, interconnect delay does not scale at the same rate with each technology node [4, 105, 106]. Die-stacking technology provides a potential solution to address the interconnect scalability problem by reducing communication penalties (by reducing interconnect length) through stacking. Die-stacking technology enables integration of multiple dies with low latency and high bandwidth interconnect. It has experienced tremendous attention during the past decade and has been employed both in industry [52, 61, 62, 66, 70, 92, 109] and academia [20, 34, 39, 53, 54, 55, 75, 77, 78, 74, 76, 102, 113, 112, 133, 139, 143]. Different implementation techniques have been proposed for die-stacking such as vertical stacking [139] and horizontal stacking on an interposer [20]. The Samsung Semiconductors has commercialized stacking multiple DRAM dies with traditional System In Package (SiP) technology [109]. Furthermore, the DRAM industry employs Through Silicon Vias (TSVs) to stack 3D DRAM chips [52, 61, 62, 70].

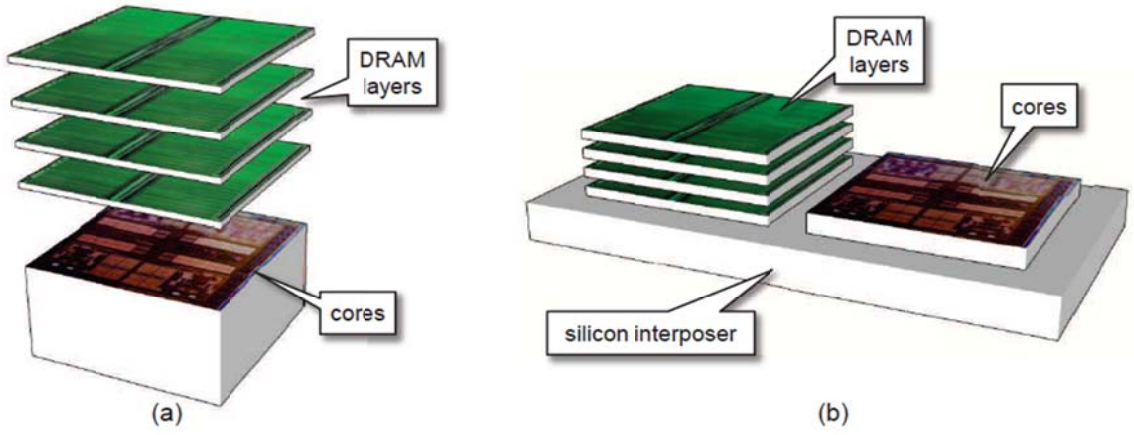


Figure 2.4: Die-stacked DRAM with multi-core chip [77, 78] implemented as (a) a vertical stack (b) silicon interposer

Die stacking technology provides a way to integrate a large amount of DRAM layers with a conventional multi-core processor chip using vertical stacking (Figure 2.4-(a)) or horizontal/2.5D stacking on an interposer (Figure 2.4-(b)). The DRAM layers can either be used as main memory [54, 55, 74, 76, 139, 143] or DRAM caches [53, 75, 77, 78, 102, 112, 113]. Using DRAM layers as main memory [54, 55, 74, 76, 139, 143] demands extensive modification to the operating system required for mapping pages to on-chip and off-chip memory. Using DRAM layers as DRAM caches retains software transparency, supports legacy software because it is not dependent on new versions of operating systems [77, 78]. Therefore, state-of-the-art [53, 75, 77, 78, 102, 112, 113] employs stacked-DRAM layers as DRAM caches because it is not possible to stack the entire system's memory on top of multi-core processor chip due to large working set sizes of complex applications [32].

2.2.2 DRAM Organization

A typical DRAM organization is shown in Figure 2.5-(a). A DRAM subsystem consists of multiple banks where each bank is arranged into rows and columns of DRAM cells, called the DRAM array. When a row is read from the DRAM array, its contents are destroyed which requires the data to be buffered. Therefore, each DRAM bank provides a row buffer (see Figure 2.5-b) that consists of SRAM cells and buffers one row of the DRAM bank (typically 2 to 8 KB). Data in a DRAM bank can only be accessed after it is fetched to the row buffer. Any subsequent accesses to the same row (row buffer hit) will bypass the DRAM array access and the data will be read from the row buffer directly. This concept is referred to as row buffer locality [69, 107]. A request to a DRAM subsystem is sent to the DRAM controller which is responsible to schedule the request to the DRAM bank as shown in Figure 2.5-(a). The DRAM controller consists of:

1. **Request Buffer** that holds a queue of pending requests,
2. **DRAM Read/Write data buffers** that holds the data that is read from/written to the DRAM bank, and
3. **Bank scheduler** that schedules the request to the DRAM bank while prioritizing request to open rows to improve row buffer locality [69, 107].

An access to a DRAM bank involves multiple steps before the read/write operation is performed as shown in Figure 2.5-(c). Assuming that the requested row is not already open (i.e. it is not in the row buffer), an activate (ACT) command is used to open the requested row in the row buffer by reading the data through the sense amplifier. When data is loaded in the row buffer (row access), a read (RD) or write (WR) command is required to access appropriate columns (typical column size is 64 byte) from the row buffer (column access). The access latency of a row buffer miss includes the time to write the contents of the previously opened row (tWR; required because reading a row from the DRAM array destroys the row's contents which needs to be written back into the DRAM array), time to activate the row (tRCD), and the column access time (tCAS). In case of a row buffer hit, only a read or write command is issued, which only requires column access time (tCAS). tCAS is the delay between the moment a DRAM cache controller requests the DRAM cache to access a particular column and the moment the data in the column is available on the DRAM bus (Figure 2.5-c). The DRAM access latency highly depends on whether an access leads to a row buffer hit or a row-buffer miss. It also depends upon the number of requests en-queued in the DRAM request buffer.

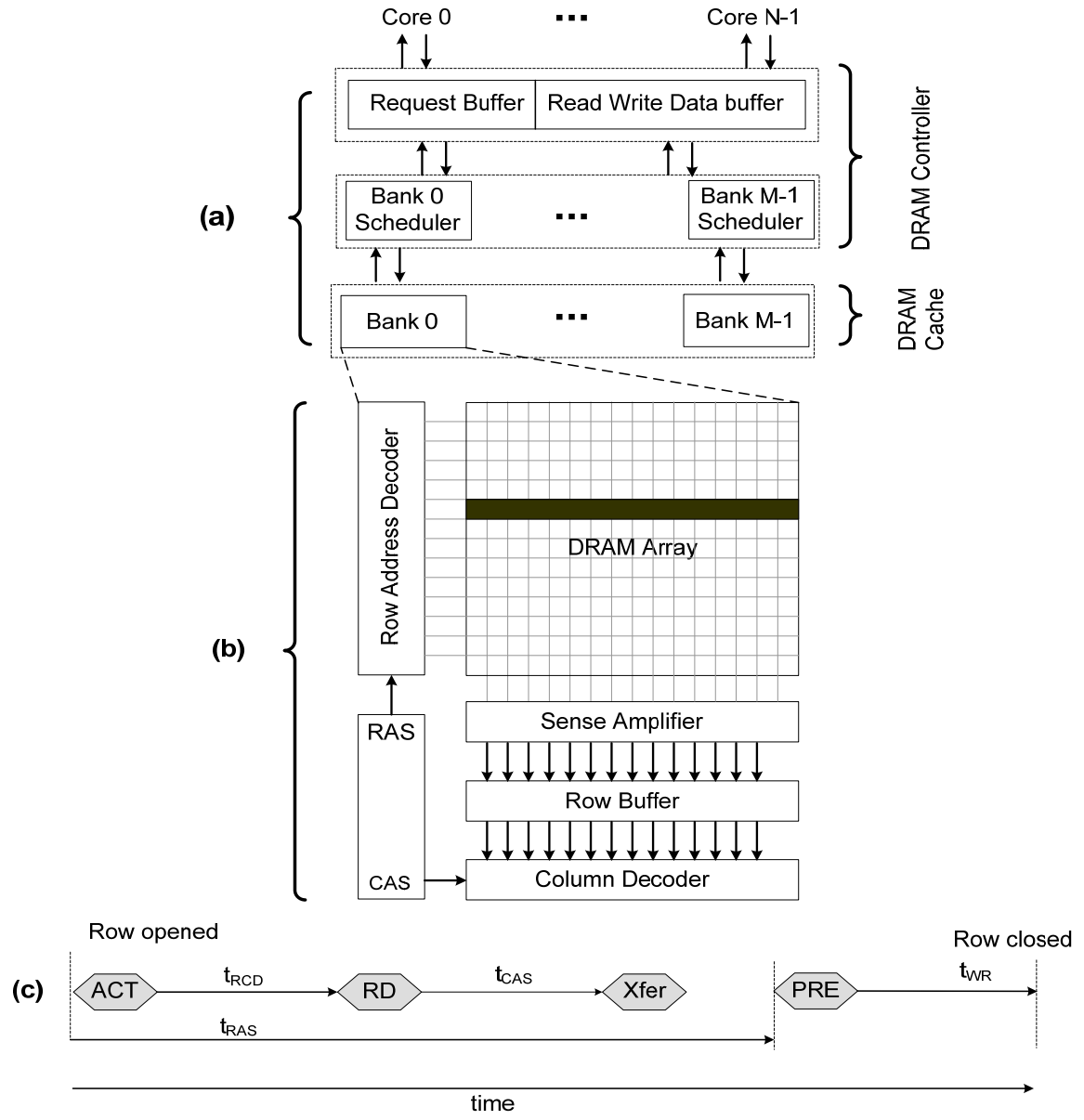


Figure 2.5: (a) DRAM organization (b) DRAM bank organization (c) Timing diagram for a DRAM bank access

2.2.3 Tag-Store Mechanism

A primary challenge in architecting a large DRAM cache is the design of the tag store which is required to identify a cache hit/miss. For instance, a 128MB DRAM cache can store 2^{21} 64-byte blocks ($2^{21} \times 64$ bytes = 134217728 bytes = 128MB), which results in a tag overhead of 12MB ($2^{21} \times 6$ bytes = 12582912 bytes = 12MB) assuming 6 bytes per tag entry [77]. Different design alternatives to architect the DRAM cache tag-store mechanism are discussed as follows:

Tags-In-SRAM: This design approach stores the tags in a separate SRAM tag array which eliminates slow DRAM access if the SRAM tag array indicates a cache miss. For a larger DRAM cache, this approach results in a high die area due to high cost-per-bit of the SRAM tag array.

Tags-In-DRAM: To reduce the die area, state-of-the-art DRAM caches [36, 77, 78, 143] store the tags in the DRAM cache as well (called Tags-In-DRAM approach). They co-locate the tags and data for an entire cache set in the same row. The tags indicate the actual location of the data stored in the row. When a request is made to a particular DRAM cache row that is not present in the row buffer, it is loaded in the row buffer. The row buffer is reserved until both tag and data are read from it. This guarantees a row buffer hit for the data access after the tags are accessed for a hit. The Tags-In-DRAM approach mitigates the storage overhead limitations of the Tags-in-SRAM approach due to low cost-per-bit of the DRAM array. However, it requires a slow DRAM access to identify a hit/miss before the request can be sent to off-chip main memory (in case of a miss) that results in increased miss latency.

2.3 Important Application and DRAM Cache Characteristics

Three important parameters that determines the performance of a DRAM cache based multi-core system are DRAM cache hit latency (D$-HL), miss rate (D$-MR) and miss latency (D$-ML). An ideal DRAM cache should simultaneously reduce all of them. This section describes important application characteristics and DRAM characteristics that have a significant impact on these metrics.

2.3.1 Inter-core Cache Contention

Inter-core cache contention in a shared cache occurs, when one core evicts a useful cache line from another core that is subsequently referenced by that core. State-of-the-art DRAM cache [77, 78, 102] suffers from inter-core cache contention because they do not consider the cache access pattern of complex applications. Inter-core cache contention primarily occurs, when ‘thrashing applications’ run concurrently with ‘non-thrashing applications’. An applications is said to have *thrashing* behavior if it exhibits poor locality that generates a large number of *zero-reuse* cache lines (i.e. cache lines that are inserted but not used before they get evicted) [51, 137, 138]. Thrashing applications have a working set size greater than the available cache size and thus get negligible benefits from the available cache capacity because the cache is not efficiently utilized due to many *zero-reuse* cache lines. However, they have a high access rate relative to the access rate of non-thrashing applications. This means, they insert a large number of cache lines in the shared cache, and as a result, they quickly evict *highly-reuse* cache lines (i.e. cache line that is reused in the near future) from other applications. This increases the contention between thrashing and non-thrashing applications.

Figure 2.6 illustrates a cache servicing a mix of thrashing and non-thrashing applications with accesses (shown in capital letters J, K, L etc.) from a thrashing application. The cache initially contains some useful highly-reuse cache lines from non-thrashing applications (shown in grey boxes with letters A, C, E etc.). On a cache miss, an incoming cache line is inserted into the most recently used (MRU) position while the cache in the least recently used (LRU) position is the candidate for eviction to make room for the incoming cache line. As the thrashing application inserts more cache lines into the cache, the highly-reuse cache lines are evicted. Subsequent accesses to these highly-reuse cache lines (letters A, C, E etc.) will result in cache misses, hereby affecting performance. State-of-the-art DRAM cache [77, 78] always insert both *zero-reuse* and

highly-reuse cache lines in the DRAM cache, which increases inter-core cache contention and leads to an increased DRAM cache miss rate.

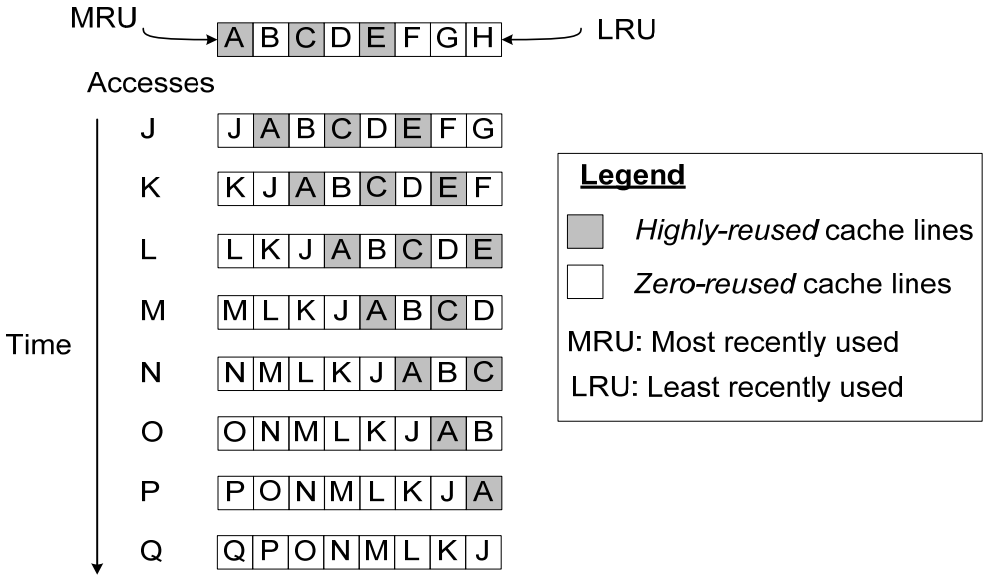


Figure 2.6: Example showing inter-core cache contention between thrashing and non-thrashing applications

2.3.2 Inter-core DRAM Interference in the DRAM cache

Simultaneous requests to the DRAM cache from multiple applications executing on a multi-core system can affect system performance in unpredictable ways and it can lead to inter-core DRAM interference among the cores, which results in poor system performance due to increased DRAM cache hit latency and DRAM cache miss rate.

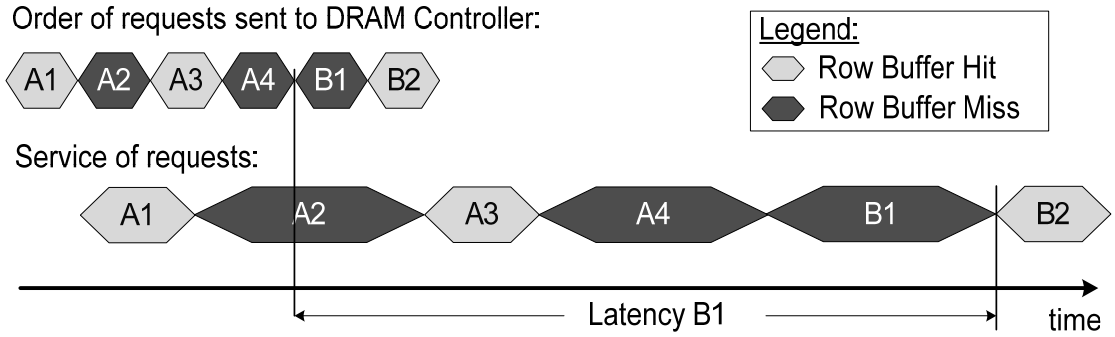


Figure 2.7: Example showing inter-core interference at the DRAM bank

Figure 2.7 presents an example showing the hit latencies for cache requests from applications A and B running on two different cores with a DRAM cache. Application A has a high cache access rate with thrashing behavior, while application B has a low cache access rate with non-thrashing behavior. The *highly-reuse* requests from application B (B1 and B2) in Figure 2.7 arrives at the DRAM controller (Figure 2.5-a) when the DRAM bank is scheduled to service the large number of zero-reuse requests from application A (A1, A2, A3, and A4). As a result, cache

requests from application B are significantly delayed by this so-called inter-core DRAM interference which can degrade the performance of application B. Inter-core DRAM interference is primarily due to unnecessary fill requests (i.e. data is filled into the cache for the first time) from thrashing applications. These unnecessary fill requests from thrashing applications may delay the critical (read or write) requests from non-thrashing applications. The contention between critical and unnecessary fill requests increases the amount of time needed to service critical requests, which increases the DRAM cache hit latency. State-of-the-art DRAM cache [77, 78, 102, 112, 113, 133, 142] suffers from increased inter-core DRAM interference that leads to increased DRAM cache hit latency because they always insert data into DRAM cache, independent on whether it is highly-reuse or zero-reuse data. Furthermore, they incur increased DRAM cache miss rate due to inter-core cache contention between thrashing and non-thrashing applications.

2.3.3 Impact of Associativity

Associativity (*A*) is a trade-off between hit latency and miss rate [45, 49]. Each set in a high associative cache (i.e. larger *A*) contains more cache lines than a set in a small associative cache (i.e. smaller *A*). There is less chance of a conflict between two memory blocks in a high associative cache compared to a small associative cache. Thus, increasing associativity has the advantage of reducing conflict misses, which reduces the miss rate. However, the miss rate reduction for a high associative cache comes at the cost of increased hit latency due to high tag lookup latency because a large number of tag entries needs to be accessed to locate the requested data within the set to identify a cache hit or miss.

2.3.4 Impact of Row Buffer Mapping

Typically a DRAM cache is composed of multiple banks, where each bank is associated with a row buffer (as shown in Figure 2.8). *Row buffer mapping* is the method by which blocks from main memory are mapped to a particular set of a particular row of a particular bank (each bank is provided with a row buffer as shown in Figure 2.8). It has a significant effect on the row buffer hit rate, which directly affects the DRAM cache hit latency. The row buffer hit rate is high when more adjacent blocks are mapped to the same row buffer, as it exploits the programs' locality that adjacent blocks are likely to be accessed in the near future. However, the primary disadvantage of mapping more adjacent blocks to the same row buffer is that it results in a reduced miss rate due to non-uniform set utilization (i.e. some sets are under-utilized while others are severely over-utilized).

2.3.5 Impact of cache line size

As described in Section 2.1, a cache line is the basic unit of cache storage. A large cache line size has a significant impact on the row buffer hit rate, hit latency, miss rate and main memory latency. A simple way to improve the row buffer hit rate is to increase the cache line size (i.e. a cache line now contain multiple blocks that are stored in the same row). This improvement occurs because a larger cache line size takes advantage of spatial locality by fetching multiple blocks (typical block size is 64 bytes) at the same time while only one block is requested by the core. The improvement in the row buffer hit rate comes at the cost of significantly increased memory bandwidth consumption that results in significantly higher main memory latency because multiple 64 bytes blocks must be transferred through a limited size memory channel (typical memory

channel size is 8/16 bytes). A larger cache line size will reduce the DRAM cache miss rate for applications with high spatial locality if such pre-fetched blocks are subsequently accessed. However, a large cache line size will increase the DRAM cache miss rate for applications with low spatial locality because the pre-fetched blocks will never be reused, which leads to inefficient cache space utilization.

2.4 State-of-the-art DRAM Cache

Existing DRAM cache designs can be classified into two categories based on the cache line sizes: block-based and page-based. Block-based DRAM caches [75, 77, 78, 112, 102, 113] use a small cache line size (i.e. 64 byte cache line size). In contrast, page-based DRAM caches [54, 55, 112, 133] use a large cache line size (i.e. 1KB/2KB cache line size). The summary and the distinction of the block and the page based DRAM cache designs is presented in Section 2.4.6.

The concepts proposed in this thesis are compared with the most recently proposed block-based designs namely LH-Cache (details in Section 2.4.1) and Alloy-Cache (details in Section 2.4.3). The page-based DRAM cache designs are discussed in Section 2.4.4. This section assumes 2KB row size and 64-byte cache line size for qualitative comparisons.

2.4.1 LH-Cache [77, 78]

Recent work, namely LH-Cache [77, 78], employs a block-based DRAM cache design. It stores the tags along with the cache lines of a set in the same row, as shown in Figure 2.8. To overcome the latency disadvantage of this Tags-In-DRAM approach (see Section 2.2.3), LH-Cache uses a low overhead SRAM-based structure named as MMap$ (MissMap cache) that accurately determines whether an access to the DRAM cache will be a hit or a miss (see Figure 2.8 and Figure 2.11). It incurs a storage overhead of only 2MB SRAM compared to the unacceptable high 12MB SRAM overhead that would be required for the Tags-in-SRAM approach (see Section 2.2.3). Details on the internal functioning of the MMap$ are given in Section 2.4.2.

The DRAM cache row organization for the LH-Cache is shown in Figure 2.8, where each DRAM row consists of one cache set. The DRAM cache row is divided into T tag blocks (T = 3/6/12 for 2KB/4KB/8KB row size) and A cache lines (i.e. A = 29/58/116 for 2KB/4KB/8KB row size). After a hit is detected by the MMap$, the row buffer is reserved until both tag and cache line are read from it. This guarantees a row buffer hit for the cache line access after the tag blocks are accessed and thus it reduces the hit latency. However, LH-Cache must first read the T tag blocks before accessing the cache line, which increases the hit latency.

Figure 2.9 shows the LH-Cache row buffer hit and miss latencies for a DRAM cache hit after a miss in the SRAM cache of the previous level is detected. LH-Cache requires 10 cycles to access the MMap$. If the DRAM cache hit will also hit in the row buffer (i.e. the requested row is already open due to a previous request), then LH-Cache requires 18 cycles for CAS (to access the three tag blocks from a particular row buffer), 12 cycles to transfer the three tag blocks (192 bytes) on the 16 bytes wide DRAM cache bus, 1 cycles for the tag check, 18 cycles for CAS (to access the cache line from the row buffer), and 4 cycles to transfer the cache line (64 bytes). If the requested row is not located in the row buffer (row buffer miss), it requires additional 18 cycles for ACT (row activation) compared to the row buffer hit latency. The DRAM cache hit la-

tency in LH-Cache is 63 cycles for a row buffer hit and 81 cycles for a row buffer miss as shown in Figure 2.9.

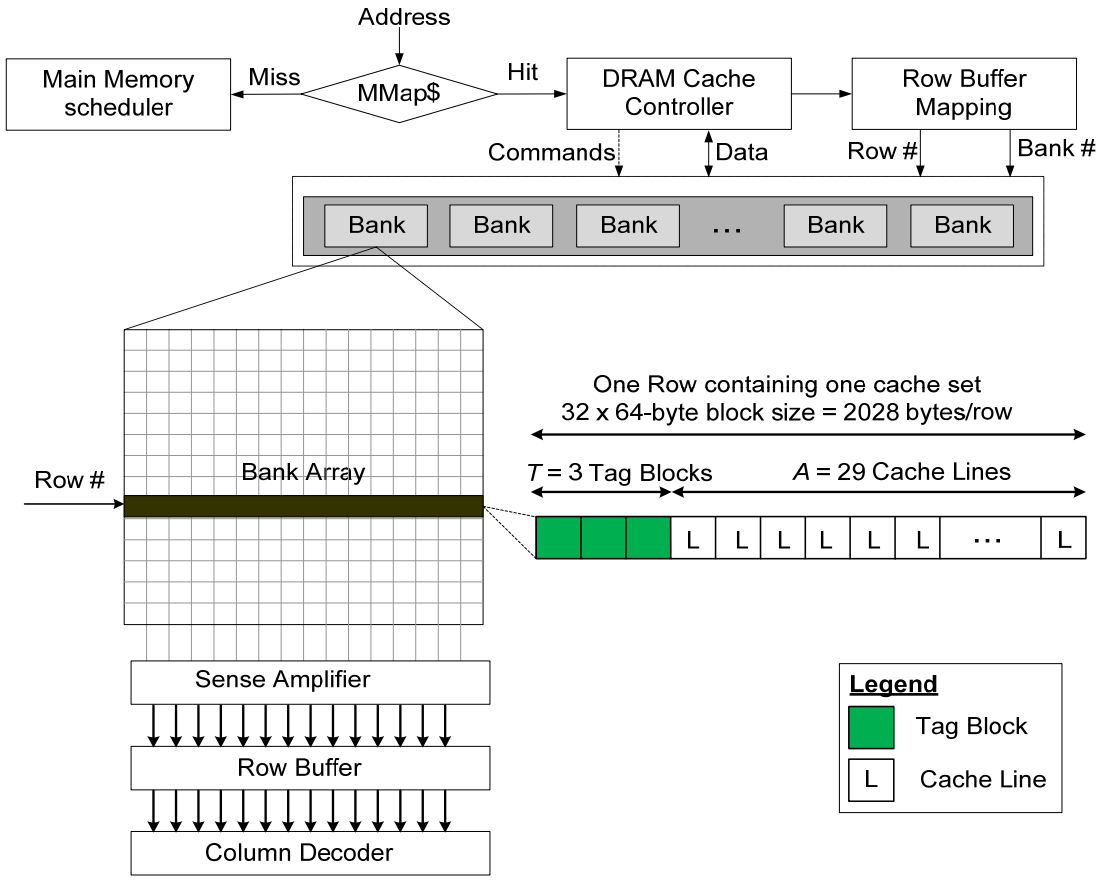


Figure 2.8: LH-Cache Cache Organization with Tags-In-DRAM for 2KB row size [77, 78]

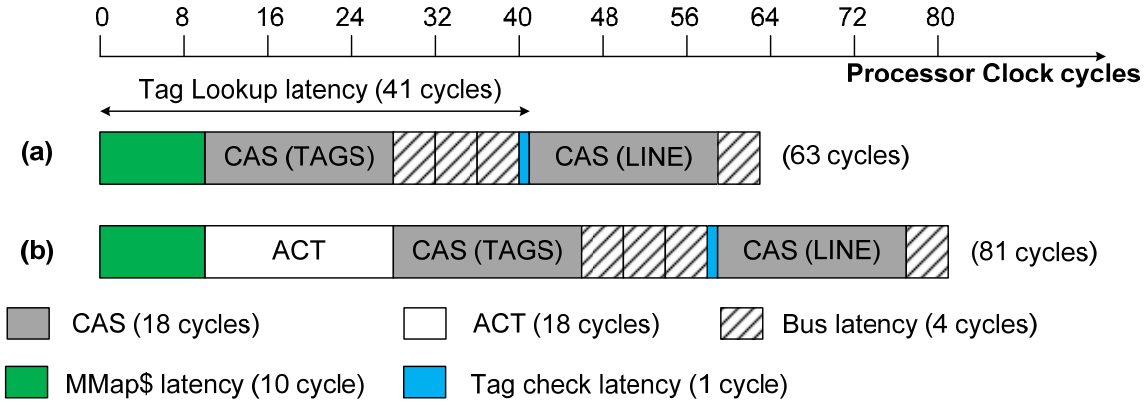


Figure 2.9: LH-Cache (a) row buffer hit latency (b) row buffer miss latency for 2KB row size with *T* = 3 and *A* = 29 (see Section 4.2 for details of DRAM cache timing parameters)

Figure 2.10 illustrates how LH-Cache maps blocks from main memory to the row buffers of banks and to the rows within a bank. The row buffer associated with a particular bank (indicated by RB-i field) and the DRAM cache row number within a bank (indicated by the "Row#" field) is determined by the main memory address. In LH-Cache, spatially close memory blocks are

mapped to different row buffers (e.g. memory blocks 0, 1, and 2 are mapped to RB-0, RB-1, and RB-2 respectively; memory block 64 is mapped to RB-0 again). Thus, the probability of temporally close accesses going to the same row is very low. This result in a reduced row buffer hit rate that leads to increased DRAM cache hit latency, because a row buffer miss has a higher latency than a row buffer hit.

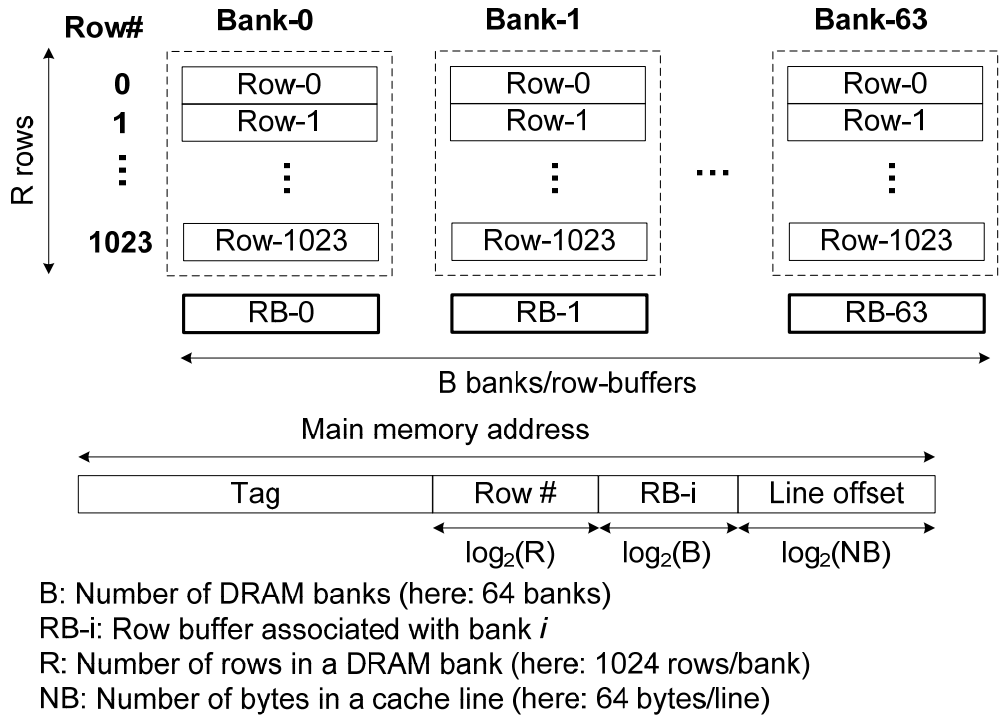


Figure 2.10: LH-Cache row buffer mapping policy

2.4.2 MMap$ Organization

The design of the MMap$ is illustrated in Figure 2.11 that precisely determines whether an access to the DRAM cache will be a hit or a miss. If the MMap$ identifies a hit, the request is sent to the DRAM cache scheduler (see Figure 2.8). A MMap$ miss (i.e. data is not available in the DRAM cache) makes DRAM cache misses faster because the DRAM cache does not need to be accessed to determine a DRAM cache miss. MMap$ logically partitions the main memory into consecutive segments of constant size. A segment is the basic unit of MMap$ storage and is a group of contiguous blocks in main memory (typical segment size is 4KB byte). Each MMap$ entry represents a segment (this thesis uses a segment size of 4KB similar to state-of-the-art [77, 78]) and tracks the presence of the blocks (this thesis uses a block size of 64 bytes similar to state-of-the-art [77, 78]) of that segment. Therefore, each MMap$ entry contains a tag (*Seg-Tag*; see "MMap$ tag-array" in Figure 2.11) corresponding to the address of the tracked memory segment and a bit vector (*Seg-BV*; see "MMap$ Data array") with one bit per block that stores the hit/miss information of the block. If a *Seg-BV* entry is 1, then the corresponding block within the segment is present in the DRAM cache, otherwise it is absent.

On a MMap$ access, the set index field (see Figure 2.11) of the requested physical address is used to index a MMap$ set in the MMap$ tag-array. All tag entries within that MMap$ set (an

associativity of 4 is shown in Figure 2.11) are then compared to the *Seg-Tag* field of the physical address to identify a segment hit/miss. A segment miss implies that the requested block is absent in the DRAM cache. Following a segment hit, the vector index field of the requested physical address is used to index the *Seg-BV* entry of the hit-segment to identify a block hit/miss.

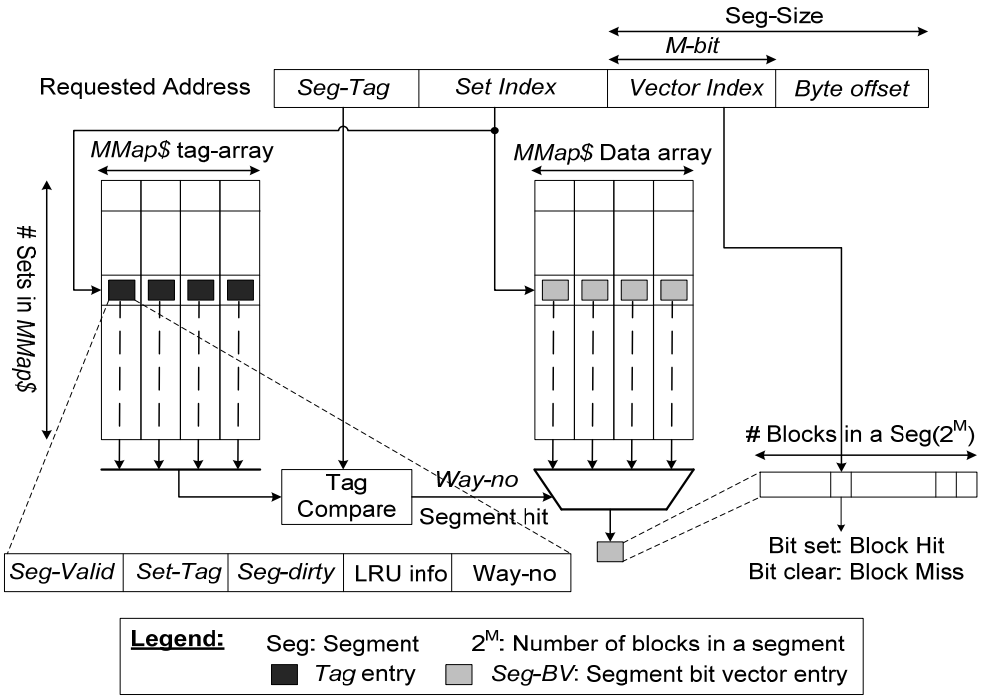


Figure 2.11: MMap$ for DRAM cache hit/miss detection [77, 78]

When a block b_i is evicted from the DRAM cache, then *Seg-BV* entry *i* (i.e. the i^{th} bit of *Seg-BV*) of the MMap$ segment entry *S* to which b_i belongs is cleared. When a block b_i is inserted into the DRAM cache, then *Seg-BV* entry *i* of segment entry *S* needs to be set. If no segment entry for *S* exists in the MMap$, a new entry is allocated for it and only its *Seg-BV* entry *i* is set. To allocate a new MMap$ entry, a victim segment is chosen using the least recently used (LRU) policy. If some *Seg-BV* entries of the victim segment *S* were set (i.e. some of its blocks are present in the DRAM cache), then all corresponding blocks must be evicted from the DRAM cache. This guarantees that the MMap$ always accurately determines whether an access to a DRAM cache will be a hit or a miss.

2.4.3 Alloy-Cache [102]

For a DRAM cache access, LH-Cache reads *T* tag blocks (requires 192 bytes for 2KB row size; i.e. $T = 3$) and one cache line (64 bytes) through a limited size DRAM channel (16 byte), which leads to increased DRAM cache hit latency for LH-Cache as shown in Figure 2.9. To reduce the DRAM cache hit latency, the Alloy-Cache [102] unifies tag and data of a cache line into a single entity called TAD (Tag And Data) as shown in Figure 2.12. Each TAD entry (8 bytes for the tags and 64 bytes for cache line) represents one set of the direct mapped cache (i.e. $A = 1$).

Alloy-Cache reduces the DRAM cache hit latency because for each DRAM cache access, it reads 72 bytes (incurs bus latency of 5 clock cycles for transferring 72 bytes on a 16 byte chan-

nel) instead of reading 256 bytes (192 bytes for the tags and 64 bytes for cache line) required for LH-Cache for 2KB row size. Furthermore, Alloy-Cache requires a single DRAM cache access to get the unified TAD entry instead of having separate accesses required for tags and cache line as required for LH-Cache. As a result, Alloy-Cache significantly reduces the DRAM cache hit latency compared to the LH-Cache. Figure 2.13 shows latencies for a row buffer hit and miss in Alloy-Cache. The row buffer hit latency is 34 clock cycles and the row buffer miss latency is 52 clock cycles (18 additional cycles ACT) as illustrated in Figure 2.13. Despite the latency advantage, Alloy-Cache incurs increased DRAM cache miss rate and DRAM cache miss latency compared to LH-Cache due to increased conflict misses, because it employs a direct mapped cache organization. The increased conflict misses result in increased contention in the memory controller, which leads to increased DRAM cache miss latency compared to LH-Cache.

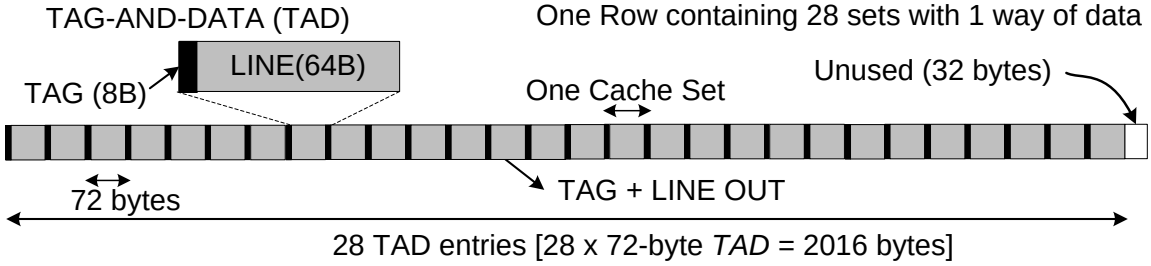


Figure 2.12: DRAM cache row organization used by Alloy-Cache for 2KB row size

Alloy-cache maps 28 consecutive memory blocks to the same DRAM row buffer (e.g. memory block-0, block-1, …, block-27 are mapped to RB-0). Thus, the probability of temporally close accesses going to the same row is very high in Alloy-Cache. This results in an increased row buffer hit rate compared to LH-Cache, which further reduces DRAM cache hit latency.

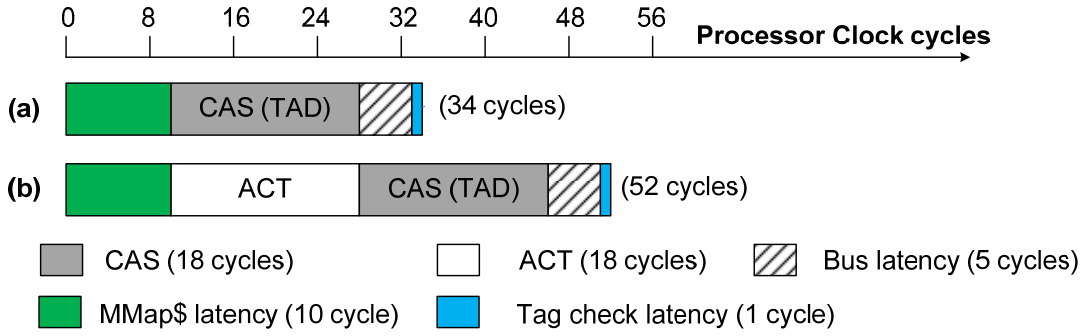


Figure 2.13: Alloy-Cache (a) row buffer hit latency (b) row buffer miss latency (see Section 4.2 for details of DRAM cache timing parameters)

2.4.4 Further Related Work in block-based DRAM Caches

The work in [75] investigated ways of reducing the DRAM cache miss rate by organizing each DRAM cache set as multiple queue structures designed for a 4-core system and 64-way associative DRAM Last-Level-Cache. However, that work is impractical for larger number of cores and smaller associative DRAM cache, because for each DRAM cache set it requires maintaining

$N + 2$ queues (one per core and two shared queues) for an N-core system. In addition, they stores the tags in an SRAM array that incurs significant area overhead for larger DRAM cache.

Ref. [112] proposes a self-balancing dispatch (SBD) mechanism that adaptively dispatches requests either to DRAM cache or to main memory, depending on the instantaneous queuing delays at the DRAM cache and main memory.

Ref. [113] focuses on improving the reliability by presenting a general approach for enabling high-level as well as configurable levels of reliability, availability and serviceability for die-stacked DRAM caches.

2.4.5 Page-based DRAM Caches

Page-based DRAM caches [54, 55, 133] use a large cache line (i.e. 1KB/2KB cache line) and a large fetch granularity (i.e. 1KB/2KB). Note that cache line is the basic unit of cache storage and fetch granularity is the unit of data transfer between cache hierarchy and main memory. The primary advantage of large cache line size/fetch granularity is that they exploit programs' spatial locality, which results in a reduced hit latency via improved row buffer hit rate. Unfortunately, the larger fetch granularity comes at the cost of significant memory bandwidth consumption due to excessive prefetching that results in a significantly high main memory latency. Furthermore, it exacerbates the performance of memory intensive applications with limited data reuse and low spatial locality. Though the large cache line size may improve the performance of less memory intensive applications with high spatial locality, it comes at the cost of an increased miss rate for memory intensive applications with reduced spatial locality. Another drawback of a large cache line size is that it suffers from inefficient resource allocation because not all blocks within the cache line are used prior to page evictions, which leads to reduced efficiency. Also, the cache miss rates in multi-core systems for large cache line size generally limits the performance due to reduced spatial locality and false sharing [126].

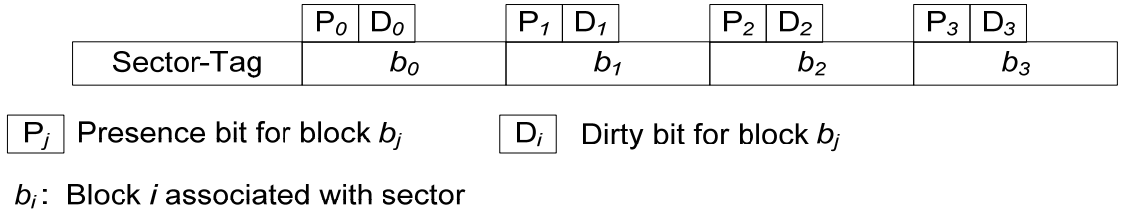


Figure 2.14: Sector organization with 4 blocks per sector

Ref. [142] proposes a sector cache organization to reduce the tag storage overhead for a large DRAM cache. The basic unit of storage in [142] is defined as a sector, which is divided into multiple blocks as shown in Figure 2.14. Each sector is associated with a tag (Sector-Tag) to determine whether a particular sector is present in the DRAM cache (sector hit) or absent (sector miss). Each block b_i of a sector is provided with a presence bit P_i and thus only some of the blocks of a sector need to be present. When a block b_i is inserted into the DRAM cache, then P_i of the sector entry S to which b_i belongs is set. If no entry for S exists in the DRAM cache, a new entry is allocated for it and only the presence bit of the requested block b_i (i.e. P_i) is set. To allocate a new sector entry, a victim sector is chosen using the least recently used (LRU) policy. If

some D_i bits (dirty bit associated with block b_i) of the victim segment S were set, then the dirty blocks from the victim segment must be written back to main memory.

Ref. [142] employs a large sector size (i.e. 256 bytes/512 bytes) together with a small block size (i.e. 64 bytes). In contrast to traditional page-based designs [54, 55, 133], it fetches data at small granularity (i.e. 64 bytes) in order to mitigate the excessive prefetching problem of page-based designs. It proposes to store the sector tags along with the blocks and uses hit/miss predictor to predict whether the requested block resides in DRAM cache or not. Each DRAM row can hold blocks from K distinct sectors (typical value of K=4) in the physical memory. For the core size (8-cores) and row size (2KB) considered in this thesis, this results in a tag overhead of 51 bytes per row (K=4 and sector size = 512 bytes).

The sector cache organization [142] has the following major disadvantages compared to the recently proposed block-based LH-Cache design [77, 78]. First, it suffers from inefficient resource allocation due to internal fragmentation because it reserves the space for the entire sector, while some of the blocks belonging to a sector may not be referenced before sector eviction. Second, reserving one 64 byte column (each 2KB DRAM row contains 32 64-byte columns) for the sector tags (i.e. they require 51 bytes for the tags) will make the other 31 columns available for blocks (of a sector). This requires non-trivial changes to row mapping because it requires the sector size to be power of 2. Third, it will incur a significantly high DRAM cache miss rate compared to the LH-Cache because it allows blocks from 4 distinct sectors to be mapped to a particular row. In contrast, LH-Cache allows 29 distinct blocks to be mapped to a particular row (i.e. they may belong to 29 distinct sectors).

2.4.6 Distinction with the state-of-the-art

There has been a considerable amount of research on DRAM caches [54, 55, 77, 78, 102, 112, 113, 133, 142] and this thesis compares the proposed policies with the most recent state-of-the-art DRAM cache designs [77, 78, 102]. Table 2.1 provides a comparisons between different DRAM cache designs with respect to the most important parameters (DRAM cache hit latency, DRAM cache miss rate, and main memory latency), application and DRAM characteristics. An ideal DRAM cache should simultaneously reduce the DRAM cache hit latency (depends upon row buffer hit rate, associativity and tag lookup latency), DRAM cache miss rate (depends upon associativity, inter-core cache contention and inter-core DRAM interference), and main memory latency (depends upon DRAM cache miss rate and memory bandwidth consumption). At the same time, it should provide efficient resource allocation with a low implementation overhead.

The DRAM cache proposed in this thesis employs a block based design that provides a low-overhead tag-store mechanism (i.e. storing tags in DRAM cache) and reduced conflict misses (via high associativity). It retains the benefits of block-based designs [77, 78, 102, 112, 113] by employing small cache line size (to mitigate internal fragmentation compared to sector cache organization [142]) and small fetch granularity (to reduce memory bandwidth consumption compared to page-based designs). For the rest of this thesis, the terms ‘cache line’, and ‘block’ are used interchangeably because this thesis employs block based design for all cache levels. Table 2.2 illustrates an overview of the proposed policies that mitigates the following major drawbacks of LH-Cache [77, 78, 112] in specific and block-based designs [77, 78, 102, 112, 113] in general.

	[77, 112]	[102, 113]	[142]	[54, 55, 133]	Proposed
Cache line size	Block	Block	Sector	Page	Block
Fetch granularity	Block	Block	Block	Page	Block
Associativity (*A*)	High (29)	Low (1)	Medium (4)	Medium (16)	High (30)
Internal Fragmentation	No	No	Yes	No	No
Requires operating system modification	No	No	No	Yes	No
Low overhead Tag-store	✓	✓	✓	×	✓
Reduced inter-core cache contention	×	×	×	✓	✓
Reduced inter-core DRAM interference	×	×	×	✓	✓
High row buffer hit rate	×	✓	✓	✓	✓
Reduced memory bandwidth consumption	✓	✓	✓	×	✓
Low DRAM cache miss rate	✓/×*	×	✓/×*	✓/×*	✓
Low DRAM cache hit latency	×	✓	✓	✓	✓
Low memory latency	✓	×	✓	×	✓

Table 2.1: Comparisons between different DRAM cache designs for a 2KB row size
***Miss ratio depends upon cache access pattern of concurrently running applications on a multi-core system**

DRAM Last-Level-Cache Polices	Advantages
Adaptive DRAM Insertion Policy (Chapter 5)	reduces inter-core cache contention reduces inter-core DRAM interference
Set Balancing Policy (Chapter 5)	provides uniform access distribution
Row Buffer Mapping Policy (Chapter 6)	improves row buffer hit rate
DRAM Tag-Cache (Chapter 6)	reduces tag lookup latency

Table 2.2: Advantages of the proposed Policies

Existing block-based designs suffers from increased inter-core cache contention that leads to increased DRAM cache miss rate via increased conflict misses. They lead to increased DRAM cache hit latency via increased inter-core DRAM interference. They exhibit a non-uniform distribution of accesses across different DRAM cache sets that leads to increased DRAM cache miss rate. LH-Cache has a reduced row buffer hit rate due to reduced spatial locality that leads to increased DRAM cache hit latency. LH-cache further worsens DRAM cache hit latency due to high tag lookup latency because it always reads the tags from the DRAM cache.

To mitigate inter-core cache contention and inter-core DRAM interference, this thesis proposes an adaptive DRAM insertion policy (details in Chapter 5) that is flexible enough to be applied to any DRAM cache organization and replacement policy. To reduce conflict misses via

improved set utilization, this thesis proposes a set-balancing policy (details in Chapter 5) that provides a uniform access distribution across DRAM cache sets. To reduce hit latency via an improved row buffer hit rate, this thesis proposes a DRAM row buffer mapping policy (details in Chapter 6) that exploits data access locality in the row buffer with a slight increase in miss rate. To further reduce the hit latency via reduced tag lookup latency, this thesis proposes a low-latency SRAM structure namely DRAM Tag-Cache (details in Chapter 6) that can quickly determine whether an access to the DRAM cache will be a hit or a miss. A short overview of the proposed policies is presented in Chapter 3.

Chapter 3 Overview of Proposed Policies

As the industry continues to increase the number of cores, multi-level caches [8, 45, 96, 128] are increasingly becoming the trend of multi-core systems in order to mitigate the widening gap between processor and memory speed [12, 60, 108, 131, 136, 144]. Furthermore, the advent of on-chip DRAM caches (thanks to die stacking technologies) has led to the evolution of multi-level SRAM/DRAM cache hierarchies comprised of increasing cache sizes and latency at each level. State-of-the-art multi-level SRAM/DRAM cache hierarchies are equipped with low latency smaller caches backed up by high latency larger caches. These hierarchies contain fast and small private L1 and L2 caches to satisfy the core's need in terms of low latency. On the other hand, they employ larger L3 SRAM and L4 DRAM caches, which are shared among all cores to reduce high latency off-chip memory accesses.

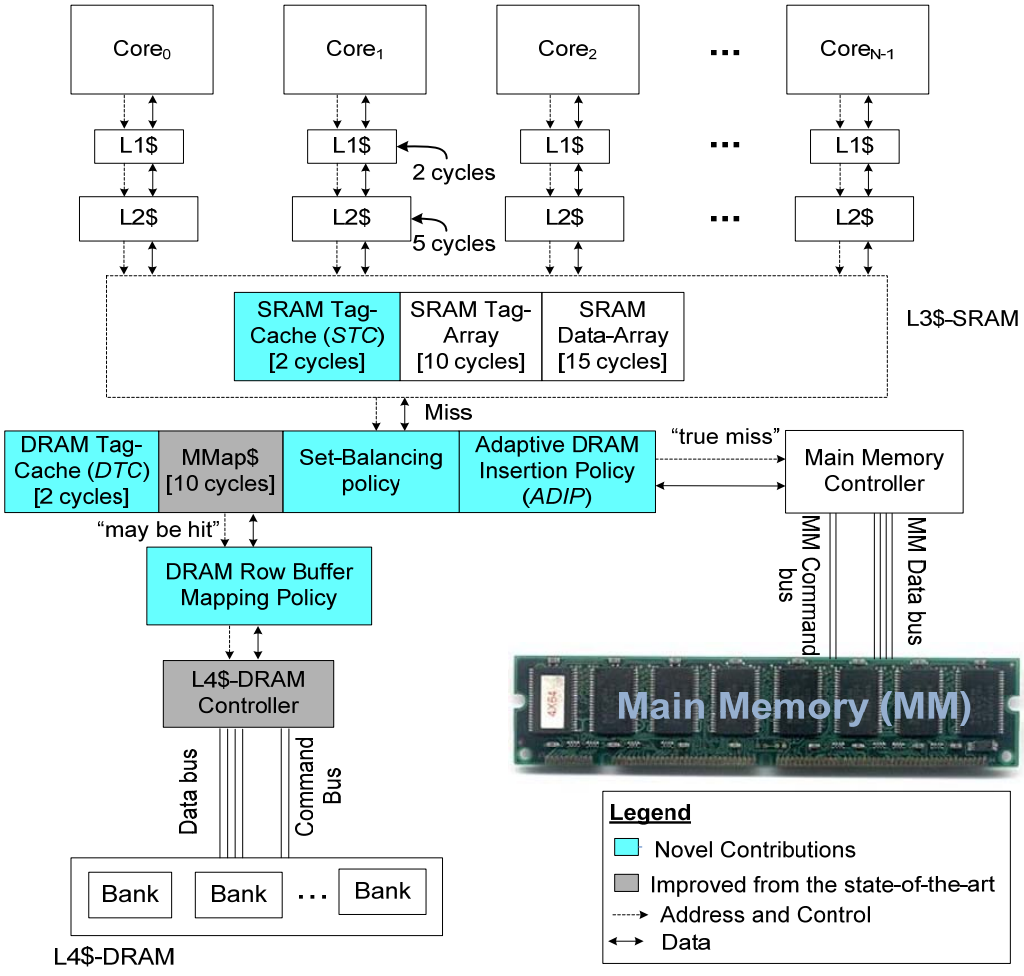


Figure 3.1: Proposed SRAM/DRAM cache hierarchy for an N-core system

Multi-core systems present new challenges in the design of an ideal cache hierarchy that not only reduces the miss rate and hit latency for each cache level, but also reduces the main memory latency. Note that the main memory latency can be reduced by decreasing the number of requests to the main memory which in turn reduces the average waiting latency per request. This chapter

provides a short overview of the proposed cache policies applied on top of L4 DRAM cache that substantially improve the performance via reduced L4 hit latency and L4 miss rate compared to state-of-the-art [77, 78, 102]. Figure 3.1 shows the organization of the proposed SRAM/DRAM cache organization, highlighting the proposed novel contributions. Similar to state-of-the-art [36, 38, 77, 78], this thesis stores the tags in the DRAM cache and employ a MMap$ (details in Section 2.4.1) to identify DRAM cache hit/miss. The novel contributions of the proposed SRAM/DRAM cache organization are highlighted in Figure 3.1 and outlined in the following.

3.1 Adaptive DRAM Insertion Policy

On a cache miss, state-of-the-art policies always insert data into the DRAM cache, independent of whether it is *highly-reuse* (i.e. data that is reused in the near future) or *zero-reuse* data (i.e. data that is never reused before it gets evicted). This leads to inefficient DRAM resource allocation. To address this problem, this thesis presents an *adaptive DRAM insertion policy* (*ADIP*) that exploits the fact that some applications (so-called thrashing application) often fetch *zero-reuse* data that does not contribute to cache hits because it is not accessed again. The proposed policy is based on the idea of restricting the number of *zero-reuse* data insertions from thrashing applications into DRAM cache and it decides at runtime whether data that is fetched from off-chip memory shall be inserted into DRAM cache or not. Figure 3.2 shows the high level overview of the adaptive DRAM insertion policy for an SRAM/DRAM cache hierarchy. After an L4 DRAM cache miss, data is brought from memory and inserted into L1, L2, and L3 SRAM caches. The data may or may not be inserted additionally in L4 DRAM cache, which is determined by the proposed adaptive DRAM insertion policy. Existing DRAM cache hierarchies [77, 78, 102, 143] always fills the data into L1, L2, L3 and L4 DRAM caches.

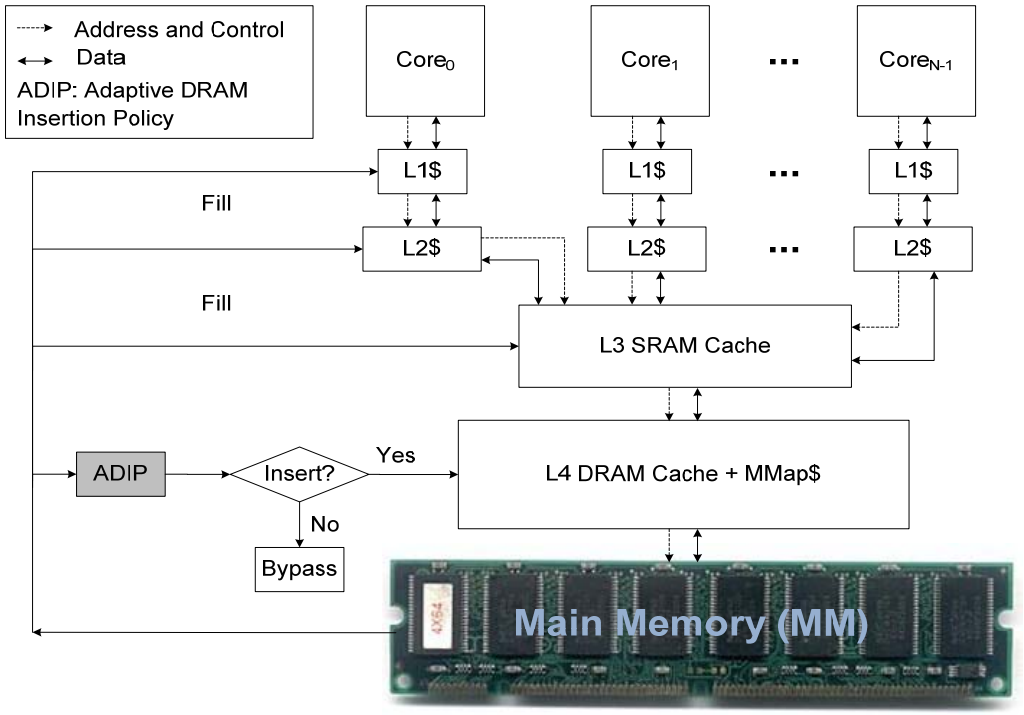


Figure 3.2: High level view of the proposed Adaptive DRAM Insertion Policy

The proposed *ADIP* uses a low overhead *Selective Probabilistic Filter* which provides an efficient filtering mechanism that filters out the majority of *zero-reuse* data but places the majority of *highly-reuse* data into DRAM cache (details in Section 5.2). The proposed *ADIP* reduces contention between *highly-reuse* and *zero-reuse* data which leads to reduced DRAM cache access latency via improved DRAM bandwidth utilization. It provides efficient resource management that leads to reduced miss rate. ADIP first identifies the thrashing applications by monitoring a few sets (so-called sampled sets) of the DRAM cache to track the runtime miss rate information of concurrently running applications. Then, it reduces the insertion rate into DRAM cache from thrashing applications to reduce their effect on other applications. The insertion policy adaptively switches at runtime, depending on monitored application characteristics. That provides performance isolation between thrashing and other applications. In contrast, state-of-the-art insertion policies always insert the data from thrashing and other applications in the DRAM cache, which leads to increased miss rate due to contention between thrashing and other applications.

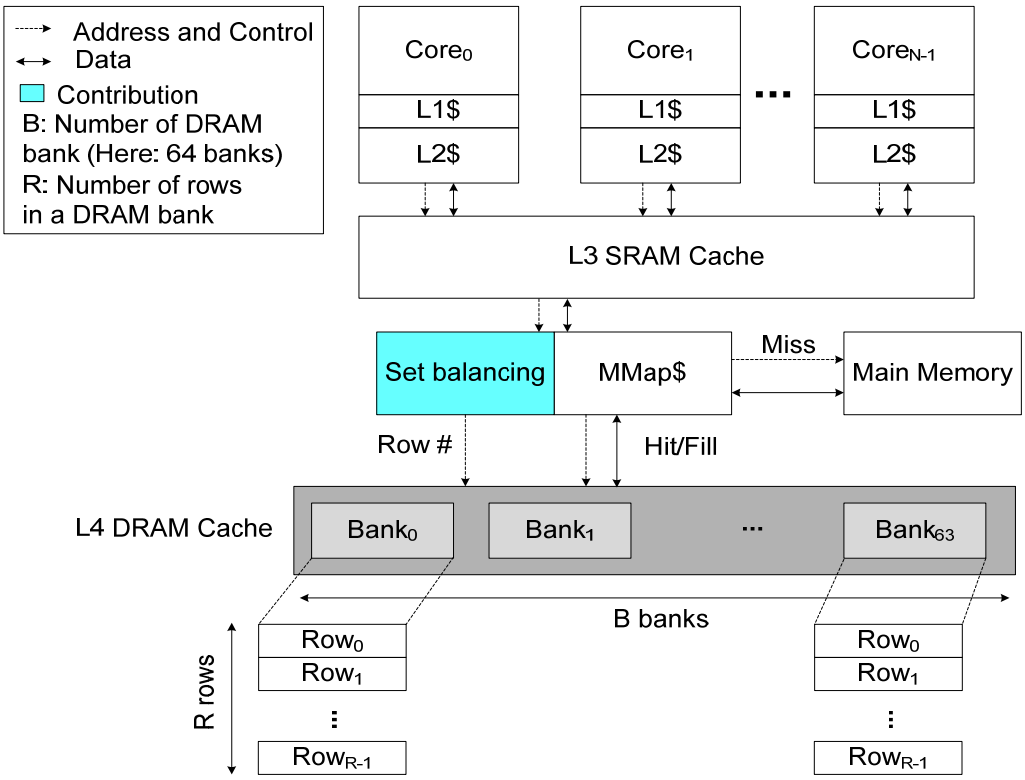


Figure 3.3: High level view of the proposed Set balancing policy

3.2 Set Balancing Policy

Typically, a DRAM cache is composed of multiple banks, where each bank consists of multiple rows as shown in Figure 3.3. Each row may contain a single set [77, 78] or multiple sets [102], where each set contains "*A*" cache lines, i.e. an associativity of *A*. State-of-the-art DRAM cache uses the least significant bits of the memory block address to select the cache set number and bank number as shown in Figure 3.4. On the other hand, they use the higher order bits of the memory block address to select the DRAM cache row number. These bits exhibit highly non-uniform distribution compared to lower order address bits. Using them to select the row number

leads to inefficient utilization of the DRAM cache because some of the DRAM cache rows may be under-utilized, whereas others may be severely over-utilized. As a result, over-utilized rows incur large miss rates due to increased conflict misses compared to under-utilized rows, which may degrade the performance.

To reduce conflict misses via improved row utilization, this thesis proposes a DRAM set balancing policy and integrates it into the MMap$ [77, 78] as shown in Figure 3.3. The proposed set balancing policy improves the DRAM cache resource utilization by assigning the DRAM cache row number to every newly requested MMap$ segments (details in Section 2.4.1) in a round robin way. The primary difference is that state-of-the-art DRAM cache statically determines the DRAM row number based on the memory block address (Figure 3.4) while the proposed set balancing policy determines it dynamically in a round robin manner based on the access sequence. For each new allocated MMap$ segment entry (after a segment miss) a new row number is generated and stored in the MMap$ as part of the segment entry (details in Section 5.3.1).

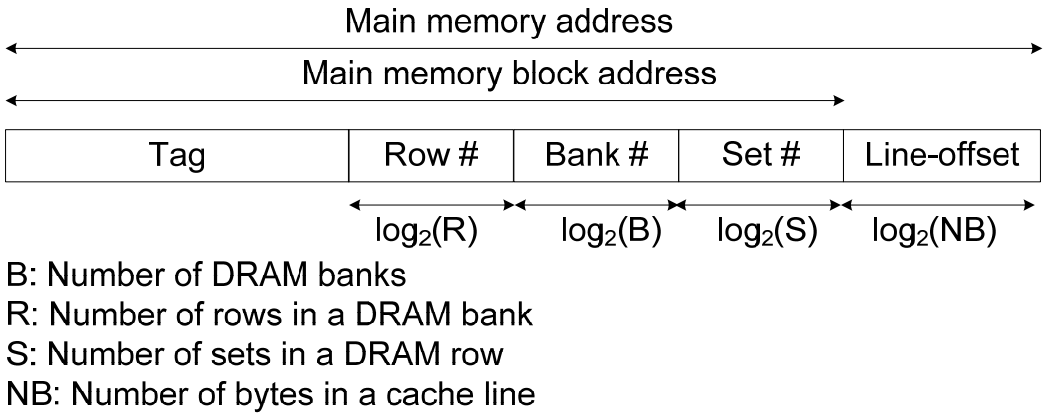


Figure 3.4: DRAM cache row mapping without set balancing

3.3 DRAM Row Buffer Mapping Policy

An important factor that can impact the latency and the miss rate of a DRAM cache is the DRAM row buffer mapping policy. *DRAM Row Buffer mapping* is the method by which memory blocks are mapped to a row buffer of a particular DRAM cache bank. The mapping of the memory block address into a DRAM cache row buffer has a significant effect on the system throughput as it directly affects the row buffer hit rate (effects DRAM cache access latency) and set-level parallelism (effects miss rate). The task of an efficient DRAM row buffer mapping policy is to minimize the probability of row buffer conflicts in temporally adjacent cache requests to improve row buffer hit rate without significantly degrading the miss rate. This thesis demonstrates that the state-of-the-art DRAM row buffer mapping policy namely LH-Cache [77, 78] has a reduced row buffer hit rate due to reduced temporal locality because it maps consecutive memory blocks to different row buffers. To address this problem, this thesis presents a novel row buffer mapping policy that maps four consecutive memory blocks to the same row buffer so that spatially close accesses hit in the row buffer. Thus, it reduces the DRAM cache hit latency via a significantly improved row buffer hit rate. The proposed row mapping policy slightly increases the DRAM cache miss rate due to a reduced set-level-parallelism compared to LH-Cache, but that is compensated by a significant reduction in DRAM cache hit latency.

3.4 Tag Cache Design

When the tags are stored in the cache, they need to be accessed quickly to identify the location of the data. A major performance bottleneck in state-of-the-art DRAM caches [77, 78] is their high tag lookup latency because they access the tags from the slower DRAM cache after MMap$ access. Note that MMap$ consists of bits indicating the presence or absence of data, which exactly determines an L4 hit/miss. But the MMap$ cannot not identify the actual location of data in a DRAM cache set because it does not store the tag information, which would require huge storage overhead. To reduce the tag lookup latency, this thesis proposes a small and low latency SRAM structure namely DRAM Tag Cache (*DTC*) that holds the tags of the rows that were recently accessed in the DRAM cache. The proposed *DTC* has a high hit rate because it exploits temporal locality provided by the proposed DRAM row buffer mapping policy (mentioned in Section 3.3). It provides fast tag lookup, because for a *DTC* hit it directly reads the tags from the low latency *DTC* in two cycles (see Figure 3.1 and Figure 3.5; one cycle required to identify *DTC* hit/miss and one cycle required to identify L4 hit/miss). The tag lookup latency is reduced for a *DTC* hit, because it does not requires MMap$ and DRAM cache access. In contrast, state-of-the-art DRAM cache always access MMap$ followed by reading the tags from the DRAM cache, which incurs high tag lookup latencies of up to 41 cycles (see Figure 2.9-a).

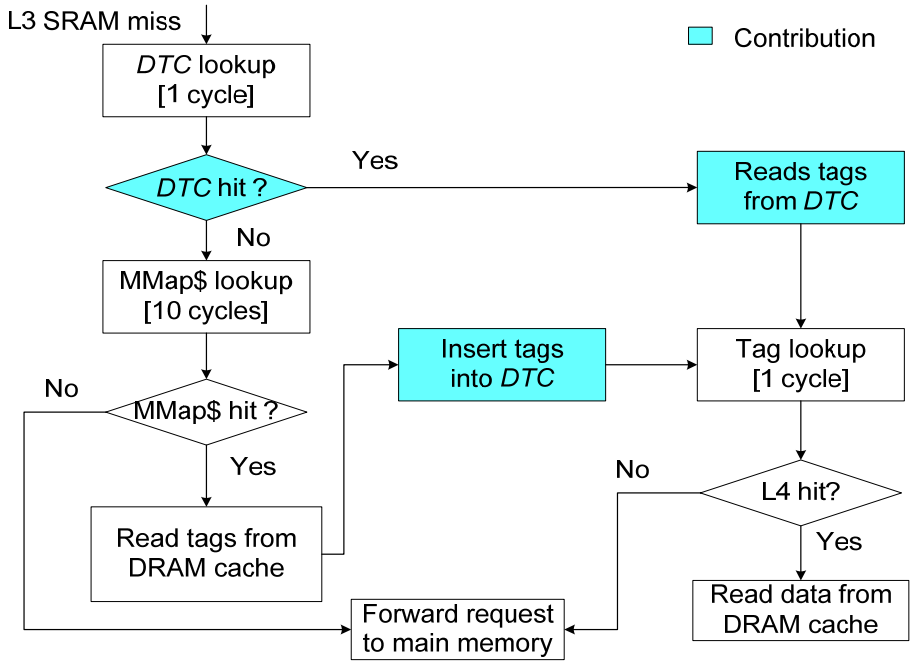


Figure 3.5: Steps involved in L4 DRAM tag lookup after an L3 SRAM miss

State-of-the-art SRAM/DRAM cache hierarchies employ a large L3 SRAM cache that accommodates a large portion of the application's working set size via high storage capacity, which in turn improves the overall performance via reduced L4 DRAM access rate. However, the increased capacity of a large L3 SRAM cache comes at the cost of higher tag latency due to wire delays between the large tag array and L3 SRAM controller [87, 124]. Therefore, reading tags from a large L3 SRAM tag array results in a high latency for L3 requests. Similar to the *DTC*, this thesis also proposes a small and low latency SRAM Tag Cache (*STC*) that holds the tags of the recently accessed sets in the L3 SRAM cache. The proposed *STC* exploits the spatial locality

by prefetching tags from adjacent cache sets. The *STC* is accessed faster (2 cycles) than a large L3 SRAM tag array (10 cycles). An *STC* hit quickly identifies L3 hit/miss to determine whether the request should be sent to the L3 SRAM data array (i.e. L3 hit) or to the next level cache (i.e. L4 DRAM cache for an L3 miss). This thesis further describes how a state-of-the-art SRAM tag array is modified to support SRAM Tag-Cache design (details in Section 6.5.5).

3.5 Super-block MMap$ (SB-MMap$)

State-of-the-art DRAM cache utilizes an SRAM structure namely MissMap cache (MMap$) [77, 78] that provides DRAM hit/miss information (details in Section 2.4.1) by maintaining presence bits (indicates whether a block is present in DRAM cache or not) at the block level. The primary advantage of MMap$ is that it does not require DRAM access for a MMap$ block miss (i.e. block is not present in the DRAM cache), before the request is sent to main memory. However, it requires a reasonably large amount of SRAM storage (e.g. 2MB for 128MB DRAM cache) to store block-level presence bits.

The proposed row buffer mapping policy (see Section 3.3) along with the DRAM Tag-Cache allows reducing the size of the MMap$. To reduce the storage overhead, this thesis proposes to use a single bit to store presence information about multiple blocks (instead of a single block) called a super-block. A super-block comprises a power of two number of blocks. The presence bit of a super-block indicates whether any (one or more) of its associated blocks are present in the DRAM cache or not. The drawback of the proposed super-block MMap$ (SB-MMap$) is that it exacerbates DRAM cache hit/miss prediction accuracy. The reason is that SB-MMap$ may wrongly predict a DRAM hit (i.e. presence bit associated with a super-block is set while the block is not present in DRAM cache) while it turns out to be a DRAM miss. However, the prediction accuracy is significantly improved via a high *DTC* hit rate (i.e. *DTC* provides the block hit/miss information instead of SB-MMap$) provided by the proposed row buffer mapping policy. Note that state-of-the-art DRAM row buffer mapping policy [77, 78] provides a reduced *DTC* hit rate due to reduced spatial locality (details in Section 6.7.3), which leads to reduced performance via poor DRAM cache hit/miss prediction accuracy when SB-MMap$ is incorporated in the cache hierarchy. This thesis further analyzes the effect of different super-block sizes (small vs. large) on the DRAM cache hit/miss prediction accuracy and the overall performance.

3.6 Summary

High DRAM cache hit latencies, increased inter-application contention (between thrashing and non-thrashing applications), and the increased working set sizes of complex applications necessitates efficient policies in order to satisfy the diverse requirements to improve the overall throughput. This work addresses how to design DRAM caches to reduce DRAM cache hit latency, DRAM cache miss rate and hardware cost at the same time, while taking into account application and DRAM characteristics. It presents novel DRAM and application aware policies for on-chip DRAM caches that simultaneously improve the DRAM hit latency and DRAM cache miss rate.

Table 3.1 illustrates an overview of the proposed policies whose details can be found in Chapter 5 and Chapter 6. Chapter 5 presents the policies for miss rate reduction, while Chapter 6 provides the details of policies for latency reduction. Note that each of the proposed policies are

complementary. Therefore, they are evaluated independently in Chapter 5 and Chapter 6. Additionally, this thesis evaluates and analyzes the combinations of the selected policies in Chapter 7. All evaluations in Chapters 5 to 7 are based on the identical experimental setup, and thus the setup is briefly explained upfront in Chapter 4. The tag lookup latency for a *DTC* hit with set balancing policy is 12 cycles (2 cycles for a *DTC* hit and 10 cycles for the MMap$ access required to get the DRAM row number). In contrast, the tag lookup for a *DTC* hit without set balancing policy is 2 cycles (i.e. it does not require MMap$ access to get the DRAM row number as it is determined by the memory block address). The latency benefits via incorporating *DTC* outweighs the miss rate benefits from the set balancing policy. For this reason, the set balancing policy is not included in Chapter 7 with the DRAM Tag Cache (*DTC*).

The proposed policies are evaluated for various applications from SPEC2006 [5] using a modified version of SimpleScalar. The combination of the proposed DRAM-aware and application-aware complementary policies work synergistically, which improves the average performance by 30.4% and 23.9% compared to [77, 78] and [102] respectively for an 8-core system while requiring 51% less hardware cost (i.e. requiring ~1MB SRAM storage overhead instead of ~2MB).

DRAM Last-Level-Cache Polices	Advantages	Reduces Hit Latency	Reduces Miss Rate
Adaptive DRAM Insertion Policy (Chapter 5)	1. reduces inter-core interference	Yes	Yes
Set Balancing Policy (Chapter 5)	2. provides uniform access distribution	No	Yes
Row Buffer Mapping Policy (Chapter 6)	3. improves row buffer hit rate	Yes	No
Tag-Cache (SRAM and DRAM Tag-Cache; Chapter 6)	4. reduces tag lookup latency	Yes	No
Super-block MMap$ (SB-MMap$; Chapter 6)	5. Reduces storage overhead	N/A	N/A
Combination of selected policies (Chapter 7)	1, 3, 4, and 5	Yes	Yes

Table 3.1: Overview of the proposed Policies

Chapter 4 Experimental Setup

Chapter 5 and Chapter 6 present policies for miss rate and latency reduction respectively, detailing their operations and implementation with embedded evaluation. Chapter 7 presents an evaluation for the combined contributions presented in Chapter 5 and Chapter 6. This thesis employs the same experimental setup for presenting the evaluation results in Chapters 5 to 7. Therefore, this chapter presents a brief overview of the simulator infrastructure as well as the description of the benchmarks and performance metrics used for evaluation.

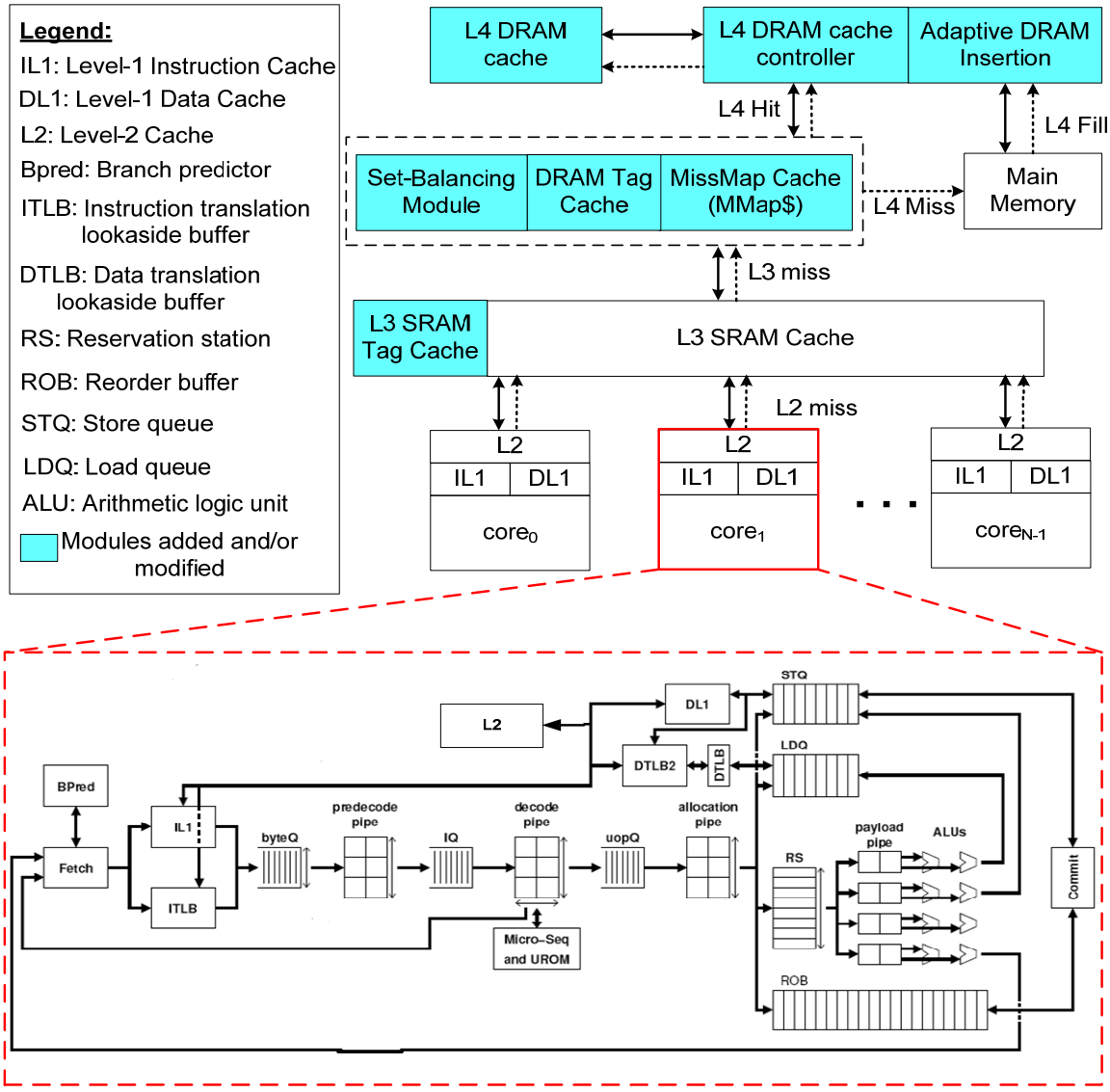


Figure 4.1: Overview of the simulator based on Zesto simulator [79]

4.1 Simulation Infrastructure

This thesis uses the x86 version of the SimpleScalar (Zesto) [79] simulator for evaluation. The Zesto simulator is built on top of the SimpleScalar toolset [13], which provides a cycle-accurate processor and detailed cache hierarchy model. The primary disadvantage of the original cycle-accurate SimpleScalar [13] simulator is that it uses a simplified memory model with fixed latency. The simplistic model is not a true representative of modern main memory. In contrast, Zesto faithfully models queuing delays and bandwidth constraints for the main memory banks and ports. Also, the SimpleScalar simulator uses a simplified processor model based on an old concept of Register Update Unit (RUU) [119] published in 1990. In contrast, Zesto provides detailed modeling of a modern x86 based microarchitecture. It provides a detailed pipelined model implementing fetch, decode, allocation, execute and commit stages similar to Intel's Pentium Pro architecture as shown at the bottom of Figure 4.1.

Several modifications and additions were made to the simulator infrastructure for this thesis, especially in the DRAM cache. Since this work deals with the cache hierarchy, the simulator has been modified to faithfully model bus contention for different cache levels including access to MMap$ [77, 78]. The modified simulator models port contention, queuing delays, bank conflicts, and other DDR3 DRAM system constraints [61] for DRAM cache. Figure 4.1 shows the block-level details of the simulation infrastructure for simulating an N-core system for a DRAM cache based hierarchy with embedded modifications. Each core is provided with its own private caches (e.g. L1 and L2 SRAM caches) that are connected to a shared L3 SRAM cache. When a core issues a read or write request, the cache hierarchy first check the private caches for a hit/miss. A request that misses in the private caches is forwarded to the L3 shared SRAM cache. After an L3 cache miss, the MMap$ is queried to determine an L4 DRAM cache hit/miss. If the MMap$ identifies a hit (i.e. requested block is present in the L4 cache), the request is sent to the DRAM cache. If the MMap$ identifies miss (i.e. requested block is not available in the L4 cache), the request is sent to main memory.

4.2 Simulation Parameters

The core, cache, and main memory parameters are listed in Table 4.1 and Table 4.2. The core is clocked at 3.2 GHz with 32 KB instruction cache and 32 KB data cache with 2 cycles latency. The core has a 128-entry reorder buffer, 32-entry reservation station, 32-entry load queue, and 24-entry store queue. Each core is able to fetch, decode and commit up to four x86 instructions in a single cycle. Similar to state-of-the-art [77, 78, 102], this thesis assumes that DRAM-cache timing latencies are approximately half of that compared to off-chip memory, which allows direct comparison with them. The latency of the SRAM caches are computed using CACTI v5.3 [124] for a 45nm technology.

Throughout this thesis, the following assumptions are considered for all evaluations:

1. Similar to state-of-the-art [36, 37, 38, 77, 78, 102], this thesis employs FR-FCFS (First Ready First Come First Serve) access scheduling [107] in the DRAM cache and memory controllers.
2. Similar to state-of-the-art [36, 77, 78, 102], this thesis assumes four DRAM cache channels and the DRAM cache bus width per channel is assumed to be 128 bits (16 bytes).

3. This thesis assumes that the tags are stored in the DRAM cache similar to state-of-the-art [36, 77, 78, 102] while a 2 MB MMap$ [77, 78] is employed to identify a DRAM cache hit/miss. MMap$ is accessed after an SRAM miss. The latency values of MMap$ is calculated using CACTI v5.3 [87, 124] for a 45nm technology.
4. Similar to state-of-the-art [35, 75, 99, 138], this thesis assumes that each core runs a single application.

Core Parameters	
Core Frequency	3.2 GHz
ROB (reorder buffer) size	128
Reservation station (RS) size	32
Load Queue (LDQ) size	32
Store Queue (STQ) size	24
Decode width	4
Commit width	4
Branch misprediction penalty	14 cycles
SRAM Cache Parameters	
Private L1 Caches (IL1 and DL1)	32KB, 8-way, 2 cycles
Private L2 Caches	256KB, 8-way, 5 cycles
Shared L3 SRAM Cache (Serialized tag and data access)	8MB, 8-way, 10 cycles Tag-Latency, 15 cycle data latency
DRAM cache Parameters	
MMap$	2MB, 10 cycles
DRAM cache size	128 MB
Row buffer size	2KB (2048 bytes)
Number of DRAM banks	64
Number of channels	4
Bus Width	128 bits per channel
Bus Frequency	1.6 GHz
t_{RAS} (Row access strobe)	72 cycles
t_{RCD} (Row to column command delay)	18 cycles
t_{RP} (Row precharge delay)	18 cycles
t_{CAS} (Column access strobe)	18 cycles
t_{WR} (Write recovery time)	18 cycles

Table 4.1: Core and cache parameters

Main Memory Parameters	
Number of channels	2
Bus Width	64 bits per channel
Bus Frequency	800 MHz
t_{RAS} (Row access strobe)	144 cycles
t_{RCD} (Row to column command delay)	36 cycles
t_{RP} (Row precharge dealy)	36 cycles
t_{CAS} (Column access strobe)	36 cycles
t_{WR} (Write recovery time)	36 cycles

Table 4.2: Main memory parameters

4.3 Benchmarks and classification

One of the key metric that determines the application cache access behavior is the Last-Level-Cache (LLC) Misses Per Thousand Instructions (MPKI). LLC MPKI is an indicator that determines how application performance is affected by the amount of cache resources available to it. The applications from SPEC2006 exhibit diverse cache access patters as illustrated in Figure 1.2 which shows LLC MPKI for different applications while varying the LLC capacity. Based on the LLC MPKI metric, this thesis classifies the applications into the following categories:

1. *Latency Sensitive* applications are very sensitive to the amount of cache resources allocated to them. Increasing the cache resources of these applications (e.g. *473.astar.train*, *437.leslie3d.train* , *471.omnetpp* etc.) provides significant reduction in MPKI.
2. *Memory Sensitive* applications have a high cache miss rate and a high cache access rate. These applications (e.g. *437.leslie3d.rain*, *433.milc*, and *450.soplex*) get negligible benefit from increasing the cache resources.

Future multi-core systems are expected to execute multiple applications with diverse cache access patterns. For evaluation, this thesis makes use of various application mixes from SPEC2006 [5] as shown in Table 4.3. These application mixes were chosen because they contain applications with different working set sizes and cache access patterns.

4.4 Simulation Methodology

This thesis uses the Simpoint tool [40, 93, 94] to select representative samples for each application. The simulation statistics are collected for 500 million instructions with a fast-forward of 500 million instructions (to warm up the caches and branch predictors in functional mode) for each application. When a shorter benchmark finishes early by completing its 500 million instructions, then it is restarted and continues to contend for the cache and bus resources. However, the simulation statistics are reported for the first 500 million instructions after the fast-forward.

Name	Benchmarks
Mix_01	433.milc(1), 437.leslie3d.ref (1), *437.leslie3d.train* (1), 450.soplex (1), *462.libquantum*(1), 470.lbm(1), *471.omnetpp*(1), *473.astar.train*(1)
Mix_02	437.leslie3d.ref (2), *437.leslie3d.train* (2), 450.soplex (2), *462.libquantum*(2)
Mix_03	433.milc(2), 470.lbm(2), *471.omnetpp*(2), *473.astar.train*(2)
Mix_04	433.milc(1), 437.leslie3d.ref (2), *437.leslie3d.train* (2), 450.soplex (1), *462.libquantum*(1), *473.astar.train*(1)

Table 4.3: Application mixes (value in parenthesis denotes the number of instances used for that particular application). Latency sensitive applications shown in italics

4.5 Performance Metric

Several performance metrics [24, 67, 75, 80, 118] have been used for the evaluation of multi-core systems when comparing old and new policies, which determines throughput and fairness measures. A throughput measure is used to determine the overall speedup of a new policy. The speedup may come from one or more applications at the cost of performance degradation of some other applications. On the other hand, a fairness measure is used to determine whether concurrently running applications on a multi-core system receive a fair performance improvement when using a new policy or not. Commonly used evaluation metrics are the 'overall instruction per cycle', 'arithmetic mean instruction per cycle' and 'harmonic mean instruction per cycle'. The first two metrics favor the throughput measure but they do not truly capture the fairness measure. The harmonic mean instruction per cycle metric has been shown to balance both fairness and throughput [24, 67, 80] because it tends to be lower if one or more applications lose performance when using a new policy. Therefore, this thesis employs the 'harmonic mean instruction per cycle' metric [24, 67, 80] for performance evaluation which is given as follows:

$$\text{Harmonic Mean Instruction per cycle} = \frac{N}{\sum_{i=1}^{N} \frac{1}{IPC_i}}$$

N is the number of applications in a particular application mix. IPC_i is the instruction per cycle (IPC) of the application when it runs concurrently with other applications.

Chapter 5 Policies for Miss Rate Reduction

On-chip DRAM Last-Level-Cache has been employed recently [53, 75, 77, 78, 102, 112, 113] to reduce the number of slower main memory accesses. A DRAM cache hit provides fast access to the requested data compared to off-chip memory access. Therefore, to maintain high performance, it is important to reduce DRAM Last-Level-Cache misses. This chapter proposes novel policies namely adaptive DRAM insertion policy and set balancing policy to reduce the DRAM cache miss rate by reducing the number of conflict misses. The integration of the proposed policies into a DRAM cache hierarchy is shown in Figure 5.1 for an N-core system.

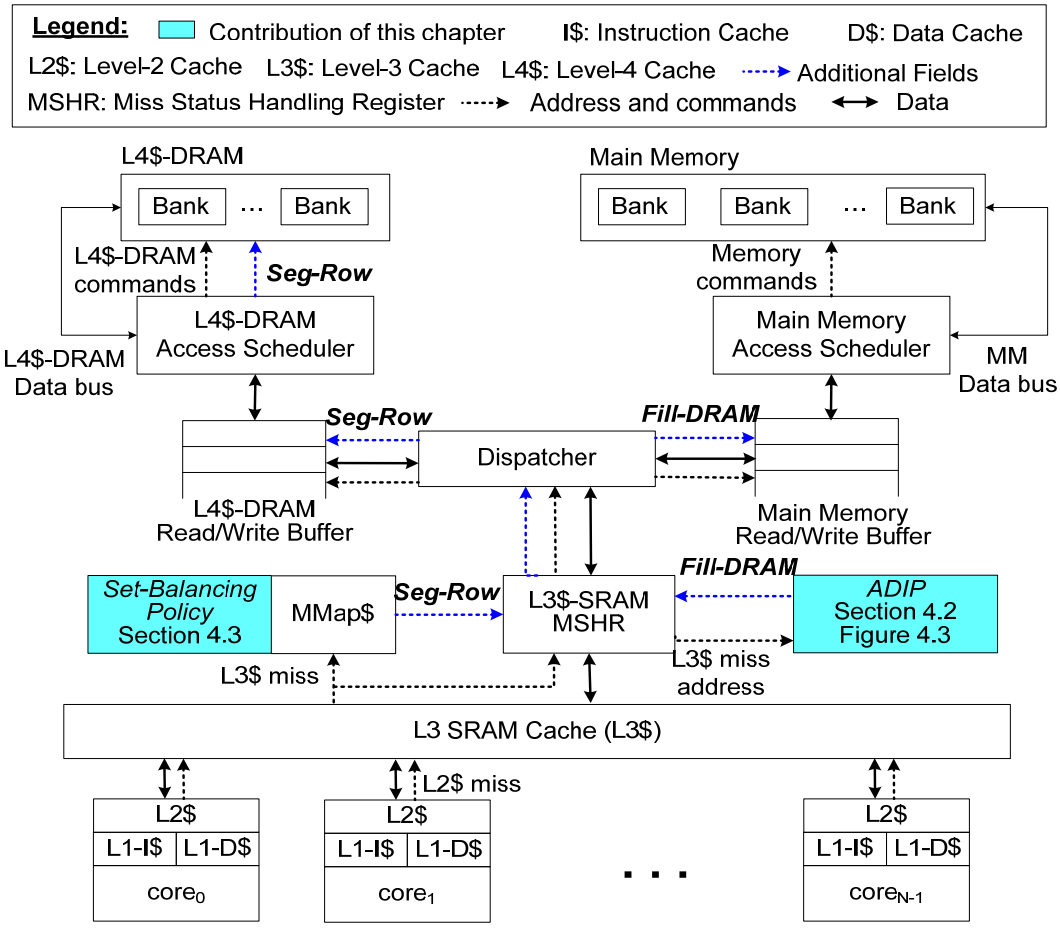


Figure 5.1: Proposed DRAM cache hierarchy for an N-core system; *Fill-DRAM* field indicates whether an incoming block from off-chip memory should be inserted into the L4$-DRAM or not; *Seg-Row* field is used for set balancing

The first section of this chapter analyzes state-of-the-art static DRAM insertion policies [77, 78, 102] to show how some applications incur increased miss rate, which leads to reduced system throughput. It demonstrates that existing policies do not work well because they use a static insertion rate (DRAM insertion rate is defined as the percentage of block insertions into DRAM cache) for all applications, i.e. they always insert requesting blocks into DRAM cache after a DRAM cache miss. The next section demonstrates that the performance can be improved if the

applications are assigned appropriate insertion rates depending upon which insertion rate performs better for a given application. It presents the novel adaptive DRAM insertion policy that provides improved system throughput compared to state-of-the-art static insertion policies [51, 77, 78, 102] via reduced miss rate. The proposed adaptive DRAM insertion policy chooses the best-performing insertion rate for each application at runtime among four different insertion rates, by tracking the miss rate information of concurrently running applications with a low overhead monitoring mechanism.

Section 5.3 of this chapter shows that multiple applications running on a multi-core system exhibit a non-uniform distribution of accesses across different DRAM cache sets, which leads to an inefficient utilization of DRAM cache capacity. To overcome this problem, this section presents a DRAM cache set balancing policy that mitigates the imbalance across DRAM cache sets to improve the performance via efficient capacity utilization.

5.1 Motivation

In a typical multi-core system, cores are concurrently running heterogeneous applications such as web-browser, text editors, scientific or data mining applications. These concurrently executing applications compete with each other for the shared resources causing inter-core interference. An important design consideration for a DRAM cache based multi-core system is the management of shared resources such as DRAM cache capacity, DRAM cache bandwidth, and off-chip memory bandwidth [12, 108, 131, 136, 144]. As the number of cores in a multi-core system increases, increased number of requests from multiple cores can cause inter-core DRAM interference (explained in Section 2.3.2) in the DRAM cache leading to an increased load on DRAM cache bandwidth. It may also result in an increased load on off-chip bandwidth via inter-core cache contention (explained in Section 2.3.1), hereby increasing DRAM cache miss rate. This section presents an example that shows how state-of-the-art DRAM insertion policies may cause increased miss rate and how a judicious DRAM insertion policy can be used to mitigate inter-core DRAM interference and inter-core cache contention.

State-of-the-art DRAM insertion policies [77, 78, 102] do not work well with applications that have a reuse distance (i.e. the number of insertions before the block is reused) larger than the cache associativity. Such applications are classified as “thrashing” applications [51, 138] (see Section 2.3.1). These applications have poor temporal locality for the available cache size, as they generate a large number of requests without being reused in the future [51, 138]. Figure 5.2 illustrates a 4-way DRAM cache with accesses (shown in capital letters E, F etc.) from a thrashing application. On a cache miss, an incoming block is inserted into the most recently used (MRU) position while the block in the least recently used (LRU) position is the candidate for eviction to make room for the incoming block.

The DRAM insertion policy used in the state-of-the-art [77, 78, 102] statically inserts the block with a probability of 1 after a DRAM cache miss, which increases the number of unnecessary fill requests for thrashing applications as illustrated in Figure 5.2-(a). For instance, blocks A, B, E and F have a reuse distance (RD) greater than the associativity of 4 and are thus never reused if inserted with a probability of 1 as illustrated in Figure 5.2-(a). This results in a high load on DRAM cache bandwidth via increased unnecessary fill requests. Additionally, the contention

between thrashing and non-thrashing application for a static DRAM insertion policy will reduce the number of hits, thus increasing the load on off-chip memory bandwidth.

For thrashing applications, the performance can be improved by inserting the blocks into DRAM cache with a low probability, thus reducing the number of fill requests. It enables thrashing applications to retain some fraction of the working set which increases the number of hits as illustrated in Figure 5.2-(b) and (c). In this example, an insertion probability of ¼ leads to the best hit rate and to reduced number of fill requests compared to higher insertion probabilities. To reduce interference between hit and unnecessary fill requests in DRAM cache, the proposed *Adaptive DRAM Insertion Policy* (*ADIP*) uses a low probability to insert an incoming block for applications with long reuse distances and uses the highest probability of 1 for applications with short reuse distances. *ADIP* adapts the DRAM insertion probabilities at run-time on a per-core basis.

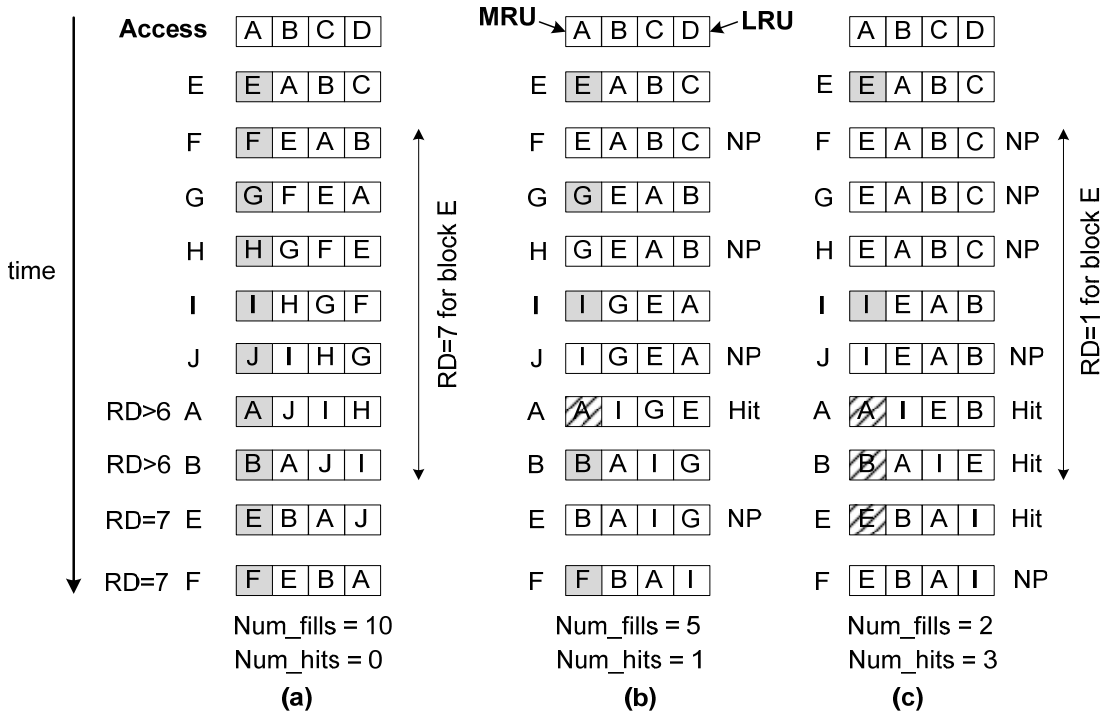


Figure 5.2: Example illustrating DRAM insertion probability of (a) 1, (b) ½, and (c) ¼ Block insertion shown as grey, hits shown as shades, NP stands for block not placed in DRAM cache, RD stands for reuse distance

5.2 Adaptive DRAM Insertion Policy (*ADIP*)

The integration of the proposed Adaptive DRAM insertion policy (*ADIP*) into the cache hierarchy is shown in Figure 5.1 and Figure 5.3 presents the *ADIP* details. The MMap$ (functionality explained in Section 2.4.1) is accessed after a miss in the L3 SRAM cache. A MMap$ hit indicates that the block is present in L4 DRAM cache. In that case, the block is read from L4 DRAM cache and inserted into L3 SRAM cache and the core private L1/L2 caches to exploit the temporal locality that the referenced block might be accessed again in the near future. Hits to these replicated blocks in L3 SRAM reduce the effective access latency by avoiding costly L4 DRAM

accesses, hereby reducing inter-core DRAM interference. On a MMap$ miss, the block is brought from memory and inserted into L3 SRAM and core private L1/L2 caches. The block may or may not be filled additionally in L4 DRAM cache, which is determined by the adaptive DRAM insertion policy (*ADIP*). Existing DRAM cache hierarchies [77, 78, 102] always insert the block into all cache levels when brought from main memory.

The adaptive DRAM insertion policy (*ADIP*) consists of two major components:

1. **Application Profiling Unit (APU):** In order to provide sufficient information for the *ADIP* policy, the APU profiles the application behavior (thrashing or non-thrashing) by tracking run-time miss rate information of all concurrently executing applications (described in Section 5.2.1).
2. **Probability Selection Unit (PSU):** reads the runtime statistics provided by the APU to determine the suitable insertion probability for each application (Section 5.2.2).

5.2.1 Application Profiling Unit (APU)

Figure 5.3 shows the details of the **A**pplication **P**rofiling **U**nit (*APU*) that is based on set dueling. Set dueling is a well established mechanism [75, 51] to adaptively choose between two competing policies P0 and P1. In set dueling, a few sampled sets of the cache are dedicated to always use policy P0 and other few sampled sets to always use policy P1. A saturating *k*-bit policy selection (PSEL) counter (counting from 0 to 2^k-1 and initialized with 2^{k-1}) estimates which of the two policies leads to a smaller number of misses. Misses in the sampled sets using P0 cause the PSEL counter to be incremented and misses in the sampled sets using P1 cause it to be decremented. If the MSB of PSEL is '0', then policy P0 is used for all non-sampled sets, if it is '1', then policy P1 is used.

This thesis employs the set dueling mechanism to adaptively choose among four DRAM insertion probabilities (p_a, p_b, p_c, and p_d). In the proposed *ADIP*, each set inserts an incoming block with a probability vector $<p_0, \ldots, p_{n-1}>$, where p_i denotes the insertion probability for requests from $core_i$. Some cache sets are "leader sets" (that contain some sampled sets per core) and other cache sets are "non-sampled sets" that follow the decisions of the leader sets. Figure 5.3 shows the *ADIP* for an N-core system where the sets are clustered into groups of *CS* sets (*CS* stands for cluster size; this thesis uses *CS* = 128). Each cluster contains *6N* leader sets (6 per core) and *CS* – *6N* non-sampled sets. The first 6 sets of each cluster are used as sampled sets for $core_0$, while the next 6 sets are used as sampled sets for $core_1$, and so on for the other cores. Out of the 6 leader sets per core, 4 sets (grey boxes in Figure 5.3) are dedicated as sampled sets with fixed insertion probabilities p_a, p_b, p_c, and p_d (this thesis uses $p_a = 1/64$, $p_b = 1/16$, $p_c = 1/4$, and $p_d = 1$). For example, $core_0$ always inserts an incoming block with a fixed probability of p_a for the first set of each cluster ($p_{0=}p_a$ for this set) and with probability p_b for the second set of the cluster ($p_{0=}p_b$). Similarly, $core_0$ always inserts an incoming block with a fixed probability of p_c for the fourth set of each cluster ($p_{0=}p_c$ for this set) and with probability p_d for the fifth set of the cluster ($p_{0=}p_d$). Each core is provided with three 10-bit policy selection counters namely $PSEL^{ab}{}_i$, $PSEL^{cd}{}_i$, and $MPSEL_i$ that determine the winning policy for each core. The 10-bit policy selection counter $PSEL^{ab}{}_i$ for $core_i$ estimates which of the two insertion probabilities (p_a or p_b) leads to the smaller number of misses. A miss incurred in the set dedicated for p_a increments $PSEL^{ab}{}_i$ while a miss incurred in the set dedicated for p_b decrements $PSEL^{ab}{}_i$. This direct comparison between p_a and p_b is used to decide the insertion probability $p_{ab,i}$ of a so-called "partial set" for $core_i$ (shown as shaded boxes in Fig-

ure 5.3). The policy section counters ($PSEL^{ab}_i$, $PSEL^{cd}_i$, and $MPSEL_i$) remain unchanged for misses in the non-leader sets of $core_i$. The next section provides the details for choosing a suitable insertion probability among four different insertion probabilities (i.e. p_a , p_b , p_c , and p_d) for each core.

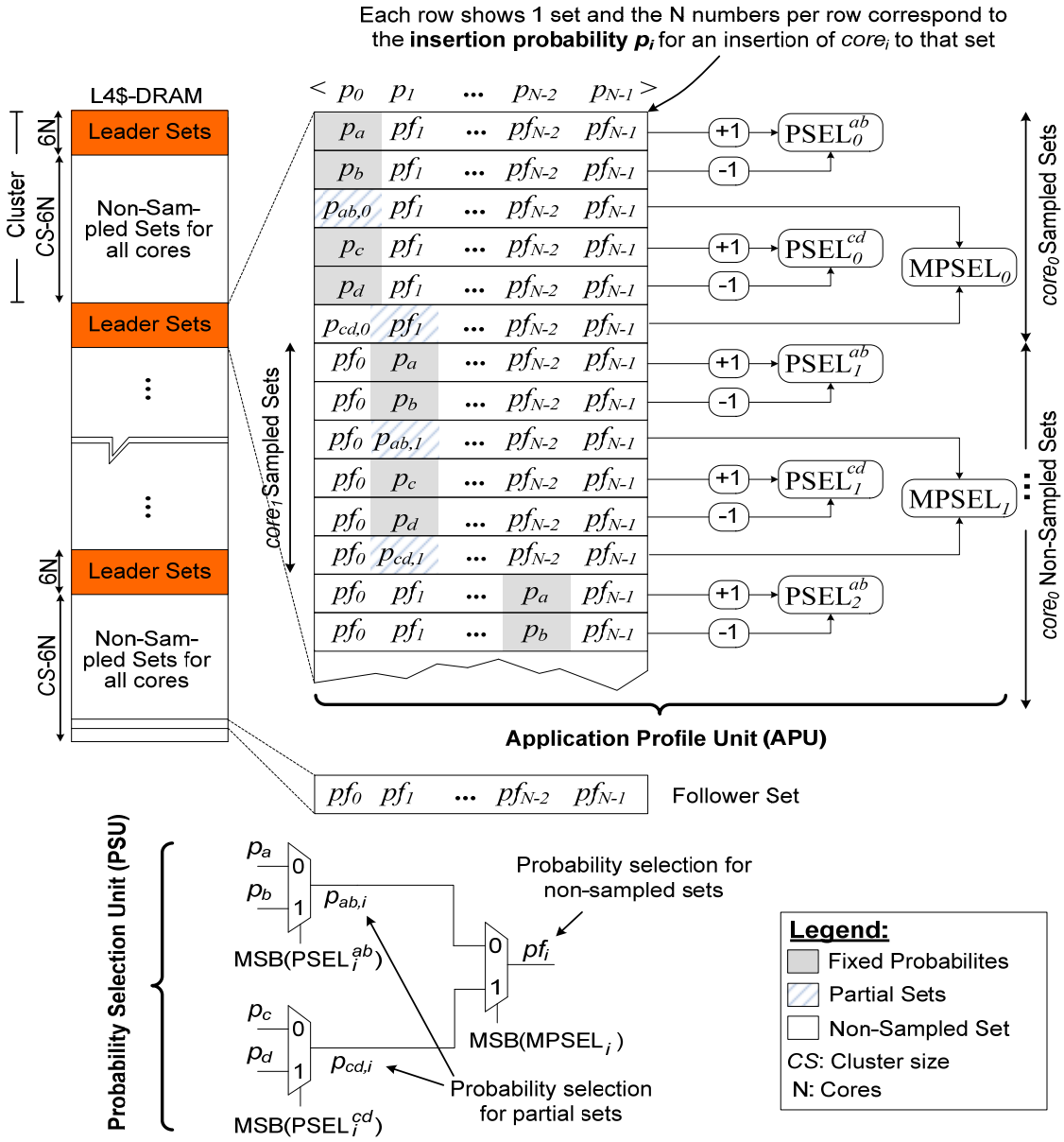


Figure 5.3: Adaptive DRAM Insertion Policy for an N-core system

5.2.2 Probability Selection Unit (PSU)

The goal of the **P**robability **S**election **U**nit (*PSU*) is to decide the insertion probability pf_i for $core_i$ at run-time for the large number of non-sampled sets. The *PSU* reads the policy selection counters ($PSEL^{ab}_i$, $PSEL^{cd}_i$, and $MPSEL_i$) for each competing application to determine the insertion probability pf_i (pf_i stands for the policy that is used by the "non-sampled sets" of $core_i$). If the

MSB of $PSEL^{ab}_i$ is 0, then $p_{ab,i}$ is set to p_a, otherwise to p_b (see multiplexors at the lower part of Figure 5.3). Similarly, $PSEL^{cd}_i$ estimates which of the two insertion probabilities p_c or p_d leads to the smaller number of misses. Finally, a meta-policy selection counter $MPSEL_i$ is associated with each $core_i$ that estimates which of the two partial insertion probabilities ($p_{ab,i}$ or $p_{cd,i}$) leads to the smaller number of misses. If the MSB of $MPSEL_i$ is 0, then the insertion probability pf_i for all non-sampled sets of $core_i$ is $p_{ab,i}$, otherwise $p_{cd,i}$.

To reduce the number of unnecessary fill requests from thrashing applications, the *ADIP* policy chooses low insertion probability for them. It chooses high insertion probability for non-thrashing applications, which in turn increases the number of hits. This reduces inter-core cache contention between thrashing and non-thrashing applications by rarely inserting blocks into DRAM cache from a thrashing application to reduce their effect on other applications.

5.2.3 Probability Realization

A target DRAM insertion probability can be implemented using a binary counter or a linear feedback shift register (LFSR). This thesis uses LFSR to realize different insertion probabilities. The primary advantage of using LSFR [28] is that it requires reduced hardware overhead (one LFSR requires six XOR gates and six flip flops) compared to a conventional binary counter. The other advantage of LFSR is that their minimum cycle time is independent of the number of bits of the counter. The proposed *ADIP* needs seven LFSRs (six for sampled sets of leader set and one for non-sampled sets) per core. To realize a target DRAM insertion probability, LFSR generates a 7-bit pseudo-random number (7-bit LFSR generates a number between 1-127 excluding zero) which is compared to threshold values ('3' for $p_a = {}^1/_{64}$, '9' for $p_b = {}^1/_{16}$, '33' for $p_c = ¼$, and '128' for $p_d = 1$). For example the insertion probability $p_b = {}^1/_{16}$ requires generating a pseudo-random number and testing whether it is smaller than 9. If this is the case, then the block is inserted in DRAM cache (i.e. 8 blocks are inserted among 127 requested blocks from main memory), otherwise the block bypasses DRAM cache. Altogether, *ADIP* performs an *adaptive DRAM insertion/bypass* decision based on the comparison of the pseudo random number generated by LFSR with the corresponding threshold value for the insertion probability.

5.3 Set Balancing Policy (*SB-Policy*)

For a large number of DRAM cache sets, the efficiency of the DRAM cache is reduced because programs exhibit a non-uniform distribution of accesses across different cache sets [104]. In such a scenario, some of the DRAM cache sets may be under-utilized, whereas others may be severely over-utilized. As a result, over-utilized sets suffer more conflict misses compared to under-utilized sets which may degrade the performance via increased miss rate. To reduce conflict misses via improved row utilization, this thesis proposes a DRAM set balancing policy and integrates it into MMap$ (details of MMap$ in Section 2.4.1; see Figure 2.11) as shown in Figure 5.4. Recent research has proposed various DRAM cache organizations namely LH-Cache [77] (details in Section 2.4.1) and Alloy-Cache [102] (details in Section 2.4.3). The *SB-policy* can be applied on top of both of them. This section performs analytical comparison of applying the *SB-policy* on top of LH-Cache and Section 5.6.5 evaluates applying it on top of Alloy-Cache. Figure 5.4 shows how a DRAM row is determined for LH-Cache with and without *SB-policy*. The primary difference is that the LH-Cache without *SB-policy* determines the DRAM row number based on the main memory address (Figure 5.4-b) while the row number in *SB-policy* is pro-

vided by the MMap$. The *SB-policy* stores the DRAM row number in the MMap$ which is assigned to each MMap$ entry after a segment miss (see Section 5.3.1).

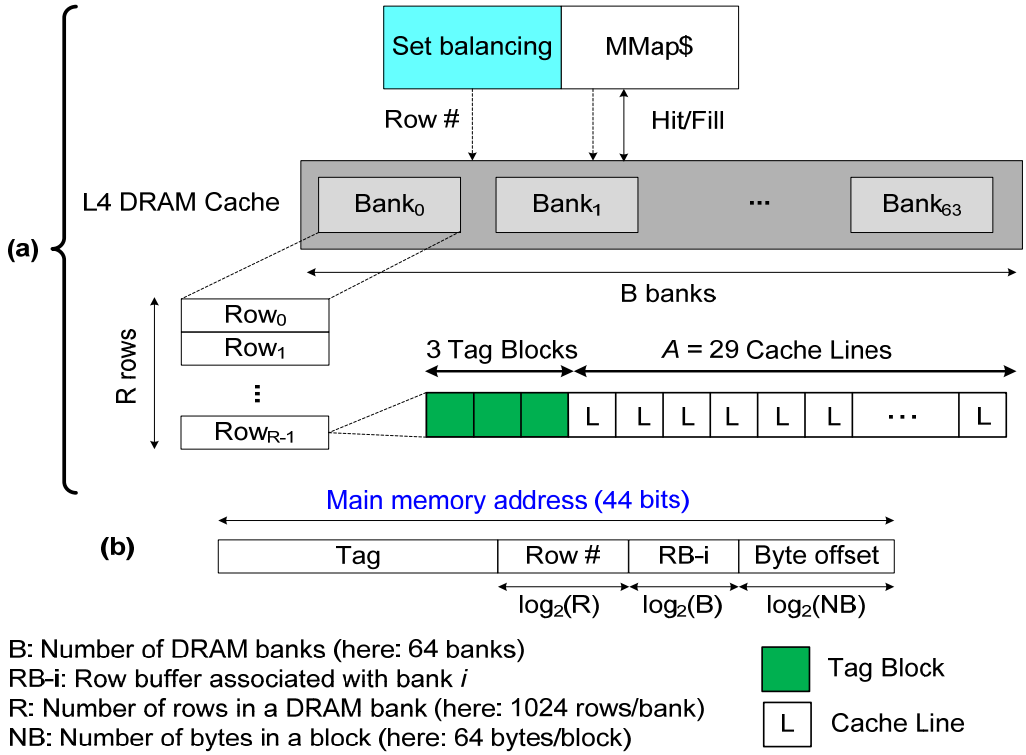


Figure 5.4: DRAM cache row mapping for LH-Cache [77] (a) with *SB-Policy* (b) without *SB-Policy*

An intuitive answer to achieve uniform cache set distribution is to assign a DRAM cache row number (each DRAM bank contains R rows as shown in Figure 5.4, where each row consists of one cache set with 29-way associativity) to each block in a round robin way after a block miss. This would require storing the row number (requires 11 bits for R = 2048) with each block in the MMap$ (details of MMap$ in Section 2.4.1; see Figure 2.11) and this would lead to an additional storage requirement of 640 bits (10 x 64 = 640 bits for R = 2048) for each MMap$ entry. Note that each MMap$ entry requires storage overhead of 94 bits including 1 valid bit, 1 dirty bit, 4 LRU bits, 4 way-number bits, 20 bits for *Seg-Tag* field, and 64-bit for *Seg-BV* field as shown in Figure 5.5. The above-mentioned approach will require up to ~7.8× more storage overhead (it requires ~15.6 MB MMap$) compared to the original MMap$ size (it requires ~2MB MMap$). Instead, the proposed Set Balancing policy (*SB-Policy*) stores the DRAM row number at coarser granularity, as described in the following.

A segment is the basic unit of the MMap$ storage and is a group of contiguous blocks in main memory (typical segment size is 4KB). Each MMap$ entry tracks the block (this thesis use a block of 64 bytes similar to state-of-the-art [77, 78]) associated with a segment (this thesis uses a segment size of 4KB similar to state-of-the-art [77, 78]). Each 4KB MMap$ segment is associated with a tag (called *Seg-Tag*) and a bit vector (called *Seg-BV*) with one bit per block as shown in Figure 5.5. The *Seg-Tag* field determines whether a particular memory segment is present in the MMap$ (segment hit) or absent (segment miss). The *Seg-BV* field determines the hit/miss of a particular block b_i within a particular segment. The proposed *SB-Policy* stores the row number

at segment level which only requires storage overhead of 10 bits (for R = 1024) for each MMap$ entry (i.e. the proposed *SB-policy* requires ~2.2 MB MMap$). For this reason, the proposed *SB-Policy* add an additional *Seg-Row* field to each MMap$ entry for set balancing. The *Seg-Row* field is assigned to each MMap$ entry after a segment miss (see Section 5.3.1).

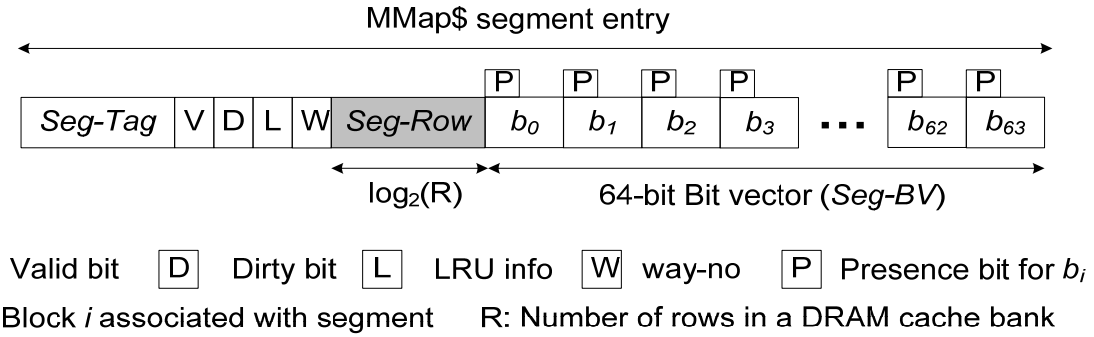


Figure 5.5: MMap$ segment entry; proposed *SB-Policy* adds an additional Seg-Row field to MMap$ entry for set balancing

5.3.1 Row Assignment

The *SB-Policy* assigns a DRAM cache row number to a MMap$ segment after a segment miss (when the segment is referenced for the first time) as shown in Figure 5.6. When an application running on $core_i$ accesses a new segment *S* that is currently absent in the MMap$ (i.e. segment miss), then a new MMap$ entry *E* is allocated for *S* and a DRAM row number (called *Seg-Row*) is assigned to *S* in a round robin manner for $core_i$. After a MMap$ segment hit (DRAM row number already assigned), the DRAM row number is provided by the MMap$ (determined by *Seg-Row* field of the MMap$ entry). The DRAM bank number (i.e. row buffer) is determined by the least significant bits of the memory block address as illustrated in Figure 5.4-(b). Since the *SB-Policy* assigns the DRAM row number in a round robin manner for each core, it leads to an **improved DRAM cache row utilization** (and hence it leads to an improved DRAM cache set utilization).

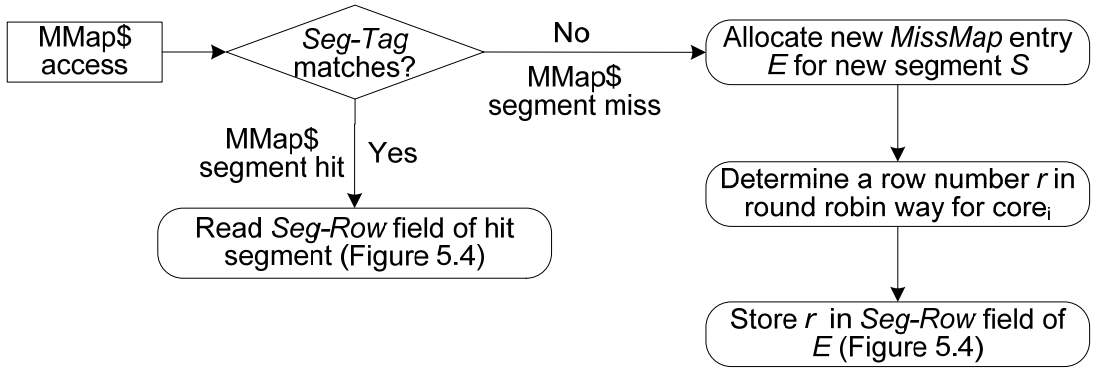


Figure 5.6: Row assignment for *SB-policy*

5.4 Implementation

Figure 5.7 shows the steps involved in cache lookup operation in the proposed *ADIP* and *SB-policy* for a new request from $core_i$, which are explained as follows:

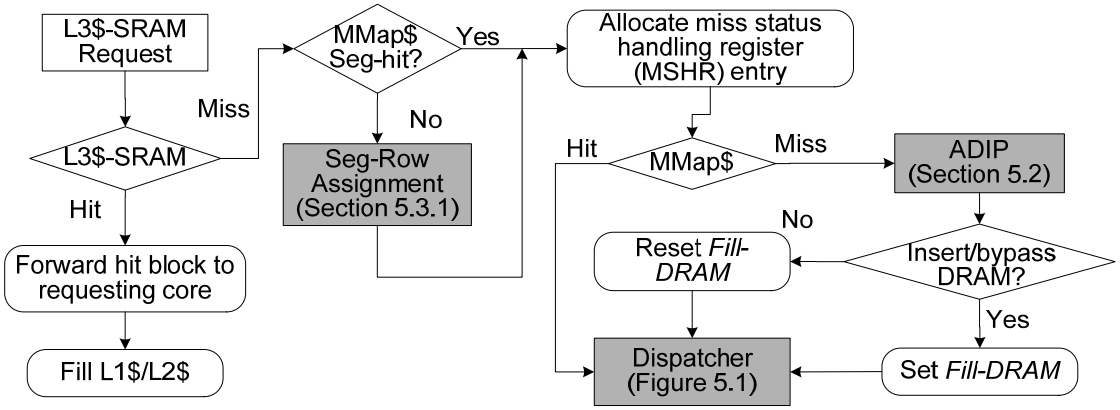


Figure 5.7: Steps involved in cache lookup operation

L3 SRAM cache hit: After an L3 SRAM cache hit, the hit block is forwarded to the requesting core and filled in its private L1/L2 caches.

L3 SRAM cache miss: After an L3 SRAM cache miss, a miss status handling register (MSHR) [71] is allocated that keeps track of the outstanding L3 cache misses (see Figure 5.1). The various fields of the MSHR entry include the *Valid-bit*, *Issued-bit* (request is issued or still pending), *Access-type* (load or store), *Value-field* (data returned or store value), and *Block-address*. To support *ADIP*, an additional single-bit field is added to MSHR and to the main memory read/write buffer (*MM-RWB*) named as *Fill-DRAM* that indicates whether an incoming block when brought from off-chip memory should be inserted into the L4 DRAM cache or not. To support the *SB-policy*, an additional field is added to MSHR and to the L4$-DRAM read/write buffer (*L4$-DRAM-RWB*) named as *Seg-Row* that indicates the DRAM cache row number that will house the relevant block.

MMap$ segment miss: *Seg-Row* is assigned to each MMap$ segment on a segment miss as illustrated in Figure 5.6.

MMap$ segment hit: *Seg-Row* is read from the MMap$ hit entry (Figure 5.5) of the MMap$ and forwarded to the dispatcher (see Figure 5.1).

L3 SRAM cache miss/MMap$ hit: For a MMap$ hit after the L3 miss, the dispatcher (see Figure 5.1 and Figure 5.7) forwards the request to the L4 DRAM cache access scheduler by allocating an entry in the DRAM read/write buffer (*L4$-DRAM-RWB*; see Figure 5.1). When the data is returned from DRAM cache, it is forwarded to the requesting core and filled in its private L1/L2 caches and the shared L3 SRAM cache.

L3 SRAM cache miss/MMap$ miss: For a MMap$ miss after the L3 miss, the *Fill-DRAM* field of the MSHR entry is determined by *ADIP* as illustrated in Figure 5.1 and Figure 5.7. The dispatcher forwards the request to the main memory access scheduler by allocating an entry in the

MM-RWB. If the *Fill-DRAM* field of the MSHR entry is 1, the dispatcher additionally allocates an entry in the *L4$-DRAM-RWB*. When the data is returned from main memory to *MM-RWB*, the *Fill-DRAM* field of the *MM-RWB* entry is checked. If the *Fill-DRAM* field is 1, then the block is forwarded to the respective *L4$-DRAM-RWB* entry so that the block is filled in L4 DRAM cache. If the *Fill-DRAM* field is 0, then the block bypasses the DRAM cache. Independent of the *Fill-DRAM* field, the data is forwarded to the requesting core and filled in L1/L2 and L3 SRAM cache.

5.5 Overhead

The proposed *ADIP* needs seven LFSRs (six for sampled sets of leader set and one for non-sampled sets; one LFSR requires six XOR gates and six flip flops), three multiplexers and three 10-bit policy selection counters per core as shown in Figure 5.3. Altogether, an *N*-core system requires 7*N* LFSR (56 LFSR for an 8-core system), 3*N* multiplexers (24 multiplexers for an 8-core system), and 3*N* 10-bit policy selection counters (24 policy selection counters for an 8-core system). It requires a single bit per *MSHR* and *MM-RWB* entry for the *Fill-DRAM* field which requires a storage overhead of 64 bits (8 bytes) for a 32-entry MSHR and a 32-entry *MM-RWB*. Storing the DRAM row number in the MMap$ for *SB-policy* increases the size of the MMap$ entry by $\log_2(R)$ bits where R is the number of rows in a DRAM bank. For a 2MB MMap$ with R = 1024, this would lead to a storage overhead of ~200KB for the *SB-policy*. The other overhead for the proposed *SB-policy* is the $\log_2(R)$-bit round-robin row selection logic for each core. Altogether, the proposed policies presented in this chapter comes with negligible hardware overhead.

5.6 Experimental Results

The parameters for the cores, caches and off-chip memory are the same as used in the experimental setup in Section 4.2 (see Table 4.1) with various workloads from SPEC2006 [5] listed in Table 4.3. This section uses 2KB row size for comparisons. However, the concepts proposed in this chapter can be applied for other row sizes (e.g. 4KB or 8KB) as well. For evaluation, this section compares the proposed Adaptive DRAM insertion policy (*ADIP*; details in Section 5.2) and *SB-policy* (details in Section 5.3) on top of state-of-the-art DRAM cache organizations namely LH-Cache [77] (discussed in Section 2.4.1) and Alloy-Cache [102] (details in Section 2.4.3). The main drawback of these works is that they statically determine the DRAM insertion policy for an incoming block and suffer from inter-core DRAM interference, whereas the proposed *ADIP* adapts the DRAM insertion probability at run-time on a per-core basis. In addition, the proposed *SB-policy* further improves the performance via improved DRAM cache set utilization.

5.6.1 *ADIP* and *SB-policy* on top of LH-Cache [78]

This subsection evaluates the performance impact of applying the proposed adaptive DRAM insertion policy (*ADIP*) and *SB-policy* on top of the state-of-the-art DRAM cache organization namely LH-Cache [78]. For evaluation, it compares the following different policies on top of LH-Cache:

1. Original LH-Cache with static DRAM insertion policy namely *LH* using traditional least recently used policy (details in Section 2.1.1)
2. State-of-the-art replacement policy for set-associative caches namely *LH-TAP*, where *TAP* stands for **T**hread **A**ware **P**lacement policy proposed in [51].
3. Proposed adaptive DRAM insertion policy (*ADIP*; details in Section 5.2) namely *LH-ADIP*
4. Proposed *ADIP* and *SB-policy* (details in Section 5.3) namely *LH-ADIP-SB*

Figure 5.8 shows the average normalized harmonic mean instruction per cycle (HM-IPC) throughput results with the speedup normalized to *LH* [78]. On average, the combination of *ADIP* and *SB-policy* improves the overall HM-IPC speed by 14.3% and 6.9% compared to *LH* [78] and *LH-TAP* [51] respectively. On average, the proposed *LH-ADIP* policy alone improves the overall HM-IPC speed by 13% and 5.8% compared to *LH* and *LH-TAP* respectively. Thus, the proposed *SB-policy* provides additional 1.3% improvement in performance compared to *LH-ADIP*. On average, the proposed *LH-ADIP-SB* policy improves the HM-IPC speed of latency sensitive applications by 15.9% and memory sensitive by 13.2% compared to *LH* for an 8-core system.

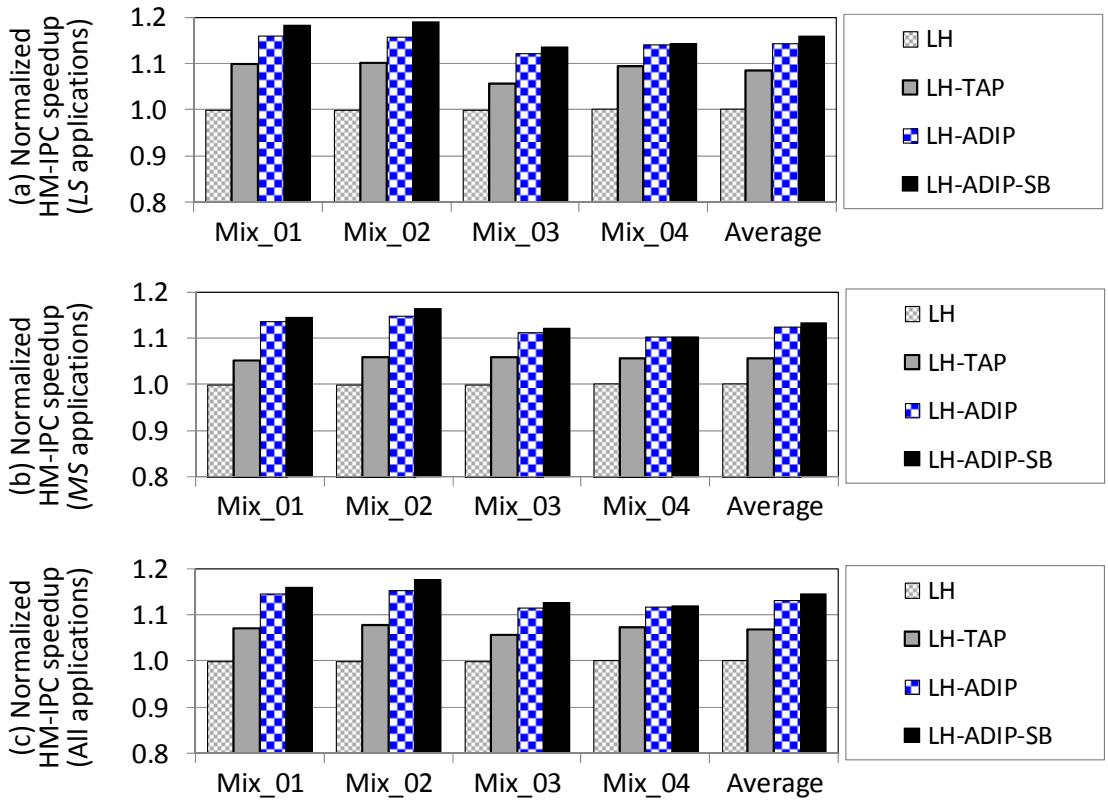


Figure 5.8: Normalized HM-IPC speedup compared to *LRU* [77] for (a) *Latency Sensitive* (*LS*) applications (b) *Memory Sensitive* (*MS*) applications (c) Both *LS* and *MS* applications

5.6.2 Impact on DRAM cache bandwidth and capacity utilization

The *LH* policy [78] does not work well with applications that have thrashing behavior and it suffers from inter-core cache contention. In the *LH* policy, thrashing applications insert a large

number of blocks in the DRAM cache and as a result, they evict useful blocks belonging to other applications. The eviction of useful blocks increases the contention between thrashing and non-thrashing applications causing inter-core cache contention. To mitigate inter-core cache contention, *LH-TAP* [51] adapts the cache replacement policy at runtime by tracking run-time miss rate information of all concurrently executing applications. However, *LH-TAP* still inserts blocks into DRAM cache with a probability of 1 which causes inter-core DRAM interference by increasing unnecessary fill requests from thrashing applications. The performance improvement of the proposed *LH-ADIP* policy over *LH-TAP* is mainly due to reduced inter-core DRAM interference because *LH-ADIP* chooses low insertion probabilities for thrashing applications with long reuse distance.

Figure 5.9 shows the distribution of DRAM cache accesses. The four bars show the different types of DRAM cache accesses as fraction of all accesses for *LH* [78], *LH-TAP* [51], *LH-ADIP* [Proposed], and *LH-ADIP-SB* [Proposed] (from left to right). The cache accesses are categorized as:

1. demand hits for read and write requests.
2. fill requests when the data is filled into DRAM cache for the first time.
3. writeback requests (i.e. when the dirty data is written back from L3 SRAM cache)

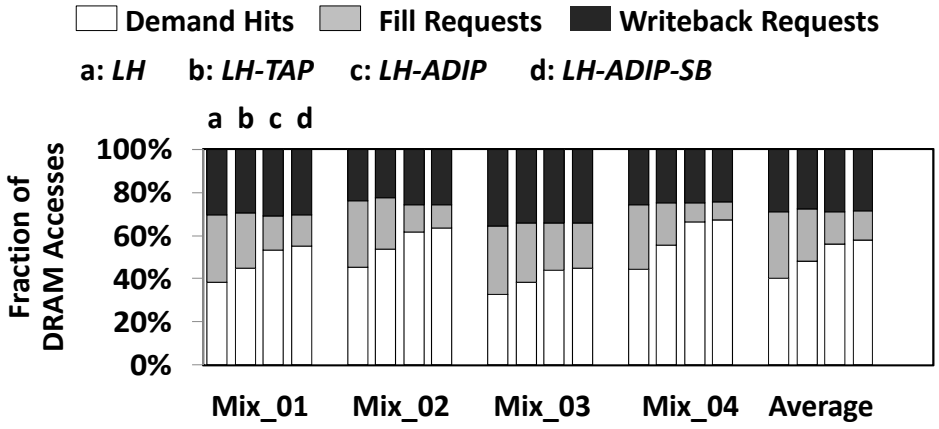


Figure 5.9: Distribution of DRAM cache accesses for different policies

Note that the distribution does not include the bypassed blocks for *LH-ADIP*, and *LH-ADIP-SB* policies. On average, the proposed *LH-ADIP-SB* increases the percentage of demand hits by 43.2%, 19.9% and 2.5% compared to *LH*, *LH-TAP*, and *LH-ADIP* [Proposed], respectively. On average, the proposed *LH-ADIP-SB* reduces the percentage of fill request by 56%, 43.8% and 8.1% compared to *LH*, *LH-TAP*, and *LH-ADIP* [Proposed], respectively. By reducing the intensity of fill requests and increasing the percentage of demand hits using *ADIP* and *SB-policy*, the proposed policies mitigate a major disadvantage of shared DRAM caches, namely inter-core DRAM interference. Thus, the proposed policies enables efficient utilization of DRAM cache capacity and bandwidth via increased demand requests and reduced fill requests, respectively.

5.6.3 Impact on miss rate

The proposed *LH-ADIP-SB* policy increases the effective DRAM cache capacity via reducing the insertion rate of rarely-reused blocks (i.e. using *ADIP*) and via the use of efficient set balancing (i.e. using *SB-policy*). The effective utilization of DRAM cache capacity reduces the miss rate, which results in reduced contention on off-chip memory bandwidth. Figure 5.10 illustrates this observation comparing different policies in terms of DRAM cache miss rate. On average, the proposed *LH-ADIP-SB* policy reduces the overall DRAM cache miss rate by 24.8% and 12.7% compared to *LH* and *LH-TAP*, respectively. In addition, the proposed *LH-ADIP-SB* policy reduces the DRAM cache miss rate of latency sensitive application by 70% and 42.4% compared to *LH* and *LH-TAP*, respectively.

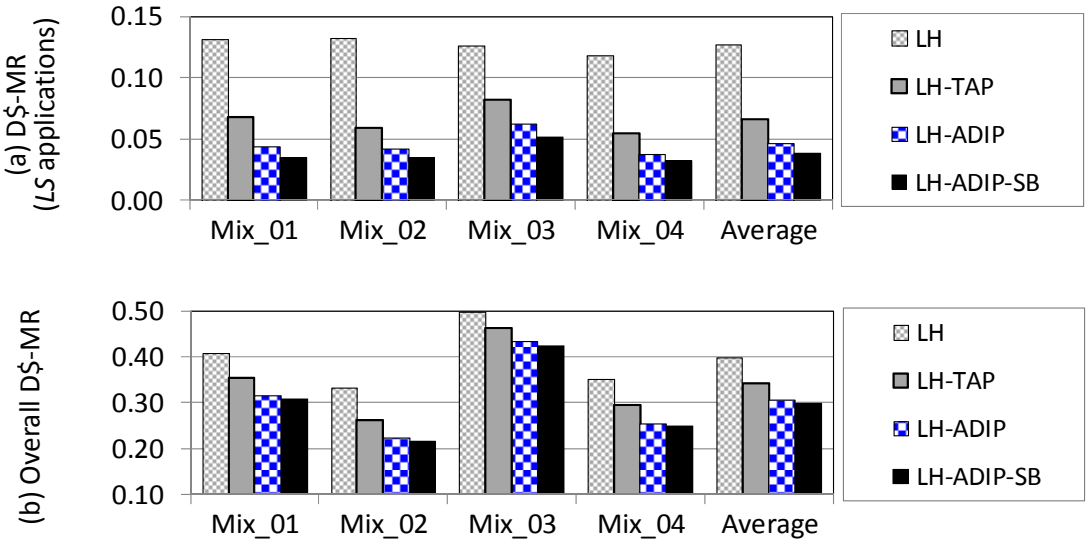


Figure 5.10: (a) DRAM cache miss rate (D$-MR) for Latency Sensitive (*LS*) applications (b) Overall DRAM cache miss rate

5.6.4 *ADIP* Run-time adaptivity

Comparing across the applications, it has been found that the reuse distance of some applications (470.lbm, 437.leslied.ref, 462.libquantum, 450.soplex) change during different phases of their execution, which shows the fundamental advantage of the proposed adaptive DRAM insertion policy. Figure 5.11 illustrates this observation showing the DRAM insertion probability that *ADIP* automatically selects at runtime for the applications running in Mix_01 (see Table 4.3). The DRAM insertion probability for each application is sampled once every 2 million cycles and shown in Figure 5.11. The proposed DRAM insertion policy chooses low insertion probabilities at runtime for memory sensitive applications (e.g. 470.lbm, 437.leslied.ref, 433.milc, and 450.soplex) for majority of their execution time. On the other hand, it chooses high insertion probabilities at runtime for latency sensitive applications (e.g. 473.astar.train, 437.leslied.train, 462.libquantum, and 471.omnetpp) for majority of their execution time.

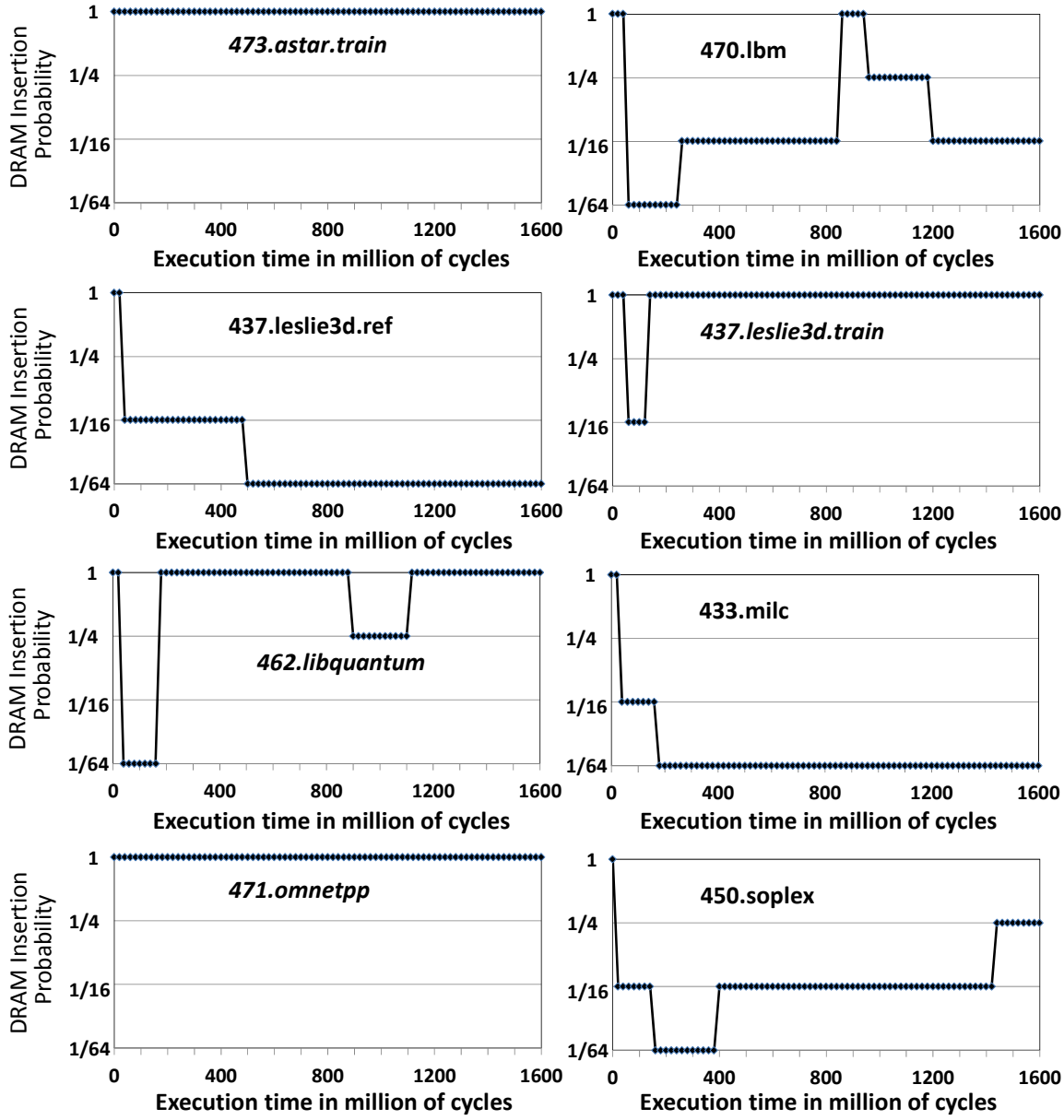


Figure 5.11: Run-time DRAM insertion probability for non-sampled sets of all applications in Mix_01 (see Table 4.3). Latency sensitive applications shown in italic and memory sensitive applications shown in non-italic

5.6.5 *ADIP* and *SB-policy* on top of Alloy-Cache [102]

The primary advantage of the proposed policies is that they can be applied irrespective of DRAM cache organizations and replacement policy and they complement each other. It implies that the proposed *ADIP* and set balancing policies are flexible enough to be applied to any replacement policy and DRAM cache organization. This section shows the performance impact of applying the proposed *ADIP* and *SB-policy* on top of the state-of-the-art DRAM cache organization namely Alloy-Cache [102]. Since Alloy-Cache (details in Section 2.4.3) employs a direct-mapped

cache organization as shown in Figure 2.12, the traditional least recently used and other replacement policies cannot be applied on top of Alloy-Cache.

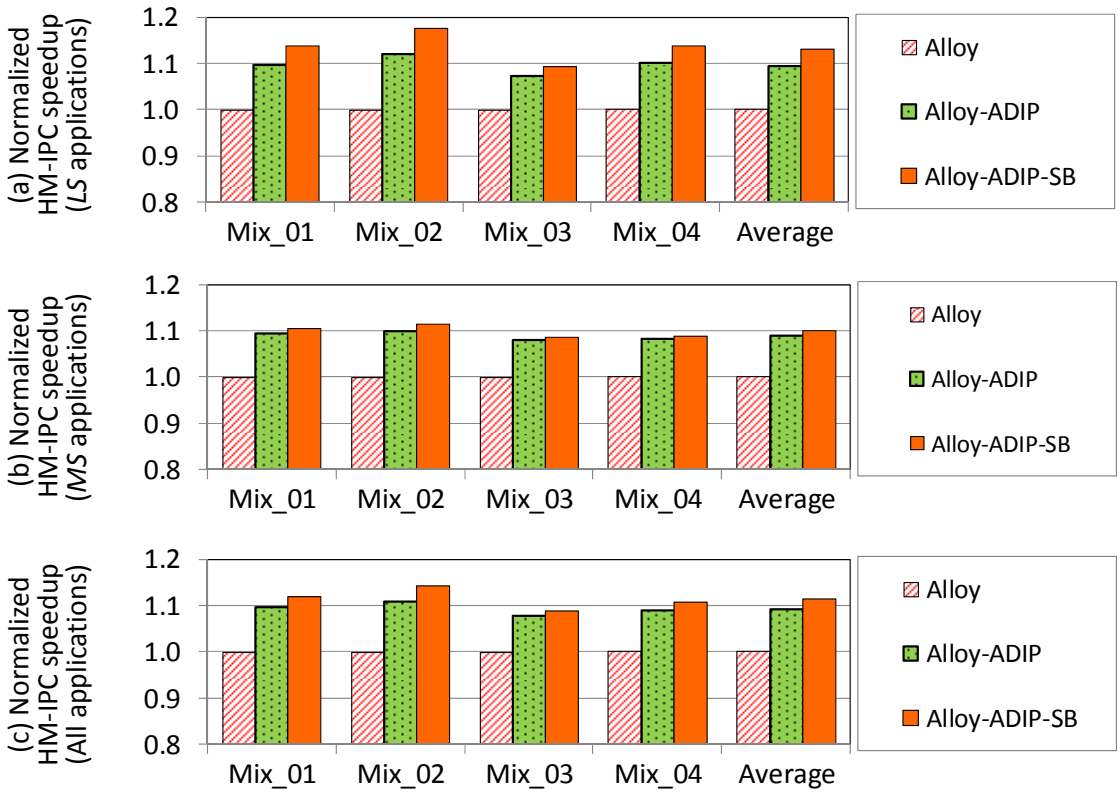


Figure 5.12: Normalized HM-IPC speedup compared to *Alloy* [102] for (a) *Latency Sensitive* (*LS*) applications (b) *Memory Sensitive* (*MS*) applications (c) Both *LS* and *MS* applications

For evaluation, this section provides the following different policies on top of Alloy-Cache:

1. Alloy-Cache with static DRAM insertion policy namely *Alloy*
2. The proposed *ADIP* applied on the top of Alloy-Cache namely *Alloy-ADIP*
3. The proposed *ADIP* and *SB-policy* applied on the top of Alloy-Cache namely *Alloy-ADIP-SB*

Figure 5.8 shows the performance improvement of the proposed policies compared to *Alloy* [78]. On average, the combination of *ADIP* and *SB-policy* improves HM-IPC speedup of latency sensitive applications by 13.1%, memory sensitive applications by 9.9% and overall HM-speedup by 11.3% compared to *Alloy* [102].

5.6.6 Impact of Set Balancing Policy (*SB-policy*)

This section evaluates the performance impact of the *SB-policy* (details in Section 5.3) by comparing *LH-ADIP* (without *SB-policy*) and *LH-ADIP-SB* (with *SB-policy*). It also evaluates the performance impact of the *SB-policy* when applied on top of Alloy-Cache [102] by comparing *Alloy-ADIP* (without *SB-policy*) and *Alloy-ADIP-SB* (with *SB-policy*). On average, *LH-ADIP-SB* reduces the overall DRAM cache miss rate by 2.4% (Figure 5.13-a) compared to *LH-ADIP* via improved set utilization (i.e. storing the DRAM row number in the MMap$ and assigning it in a

round robin fashion). Similarly, *Alloy-ADIP-SB* reduces the overall DRAM cache miss rate by 4% (Figure 5.13-a) compared to the *Alloy-ADIP*. When applied on top of LH-Cache, set balancing improves the HMIPC throughput (Figure 5.8-a) of latency sensitive applications by 1.8% and overall HMIPC throughput (Figure 5.8-c) by 1.3%. For Alloy-Cache, set balancing provides better speedup (3.3% for latency sensitive application with 1.9% overall speedup) as shown in Figure 5.12. When the overall DRAM cache miss rate is high (*Alloy-ADIP* has a higher miss rate compared to *LH-ADIP* as shown in Figure 5.13), set balancing provides greater performance improvement for Alloy-Cache (3.3% improvement in performance of latency sensitive applications via set balancing) compared to LH-cache (1.8% speedup for latency sensitive applications).

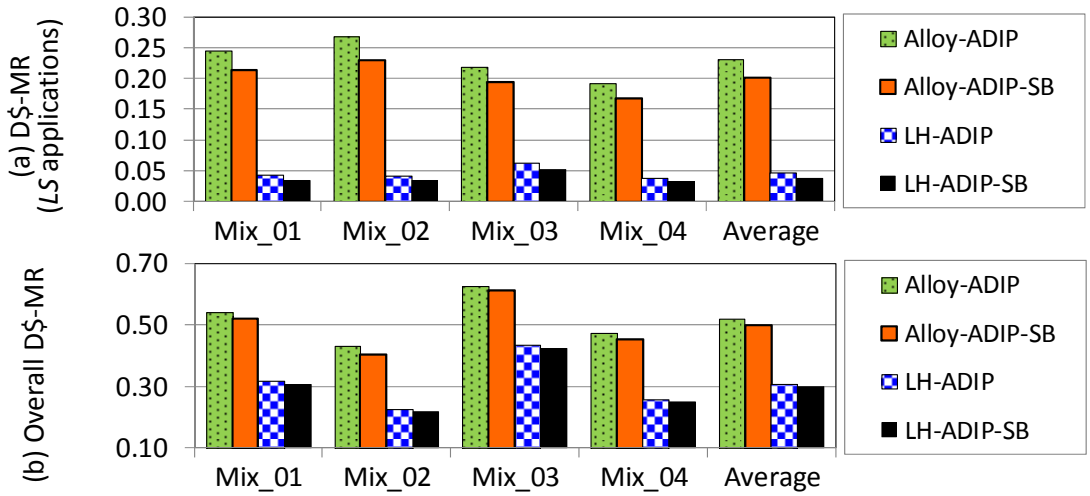


Figure 5.13: (a) DRAM cache miss rate (D$-MR) for Latency Sensitive (*LS*) applications (b) Overall DRAM cache miss rate

5.7 Summary

This chapter showed that that inter-core DRAM interference can cause performance degradation in existing DRAM cache hierarchies [77, 102] when the cache access rate from multiple applications varies significantly. It showed that in order to mitigate inter-core DRAM interference it is necessary to minimize the number of DRAM fill requests from thrashing applications that have large working set sizes. This chapter proposed application and DRAM aware policies for multi-core systems that reduce DRAM fill requests from thrashing applications via an adaptive DRAM insertion policy, thereby reducing inter-core DRAM interference. It also presented a set balancing policy that reduces DRAM cache miss rate via improved capacity utilization, thereby reducing the load on the off-chip bandwidth. This chapter evaluated the proposed policies for various workload mixes and compared it to state-of-the art. The experiments showed that the proposed policies increase the performance (harmonic mean instructions throughput) by 14.3% and 6.9% compared to *LH* [77] and *LH-TAP* [51] at negligible hardware overhead when applied on the top of LH-Cache [77]. They also improve harmonic mean instructions throughput by 11.3% when applied on the top of direct mapped Alloy-Cache [102].

Chapter 6 Policies for Latency Reduction

Memory speed has become a major performance bottleneck as more and more cores are integrated on a multi-core chip. The widening latency gap between high speed cores and memory has led to the evolution of multi-level SRAM/DRAM cache hierarchy comprised of increasing cache sizes and latency at each level. These multi-level SRAM/DRAM cache hierarchies exploit the latency benefits of smaller caches (e.g. private L1 and L2 SRAM caches) and the capacity benefits of larger caches (e.g. shared L3 SRAM and shared L4 DRAM cache) as shown in Figure 6.1. However, they incur high latencies for the larger cache levels due to high tag lookup latency, which may degrade the performance. Therefore, to improve the overall instruction throughput, it is important to reduce the latency of L3 SRAM and L4 DRAM cache. To solve this problem, this chapter proposes policies (highlighted in Figure 6.1) for latency reduction in the cache hierarchy.

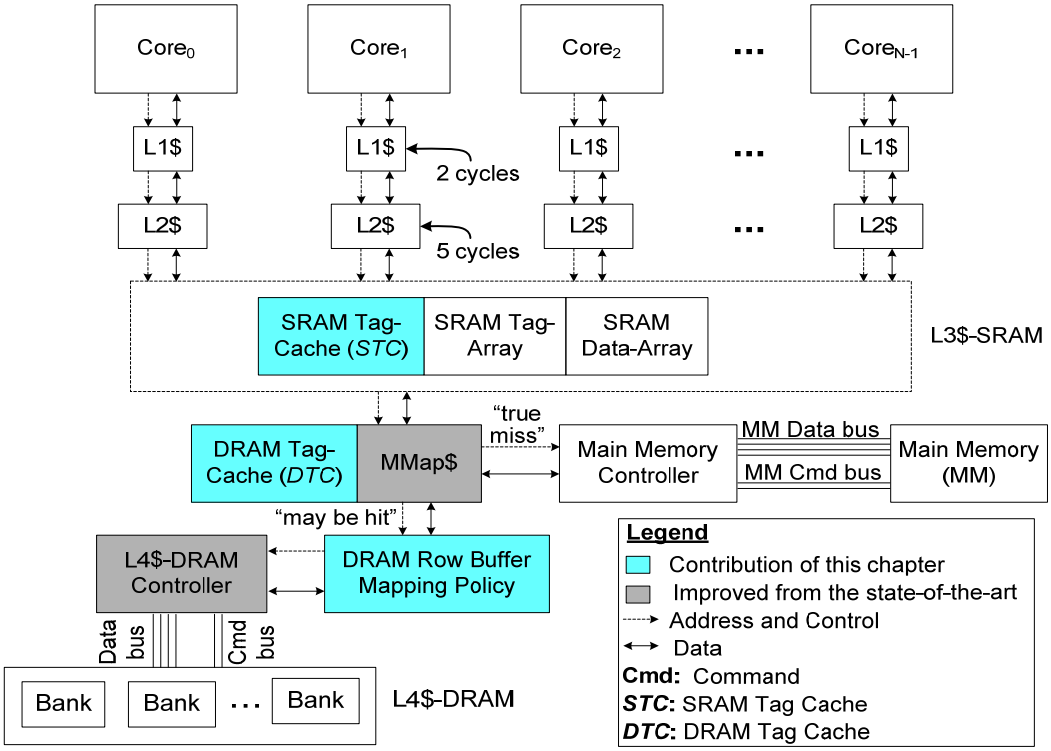


Figure 6.1: SRAM/DRAM cache hierarchy highlighting the novel contributions

This chapter analyzes the design trade-offs in architecting an SRAM/DRAM cache hierarchy and presents different policies for latency reduction. The first section demonstrates that the *DRAM row buffer mapping policy* (i.e. the method by which blocks from main memory are mapped to the row buffer of a particular DRAM cache bank) plays a significant role in determining the overall instruction throughput because it effects L4 DRAM cache hit latency and L4 DRAM cache miss rate. It demonstrates that state-of-the-art row buffer mapping policies [77, 78, 102] are not well suited for improving the aggregate performance of a multi-core system running heterogeneous applications because they are either optimized for L4 hit latency [102] or for L4 miss rate [77, 78]. None of these policies provides a good L4 hit latency and L4 miss rate at the same time. The second section gives an overview of the proposed SRAM/DRAM cache organi-

zation highlighting the novel contributions proposed in this thesis. The third section presents novel row buffer mapping policies that simultaneously target L4 hit latency and L4 miss rate with the goal of achieving the best of both. It provides detailed qualitative comparisons of different row buffer mapping policies and their impact on important parameters such as DRAM cache row buffer hit rate, DRAM cache hit latency, and DRAM cache miss rate. Section 6.5 presents novel low latency SRAM structures namely DRAM Tag-Cache (*DTC*) and SRAM Tag-Cache (*STC*). The *STC* and *DTC* hold the tags of the sets that were recently accessed in L3 and L4 caches, respectively. They provide fast lookup because for a Tag-Cache hit, they quickly identify hit/miss for the larger caches. This chapter further analyzes the effect of different DRAM row buffer mapping policies on the *DTC* hit rate and the overall performance.

6.1 Problems of the State-of-the-art

Recently proposed DRAM row buffer mapping policies for DRAM caches are predominantly optimized for either L4 DRAM cache hit latency or L4 DRAM cache miss rate. Figure 6.2 illustrates this observation by comparing state-of-the-art row buffer mapping policies proposed in [77, 78, 102].

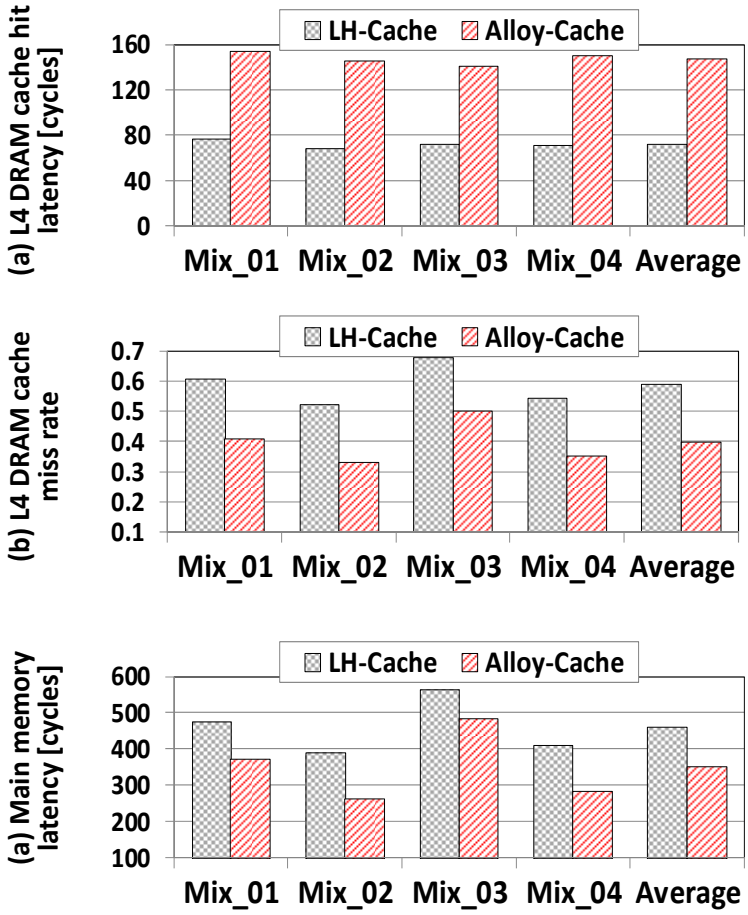


Figure 6.2: (a) L4 DRAM hit latency (b) L4 DRAM miss rate (c) main memory latency

LH-Cache [77, 78] (details in Section 2.4.1) is optimized for L4 DRAM miss rate by providing a high associativity (high associativity reduces the L4 miss rate). The downside of the LH-

Cache row buffer mapping policy is that it has a serialized tag-and-data access with reduced row buffer hit rate that leads to increased L4 DRAM hit latency. The Alloy-Cache row buffer mapping policy (explained in Section 2.4.3) [102] optimizes the L4 DRAM hit latency because it provides fast tag lookup and improved row buffer hit rate. This comes at the cost of increased L4 miss rate because it employs direct mapped cache. Figure 6.2 shows the L4 hit latency, L4 miss rate and main memory latency experienced by LH-Cache and Alloy-Cache for an 8-core system. The parameters for the cores, caches and off-chip memory are the same as used in the experimental setup in Chapter 3 (see Table 4.1 and Table 4.2) with various workloads from SPEC2006 [5] listed in Table 4.3. On one extreme, LH-Cache [77, 78] has a high L4 hit latency compared to the Alloy-Cache [102] as depicted in Figure 6.2-(a). On the other extreme, Alloy-Cache has a high L4 miss rate compared to LH-Cache as depicted in Figure 6.2-(b). The higher L4 miss rate of Alloy-Cache also leads to a higher main memory access latency compared to LH-Cache (Figure 6.2-c) due to increased contention in the main memory controller. This chapter proposes policies for the DRAM cache that minimize both L4 hit latency (via improved row buffer and *DTC* hit rate) and L4 miss rate (via high associativity) at the same time in order to improve the overall instruction throughput.

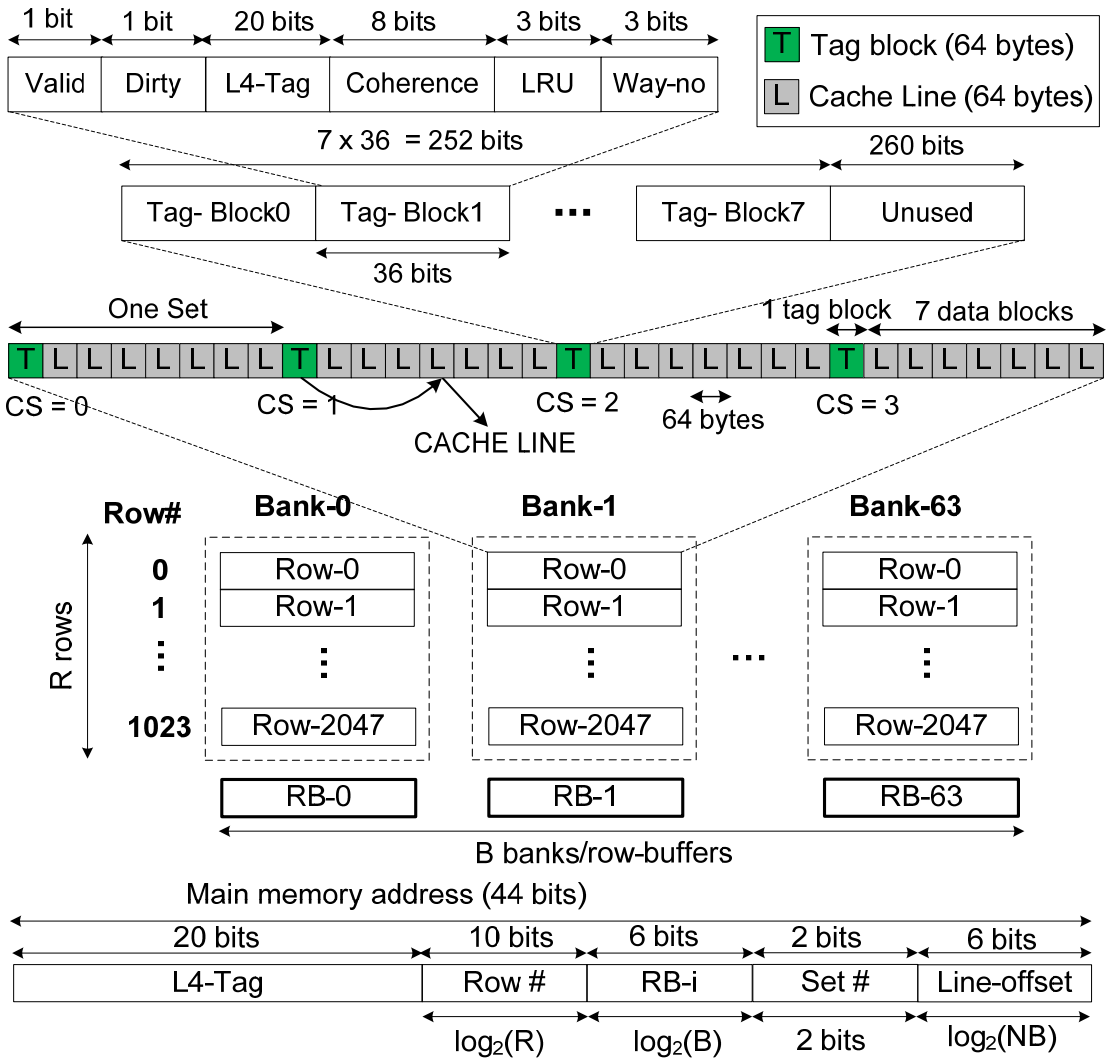
6.2 Proposed SRAM/DRAM Cache Organization

Figure 6.1 shows the organization of the proposed SRAM/DRAM cache organization along with a MMap$ (details in Section 2.4.1), highlighting the novel contributions proposed in this chapter. Similar to state-of-the-art [77, 78, 102], the proposed approach stores the tags in the DRAM cache and employs MMap$ to identify DRAM cache hit/miss. This chapter presents the following novel contributions:

1. This chapter proposes novel DRAM row mapping policies (details in Section 6.3) after analyzing that state-of-the-art row buffer mapping policies [77, 78, 102] do not work well because they are optimized only for a single parameter (L4 DRAM hit latency or L4 DRAM miss rate). The proposed row buffer mapping policies reduce the L4 DRAM hit latency via improved row buffer hit rates and they reduce the L4 DRAM miss rate via a high associativity.
2. This chapter proposes a small and low latency SRAM structure namely DRAM Tag-Cache (*DTC*; details in Section 6.5.1) that allows most L4 DRAM tag accesses to be serviced at reduced access latency compared to when tags are accessed from the DRAM cache. Accesses that hit in the *DTC* are serviced with a lower latency compared to accesses that miss in the *DTC*. This chapter further analyze the effect of different row buffer mapping policies from state-of-the-art [77, 78] and the proposed row mapping policy on *DTC* hit rate.
3. The proposed row buffer mapping policy allows reducing the size of existing MissMap cache (MMap$) [77, 78] organization which is used by state-of-the-art DRAM cache to provide hit/miss information (details in Section 2.4.1). Reducing the size of the MMap$ may reduce the DRAM cache hit/miss prediction accuracy. This chapter analyze the effect of different MMap$ sizes on the DRAM cache hit/miss prediction accuracy.
4. This chapter modifies the DRAM cache controller (detail in Section 6.5.3), which further reduces the L4 DRAM hit latency for the various row buffer mapping policies proposed in this chapter.

6.3 DRAM Row Buffer Mapping Policies

This section introduces novel DRAM-aware row buffer mapping policies that simultaneously optimize L4 DRAM hit latency and L4 DRAM miss rate. The proposed row buffer mapping policies are based on the notion that the latency of L4 DRAM can be reduced by improving row buffer hit rate. This section assumes 2KB row size for qualitative comparisons because the same row buffer size is used by state-of-the-art for DRAM cache [77, 78, 102]. However, the concepts proposed in this section can also be applied for other row sizes.



B: Number of DRAM banks (here: 64 banks)
RB-**i**: Row buffer associated with bank *i* (size of row buffer is 2KB = 2048 bytes)
R: Number of rows in a DRAM bank (here: 1024 rows/bank)
Row #: DRAM cache Row number with in a particular
Set #: Set number with in a particular row
NB: Number of bytes in a cache line (here: 64 bytes/line)
CS: Cache Set number determined by least significant two bits of memory block address

Figure 6.3: Row Buffer Mapping with Associativity of 7 (RBM-A7)

6.3.1 Row Buffer Mapping Policy with an Associativity of Seven (RBM-A7)

The DRAM row organization for the proposed **R**ow **B**uffer **M**apping policy with an **A**ssociativity of **7** (RBM-A7) is illustrated in Figure 6.3, where each 2KB DRAM row comprises 4 cache sets with a 7-way set associativity. Each cache set consists of 1 tag block (64 bytes = 512 bits) and 7 cache lines. The 7 cache lines need 7 x 36 = 252 bits for their tag entries with 260 bits left unused. An L4 DRAM cache access must first read the tag block before accessing the cache line. After an L4 DRAM hit is detected by the MMap$, the row buffer is reserved until the tag block and cache line are both read from it. This guarantees a row buffer hit for the cache line access after the tag block is accessed.

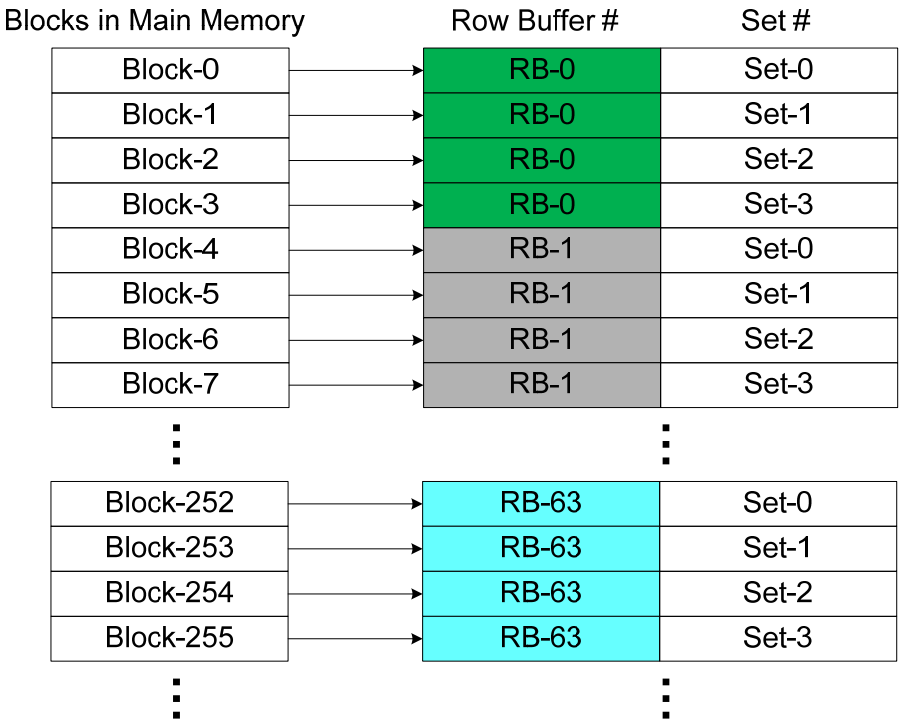


Figure 6.4: Memory block mapping for the proposed RBM-A7 policy

Figure 6.3 illustrates how RBM-A7 maps blocks from main memory to the row buffers. The DRAM cache row number within a bank (indicated by the "Row#" field) is determined by the memory block address. A high row buffer hit rate can effectively amortize the high cost of a DRAM array access by reducing the hit latency. RBM-A7 exploits the row buffer locality by mapping 4 spatially close blocks to the same row buffer. The DRAM cache set number (each DRAM row contains 4 cache sets as shown in Figure 6.3) within a row is determined by the least significant two bits of the memory block address as illustrated in Figure 6.3.

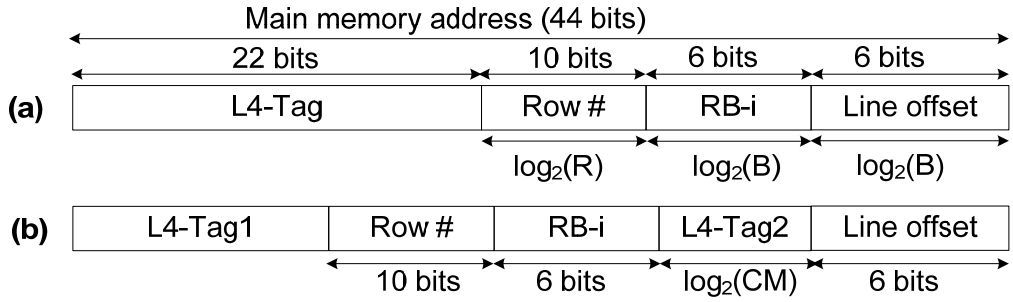
Figure 6.4 illustrates how memory blocks are mapped to a row buffer and to a DRAM cache set within a row buffer. The RBM-A7 policy maps 4 consecutive memory blocks to the same DRAM row buffer. For instance, blocks 0-3 from main memory are mapped to RB-0 (i.e. row buffer associated with Bank-0) and blocks 4-7 are mapped to RB-1 (i.e. row buffer associated with Bank-1) as depicted in Figure 6.4. The DRAM cache set number within a row is determined

by the two least significant bits of the memory block address as illustrated in Figure 6.4. For instance, block-4 is mapped to set-0 and block-7 is mapped to set-3 of RB-1.

The proposed RBM-A7 policy reduces the L4 DRAM hit latency via reduced tag serialization latency (details in Section 6.3.3) and high row buffer hit rate (evaluated in Section 6.7.2) compared to the LH-Cache [77, 78] (details of LH-Cache in Section 2.4.1). A cache access in the RBM-A7 policy must first access one tag block (in contrast to three tag blocks accesses in the LH-Cache) before having an access to the cache line. This reduces tag serialization latency compared to the LH-Cache. The row buffer hit rate is improved compared to the LH-Cache because RBM-A7 maps four spatially close blocks to the same row buffer. In contrast, the LH-Cache has a reduced row buffer hit rate due to reduced spatial locality because it maps consecutive blocks to different row buffers. The primary advantage of the RBM-A7 policy over the Alloy-Cache [102] (details in Section 2.4.3) is that it minimizes the L4 DRAM miss rate via high associativity (7-way associative cache) compared to the direct mapped (1-way associative cache) Alloy-Cache.

6.3.2 Configurable Row Buffer Mapping Policy (CRBM)

The major difference between the configurable row buffer mapping policy and the LH-Cache [77, 78] is explained in the following.



NB: Number of bytes in a cache line (here: 64 bytes/line)
B: Number of DRAM banks (here: 64 banks)
RB-i: Row buffer associated with bank *i*
R: Number of rows in a DRAM bank (here: 1024 rows/bank)
CM: Consecutive blocks mapped to the same row
L4-Tag [22 bits] = {L4-Tag1, L4-Tag2}

Figure 6.5: Row buffer mapping policy used by (a) LH-Cache [77] (b) CRBM policy [Proposed]

(a) Row buffer mapping

The configurable row buffer mapping policy is based on the observation that the latency of L4 DRAM can be reduced by improving the row buffer hit rate. The differences between the row buffer mapping policy employed in LH-Cache [77, 78] and the one used in the configurable row buffer mapping policy is shown in Figure 6.5 (a-b). To exploit spatial and temporal locality, the row buffer hit rate can be improved by mapping more consecutive blocks to the same DRAM cache row buffer. Figure 6.5-(b) shows the configurable row buffer mapping policy where the row buffer hit rate depends upon the parameter CM (defined as number of consecutive memory blocks mapped to the same row). CM is chosen as a power of two. If CM is equal to 1, then spa-

tially close blocks are mapped to different row buffers and the row buffer mapping policy is the same as employed by LH-Cache.

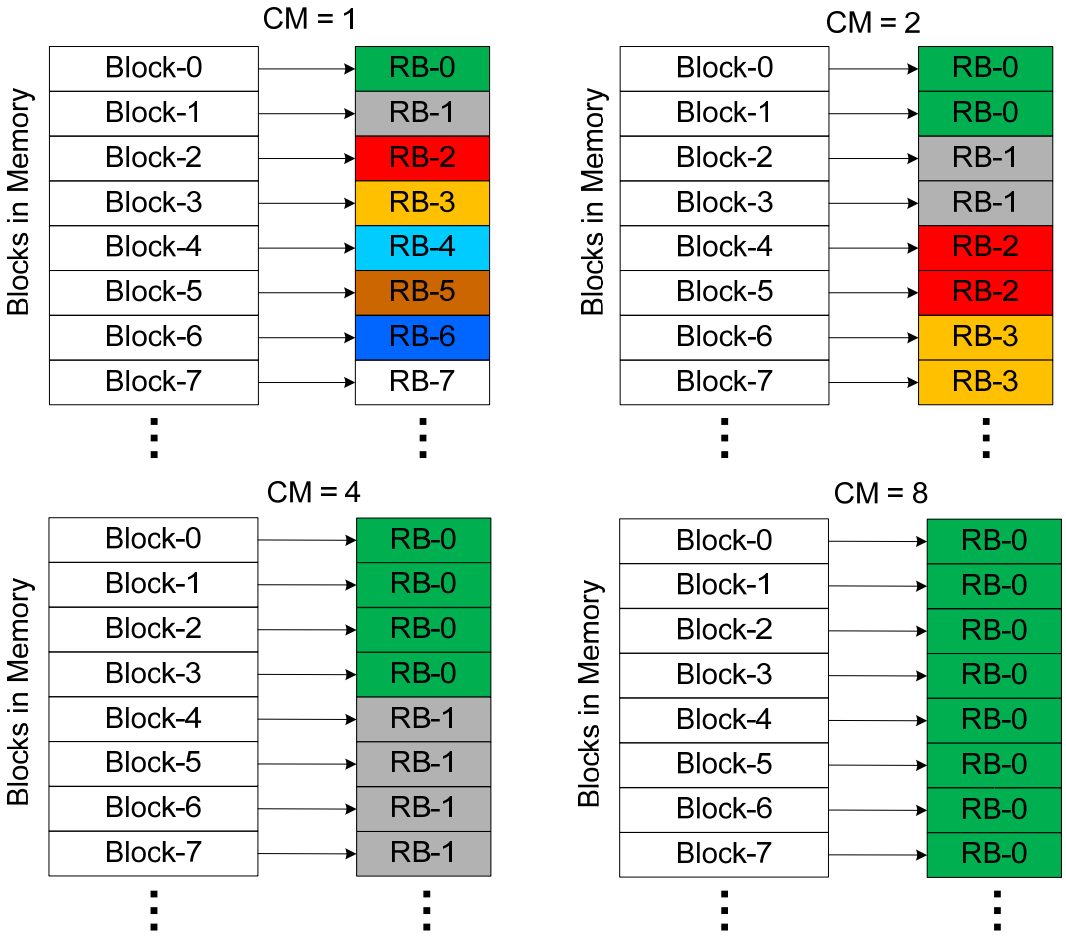


Figure 6.6: Block mapping for configurable row buffer mapping policy with different values of CM

Figure 6.6 shows how blocks from main memory are mapped to a row buffer for different values of CM. Figure 6.7 shows the row buffer hit rate for different values of CM for an 8-core system. Increasing CM improves the row buffer hit rate (1.1% for CM = 1, 14.1% for CM = 2, 22.2% for CM = 4, 28.5% for CM = 8, and 35% for CM = 16) which comes at the cost of increased DRAM cache miss rate due to reduced set-level-parallelism (because high order address bits are used to select DRAM cache row/set) and increased conflict misses (spatially close main blocks increases conflict misses within a set). The result section explores the trade-offs between L4 DRAM hit latency (depends upon row buffer hit rate) and L4 DRAM miss rate for different values of CM (1, 2, 4, 8, 16) and its impact on the overall performance. The primary advantage of the configurable row buffer mapping policy is that it benefits from a high associativity (30-way associativity; see Figure 6.10-b) that reduces conflict misses compared to Alloy-Cache (1-way associativity) and RBM-A7 policy (7-way associativity). At the same time, it can provide a high row buffer hit rate compared to the LH-Cache via judicious selection of CM without significantly degrading the L4 DRAM miss rate.

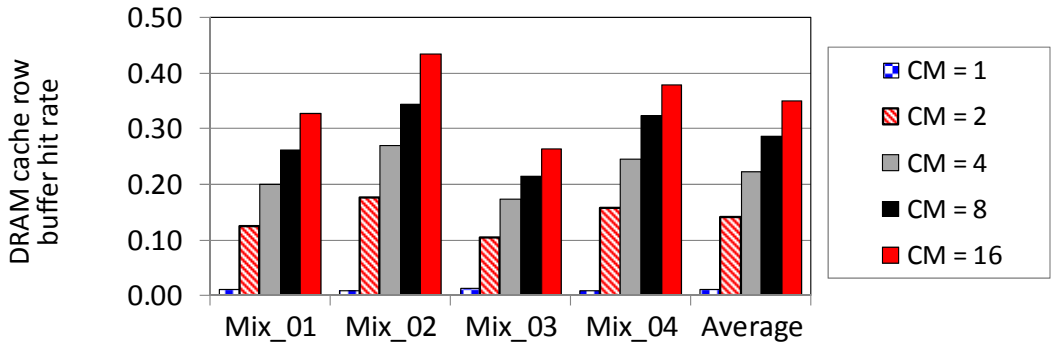


Figure 6.7: DRAM cache row buffer hit rate for different values of CM

(b) Tag block mapping and organization

The LH-Cache [77, 78] uses the traditional least recently used (LRU) replacement policy (details in Section 2.1.1) for cache replacement. The overhead of the LRU policy is 42 bits per cache line (1 valid bit, 1 dirty bit, 22 tag bits to identify presence/absence, 5 bits to track the priority of the cache line using LRU bits, 5 bits to track the location of the cache line, and eight coherence bits for an 8-core system) as shown in Figure 6.8-(a).

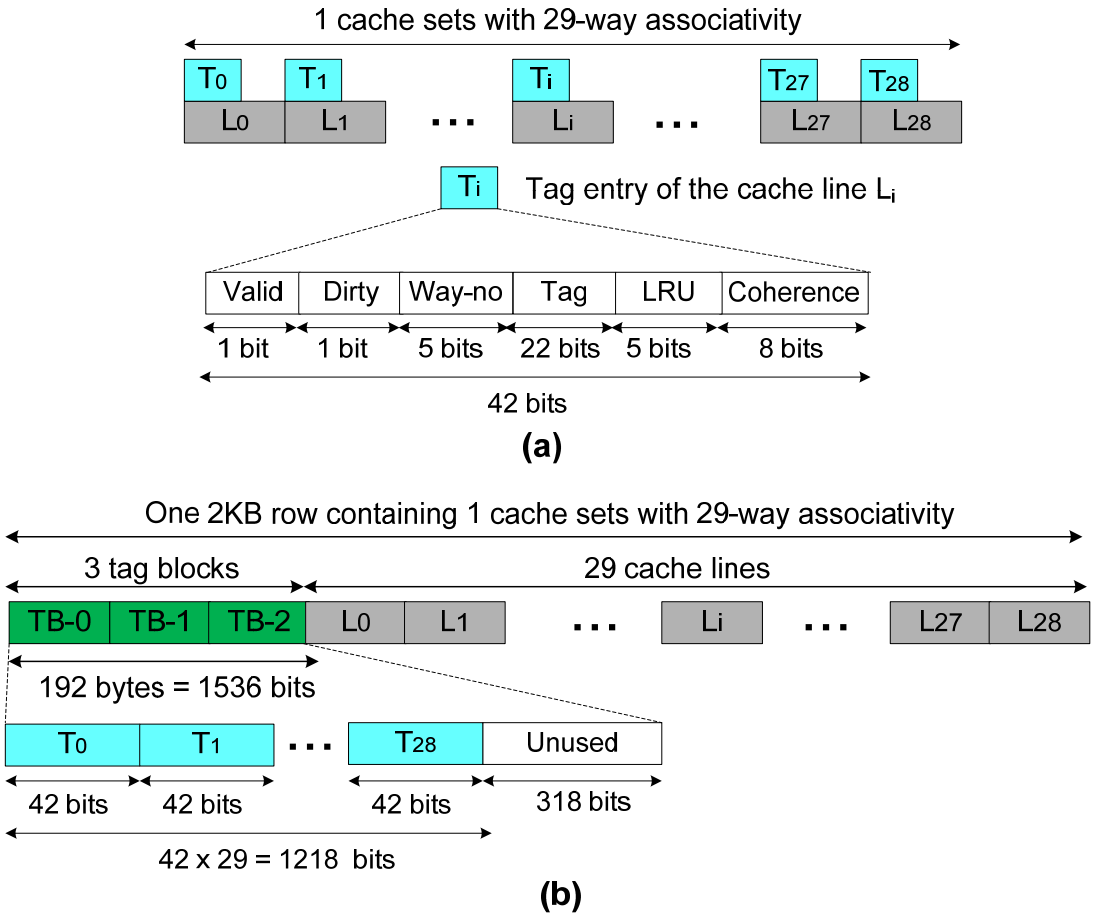


Figure 6.8: (a) Overview of LRU policy with 29-way associative cache (b) how the tag entry fields are organized in LH-Cache [77, 78]

Figure 6.8-(a) shows the logical organization of LH-Cache where one DRAM cache set consists of 29 cache lines entries. Figure 6.8-(b) shows how the tag entry fields are mapped into the blocks of the DRAM cache in LH-Cache. The tag block mapping used by LH-Cache leads to increased L4 DRAM cache hit latency because it always requires reading 3 tag blocks from the DRAM cache before reading the data block (see Figure 6.9). LH-Cache also unnecessarily consumes DRAM cache bandwidth as it always requires writing 3 tag blocks (192 bytes) on a limited-size (16 bytes) DRAM cache channel (see Figure 6.9) when the tag blocks are modified. Note that the LH-Cache requires writing 3 tag blocks on a hit to update the LRU information which are stored in TB-0, TB-1, and TB-2 (see Figure 6.8-c).

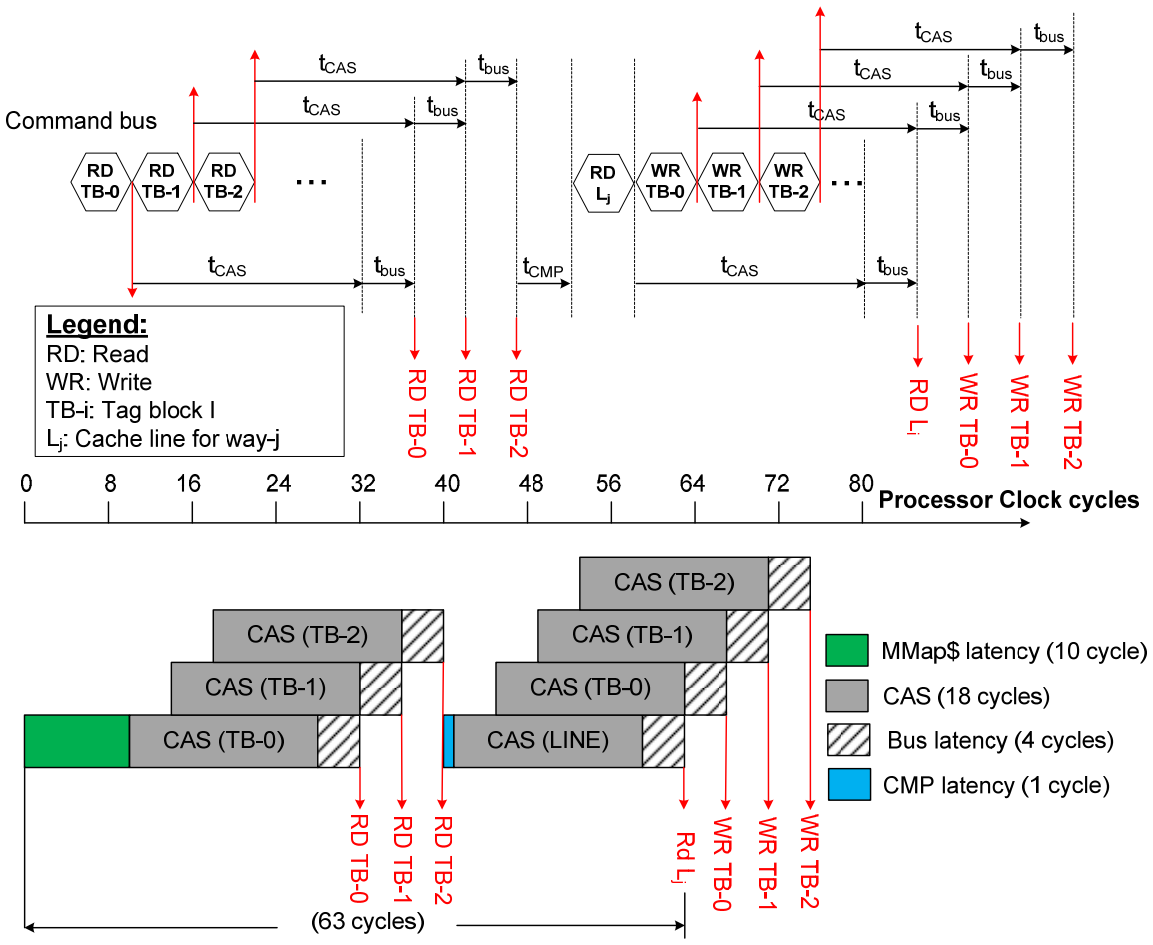


Figure 6.9: Timing and sequence of commands for L4 DRAM hit that his in the row buffer for LH-Cache

To reduce the L4 DRAM hit latency and to improve the DRAM cache bandwidth utilization, the configurable row buffer mapping policy employs the clock-based "pseudo-LRU" replacement policy [111] that reduces the overhead compared to the LRU policy. The overhead of the pseudo-LRU policy is 33 bits per cache line (1 valid bit, 1 dirty bit, 1 used bit, eight coherence bits for an 8-core system, and 22 tag bits) as shown in Figure 6.10-(a). In contrast, the overhead of LRU policy (used by LH-Cache) is 42 bits per cache line as shown in Figure 6.8-(a). The valid bit indicates whether a cache line contains a valid (valid bit is 1) or invalid block (valid bit is 0) from main memory. All the valid bits of each cache line are set to zero on power or cache reset. The dirty bit of cache line indicates whether the block from main memory has been modified by the

processor (dirty bit is 1) or remained unchanged (dirty bit is 0) since it was fetched from main memory.

For each cache set, the pseudo-LRU policy requires a single counter (5 bits) called "clock pointer" to track the current clock position. On insertion, a cache line clears its used bit and the clock pointer points to the next cache line, while the tag of the cache line is inserted. A dirty bit is set on cache line writeback and the used bit is set on cache line hit. On eviction, the cache line pointed by "clock pointer" is checked. If its used bit is zero, the cache line is evicted. Otherwise the policy clears the used bit and gives a second chance to the cache line by advancing the clock pointer. It repeats the same check until it finds a cache line with a used bit with value zero.

Figure 6.10-(a) shows the logical organization of one DRAM cache set for the configurable row buffer mapping policy that consists of 30 cache lines. Each cache set consists of 2 tag blocks and 30 cache lines as shown in Figure 6.10-(b). Each tag entry in the "pseudo-LRU" replacement policy requires 22-bits tag and 11 bits for replacement flags (valid bit, dirty bit, used, and coherence bits) to identify hit/miss. These replacement flags are updated on a cache hit (i.e. set used bit), cache writeback (set dirty bit) and updating coherence bits while the tag bits remain unchanged. Both, the replacement flags and the tags are updated on cache fill (i.e. when the data is filled in the DRAM cache).

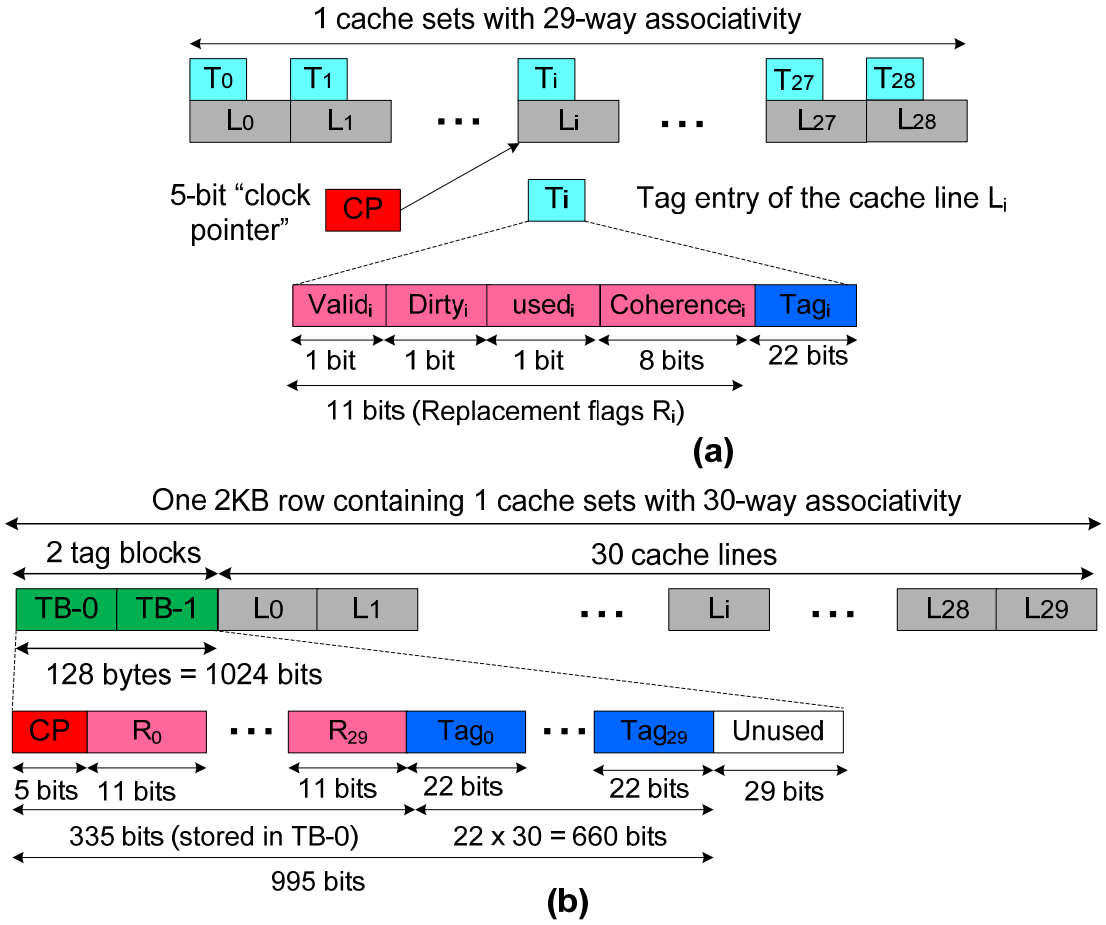


Figure 6.10: (a) Overview of "pseudo LRU" policy (b) how the tag entry fields are organized in configurable row buffer mapping policy

An L4 DRAM access in the configurable row buffer mapping policy must first access two tag blocks (in contrast to three tag block accesses in the LH-Cache) before having an access to the cache line. This reduces the L4 DRAM hit latency for a read request compared to the LH-Cache. The L4 DRAM hit latency for a row buffer hit in the LH-Cache is 63 cycles as shown in Figure 6.9. In contrast, the L4 DRAM hit latency for a row buffer hit in the configurable row buffer mapping policy is 59 cycles as shown Figure 6.11.

Figure 6.10-(b) shows how the cache line and the tag entry fields are organized in the configurable row buffer mapping policy (CRBM). The replacement flags in CRBM are stored in TB-0 (tag block zero), while the tags are stored in TB-0 and TB-1 as illustrated in Figure 6.10-(b). In the configurable row buffer mapping policy, only TB-0 needs to be written back on a cache hit, cache writeback, and on updating coherence information as shown in Figure 6.11. This would require 64 bytes to be transferred on the DRAM cache channel instead of transferring 192 bytes required for the LH-Cache. However, both tag blocks TB-0 and TB-1 need to be written into DRAM cache on a cache fill (requires 128 bytes to be transferred on DRAM cache channel). Since, the percentage of fill request (as fraction of all DRAM cache accesses) is very low (less than ~15% on average), this optimization avoids unnecessary DRAM cache bandwidth wastage for writing tag blocks that are not changed. This leads to a reduced L4 DRAM hit latency compared to LH-Cache via efficient bandwidth utilization.

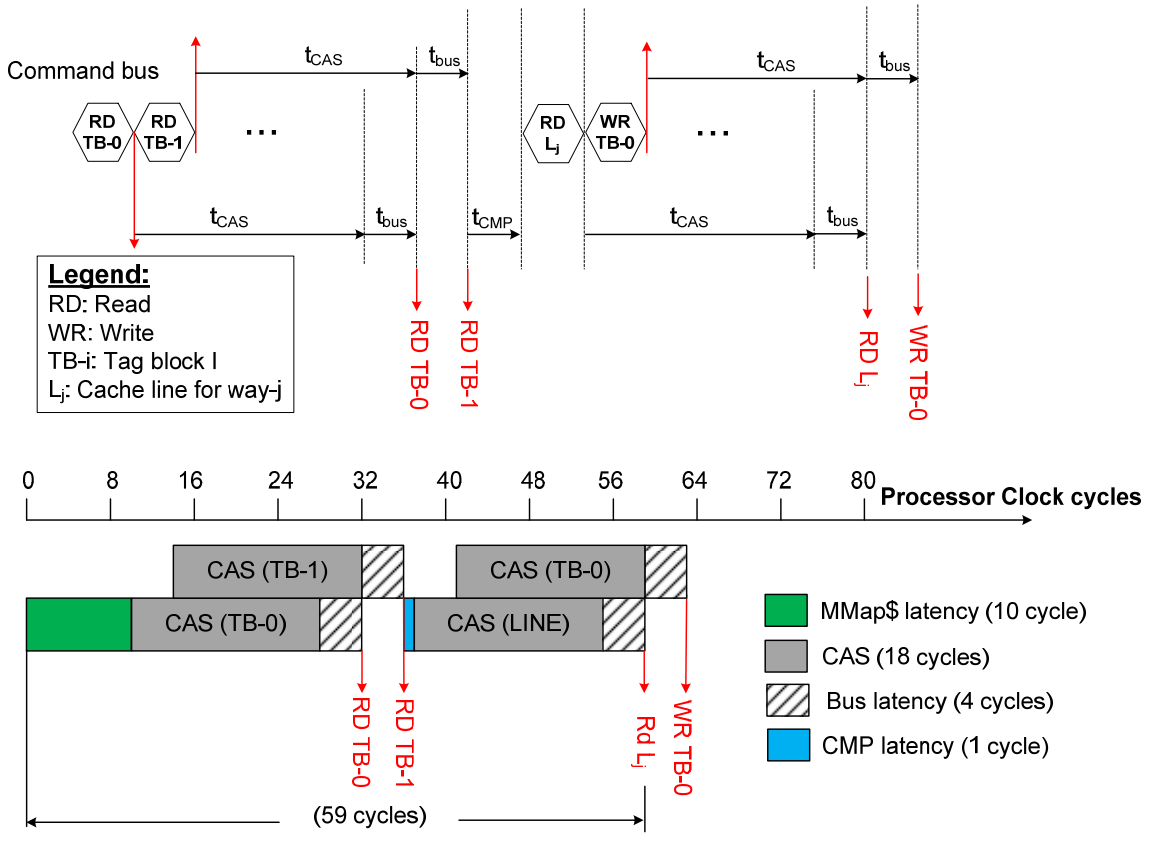


Figure 6.11: Timing and sequence of commands for L4 DRAM read hit that his in the row buffer for configurable row buffer mapping policy

6.3.3 Latency breakdown

This section analyzes the row buffer hit latencies (i.e. the accessed data is in the row buffer) and row buffer miss latencies (i.e. the accessed data is not in the row buffer) for different row buffer mapping policies. Figure 6.12 shows the latency breakdown (in terms of processor clock cycles) for different row buffer mapping policies for a 2KB row size. Note that the latency breakdown does not show the latency of the DRAM cache controller (time spent in the DRAM cache controller before having an access to a DRAM bank). This section assume identical latency values for all DRAM cache parameters which are listed in Table 4.1. All of the row buffer mapping policies require 10 clock cycles to access the MMap$ to identity DRAM cache hit/miss before the request can be sent to DRAM cache (MMap$ hit) or main memory (MMap$ miss).

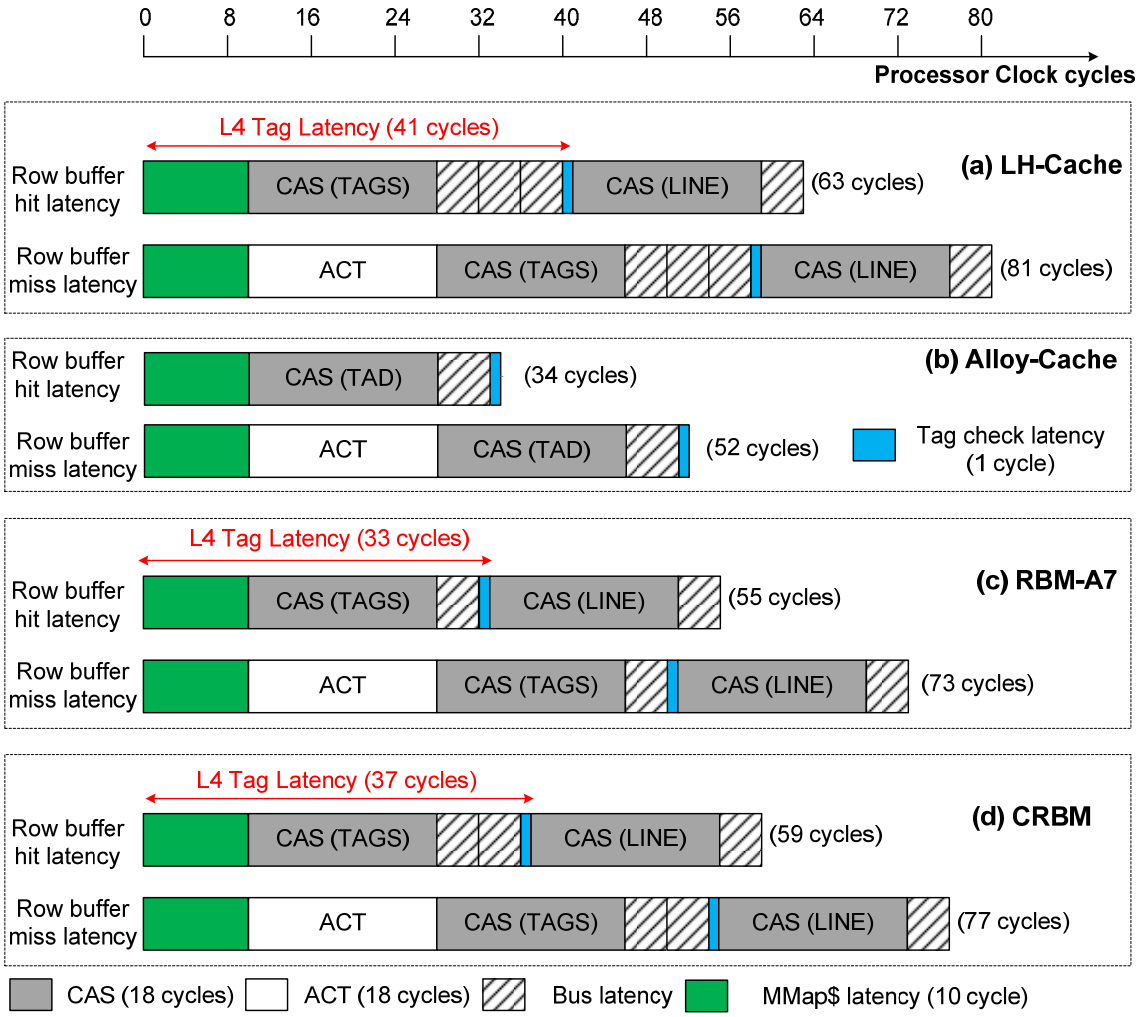


Figure 6.12: L4 DRAM cache Latency breakdown for a 2KB row size (a) LH-Cache [77, 78] (b) Alloy-Cache [102] (c) RBM-A7 [proposed] (d) CRBM [proposed]

If the data is already in the row buffer (i.e. row buffer hit), the LH-Cache [77, 78] requires 18 clock cycles for CAS (to access the tags from the row buffer), 12 cycles to transfer the three tag blocks (see Figure 6.8-a) on the bus (64 x 3 = 192 bytes need to be transferred on 16 byte wide bus that incurs 12 clock cycles for the bus latency 192/16 = 12), 1 clock cycles for the tag check, 18 clock cycles for the CAS (to access the data from the row buffer), and 4 clock cycles to trans-

fer the cache line. If the data is not in the row buffer (i.e. row buffer miss), it would require an additional latency of 18 ACT clock cycles. For the LH-Cache, the row buffer hit latency is 63 clock cycles and the row buffer miss latency is 81 clock cycles as illustrated in Figure 6.12-(a).

For the Alloy-Cache [102], the row buffer hit latency is 34 clock cycles (10 cycle for MMap$ access, 18 clock cycles for CAS, 1 cycle tag-check, and 5 bus cycles for the *TAD* (Tag And Data) entry and the row buffer miss latency is 52 clock cycles (18 additional cycles required for row activation) as illustrated in Figure 6.12-(b).

For the proposed RBM-A7 policy, the access latency of a row buffer hit includes the time to access MMap$ (10 cycles), time to access the tags (18 clock cycles for CAS tags), time to read the tags through DRAM bus (4 cycles for the tags), time to check the tag (1 clock cycle), time to access the cache line (18 clock cycles for CAS LINE) and time to read the cache line through the DRAM bus (4 cycles for the cache line). Thus, the row buffer hit latency is 55 clock cycles and the row buffer miss latency is 73 clock cycles (18 additional cycles required for row activation) for RBM-A7 policy as illustrated in Figure 6.12-(c).

For the proposed CRBM policy, the row buffer hit latency is 59 clock cycles and the row buffer miss latency is 77 clock cycles as shown in Figure 6.12-(d) because it requires 4 additional clock cycles to transfer the extra tag block compared to the RBM-A7 policy.

6.3.4 Comparisons of different row buffer mapping policies

Table 6.1 shows the impact of different row buffer mapping policies on the L4 DRAM miss rate (depends upon associativity; higher the associativity, lower the miss rate) and L4 DRAM hit latency (depends upon row buffer hit rate and L4 tag latency).

Row Buffer Mapping Policy	Associativity	Row Buffer Hit rate	L4 Tag Latency
LH-Cache [77, 78]	29 (Great)	Worst	41 cycles
Alloy-Cache	1 (Worst)	Great	11 cycles
RBM-A7	7 (Good)	Good	33 cycles
Configurable row buffer mapping policy	30 (Great)	depends upon CM	37 cycles

Table 6.1: Impact of row buffer mapping policy on associativity and latency

L4 DRAM cache hit latency highly depends on whether an access leads to a row buffer hit or a row buffer miss (details in Section 6.3.3; see Figure 6.12). Row buffer hits have a reduced access latency compared to row buffer misses for all row buffer mapping policies as illustrated in Figure 6.12.The LH-Cache [77, 78] does not exploit the full potential of row buffer locality and its disadvantageous row buffer hit rate leads to a high L4 DRAM cache hit latency. The Alloy-Cache [102] on the other hand employs a direct mapped cache that reduces the L4 DRAM hit latency via a reduced tag access latency (see Figure 6.12-b) and an increased row buffer hit rate (evaluated in Section 6.7.2). However, that reduction comes at the cost of an increased L4 DRAM miss rate due to increased conflict misses. The proposed RBM-A7 and configurable row buffer mapping policies benefit from a high associativity with a significantly reduced L4 DRAM miss rate compared to the Alloy-Cache (evaluated in Section 6.7.1). At the same time, they pro-

vide reduced L4 DRAM hit latency via improved row buffer hit rate and reduced tag access latency compared to the LH-Cache (evaluated in Section 6.7.2).

6.3.5 Impact of parameter CM

For the configurable row buffer mapping policy, the L4 DRAM hit latency can be reduced by improving the row buffer hit rate that depends upon the parameter CM (see Figure 6.5-b). Increasing CM improves the row buffer hit rate (which reduces the L4 hit latency), but comes at the cost of increased L4 DRAM miss rate due to reduced set-level-parallelism (because high order address bits are used to select the L4 DRAM set) and increased conflict misses (spatially close blocks increases the conflict misses within a set). The result section explores the trade-offs between the L4 DRAM hit latency (depends upon row buffer hit rate) and main memory latency (depends upon L4 DRAM miss rate) for different values of CM (1, 2, 4, 8, 16) and its impact on the overall performance.

6.4 Super-block MMap$ (SB-MMap$)

The MissMap Cache (MMap$) was proposed by Loh and Hill [77, 78] to precisely determine whether an access to a DRAM cache will be a hit or a miss. It is organized as a set of memory segments (typical memory segment size is 4KB) and there is a tag associated with each segment (called *Seg-Tag* as shown in Figure 6.13). Each memory segment itself is divided into blocks (typical block size is 64 bytes). Each MMap$ segment entry maintains a bit vector (called *Seg-BV*) representing whether or not the corresponding block of the relevant segment is present in DRAM cache.

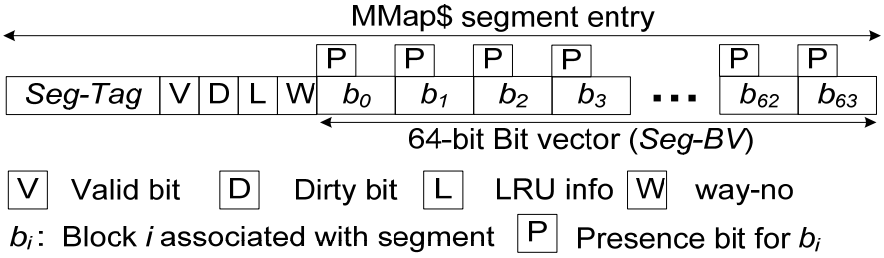


Figure 6.13: MMap$ entry covering a 4KB memory segment for LH-Cache [77, 78]

Let's assume there is a request for block b_i that belongs to a segment *S*. When there is a miss to a segment (*Seg-Tag* field does not matches with tag of *S*), a resident segment is evicted and *Seg-Tag* is set corresponding to the new segment *S*. When there is a hit to a segment (*Seg-Tag* field matches), then the MMap$ checks the *Presence bit* (P-bit) associated with the block b_i (see Figure 6.13) of the hit segment *S*. If the P-bit of the hit segment *S* is set (block b_i is present in the DRAM cache), the MMap$ forwards the request to the DRAM cache. Otherwise, it is forwarded to the main memory, bypassing the DRAM cache. When the block b_i is filled in the DRAM cache (the tags/data of block b_i is stored in the DRAM cache), then the P-bit entry of block b_i corresponding to segment *S* is set. When a block b_i is evicted from the DRAM cache, then the P-bit entry of block b_i corresponding to segment *S* is reset. When a segment is evicted from the MMap$, its tag and data must also be evicted from DRAM cache. This guarantees that the

MMap$ always precisely determines whether an access to a DRAM cache will be a hit or a miss (i.e. it provides 100% precise information about L4 DRAM hit/miss).

The proposed row buffer mapping policies allow reducing the size of the MMap$ with negligible performance degradation. The size of the MMap$ *Seg-BV* entry is reduced by using coarse-grained presence information. Therefore, a super-block is defined as a set of adjacent blocks. Figure 6.14-(a) and Figure 6.14-(b) show a super-block comprising two and four adjacent blocks, respectively. The proposed super-block MMap$ (SB-MMap$) organization assigns a presence bit to each coarse-grained super-block instead of having a separate bit for each fine-grained block. This reduces the MMap$ *Seg-BV* entry size by half (for super-block size of 2) and by quarter (for super-block size of 4) which leads to a reduced MMap$ storage overhead. However, storing presence information at coarse-grained super-block level effects the DRAM cache hit/miss prediction accuracy compared to a MMap$ that stores the presence information at the fine-grained block level.

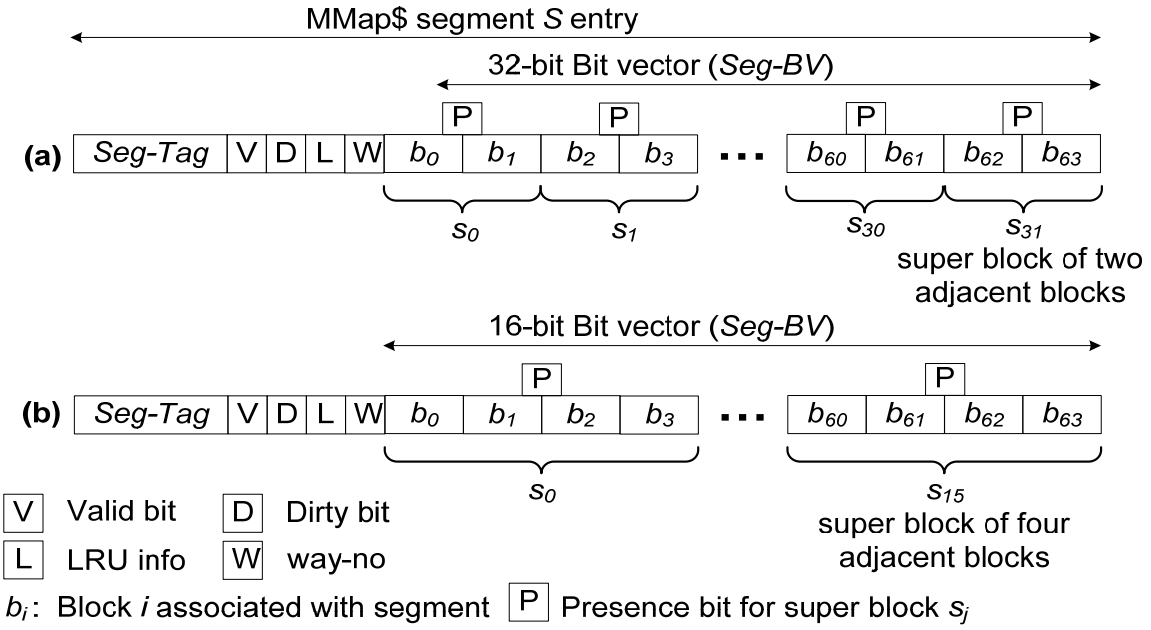


Figure 6.14: Proposed SB-MMap$ entry representing a 4KB memory segment for a super-block containing (a) two adjacent blocks (b) four adjacent blocks

In the proposed SB-MMap$ organization, when a block b_i that belongs to a segment *S* is filled in the DRAM cache, then the P-bit entry of the super-block to which b_i belongs is set. When block b_i is evicted from DRAM cache, then resetting the P-bit entry requires a tag-lookup for the adjacent blocks to determine whether adjacent blocks are present in the DRAM cache or not. Since in the configurable row mapping policy (Section 6.3) consecutive blocks are mapped to the same DRAM cache row, resetting the P-bit entry after block eviction requires a single DRAM cache row lookup. However, employing SB-MMap$ size for the LH-Cache [77, 78] (Section 2.4.1) would require additional row lookups for resetting the P-bit after block eviction, because they map consecutive blocks to different cache rows. The number of row lookups in LH-Cache will depend upon the size of the super-block.

The proposed SB-MMap$ detects either a “true miss” or a “maybe hit” after an L3 SRAM miss. A “true miss” (i.e. super-block P-bit is zero and all blocks belonging to the super-block are absent in the DRAM cache) indicates that the requested block is not present in the DRAM cache. A “true miss” does not require a DRAM cache lookup, so the request is directly sent to the main memory controller (see Figure 6.1). A “maybe hit” (i.e. super-block P-bit is set) indicates that the block may or may not be present in the DRAM cache. A “maybe hit” requires a DRAM cache lookup to identify a hit/miss to determine whether the request should be sent to the main memory controller or not. The L4 DRAM hit/miss prediction accuracy of the SB-MMap$ depends upon the super-block size. If the super-block size is 1 (traditional MMap$ proposed by [77, 78]), then it provides 100% precise information about L4 DRAM hit/miss. Increasing the super-block size reduce SB-MMap$ storage overhead at the cost of a reduced SB-MMap$ hit/miss prediction accuracy. The result section will explore the impact of different super-block sizes on the SB-MMap$ hit/miss prediction accuracy and the overall performance for the configurable row buffer mapping policy. The super-block size is chosen to be smaller than CM (consecutive memory blocks mapped to the same row) in the configurable row buffer mapping policy because it then only requires a single row lookup to access all adjacent blocks of a super-block. In addition, a small on-chip SRAM structure namely DRAM Tag-Cache (Section 6.5.1) is added to improve the L4 DRAM hit/miss prediction accuracy with SB-MMap$.

6.4.1 Impact of super-block size on storage reduction

This thesis employs traditional least recently used (LRU) replacement policy for the MMap$. Each MMap$ entry requires 1 valid bit, 1 dirty bit, 4 LRU bits, 4 way-number bits, 20 bits for *Seg-Tag* field, and 64-bit for *Seg-BV* field. This leads to a storage overhead of 94 bits required for each MMap$ entry. A super-block size of 2 will reduced the SB-MMap$ *Seg-BV* size by half (it requires 32 bits for *Seg-BV* field), which leads to storage requirement of 62 bits for each SB-MMap$ entry. Similarly, a super-block size of 4 requires 16 bits for the *Seg-BV* field, which leads to storage requirement of 46 bits for each SB-MMap$ entry. A super-block of size 2 or 4 reduces the SB-MMap$ storage overhead by 34% or 51%, respectively compared to the original MMap$.

6.5 Innovative Tag-Cache Organization for larger caches

Multi-core systems with an on-chip SRAM/DRAM cache hierarchy typically employ larger L3 SRAM and L4 DRAM caches to accommodate the large working set sizes of emerging applications [32]. The larger L3 SRAM incurs high access latencies due to long interconnect delays [87]. On the other hand, the larger L4 DRAM caches incur high access latencies due to slower DRAM cache access [54, 77, 78, 102]. This section proposes several architectural innovations to minimize L3 SRAM and L4 DRAM hit latencies to improve the overall instruction throughput.

6.5.1 DRAM Tag-Cache (*DTC*) Organization

The tag latency for an L4 DRAM hit in the proposed configurable row buffer mapping policies is 37 cycles (see Figure 6.12). To reduce it, this thesis adds a small low latency on-chip SRAM structure named as DRAM Tag-Cache (*DTC*) that holds the tags of recently accessed rows in the DRAM cache. The integration of the *DTC* into the cache hierarchy is shown in Figure 6.1 and

Figure 6.15 presents the *DTC* details for the configurable row buffer mapping policy. Note that the *DTC* only stores the tag blocks of recently accessed rows and does not contain any data. The *DTC* has a fast access latency due to its small size. It is accessed right after an L3 SRAM miss and, in case of a *DTC* hit, it reduces the L4 hit latency because it avoids the high latency MMap$ access to identify a L4 hit/miss and it avoids reading the tag block from the DRAM cache. The proposed *DTC* also reduces the L4 miss latency because the request is sent immediately (requires 2 clock cycles; one cycle to identify a *DTC* hit and one cycle to identify an L4 hit/miss) to the main memory controller after an L4 DRAM miss is detected by the *DTC*. In contrast, state-of-the-art [77, 78] requires 10-cycles for the MMap$ access to identify a miss before the request can be sent to the memory controller.

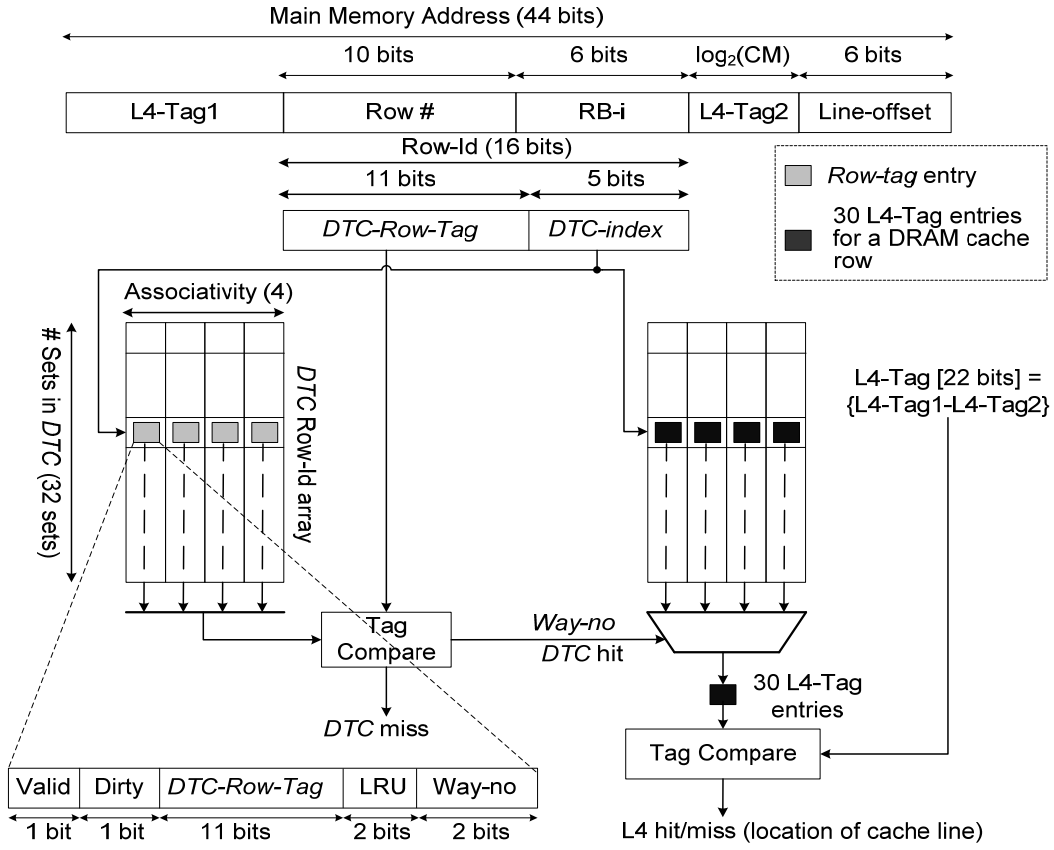


Figure 6.15: DRAM Tag Cache (*DTC*) Organization for configurable row buffer mapping policy

Figure 6.15 shows the *DTC* organization with 32 sets and 4-way associativity where the data payload of each entry contains the tags for a particular Row-id. On a *DTC* access, the *DTC-Index* field is used to index a *DTC* set in the "*DTC* Row-Id" array. All 4 *Row-Tag* entries (grey blocks in Figure 6.15) within that *DTC* set are then compared to the *DTC-Row-Tag* field from the memory block address to identify a *DTC* hit/miss.

The proposed *DTC* has the following major advantages.

1. Accesses that hit in the *DTC* incur a reduced L4 tag access latency (Figure 6.16-a) compared to a *DTC* miss (Figure 6.16-b) because they do not require DRAM cache access to read the

tags and they do not require MMap$ access to identify a L4 DRAM hit/miss. Similarly, the row buffer miss latency is also reduced for a *DTC* hit (Figure 6.16-c) compared to a *DTC* miss (Figure 6.16-d). A *DTC* hit reduces the DRAM cache bandwidth consumption compared to a *DTC* miss because it reads 64 bytes (required for a cache line access) instead of reading 192 bytes (128 bytes for the tags and 64 bytes for the cache line) in case of a *DTC* miss for the configurable row buffer mapping policy.

2. Accesses that hit in the *DTC* precisely identify whether they lead to an L4 DRAM hit or a miss for a "maybe hit" signal detected by the proposed reduced SB-MMap$, which improves the DRAM cache hit/miss prediction accuracy.
3. If an L4 DRAM miss is identified after a *DTC* hit, the request can be sent immediately to the main memory controller (without requiring MMap$ access), which reduces the L4 DRAM miss latency.

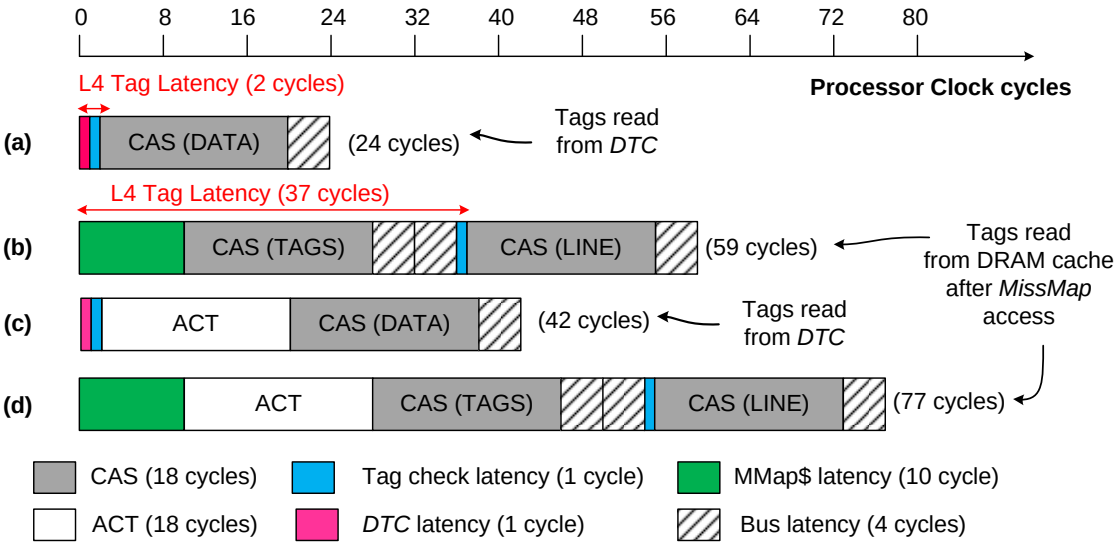


Figure 6.16: L4 DRAM row buffer hit latency for (a) *DTC* hit (b) *DTC* miss L4 DRAM row buffer miss latency for (c) *DTC* hit (d) *DTC* miss

6.5.2 *DTC* Implementation with SB-MMap$

Figure 6.17 shows the steps involved after an L3 SRAM miss in the proposed SRAM/DRAM cache organization. The *DTC* and SB-MMap$ is accessed after an L3 SRAM miss. If the *DTC* hits, then the tags from *DTC* are accessed to identify an L4 DRAM hit/miss and to identify the location of the cache line (see Tag-Compare at the right bottom part of Figure 6.15). An L4 DRAM hit (i.e. the tag matches with an incoming cache line tag) requires only data access from the DRAM cache. However, if the *DTC* misses, then the SB-MMap$ needs to be queried to identify a "true miss" or a "maybe hit" (details in Section 6.4). A "true miss" requires a main memory access to get the data. If the SB-MMap$ identifies a "true miss", the request is forwarded to the main memory, bypassing the DRAM cache access. If the SB-MMap$ identifies a "maybe hit", then the request is forwarded to the DRAM cache to eventually identify a hit or a miss. When the tags are read from the DRAM cache after a *DTC* miss, then they are inserted into the *DTC* in order to exploit the temporal locality that these tags will be accessed in the near future. The results in Section 6.7.3 show that the proposed configurable row buffer mapping policy has an improved

DTC hit rate compared to the LH-Cache (i.e. when *DTC* is incorporated with LH-Cache) [77, 78].

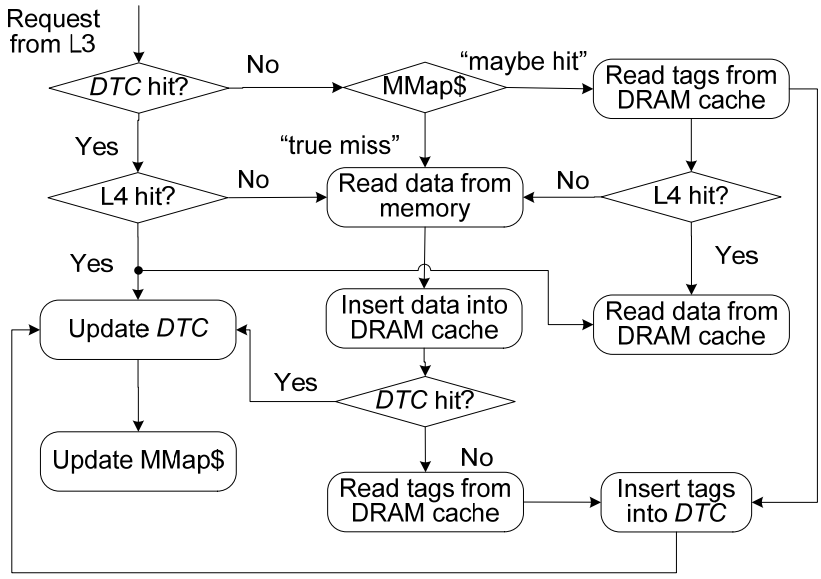


Figure 6.17: *DTC* and MMap$ lookup following an L3 SRAM miss

6.5.3 Writing tag-blocks for a *DTC* hit

When the tags are updated in the *DTC* (cache hit, cache writeback, cache fill etc.), then the cache line in the DRAM cache must be accessed (read/write for a cache hit, write for a cache writeback, and insert new cache line for a cache fill). The following modification has been made to the DRAM cache controller to efficiently write the dirty tags by exploiting the row buffer locality. The modified DRAM cache controller writes the updated *DTC* tags into the DRAM cache along with the cache line so that writing updated tag information is guaranteed to have a row buffer hit. Figure 6.18 shows the timing and sequence of commands involved in the proposed DRAM cache controller for different requests that hit in the *DTC* and that are explained as follows.

L4 DRAM read hit: For an L4 DRAM read hit request that hits in the row buffer (see Figure 6.18-a), the controller issues a read request to read the requested block whose location is determined by the tags of the *DTC* followed by a subsequent request to update the dirty tags. The requested data is read from the DRAM cache and forwarded to the requesting core followed by updating tag block TB-0 (e.g. used bit is set to 1 which is stored in TB-0; see Section 6.3.2-b and Figure 6.10-b). The latency incurred in updating TB-0 is only 4 cycles as illustrated in Figure 6.18-a.

L4 DRAM write/writeback hit: For an L4 DRAM write/writeback hit request that hits in the row buffer (see Figure 6.18-b), the controller issues a write request to update tag block TB-0 (to set the dirty bit for cache write and cache writeback and to set the used bit for cache write; these bits are stored in TB-0; see Section 6.3.2-b and Figure 6.10-b) followed by a subsequent write request to the cache line (whose location is determined by the *DTC* tags).

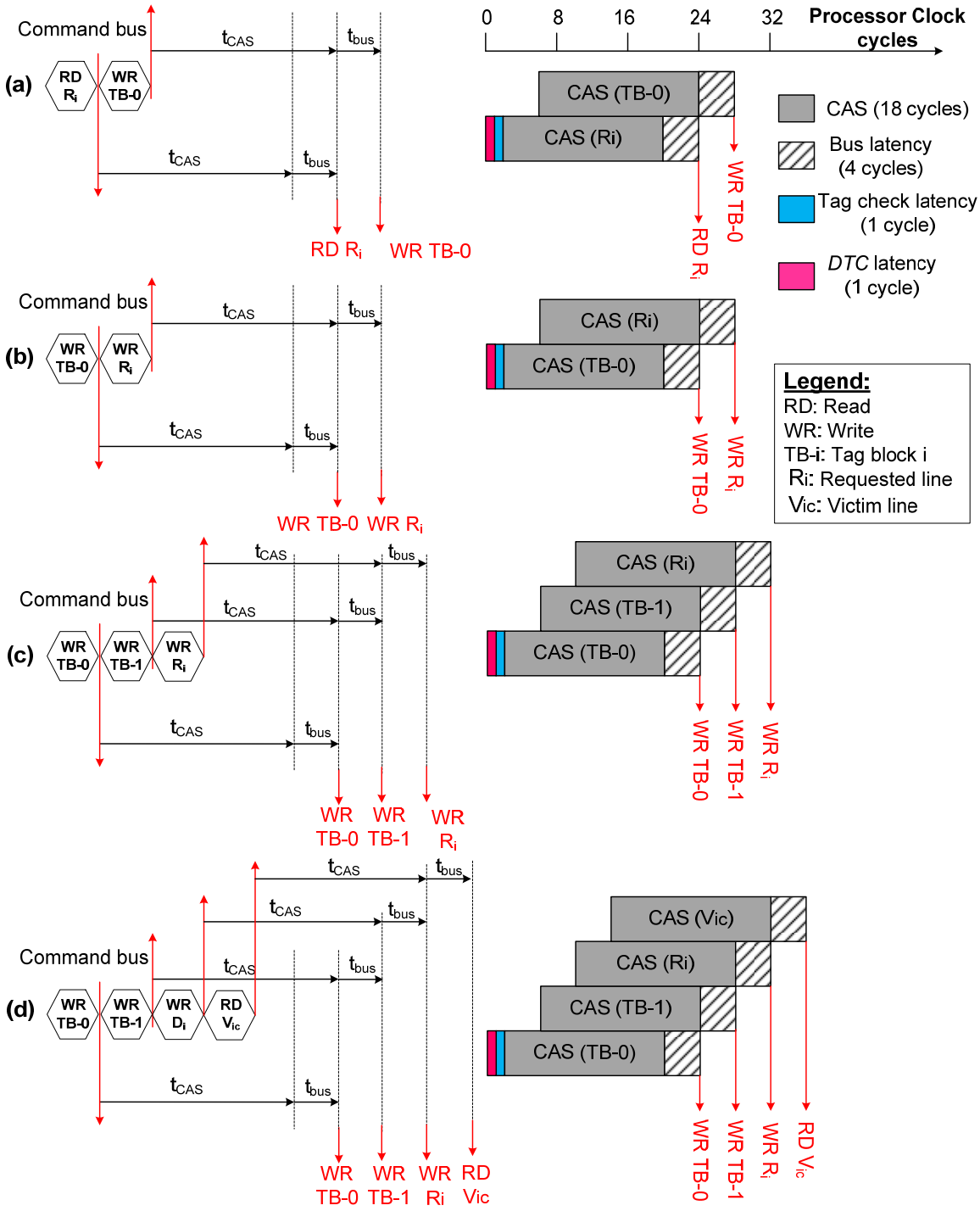


Figure 6.18: Timing and sequence of commands to update the tags in the DRAM cache after a *DTC* hit in the configurable row buffer mapping policy for an L4 DRAM row buffer hit (a) read request (b) write/write-back request (c) fill request with clean victim block eviction (d) fill request with dirty victim block eviction

L4 DRAM cache fill with clean victim line eviction: For an L4 DRAM fill request that hits in the row buffer (see Figure 6.18-c), the controller issues two write requests to update the tag

blocks TB-0 and TB-1 (to store the tags of an incoming cache line, to clear the used bit and to update the clock pointer). There is no need to write a clean victim line (i.e. dirty bit is 0) to main memory because the main memory contains the most recent copy of the data.

L4 DRAM cache fill with dirty victim line eviction: For an L4 DRAM fill request that hits in the row buffer (see Figure 6.18-d), the controller issues two write requests to update the tag blocks TB-0 and TB-1. Also, an additional read request is issued to read the dirty victim line (i.e. dirty bit is 1) that is written to main memory because the victim line has been modified since it was fetched from main memory.

6.5.4 *DTC* organization for RBM-A7 policy

Figure 6.15 shows the *DTC* organization for the configurable row buffer mapping policy (details in Section 6.3.2), while Figure 6.19 shows the *DTC* organization for the RBM-A7 policy (details in Section 6.3.1).

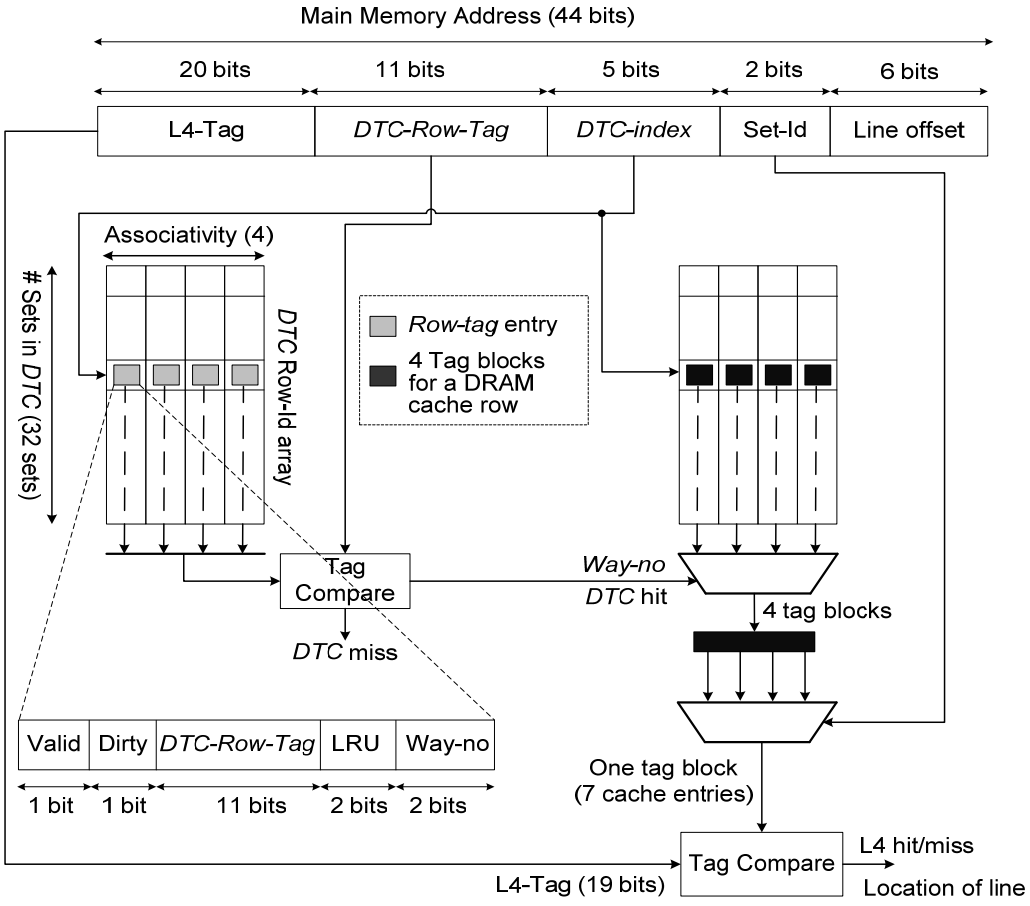


Figure 6.19: DRAM Tag Cache (*DTC*) Organization for the RBM-A7 policy

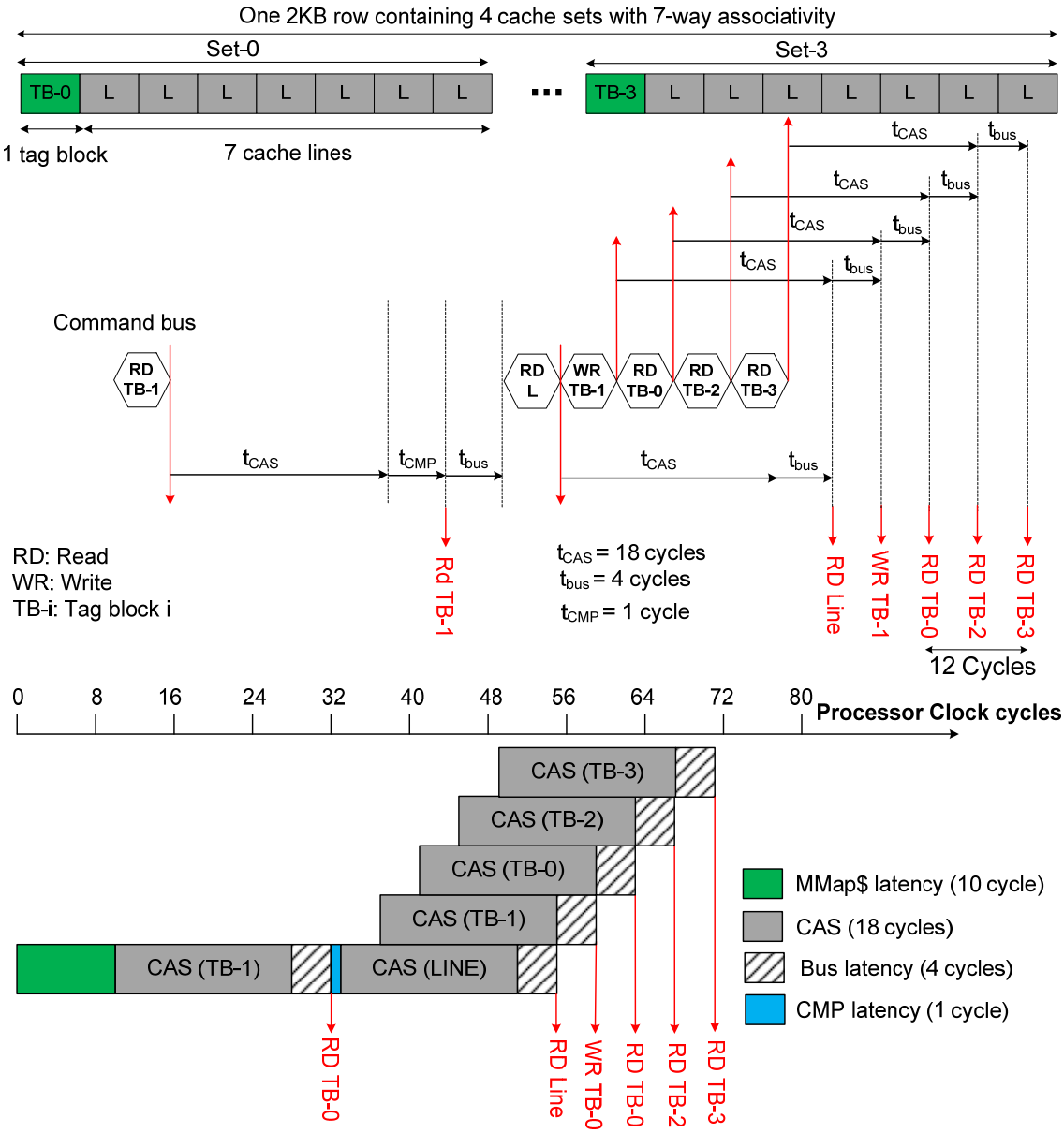


Figure 6.20: Timing and sequence of commands to fill the *DTC* after a *DTC* miss for a data block that belongs to Set-1 in the RBM-A7 policy

Each 2KB DRAM row in the RBM-A7 policy consists of 4 cache sets, where each cache set consists of one tag blocks and 7 cache lines as shown in Figure 6.3. The inclusion of *DTC* requires an efficient DRAM cache controller implementation in order to insert 4 tag blocks into *DTC* after a *DTC* miss for the RBM-A7 policy. In the proposed controller implementation, a read operation is performed for the requested cache line first, and then the non-requested tag blocks are also read from the DRAM cache to be filled in the *DTC*. Figure 6.20 shows an example of the sequence of commands to fill the *DTC* after a *DTC* miss with low latency overhead. Let us assume that there is a DRAM cache read hit for a cache line that belongs to Set-1. In the proposed implementation, the controller issues a read request to read the tag block TB-1 (tag block associated with Set-1) which indicates the location of the cache line in Set-1. The requested cache line is read from the DRAM cache and forwarded to the requesting core followed by updating tag

block TB-1 (e.g. updating LRU information). After that, subsequent read commands are sent to access the remaining tag blocks (i.e. TB-0, TB-2, and TB-3) which are filled into *DTC*. All of these operations are performed on the row buffer. In the proposed controller implementation, the non-requested tag block transfers are performed off the critical path so that they do not affect the latency of the demand request. The extra bus latency incurred to read the remaining 3 tag blocks is 12 cycles as shown in Figure 6.20. The latency overhead to read additional 3 blocks (to fill *DTC*) is compensated by future hits in the *DTC* exploiting the fact that adjacent tag blocks are likely to be accessed in the near future (see Section 6.7.3 for evaluation).

6.5.5 SRAM Tag-Cache (*STC*) Organization

A large SRAM tag array (e.g. an 8 MB L3 SRAM requires a tag storage of ~512 KB) is composed of multiple banks [124], where each bank consists of multiple sub-banks with one sub-bank being activated per access as shown in Figure 6.21. Each sub-bank is composed of multiple identical mats, where all mats in a sub-bank are activated per access. Each mat is an array of SRAM cells with associated peripheral circuitry. Each row in a mat contains the tags of one cache set. State-of-the-art SRAM/DRAM cache organizations [36, 38, 77, 78, 102] always read the tags from that large L3 SRAM tag array (see Figure 6.21) which incurs a high L3 tag latency. To reduce the L3 tag latency, a small SRAM Tag-Cache (*STC*) organization with 16 sets and 4-way associativity is added to the cache hierarchy to identify L3 hit/miss in a single cycle. The *STC* organization (see Figure 6.22) is similar to the *DTC* organization (see Figure 6.16) except that it holds the tags of the 8 adjacent sets (i.e. belonging to the same row of 8 mats) that were recently accessed in the L3 SRAM tag array. Entries in *STC* are indexed by the *STC-index* field of the memory block address as shown in Figure 6.22.

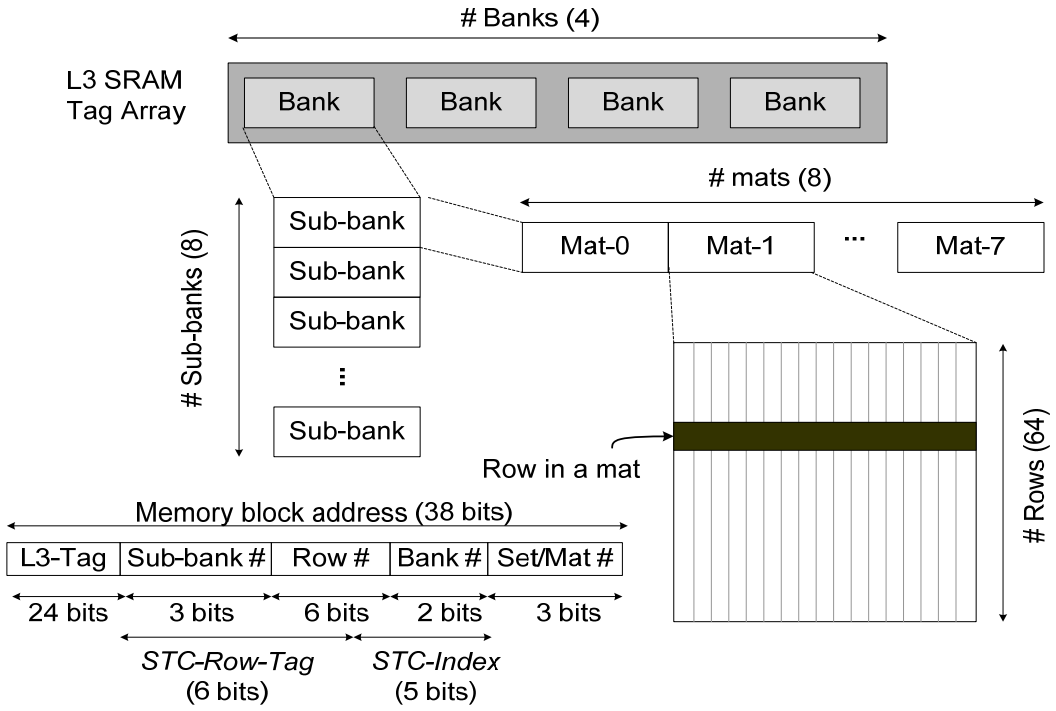


Figure 6.21: Layout of a large L3 SRAM tag array [124]

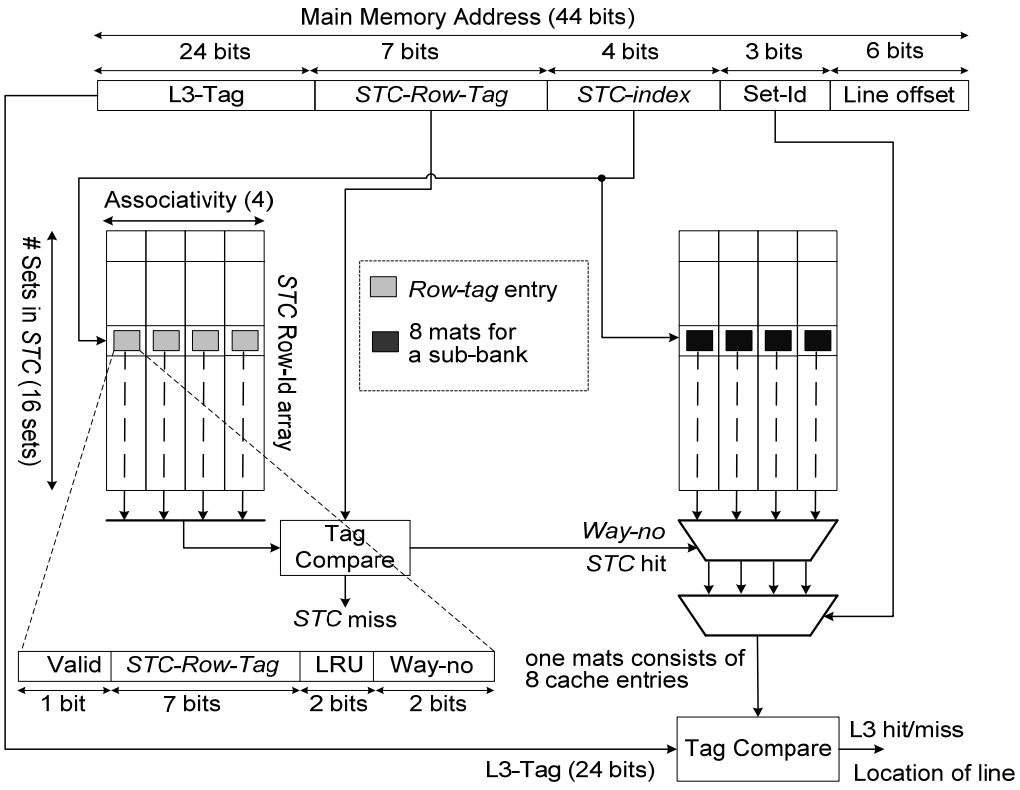


Figure 6.22: SRAM Tag-Cache (*STC*) organization

6.6 Storage Overhead

Table 6.2 summarizes the storage overhead for the DRAM Tag-Cache (*DTC*) and SRAM Tag-Cache (*STC*). The proposed *DTC* with 32 sets and 4-way associativity requires a storage overhead of ~16KB for a 2KB DRAM cache row size, while the proposed *STC* with 16 sets and 4-way associativity requires a storage overhead of ~20KB. The total storage overhead required for both is ~36KB which is negligible compared to the large L3 SRAM cache (8 MB for L3 data array) and MMap$ (2MB required for DRAM cache hit/miss prediction). Note that the proposed SB-MMap$ requires 51% less storage overhead compared to state-of-the-art.

6.7 Evaluation and Analysis

This section evaluates the proposed row buffer mapping policies (RBM-A7 and configurable row buffer mapping policy; details in Section 6.3) with state-of-the-art row mapping policies namely LH-Cache [77, 78] (details in Section 2.4.1) and Alloy-Cache [102] (details in Section 2.4.3). The drawback of these policies is that they are optimized for one parameter (LH-Cache is optimized for L4 DRAM miss rate and Alloy-Cache is optimized for L4 DRAM hit latency). In contrast, the proposed row buffer mapping policies optimize both L4 miss rate and L4 hit latency. This section also evaluates the impact of the parameter CM (number of consecutive memory blocks mapped to the same row; details in Section 6.3.5) on the L4 miss rate and L4 hit latency.

DRAM Tag-Cache (*DTC*) Overhead for configurable row buffer mapping policy	**Overhead**
Size of each L4-Tag entry in DRAM cache (**Figure 6.10**-a) (1 valid bit + 1 dirty bit + 1 used bit + 8 coherence bits + 22 L4-Tag bits)	33 bits
Tag size for each row in L4 DRAM cache (**Figure 6.10**-b) (5 bits "clock-pointer" + 30 way * 33 bits/way = 990 bits)	995 bits
Size of each *DTC* entry [1 valid bit + 1 dirty bit + 11-bits *DTC-Row-Tag* + 2-bit LRU + 2-bit way-no + 995 bits for each row in L4 DRAM cache] (**Figure 6.15**)	1012 bits
Total *DTC* storage overhead [32 sets * 4 entry/set * 1012 bits/entry = 129536 bits = 15.8125 KB]	~16 KB
DRAM Tag-Cache (*DTC*) Overhead for RB-A7 policy	**Overhead**
Size of each L4-Tag entry in DRAM cache (**Figure 6.3**) (1 valid bit + 1 dirty bit + 3-bit LRU + 3-bit way-no + 8 coherence bits + 20 L4-Tag bits)	36 bits
Tag size for each row in L4 DRAM cache (**Figure 6.3**) (4 sets * 7 way/set * 35 bits/way = 980 bits)	1008 bits
Size of each *DTC* entry [1 valid bit + 1 dirty bit + 11-bits *DTC-Row-Tag* + 2-bit LRU + 2-bit way-no + 1008 bits for each row in L4 DRAM cache] (**Figure 6.19**)	1025 bits
Total *DTC* storage overhead [32 sets * 4 entry/set * 1025 bits/entry = 131200 bits = 16.01 KB]	~16 KB
SRAM Tag-Cache (*STC*) Overhead	**Overhead**
Size of each L3-Tag entry in SRAM cache (1 valid bit + 1 dirty bit + 3-bit LRU + 3-bit way-no + 8 coherence bits + 24 L4-Tag bits)	40 bits
Tag size for each sub-bank in L3 SRAM cache (**Figure 6.21**) (8 sets * 8 way/set * 40 bits/way = 2560 bits)	2560 bits
Size of each *STC* entry [1 valid bit + 7-bits *DTC-Row-Tag* + 2-bit LRU + 2-bit way-no + 2560 bits for each row in L3 SRAM cache] (**Figure 6.22**)	2572 bits
Total *STC* storage overhead [16 sets * 4 entry/set * 2572 bits/entry = 164608 bits = 20.1 KB]	~20 KB

Table 6.2: Storage overhead of DRAM Tag-Cache (*DTC*) and SRAM Tag-Cache (*STC*)

The parameters for the cores, caches and off-chip memory are the same as used in the experimental setup in Section 4.2 (see Table 4.1 and Table 4.2) with various workloads from SPEC2006 [5] listed in Table 4.3. This chapter uses 2KB DRAM row size for comparisons. However, the concepts proposed in this chapter can also be applied for other row sizes (e.g. 4KB and 8KB). This chapter employs the adaptive DRAM insertion policy (*ADIP*; details in Section 5.2) on top of all row buffer mapping policies. However, the concept proposed in this chapter are flexible enough to be applied to any replacement policy. The following row buffer mapping policies are compared:

1. State-of-the-art LH-Cache [77, 78] (details in Section 2.4.1) namely *LH-ADIP* which is optimized for L4 DRAM miss rate.

2. State-of-the-art Alloy-Cache [102] namely *Alloy-ADIP* (details in Section 2.4.3) which is optimized for L4 DRAM hit latency.
3. The proposed RBM-A7 policy (details in Section 6.3.1) namely *RBM-A7-ADIP*.
4. The proposed configurable row buffer mapping policy (details in Section 6.3.2) namely *CRBM-ADIP-CM*, where CM (details in Section 6.3.2 and Section 6.3.5) corresponds to the number of consecutive blocks from main memory that are mapped to the same L4 DRAM row.

The latency and performance impact of DRAM Tag-Cache (*DTC*; details in Section 6.5.1) for above row buffer mapping policies is presented in Section 6.7.4 and 6.7.6 respectively. Section 6.8.2 explores the performance benefits of incorporating SRAM Tag-Cache (*STC*; details in Section 6.5.5) in the cache hierarchy.

6.7.1 Impact on L4 DRAM miss rate

This sections compares the L4 DRAM miss rate (lower is better) for different row buffer mapping policies. First, the miss rate optimized *LH-ADIP* [77, 78] and the latency optimized *Alloy-ADIP* (with worst L4 miss rate) [102] policies are compared. Then, the impact of parameter CM (2, 4, 8, 16) on the L4 DRAM miss rate it evaluated for the configurable row buffer mapping policy.

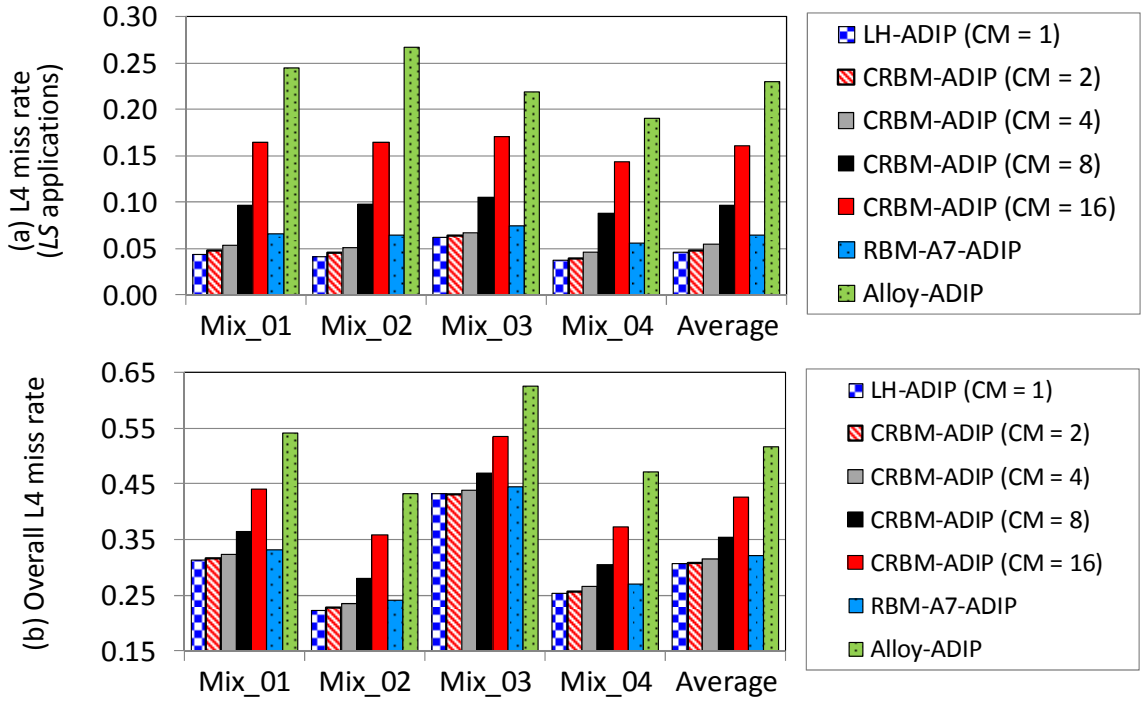


Figure 6.23: L4 DRAM miss rate (a) for Latency Sensitive applications (b) Overall miss rate; for different row buffer mapping policies

Figure 6.23-(a) shows the L4 miss rate for latency sensitive applications, while Figure 6.23-(b) shows the overall L4 miss rate for different row buffer mapping polices. The *Alloy-ADIP* policy (employs direct mapped cache) suffers more misses compared to the *LH-ADIP* policy (em-

ploys 29-way associative cache), which leads to an increased L4 miss rate. The *LH-ADIP* policy reduces the overall L4 miss rate by 40.8% compared to the *Alloy-ADIP* policy via high associativity and reduced conflict misses. On the other hand, the *Alloy-ADIP* policy improves the L4 DRAM row buffer hit rate (see Section 6.7.2) and reduces the L4 tag latency (details in Section 6.3.3), which leads to a significant reduction in L4 hit latency (50.1%) compared to the *LH-ADIP* policy (details in Section 6.7.4).

The proposed configurable row buffer mapping policy slightly increases the overall L4 miss rate (0.7% for CM = 2, 3% for CM = 4 and 15.8% for CM = 8) compared to the miss rate optimized *LH-ADIP* policy, but that is compensated by a significant reduction in L4 hit latency. A higher value of CM reduces the L4 hit latency (lower is better) via an improved row buffer hit rate (details in Section 6.7.2) but it suppresses the set-level-parallelism, which results in an increased L4 miss rate. For instance, CM = 16 incurs a significantly increased L4 miss rate (39.1% compared to the *LH-ADIP* policy), because it maps a large contiguous memory space (with 16 blocks) to a single set, which leads to a lot of conflict misses due to reduced set-level-parallelism. The proposed *RBM-A7-ADIP* policy increases the overall L4 miss rate by 5.1% compared to the *LH-ADIP* policy, because it employs an 7-way associative DRAM cache compared to the 29-way associative *LH-ADIP* policy. On the other hand, the *RBM-A7-ADIP* policy improves the row buffer hit rate (see Section 6.7.2) for the DRAM cache, which reduces the L4 hit latency by 14.9% compared to the *LH-ADIP* policy (details in Section 6.7.4).

6.7.2 Impact on the L4 DRAM row buffer hit rate

This section compares the L4 DRAM row buffer hit rate (higher is better) for all evaluated configurations. It also evaluates the impact of parameter CM on the L4 DRAM cache hit rate for the configurable row buffer mapping policy. The *Alloy-ADIP* policy maps 28 consecutive blocks to the same DRAM cache row, which leads to a significantly higher row buffer hit rate (39.4%) compared to other row buffer mapping policies as shown in Figure 6.24. However, this improvement comes at the cost of an increased L4 miss rate (Figure 6.23).

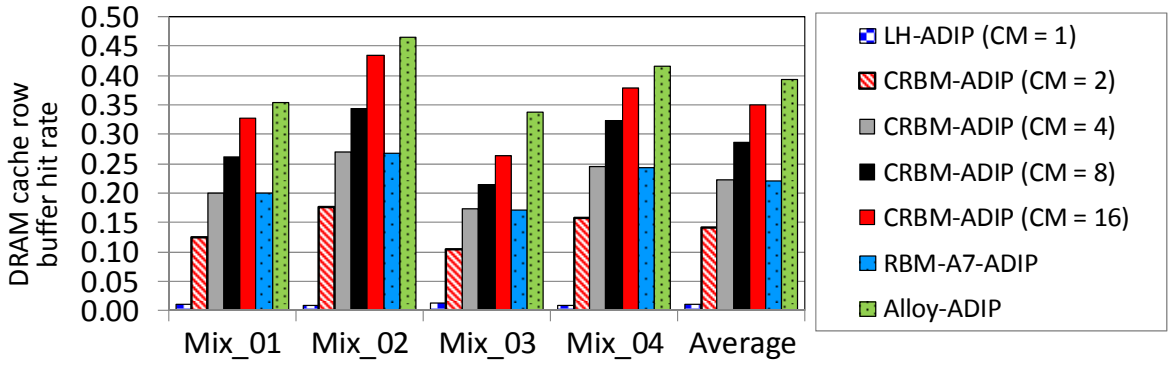


Figure 6.24: DRAM cache row buffer hit rates for different row buffer mapping policies

To exploit spatial and temporal locality, the row buffer hit rate can be improved by mapping more consecutive memory blocks to the same DRAM row buffer. Figure 6.24 illustrates this observation showing the DRAM cache row buffer hit rate for different row buffer mapping policies. The proposed configurable row buffer mapping policy benefits from a high associativity (30-way associativity) with correspondingly reduced L4 miss rate (Figure 6.23) compared to the *Alloy-*

ADIP policy. It improves the DRAM cache row buffer hit rate (14.1% for CM = 2, 22.2% for CM = 4, 28.5% for CM = 8, and 35.1% for CM = 16) compared to the *LH-ADIP* policy (1.2%). The impact of the DRAM cache row buffer hit rate on the L4 DRAM cache hit latency is evaluated in Section 6.7.4.

6.7.3 Impact on the *DTC* hit rate

This sections compares the *DTC* hit rate (higher is better) for all evaluated configurations when *DTC* is incorporated in the DRAM cache hierarchy. It also evaluates the impact of parameter CM on the *DTC* hit rate for the configurable row buffer mapping policy. As CM increases, the DRAM cache row buffer hit rate increases (Figure 6.24), which leads to a reduced L4 hit latency. However, the performance of a DRAM cache with *DTC* depends upon the *DTC* hit rate. Figure 6.25 shows the *DTC* hit rate for different row buffer mapping policies. The *LH-ADIP* policy (CM = 1) has a reduced *DTC* hit rate (1.6%) compared to the configurable row buffer mapping policy (*DTC* hit rate is 33.4% for CM = 2, 57.7% for CM = 4, 73.2% for CM = 8 and 78.8% for CM = 16). Increasing CM improves the *DTC* hit rate, which leads to a reduced L4 DRAM hit latency for the configurable row buffer mapping policy, which is evaluated in the next section.

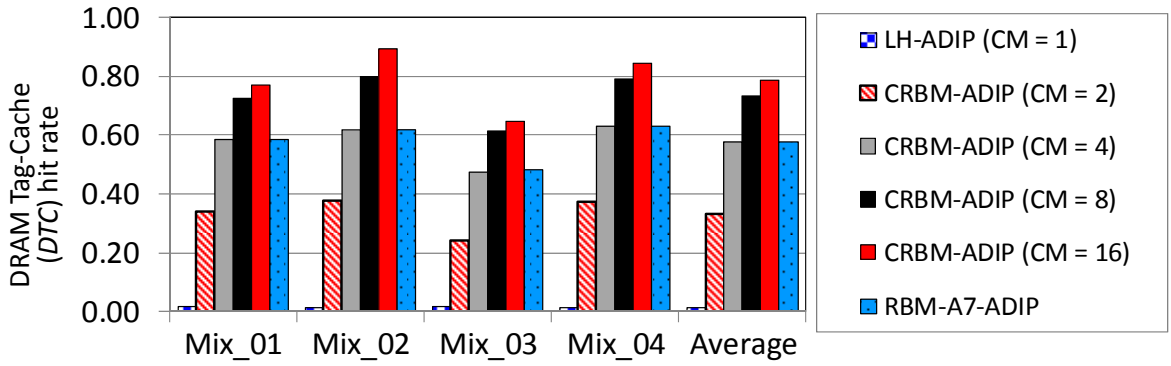


Figure 6.25: DRAM Tag-Cache hit rates for different row buffer mapping policies

6.7.4 Impact on the L4 DRAM hit latency

This sections compares the L4 DRAM cache hit latency (lower is better) for different row buffer mapping policies. It also evaluates the impact of parameter CM and *DTC* on the L4 hit latency. First, this section compares the L4 hit latency for the latency optimized *Alloy-ADIP* [102] and the miss rate optimized *LH-ADIP* (with worst L4 hit latency) [77, 78] policies. Then, it evaluates the impact of parameter CM (2, 4, 8, 16) on the L4 hit latency.

Figure 6.26-(a) shows the L4 hit latency without incorporating *DTC* into the cache hierarchy, while Figure 6.26-(b) shows the L4 hit latency when *DTC* is incorporated in the cache hierarchy. The *Alloy-ADIP* policy optimizes the L4 hit latency due to reduced L4 tag latency and a high DRAM cache row buffer hit rate compared to the *LH-ADIP* policy. Thus, the *Alloy-ADIP* policy leads to a significant reduction in L4 hit latency (50.1%) compared to the *LH-ADIP* policy as shown in Figure 6.26-(a).

As CM increases for the proposed configurable row buffer mapping policy, the DRAM cache row buffer hit rate increases (Figure 6.24), which leads to a reduced L4 hit latency (Figure 6.26-a). The reduction in L4 hit latency compared to the *LH-ADIP* policy without *DTC* is 6.8% for CM = 2, 12.4% for CM = 4, 17.3% for CM = 8, and 22.5% for CM = 16 as illustrated in Figure 6.26-(a). On the other hand, the reduction in L4 hit latency with *DTC* compared to the *LH-ADIP* policy (with *DTC*) is 15.1% for CM = 2, 24.3% for CM = 4, 30.2% for CM = 8, and 34.8% for CM = 16 due to a high *DTC* hit rate. Note that the *LH-ADIP* policy gets negligible latency benefits from *DTC* because the *DTC* hit rate is only 1.7% for the *LH-ADIP* policy.

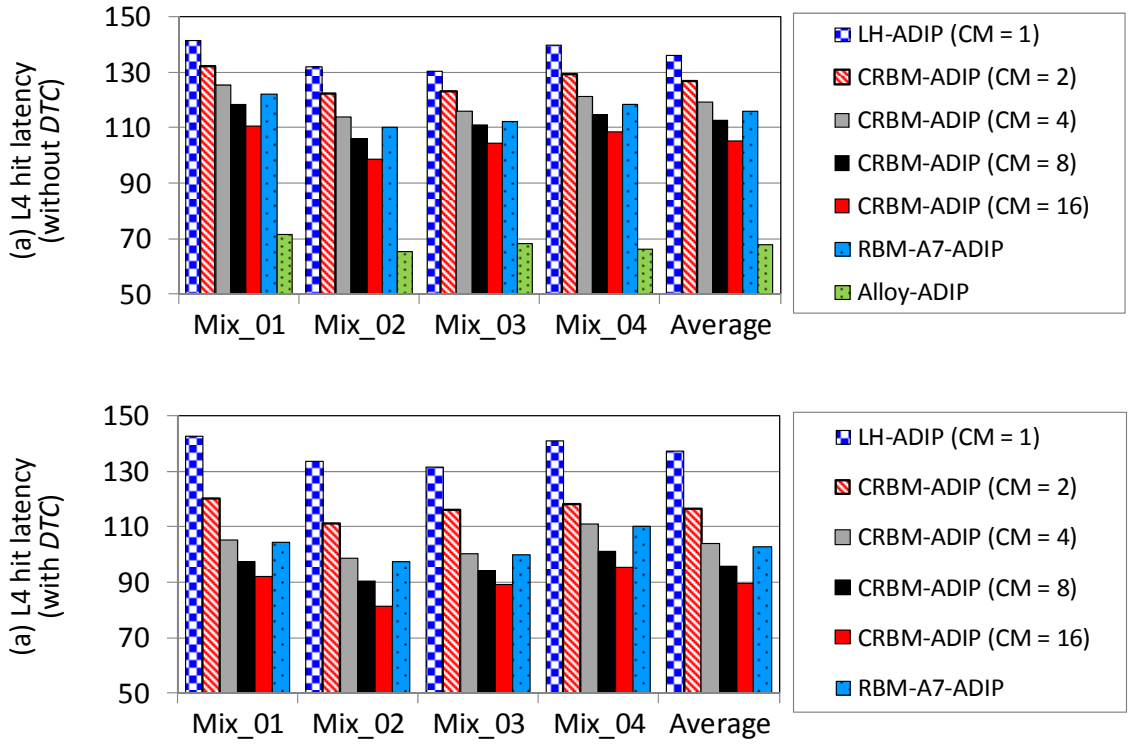


Figure 6.26: L4 DRAM cache hit latency (a) without *DTC* (b) with *DTC*; for different row buffer mapping policies

6.7.5 Performance improvement without *DTC*

Figure 6.27 shows the average normalized harmonic mean instruction per cycle (HM-IPC) throughput results for different values of CM with the speedup normalized to the *Alloy-ADIP* policy without *DTC*. On average, the configurable row buffer mapping policy without *DTC* improves the HM-IPC speed of latency sensitive applications by 17.2%/21.2%/14.2%/5.2% compared to the *Alloy-ADIP* policy for CM = 2/4/8/16, respectively. It improves the overall HM-IPC speedup by 14.4%/17.5%/10.4%/2.4% compared to the *Alloy-ADIP* policy for CM = 2/4/8/16, respectively.

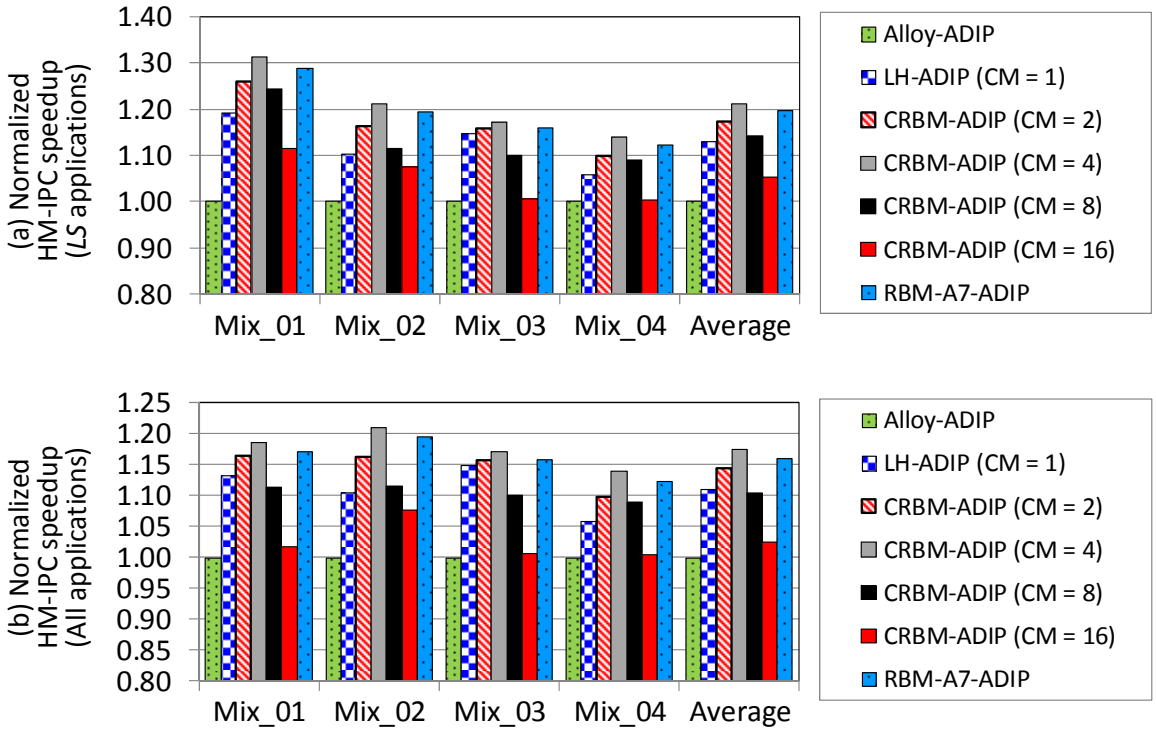


Figure 6.27: Normalized HM-IPC speedup compared to Alloy-ADIP for different row buffer mapping policies without DRAM Tag-Cache (*DTC*) for (a) latency sensitive applications (b) all applications

6.7.6 Performance improvement with *DTC*

The proposed configurable row buffer mapping policy gets significant performance benefits from adding a *DTC*. Figure 6.28 illustrates this observation showing the HM-IPC speed compared to the *Alloy-ADIP* policy. On average, the configurable row buffer mapping policy with *DTC* improves the HM-IPC speed of latency sensitive applications by 22.6%/29.1%/22.7%/16.1% compared to the *Alloy-ADIP* policy for CM = 2/4/8/16, respectively. It improves the overall HM-IPC speedup by 19%/24.8%/18.4%/13% compared to *Alloy-ADIP* policy for CM = 2/4/8/16, respectively.

The *LH-ADIP* policy has a significantly low DRAM cache row buffer hit rate (1.1%) and low *DTC* hit rate (1.6%), which results in a high L4 hit latency. Choosing a suitable value of CM is a compromise between L4 miss rate and L4 hit latency for the configurable row buffer mapping policy. Setting CM to a value of 4 provides the best performance improvement because it has a high DRAM row buffer hit rate (22.2%) and a high *DTC* hit rate (57.7%) compared to the *LH-ADIP* policy (DRAM row buffer hit rate is 1.1% and *DTC* hit rate is 1.6%). For this reason, CM = 4 significantly reduces the L4 hit latency by 24.3% compared to *LH-ADIP* policy with a negligible increase (3%) in L4 miss rate. Thus, the configurable row buffer mapping policy with CM = 4 and *DTC* significantly improves the HM-IPC speed of latency sensitive applications by 14.5% and the overall HM-IPC speed by 12.5% compared to *LH-ADIP* policy. The *LH-ADIP* policy gets negligible benefits (0.2%) from *DTC*, while the configurable row buffer mapping with CM = 4 provides additional 6.2% improvement in performance from *DTC*.

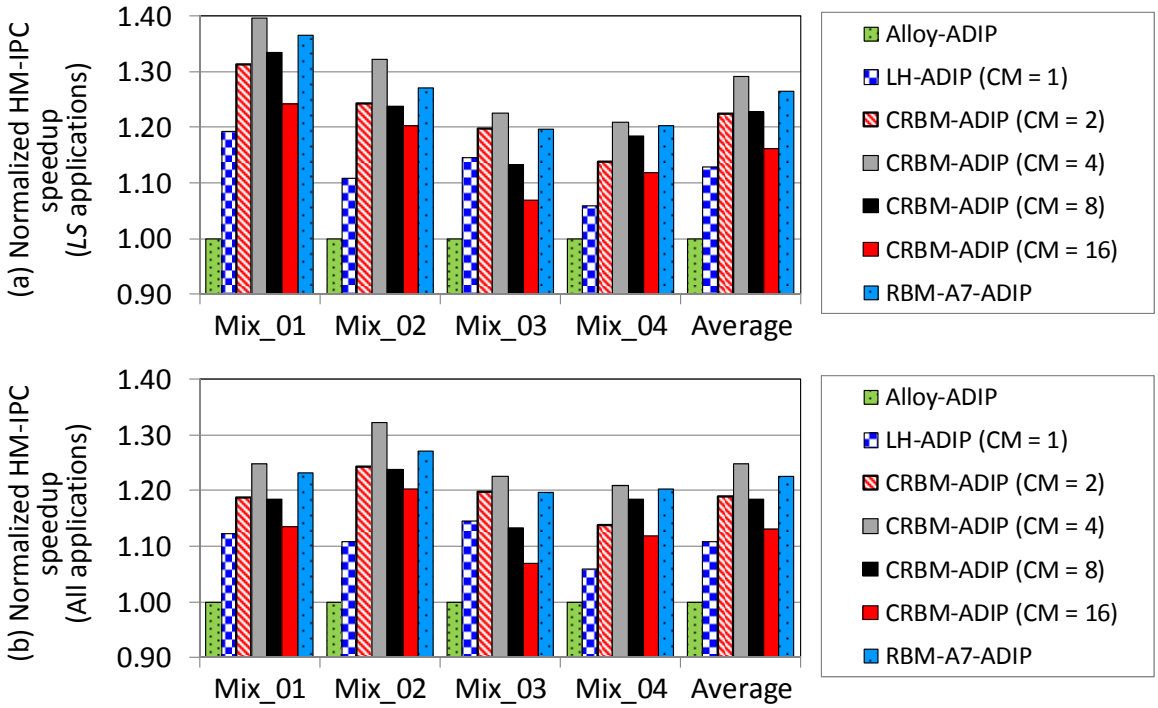


Figure 6.28: Normalized HM-IPC speedup compared to Alloy-ADIP for different row buffer mapping policies with DRAM Tag-Cache (*DTC*) (a) latency sensitive applications (b) all applications

6.7.7 Comparison of proposed policies

This section compares the performance of proposed configurable row buffer mapping policy with CM = 4 (namely *CRBM-ADIP-CM=4*) and the *RBM-A7-ADIP* policy. Both *CRBM-ADIP-CM=4* and *RBM-A7-ADIP* polices map 4 consecutive memory blocks to the same 2KB DRAM cache row, which results in almost similar row buffer and *DTC* hit rates as illustrated in Figure 6.29 (a-b).

The L4 tag latency for a *DTC* miss is 33 and 37 clock cycles respectively for the *RBM-A7-ADIP* and *CRBM-ADIP-CM=4* policies as shown in Figure 6.12. The L4 tag latency is reduced to 2 clock cycle for a *DTC* hit for both *RBM-A7-ADIP* and *CRBM-ADIP-CM=4* policies as illustrated in Figure 6.16-(a). A high *DTC* hit rate for the *CRBM-ADIP-CM=4* policy results in almost similar L4 hit latency compared to the *RBM-A7-ADIP* policy as illustrated in Figure 6.29-(c). The slight increase in L4 hit latency (0.7% as shown in Figure 6.29-c) for the *CRBM-ADIP-CM=4* policy compared to the *RBM-A7-ADIP* policy is compensated by a significant miss rate reduction (20.8% as shown in Figure 6.29-d) of latency sensitive applications. The miss rate is reduced because the *CRBM-ADIP-CM=4* policy provides a higher associativity (30-way associativity) compared to the *RBM-A7-ADIP* policy (7-way associativity). Thus, the *CRBM-ADIP-CM=4* policy improves the HM-IPC speed of latency sensitive applications by 2.2% (Figure 6.30-a) and the overall HM-IPC speed by 1.95% (Figure 6.30-b) compared to the *RBM-A7-ADIP* policy.

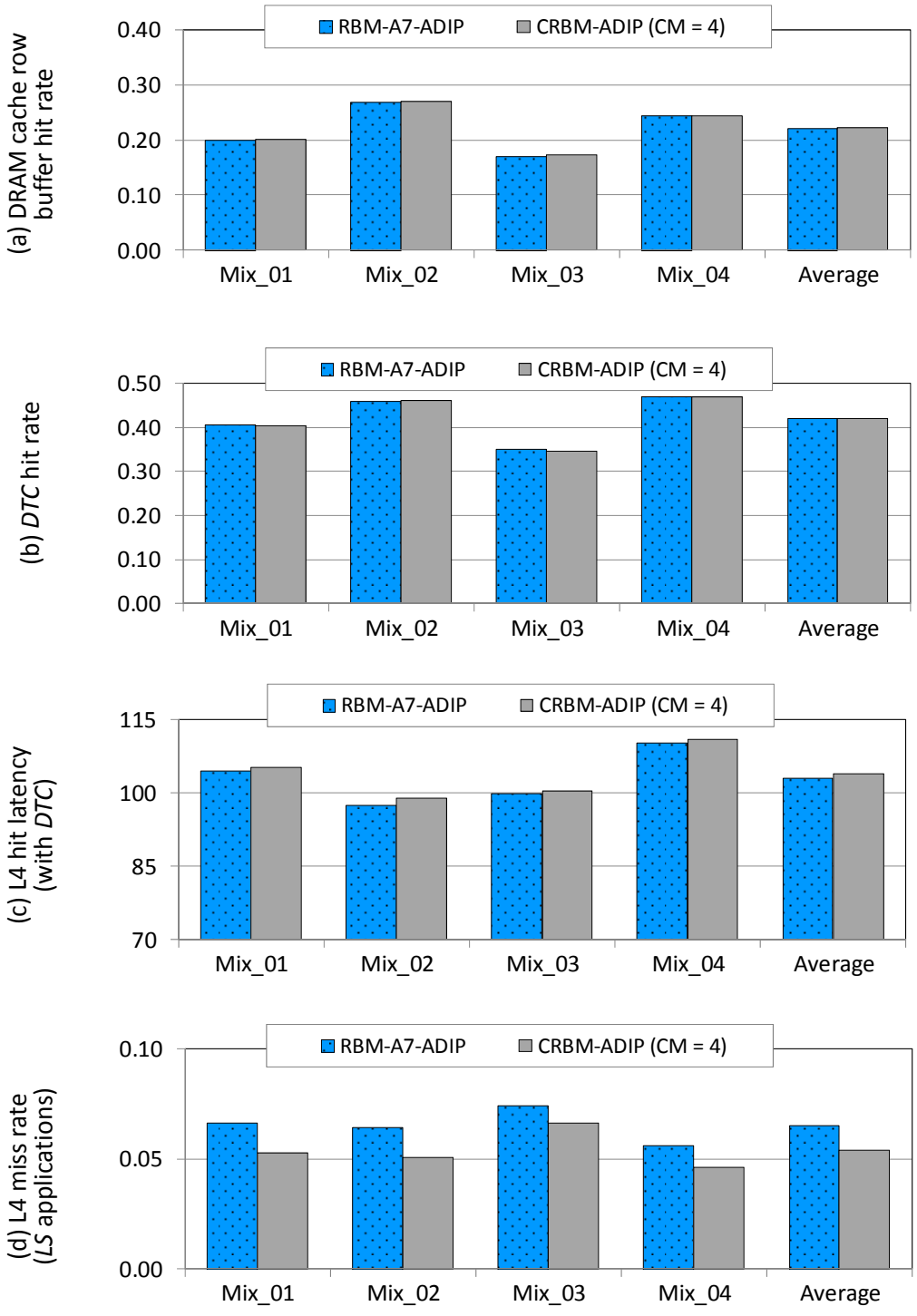


Figure 6.29: Average (a) DRAM cache row buffer hit rate (b) *DTC* hit rate (c) L4 hit latency with *DTC* (d) L4 miss rate for latency sensitive applications

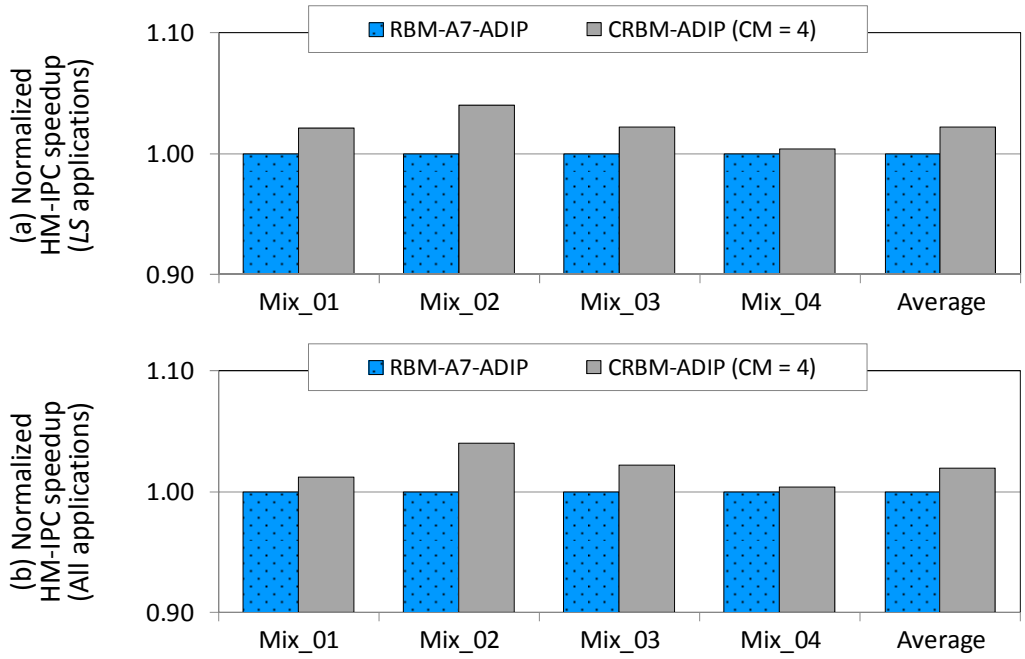


Figure 6.30: Normalized HM-IPC speedup compared to the *RBM-A7-ADIP* policy with DRAM Tag-Cache (*DTC*) (a) latency sensitive applications (b) all applications

6.8 Evaluating CRBM policy

6.8.1 Impact of row buffer mapping policy

For the rest of this chapter, the term *proposed-ADIP* refers to the configurable row buffer mapping policy with CM = 4, while the adaptive DRAM insertion policy is applied on top of it. This section evaluates and investigates the performance benefits of the *proposed-ADIP* policy compared to the state-of-the-art *Alloy-ADIP* (optimized for L4 hit latency with worst L4 miss rate) and *LH-ADIP* (optimized for L4 miss rate with worst L4 hit latency) policies. The main drawback of these policies is that they are optimized for a single parameter (L4 hit latency or L4 miss rate). In contrast, the *proposed-ADIP* simultaneously reduces L4 hit latency and L4 miss rate at the same time. On average, the *proposed-ADIP* without *DTC* improves the overall HM-IPC speedup by 17.5% and 5.9% compared to *Alloy-ADIP* and *LH-ADIP* policies respectively as illustrated in Figure 6.31.

6.8.2 Impact of Tag-Cache on performance

This section evaluates the performance benefits of incorporating a DRAM Tag-Cache (*DTC*; details in Section 6.5.1) on top of the *proposed-ADIP* namely *proposed-ADIP-DTC*. It also presents the performance benefits of incorporating an SRAM Tag-Cache (*STC*; details in Section 6.5.5) on top of the *proposed-ADIP-DTC* namely *proposed-ADIP-DTC-STC*. On average, the *proposed-ADIP-DTC* improves the overall HM-IPC speed by 24.9%, 12.5%, and 6.3% compared to *Alloy-ADIP*, *LH-ADIP*, and *proposed-ADIP* respectively as illustrated in Figure 6.31. On average, the *proposed-ADIP-DTC-STC* improves the overall HM-IPC speed by 26.5%, 14%, 7.6%,

and 1.26% compared to *Alloy-ADIP*, *LH-ADIP*, *proposed-ADIP*, and *proposed-ADIP-DTC* respectively as illustrated in Figure 6.31. The incorporation of the low latency DRAM Tag-Cache (*DTC*) on top of the *proposed-ADIP* provides additional 6.3% improvement in performance due to a high *DTC* hit rate (57.7% as shown in Figure 6.29-b) which avoids high latency DRAM cache and MMap$ access for a *DTC* hit. Similarly, the incorporation of the low latency *STC* on top of the *proposed-ADIP-DTC* provides an additional 1.26% improvement in performance due to a high *STC* hit rate (average *STC* hit rate is 61.2%).

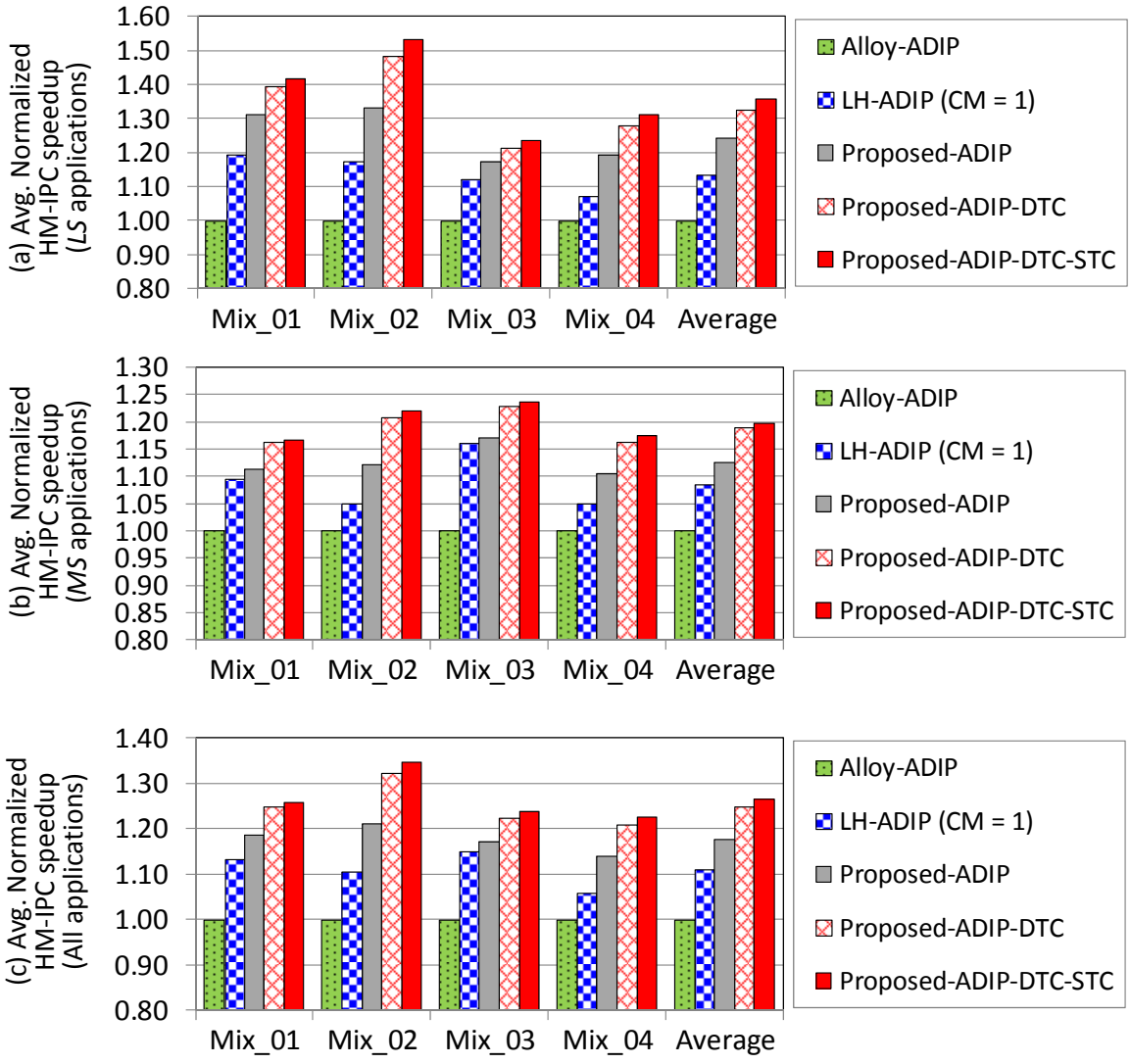


Figure 6.31: Normalized HM-IPC speedup compared to Alloy-ADIP for (a) Latency Sensitive (*LS*) applications (b) Memory Sensitive (*MS*) applications (c) Both *LS* and *MS* applications

6.8.3 Impact of the super-block size on performance

This section evaluates the impact of the super-block size (2 and 4; details in Section 6.4.1) on the overall performance by evaluating two policies namely the *proposed-ADIP-DTC-STC-sb-2* and the *proposed-ADIP-DTC-STC-sb-4* for super-block (sb) sizes of 2 and 4 respectively on top of

the *proposed-ADIP-DTC-STC* policy. Note that a super-block of size 2 (i.e. *proposed-ADIP-DTC-STC-sb-2* policy) reduces the MMap$ size by 34% and a super-block of size 4 (i.e. *proposed-ADIP-DTC-STC-sb-4* policy) reduces the MMap$ size by 51% (details in Section 6.4.1). Reducing the MMap$ size reduces the MMap$ latency. The MMap$ latency is assumed to be 10 clock cycles (for 2MB MMap$) for all configurations, although a reduced MMap$ is employed for the *proposed-ADIP-DTC-STC-sb-2* (1.33 MB) and the *proposed-ADIP-DTC-STC-sb-4* (1MB MMap$) policies (i.e. a conservative comparison).

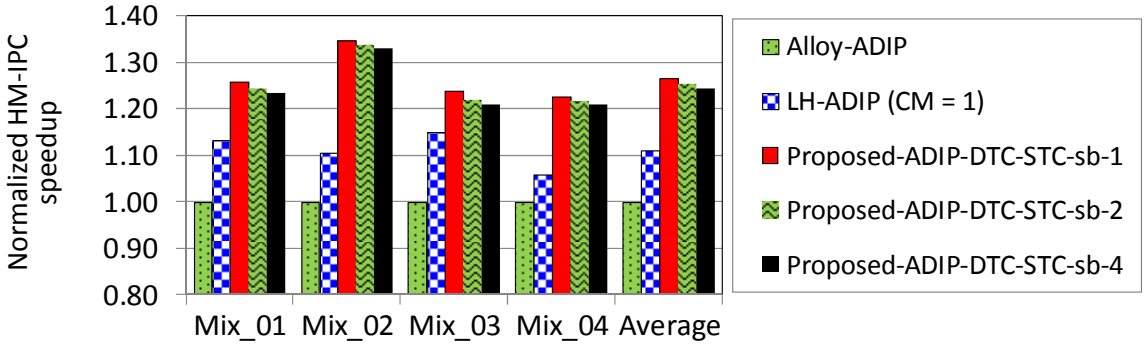


Figure 6.32: Normalized HM-IPC speedup for different super-block (sb) sizes

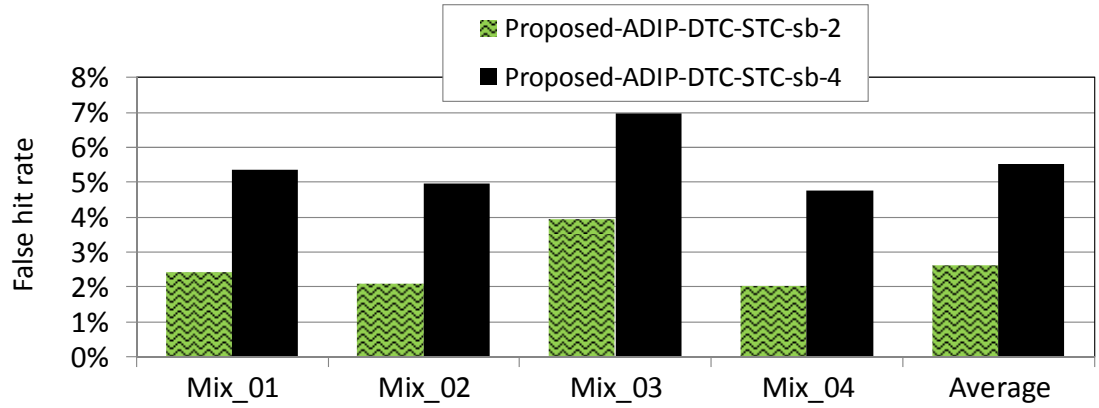


Figure 6.33: Percentage of false hits for super-blocks (sb) of size 2 and 4

On average, the *proposed-ADIP-DTC-STC-sb-2* improves the overall HM-IPC speedup by 25.3% and 12.9% compared to the *Alloy-ADIP* and *LH-ADIP* policies respectively as shown in Figure 6.32. On average, the *proposed-ADIP-DTC-STC-sb-4* improves the HM-IPC speedup by 24.4% and 12.1% compared to the *Alloy-ADIP* and *LH-ADIP* respectively. Figure 6.33 shows the percentage of false hits detected for the *proposed-ADIP-DTC-STC-sb-2* and *proposed-ADIP-DTC-STC-sb-4* policies. The percentage of false hits is 2.6% and 5.5% for the *proposed-ADIP-DTC-STC-sb-2* and *proposed-ADIP-DTC-STC-sb-4* respectively. Note that most of the false hits are eliminated by the *DTC* (*DTC* hit rate is 57.7% for CM = 4) if "maybe hit" is identified by the proposed SB-MMap$.

6.9 Summary

This chapter presented a novel DRAM row buffer mapping policy for on-chip DRAM caches that simultaneously improves the DRAM cache miss rate and the DRAM cache hit latency. Along with that it proposed the concept of a DRAM Tag Cache, a small and low latency SRAM structure that further improves the DRAM cache hit latency. This chapter further applied the concepts of the Tag-Cache architecture on top of L3 SRAM cache. This chapter performed extensive evaluations and compared the performance of the proposed approaches with two state-of-the-art row buffer mapping policies for on-chip DRAM caches. For an 8-core system, the proposed policies improve the harmonic mean instruction per cycle throughput by 24.4% and 12.1%, respectively. At the same time, it requires 51% less storage overhead to determine the DRAM cache hit/miss prediction.

The detailed analysis showed that it is the combination of improved miss rate and hit latency that provides the general performance improvement. As the proposed DRAM Tag cache architecture allows reducing the size of the MMap$ structure (used in DRAM caches to provide hit/miss prediction), the performance improvement comes at reduced area overhead which makes it generally applicable for a wide range of applications and architectures.

Chapter 7 Putting It All together: DRAM Last-Level-Cache Policies

The performance of an SRAM/DRAM cache hierarchy can be improved by reducing the average latency of a read request which can be reduced by reducing the latency and miss rate at different levels of the cache hierarchy. This thesis has presented several techniques to reduce the average access latency by reducing the L3 SRAM hit latency, L4 DRAM hit latency and L4 DRAM miss rate via novel policies while reducing MissMap Cache (MMap$) storage overhead. The performance benefits of the proposed policies along with the storage reduction benefits are presented in Chapter 5 and Chapter 6 in detail. Each of the proposed policy is employed on different on-chip hardware structures of the SRAM/DRAM cache hierarchy including L3 SRAM, L4 DRAM and MMap$ caches. Therefore, the proposed policies are complementary.

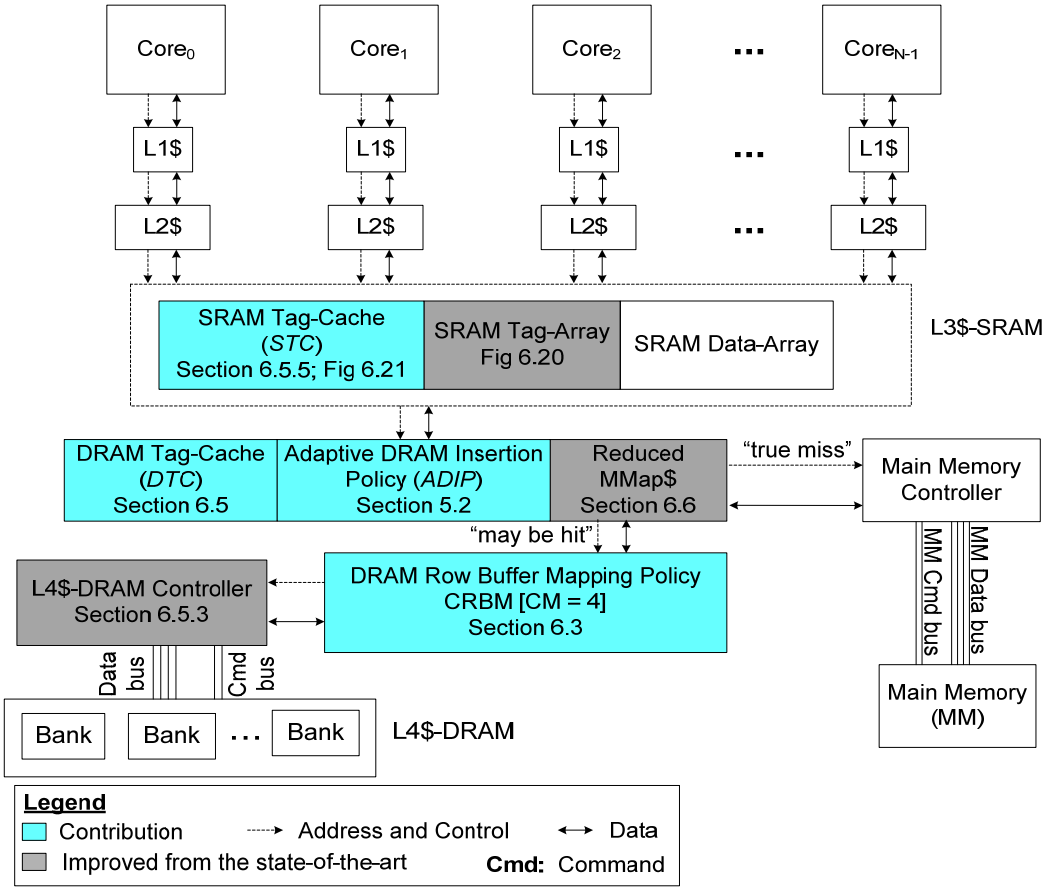


Figure 7.1: Proposed SRAM/DRAM cache hierarchy showing integration of selected policies

Figure 7.1 shows the proposed organization of the four-level SRAM/DRAM cache hierarchy illustrating the integration of selected policies (shown in highlighted areas) from Chapter 5 and Chapter 6. The proposed policies work synergistically, which improves the performance compared to state-of-the-art SRAM/DRAM cache hierarchies. This chapter discusses and evaluates

the performance gains, miss rate and latency benefits of the selected contributions (highlighted in Figure 7.1) when they are incorporated in the cache hierarchy.

7.1 Evaluation

For the evaluation on a multi-core system where each core is based on the x86 microarchitecture, this thesis has compared the proposed policies with state-of-the-art DRAM cache namely LH-Cache [77, 78] and Alloy-Cache [102], which share the same philosophy of reducing miss rate via increased DRAM cache capacity. However, they do not fully exploit the application and DRAM characteristics, which can cause inter-core interference. In addition, they incur a large area overhead required for DRAM cache hit/miss prediction, which reduces the area advantage of a DRAM cache.

	LH	***LH-ADIP***	***CM=4-ADIP***	***CM=4-ADIP-TAG$-RS***
Adaptive DRAM Insertion Policy (*ADIP*) (Details in Section 5.2)	×	✓	✓	✓
Configurable row Buffer Mapping Policy with CM= 4 (Section 6.3.2 and Section 6.3.5)	×	×	✓	✓
Tag-Cache (*TAG$*) (Details in Section 6.5)	×	×	×	✓
Reducing Storage (*RS*) size (Details in Section 6.4)	×	×	×	✓

Table 7.1: Overview of different configurations with their incorporated policies

For evaluation, this chapter compares the following different configurations:

1. State-of-the-art row buffer mapping policy with static DRAM insertion policy namely *Alloy* [102].
2. State-of-the-art row buffer mapping policy with static DRAM insertion policy namely *LH* [77, 78].
3. Proposed adaptive DRAM insertion policy (*ADIP*; details in Section 5.2) on top of *LH* [77, 78] configuration namely *LH-ADIP*.
4. Proposed *ADIP* on top of proposed configurable row buffer mapping policy with CM = 4 (details in Section 6.3.2 and Section 6.3.5) namely *CM=4-ADIP*.
5. Proposed DRAM Tag-Cache (details in Section 6.5.1) and SRAM Tag-Cache organizations (details in Section 6.5.5) incorporated in the cache hierarchy along with reduced storage overhead (using super block of size 4; details in Section 6.4 and Section 6.8.3) on top of *CM=4-ADIP* configuration namely *CM=4-ADIP-TAG$-RS*. The acronym *TAG$* and *RS* stands for Tag-Cache (SRAM and DRAM Tag-caches) and reduced storage, respectively. For this configuration, modifications have been made in the SRAM Tag-array (to incorporate the

SRAM Tag-Cache; details in Section 6.5.5), in the MMap$ (to reduce the storage overhead; details in Section 6.4), and in the DRAM cache controller (to incorporate the DRAM Tag-Cache; details in Section 6.5.3).

Table 7.1 shows an overview of different configurations with their incorporated policies. The first column of Table 7.1 shows the proposed policies while the first row represents the evaluated configurations. The second column in the table shows the state-of-the-art *LH* policy. Each additional column introduces an additional policy or enhancement in the cache hierarchy when built on top of the previous configuration. For example, the configuration shown in the third column (*LH-ADIP*) applies the adaptive DRAM insertion policy on top of the configuration shown in the second column (*LH*).

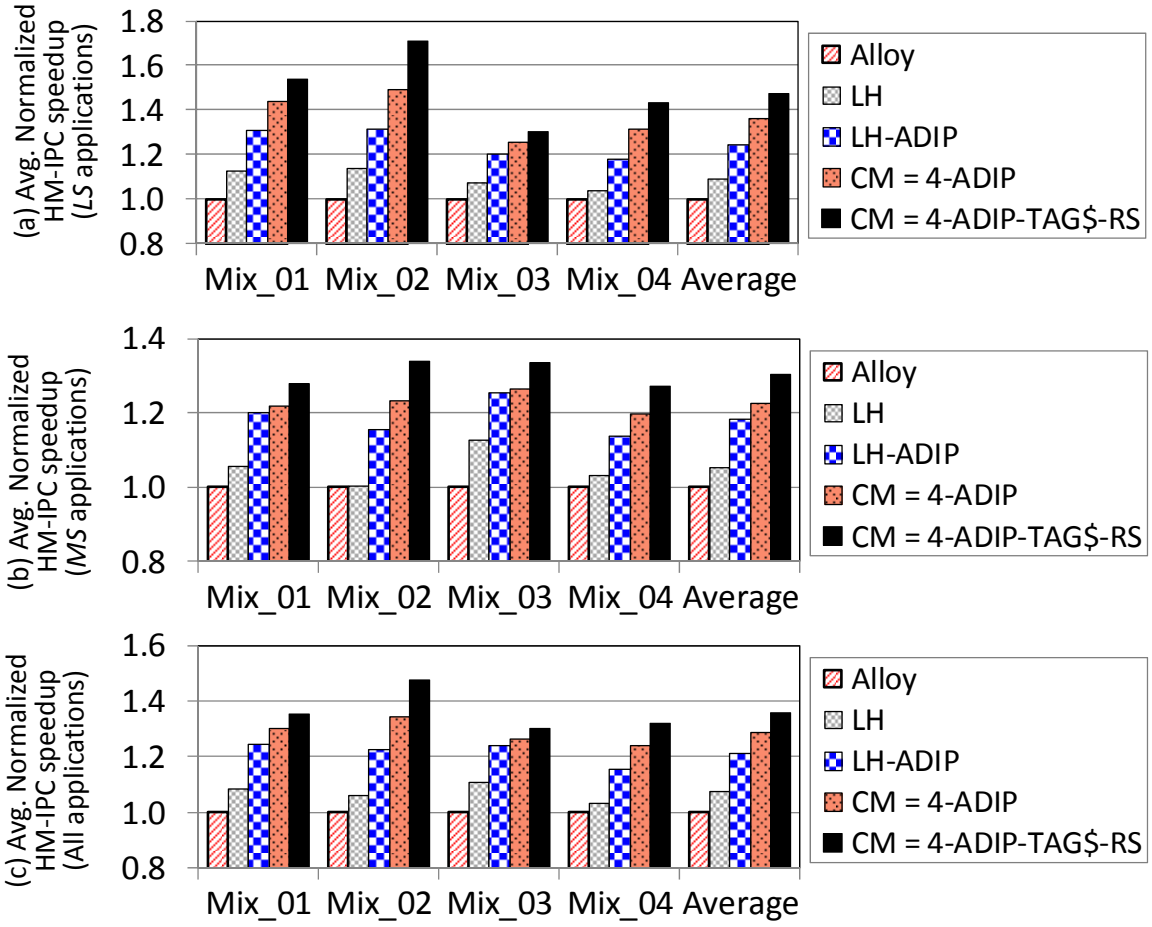


Figure 7.2: Normalized HM-IPC speedup compared to Alloy for (a) Latency Sensitive (*LS*) applications (b) Memory Sensitive (*MS*) applications (c) Both *LS* and *MS* applications

7.1.1 Performance benefits

Figure 7.2 shows the average normalized harmonic mean instruction per cycle (HM-IPC) throughput results for various configurations with the speedup normalized to the *Alloy* policy. On average, the combination of the proposed policies improves the HM-IPC speed of latency sensitive applications by 47.1% and 35% compared to the *Alloy* and *LH* policies respectively

(Figure 7.2-a). At the same time, it improves the HM-IPC speedup of memory sensitive applications by 30.4% and 23.9% compared to the *Alloy* and *LH* policies respectively (Figure 7.2-b). This results in an overall HM-speedup of 35.9% and 26.7% compared to the *Alloy* and *LH* policies respectively (Figure 7.2-c).

7.1.2 DRAM Aware Last-Level-Cache Policies are complementary

Figure 7.2 demonstrates that the proposed policies are complementary when they are employed in the cache hierarchy. Each newly added policy provides additional performance improvements compared to the identical configuration but without that policy as described in the following.

1. The *LH-ADIP* configuration (with the adaptive DRAM insertion policy) improves the overall HM-speed by 13.1% compared to the *LH* configuration (without adaptive DRAM insertion policy).
2. The *CM=4-ADIP* configuration (with the proposed configurable row buffer mapping policy with CM = 4) improves the overall HM-speedup by 5.9% compared to the *LH-ADIP* configuration (with state-of-the-art row buffer mapping policy from [77, 78]).
3. The *CM=4-ADIP-TAG$-RS* configuration (with the proposed Tag-Cache architecture along with the storage reduction technique that reduces the MMap$ storage overhead by 51%) improves the overall HM-speedup by 5.8% compared to the *CM=4-ADIP* configuration (without Tag-cache architecture and storage reduction).

7.2 Result analysis

The performance of a DRAM cache based multi-core system depends upon the L4 DRAM miss rate, L4 DRAM hit latency and off-chip memory latency. The proposed SRAM/DRAM cache hierarchy simultaneously optimizes all of the above mentioned metrics to improve the overall instruction throughput. This section describes and evaluates the miss rate and latency benefits of the proposed policies.

7.2.1 Miss rate reduction

The combination of the proposed policies namely the *CM=4-ADIP-TAG$-RS* configuration significantly reduces the L4 miss rate by 46.3% and 20.5% compared to *Alloy* [102] and *LH* [77, 78] respectively as shown in Figure 7.3. The miss rate is primarily reduced due to the adaptive DRAM insertion policy (*ADIP*) that mitigates inter-core interference via reducing the insertion rate of rarely-reused blocks. Note that the configurations with *ADIP* have a reduced miss rate compared to the configurations without *ADIP* as shown in Figure 7.3. The proposed adaptive DRAM insertion policy reduces the number of fill requests by 52.1% and increases the number of useful demand requests by 39.6% compared to the static DRAM insertion policy. Thus, the proposed *ADIP* policy improves the DRAM cache bandwidth and capacity utilization, which significantly reduces the L4 DRAM miss rate by 20.5% compared to the static DRAM insertion policy.

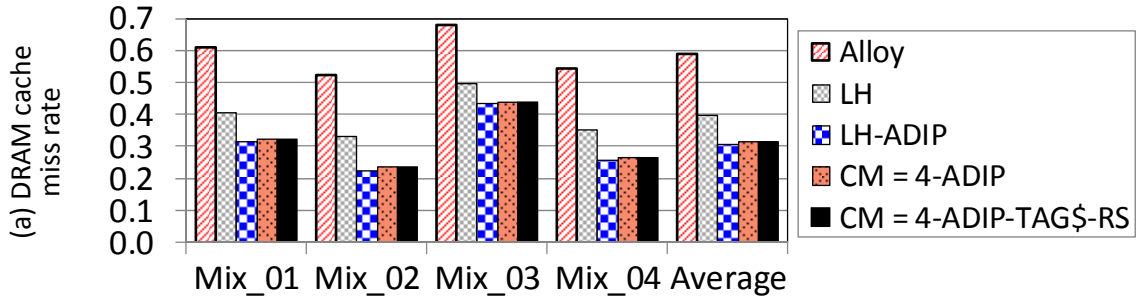


Figure 7.3: L4 DRAM cache miss rate

7.2.2 Off-chip memory latency reduction

The proposed *CM=4-ADIP-TAG$-RS* configuration reduces the off-chip memory latency by 32.1% and 10.7% compared to *Alloy* [102] and *LH* [77, 78] respectively as shown in Figure 7.4. The off-chip memory latency is directly dependent on the L4 DRAM miss rate, which increases with an increased number of misses due to increased contention in the memory controller. The main memory access latency is reduced (see Figure 7.4) for all configurations using the adaptive DRAM insertion policy due to reduced main memory controller queuing/scheduling delay and contention.

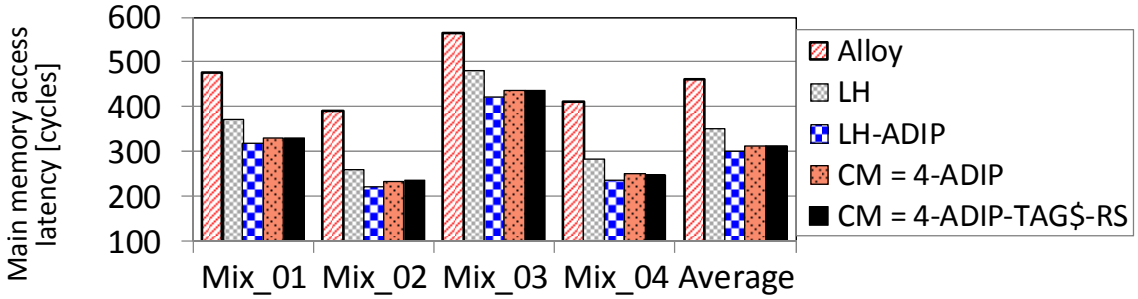


Figure 7.4: Off-chip main memory access latency

7.2.3 L4 DRAM hit latency reduction

The L4 DRAM hit latency depends upon the L4 tag latency (lower is better; see Figure 6.12 and Figure 6.16), the DRAM cache row buffer hit rate (higher is better; see Figure 7.5-b), and the contention in the DRAM cache controller. The L4 hit latency also depends upon the DRAM Tag-Cache (*DTC*) hit rate (higher is better) for the *CM=4-ADIP-TAG$-RS* configuration that incorporates *DTC* in the cache hierarchy.

The *Alloy* configuration (based on the Alloy-Cache [102]; details in Section 2.4.3) is optimized for L4 hit latency due to fast tag lookup (Figure 6.12 and Table 6.1) compared to other configurations. At the same time, it significantly improves the DRAM row buffer hit rate compared to other configurations as shown in Figure 7.5-(b). However, the improvement in L4 hit latency for the direct mapped *Alloy* configuration (Figure 7.5-a) comes at the cost of increased L4

miss rate (Figure 7.3) and off-chip memory latency (Figure 7.4), which leads to significant performance degradations compared to other configurations (Figure 7.2).

The proposed configurations (i.e. *LH-ADIP*, *CM=4-ADIP*, and *CM=4-ADIP-TAG$-RS*) benefit from a high associativity (30-way) with significantly reduced L4 miss rate and off-chip memory latency compared to the *Alloy* configuration as illustrated in Figure 7.3 and Figure 7.4 respectively.

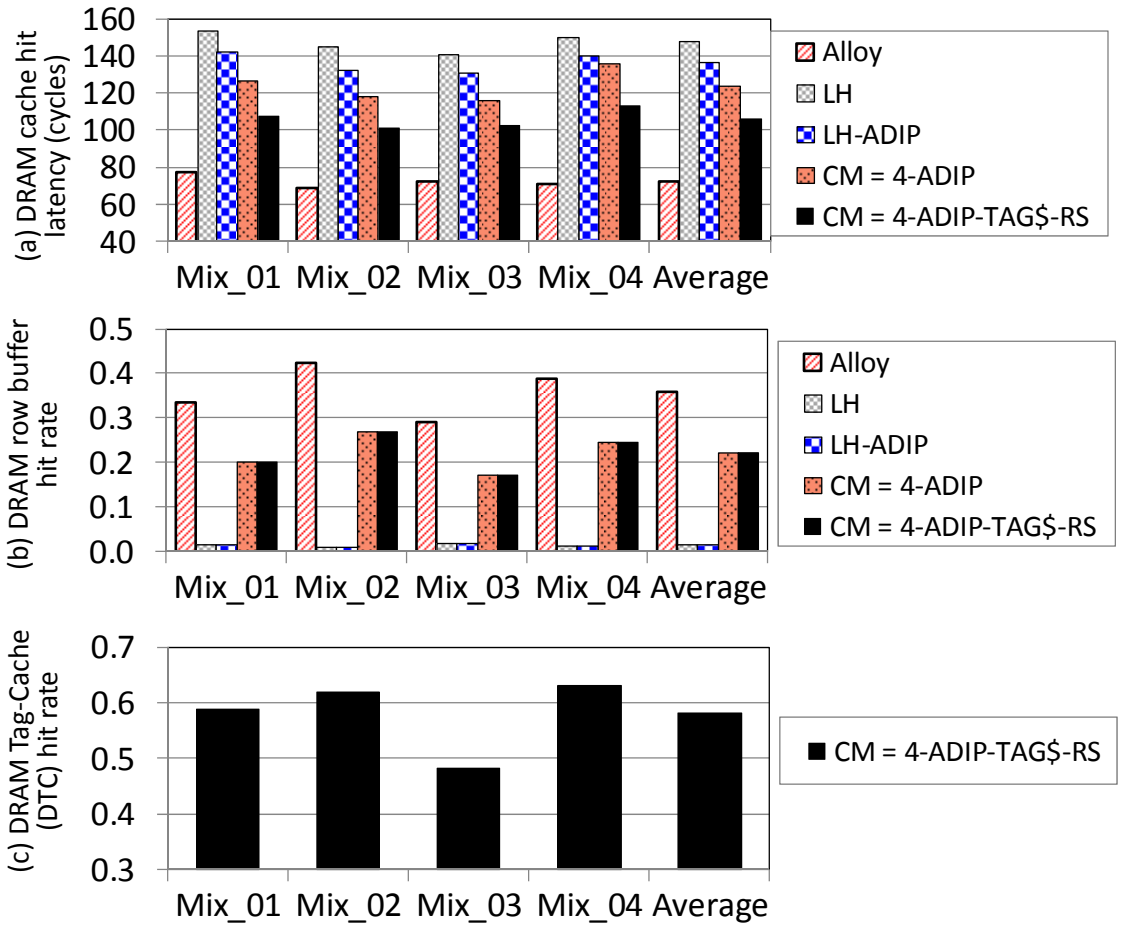


Figure 7.5: (a) L4 DRAM hit latency (b) DRAM row buffer hit rate (c) DRAM Tag-Cache hit rate

Each newly added policy provides additional reduction in L4 hit latency compared to the *LH* configuration (employed in LH-Cache [77, 78]) as described in the following.

1. The L4 hit latency reduction via the adaptive DRAM insertion policy (*ADIP*) is 7.7% compared to a static DRAM insertion policy, while comparing *LH* and *LH-ADIP* configurations. The L4 hit latency is reduced due to reduced contention in the DRAM cache controller because the adaptive DRAM insertion policy reduces the number of fill requests by 52.1% and increases the number of useful demand requests by 39.6% compared to the static DRAM insertion policy.
2. The L4 hit latency reduction via the *CM=4-ADIP* configuration (proposed configurable row buffer mapping policy with CM = 4) is 8.9% compared to the *LH-ADIP* configuration (state-

of-the-art row buffer mapping policy in [77, 78]). The latency is reduced because the *CM=4-ADIP* configuration (row buffer hit rate is 22.1%) significantly improves the row buffer hit rate compared to the *LH-ADIP* configuration (row buffer hit rate is only 1.4%) as shown in Figure 7.5-(b). The row buffer hit rate is improved because the proposed row buffer mapping policy reduces row buffer conflicts via exploiting data access locality in the row buffer by mapping four consecutive blocks to the same DRAM cache row buffer. In contrast, the state-of-the-art row buffer mapping policy in [77, 78] does not fully exploit data access locality because it maps consecutive blocks to different row buffers with significantly lower row buffer hit rate.

3. The L4 DRAM hit latency reduction via the *CM=4-ADIP-TAG$-RS* configuration (with the proposed Tag-Cache architecture along with the storage reduction technique) is 14.5% compared to the *CM=4-ADIP* configuration (without Tag-cache architecture). The latency reduction is due to the fast tag lookup (i.e. L4 tag latency is reduced) via high DRAM Tag-Cache hit rate (58%) as shown in Figure 7.5-(c). The L4 tag latency is reduced for a DRAM Tag-Cache (*DTC*) hit because the tags are read from the low latency *DTC* in two cycles for the *CM=4-ADIP-TAG$-RS* configuration. In contrast, the *CM=4-ADIP* configuration always access the MMap$ followed by reading the tags from the slower DRAM cache, which incurs high L4 tag latency of 37 cycles (see Figure 6.16).

7.3 Summary

DRAM cache management has become more challenging in multi-core systems because of increased inter-core cache contention (leads to increased DRAM cache miss rate), increased inter-core DRAM interference (leads to increased DRAM cache hit latency), limited off-chip memory bandwidth (leads to increased off-chip memory latency), and DRAM organization (DRAM cache is slower compared to SRAM cache). This chapter described and evaluated the performance benefits of application and DRAM aware complementary policies to address the above mentioned challenges. This chapter showed that the application aware adaptive DRAM insertion policy can mitigate inter-core DRAM interference and inter-core cache eviction, which lead to reduced DRAM cache miss rate and hit latency. It further explored and investigated the latency benefits of the novel DRAM aware policies (i.e. novel DRAM row buffer mapping policy, Tag-cache architecture, and DRAM controller optimizations) that further improved the performance of DRAM cache based multi-core systems.

This chapter performed extensive evaluations and compared the performance of complementary policies with two state-of-the-art proposals for on-chip DRAM caches. Through detailed performance analysis, this chapter showed that the proposed policies provide simultaneous reduction in DRAM cache miss rate, DRAM cache hit latency and off-chip memory access latency compared to state-of-the-art DRAM cache, resulting in substantial performance benefits. For an 8-core system, the combination of the proposed policies improves the performance of latency sensitive applications by 47.1% and 35% compared to two state-of-the-art proposals for on-chip DRAM caches. At the same time, it requires 51% less storage SRAM overhead required to manage on-chip DRAM cache.

Chapter 8 Conclusion and Outlook

As the pressure on the off-chip memory tends to increase due to large application footprints of complex applications [32], a traditional SRAM-based cache hierarchy cannot satisfy the capacity requirements of these applications with large working set sizes. The problem worsens for multi-core systems with increasing number of cores causing memory bandwidth problems [12, 60, 108, 131, 136, 144]. To mitigate these problems, various proposals for on-chip DRAM caches have been proposed because of the DRAM capacity advantages compared to traditional SRAM caches and the low latency advantage compared to off-chip main memory. Although on-chip DRAM cache provides high capacity (compared to SRAM caches) and large bandwidth (compared to off-chip memory), it is not simple to actually integrate it in the cache hierarchy due to its higher latency (compared to SRAM caches) and complex management. Before summarizing the thesis contributions in Section 8.1, the challenges that need to be addressed to employ the emerging on-chip DRAM cache in an SRAM/DRAM based cache hierarchy are summarized as follows:

1. Efficient management of DRAM cache capacity and bandwidth is required to mitigate inter-core cache eviction and inter-core DRAM interference respectively.
2. The data access locality in the row buffer needs to be exploited in order to reduce the DRAM cache hit latency via improved row buffer hit rate.
3. The tag lookup latency needs to be minimized, which is the dominant factor of a DRAM cache hit latency. Similarly, the tag lookup latency for the large shared SRAM cache should be minimized.
4. The total SRAM storage required for DRAM cache management should be minimal, as it incurs high system cost.

8.1 Thesis Summary

This thesis addresses the challenges of state-of-the-art DRAM cache hierarchies that limit the overall instruction throughput. It proposes low-overhead policies in order to provide improved performance for DRAM cache based multi-core systems. The proposed policies exploit the capacity benefits of emerging on-chip DRAM cache at the architectural level to achieve high performance while simultaneously considering the application (e.g. cache access pattern) and DRAM system characteristics (e.g. bandwidth and row buffer locality etc.).

To efficiently manage on-chip DRAM cache capacity and bandwidth, an *adaptive DRAM insertion* policy (*ADIP*) has been presented that adapts the DRAM insertion rate at runtime based on the miss rate information provided by a low overhead hardware monitoring unit. The proposed *ADIP* is not only capable of adapting the DRAM insertion rate of concurrently running applications on a multi-core system, but it is also able to dynamically adjust the DRAM insertion rate during different execution phases of the same application. *ADIP* restricts the number of *zero-reuse* data (i.e. data that is not reused before it gets evicted) insertions into the DRAM cache, which reduces inter-core DRAM interference via reduced contention in the DRAM cache controller. It reduces the number of evictions for the *highly-reuse* data (i.e. data that is likely to be reused in the near future), which reduces inter-core cache evictions. It maximizes DRAM cache

bandwidth and capacity utilization for highly-reuse data and it minimizes the negative effect of zero-reuse data. It has been shown in Chapter 5 that the proposed adaptive DRAM insertion policy significantly outperforms the existing static DRAM insertion policy and requires negligible hardware overhead. It has been demonstrated in Chapter 5 that the proposed *ADIP* can be applied to any DRAM cache organization.

To maximize the DRAM row buffer locality, a novel row buffer mapping policy has been proposed that simultaneously optimizes the DRAM cache hit latency and DRAM cache miss rate. The proposed row buffer mapping policy maps four consecutive memory blocks to the same row buffer, which results in a significantly higher row buffer hit rate with negligible increase in miss rate compared to state-of-the-art. It reduces the DRAM cache hit latency via an improved row buffer hit rate while exploiting programs spatial and temporal locality. At the same time, it reduces the DRAM cache miss rate via a higher associativity. It has been demonstrated in Chapter 6 that the proposed row buffer mapping policy outperforms state-of-the-art row buffer mapping policies that are either optimized for DRAM cache hit latency or for DRAM cache miss rate.

To minimize the tag lookup latency for a large shared DRAM cache, this thesis proposes a low-overhead and low-latency SRAM structure namely DRAM Tag-Cache (*DTC*) that can quickly determine whether an access to the large DRAM cache will be a hit or a miss. However, the performance of the proposed DRAM Tag-Cache depends upon the *DTC* hit rate. This thesis demonstrates that integrating a *DTC* into a recently proposed hierarchy for on-chip DRAM cache [77, 78] provides negligible latency and performance benefits due to their low *DTC* hit rates (they map consecutive memory blocks to different row buffers, which leads to a significantly low *DTC* hit rate). However, when the *DTC* is integrated with the proposed row buffer mapping policy that exploits programs spatial and temporal locality by mapping four consecutive memory blocks to the same row buffer, it exhibits a high hit rate. In contrast to the previous proposal [77, 78] for on-chip DRAM cache that always reads the tags from the slower DRAM cache after a MissMap cache (MMap$) access (requires 41 cycles for both accesses), the proposed *DTC* provides fast tag lookup that incurs only two cycles for a *DTC* hit. Similarly, to minimize the tag lookup latency for a large shared SRAM cache, this thesis applies the concept of the Tag-Cache architecture on top of SRAM cache. The latency and performance benefits using the proposed Tag-Cache architecture (for SRAM and DRAM cache) are discussed in Chapter 6.

To reduce the hardware cost required for DRAM cache management with minimal impact on the overall performance, this thesis reduces the storage overhead of the recently proposed MMap$ (provides precise information about DRAM cache hit/miss using fine-grained presence information). The MMap$ overhead is reduced by storing presence information at coarser level instead of storing fine-grained presence information. The main drawback of storing coarse-grained presence information is that it increases the number of false DRAM cache hits. However, incorporating the DRAM Tag-cache (*DTC*) along with the proposed row buffer mapping policy reduces the number of false hits via a high *DTC* hit rate. Thus, the proposed approach significantly reduces the storage overhead of the existing MMap$ by 51% with a negligible performance degradation of only 1.6% (compared to a larger precise MMap$) due to a false hit rate of 5.5%.

The policies proposed in this thesis are able to efficiently mitigate inter-application interference, maximize DRAM cache capacity and bandwidth utilization, exploit DRAM row buffer locality, and reduce tag lookup latency. It has been demonstrated in Chapter 7 that when the pro-

posed policies are combined together, they synergistically improve the overall performance of SRAM/DRAM based cache hierarchies.

The proposed policies are evaluated for various applications from SPEC2006 [5] using a modified version of a cycle accurate performance simulator [79] that supports a detailed cache and memory hierarchy model. Experimental results demonstrate that the synergistic combination of the proposed complementary policies improves the harmonic mean instruction per cycle throughput of latency sensitive applications by 47.1% and 35% compared to Alloy-Cache [102] and LH-Cache [77, 78], respectively. At the same time, it requires 51% less storage overhead for the MMap$.

8.2 Future Work

The latency and miss rate advantages, the comparisons with state-of-the-art proposals for on-chip SRAM/DRAM cache hierarchy, and the experimental results using SPEC2006 [5] workloads demonstrate that the proposed policies are capable of improving the overall instruction throughput of SRAM/DRAM cache based multi-core systems. These encouraging results open up new directions for research in on-chip cache hierarchies, which are summarized as follows.

Emerging memory technologies: The concepts proposed in this thesis are not limited to DRAM based memory technology only. They are flexible enough to be applied to any other emerging memory technologies (e.g. Phase change memory [72, 100, 101, 103], Spin-transfer torque RAM [84, 117, 121, 145], magnetic RAM [21, 134], and resistive RAM [83]) that exhibit characteristics similar to DRAM. For instance, the novel adaptive DRAM insertion policy proposed in this thesis can be applied to the phase change memory (PCM) to mitigate inter-application interference and finite PCM endurance. The proposed row buffer mapping policy along with the Tag-cache architecture can be extended to PCM to mitigate long PCM latencies via improved row buffer locality and high Tag-Cache hit rates. Thus, the novel concepts proposed in this thesis can be extended to multi-core cache hierarchies that use new emerging memory technologies.

Emerging cache hierarchies: The on-chip cache hierarchy has remained relatively simple in the past, consisting of traditional SRAM based caches. However, the continues improvement in process technology (e.g. die stacking [52, 61, 62, 66, 70, 92, 109] and heterogeneous integration [81, 146]) and memory technologies [72, 83, 84, 100, 101, 103, 117, 121, 145] have led to the evolution of emerging hybrid cache hierarchies [19, 134, 135]. These hybrid cache hierarchies will likely be composed of different memory technologies to exploit their capacity (e.g. DRAM, PCM, STT-RAM, and RRAM etc.) and latency (e.g. SRAM and embedded DRAM etc.) benefits. These cache hierarchies provide different latency, area and power trade-offs when compared with traditional SRAM based cache hierarchies. The policies presented in this thesis need to be modified for these future cache hierarchies in order to exploit their latency and miss rate benefits more effectively.

Compile-time DRAM insertion policy: The performance of a DRAM cache based multi-core system depends upon memory access patterns. The proposed adaptive DRAM insertion policy allocates the DRAM cache resources to concurrently running applications at runtime while considering their memory access patterns. However, a compile time DRAM insertion policy

considering DRAM (e.g. considering DRAM row buffer locality) and application (e.g. considering applications' memory access pattern) characteristics can further improve DRAM capacity utilization by giving hints about application behavior at the page level to the operating system. This approach can increase the programming complexity. However, it can improve the overall instruction throughput via reduced off-chip accesses, if efficiently managed by the compiler.

Operating-system based DRAM insertion policy: The proposed adaptive DRAM insertion policy performs the on-chip DRAM cache insertion/bypass decision at the block level. However, it is also possible to carry out the insert/bypass decision at the page-level by exposing the on-chip DRAM cache and off-chip memory resources to the operating system. To perform page-level on-chip DRAM cache management, efficient operating system based techniques are required to determine which pages are critical to be inserted in the on-chip DRAM cache. For instance, many applications have small working set sizes that can fit within the limited capacity of on-chip DRAM memory. Inserting the pages of these applications in the on-chip DRAM cache will lead to a significant reduction in off-chip memory traffic. Even for the applications with large working set size that do not fit into the on-chip DRAM memory, there exists a significant variation in the page usage. For instance, some pages are frequently used by the application (classified as hot pages), while other pages are rarely used by the application (classified as cold pages). The operating system can track the page usage statistics to determine whether a recently accessed pages should be inserted into on-chip DRAM cache or not. Inserting the most frequently accessed hot pages in the on-chip DRAM cache while bypassing the rarely reused cold page can lead to a noticeable performance improvement, if effectively handled by the operation system.

Reducing row buffer conflicts: This thesis has proposed a novel row buffer mapping policy to mitigate row buffer conflicts compared to state-of-the-art while exploiting programs' spatial and temporal locality. However, when the DRAM cache banks are shared among a large number of applications, it results in increased row buffer conflicts in the same bank via increased inter-application contention, which limits the performance. One possibility is to allow each bank to be accessed by a limited number of applications to reduce row buffer conflicts. This approach would require an intelligent mapping of the application pages to different DRAM cache banks that can be handled at the operating system level. An intelligent application to bank mapping can potentially reduce the negative impact of inter-application interference via reduced row buffer conflicts.

Although this thesis demonstrated the latency, miss rate and performance benefits of the proposed policies for an SRAM/DRAM cache hierarchy, there is still a lot of room for interesting future work in the area of on-chip caching for multi-core systems.

Bibliography

[1] Intel Inc. http://ark.intel.com.

[2] Intel Inc. http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[3] Intel® Xeon® Processor E5-2690. http://ark.intel.com/products/64596/intel-xeon-processor-e5-2690-20m-cache-2_90-ghz-8_00-gts-intel-qpi.

[4] International Technology Roadmap for Semiconductors. http://www.itrs.net.

[5] Standard Performance Evaluation Corporation. http://www.spec.org.

[6] Agarwal A., J. Hennessy, and M. Horowitz. Cache Performance of Operating System and Multiprogramming Workloads. *ACM Transactions on Computer Systems (TOCS)*, 6(4):393–431, November 1988.

[7] R. X. Arroyo, R. J. Harrington, S. P. Hartman, and T. Nguyen. IBM POWER7 Systems. *IBM Journal of Research and Development*, 55(3):2:1 – 2:13, 2011.

[8] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-level Cache Hierarchies. *SIGARCH Computer Architecture News*, 16(2):73–80, May 1988.

[9] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J Zook. TILE64 Processor: A 64-Core SoC with Mesh Interconnect. In *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 88–598, February 2008.

[10] B.N. Bershad, D. Lee, T.H. Romer, and J. B. Chen. Avoiding Conflict Misses Dynamically in Large Direct-mapped Caches. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 158–170, 1994.

[11] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, and L. Jiang. Die-Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–479, December 2006.

[12] S. Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[13] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[14] John Callaham. Intel Announces its First i7 8-core Extreme Edition Processor. http://-www.neowin.net/news/intel-announces-its-first-i7-8-core-extreme-edition-processor, March 2014.

[15] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Contention on a Chip Multi-Processor Architecture. In *Proceedings of the 11th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 340–351, February 2005.

[16] J. Chang and G. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, pages 264–276, June 2006.

[17] J. Chang and G. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *Proceedings of the 21st International Conference on Supercomputing (ICS)*, pages 242–252, June 2007.

[18] M. Chang, P. Rosenfeld, S. Lu, and B. Jacob. Technology Comparison for Large Last-Level Caches (L3Cs): Low-Leakage SRAM, Low Write-Energy STT-RAM, and Refresh-Optimized eDRAM. In *Proceedings of the 19th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 143–154, 2013.

[19] Y. T. Chen, J. Cong, H. Huang, L. Chunyue, M. Potkonjak, and G. Reinman. Dynamically Reconfigurable Hybrid Cache: An Energy-Efficient Last-Level Cache Design. In *Proceedings of the 15th conference on Design, Automation and Test in Europe*, DATE ’12, pages 45–50, March 2012.

[20] Y. Deng and W. Maly. Interconnect Characteristics of 2.5-D System Integration Scheme. In *Proceeding of the International Symposium on Physical Design (ISPD)*, pages 171–175, April 2001.

[21] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen. Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) As a Universal Memory Replacement. In *Proceedings of the 45th Design Automation Conference (DAC)*, pages 554–559, 2008.

[22] H. Dybdahl and P. Stenström. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proceedings of the 13th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 2–12, February 2007.

[23] T. Ebi, H. Rauhfuss, A. Herkersdorf, and J. Henkel. Agent-based Thermal Management using Real-time I/O Communication Relocation for 3D Many-cores. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization, and Simulation workshop (PATMOS’11)*, 2011.

[24] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE MICRO*, 28(3):42–53, May 2008.

[25] J. Feehrer, S. Jairath, P. Loewenstein, R. Sivaramakrishnan, D. Smentek, S. Turullols, and A. Vahidsafa. The Oracle Sparc T5 16-Core Processor Scales to Eight Sockets. *IEEE MICRO*, 33(2):48–57, 2013.

[26] David Geer. Chip Makers Turn to Multicore Processors. *Computer, IEEE Computer Society*, 38(5):11–13, 2005.

[27] David Geer. For Programmers, Multicore Chips Mean Multiple Challenges. *Computer, IEEE Computer Society*, 40(9):17–19, 2007.

[28] S.W. Golumb. Shift Register Sequences, 1982.

[29] J. R. Goodman. Using Cache Memory to Reduce Processor-memory Traffic. *SIGARCH Computer Architecture News*, 11(3):124–131, June 1983.

[30] J. R. Goodman. Using Cache Memory to Reduce Processor-memory Traffic. In *Proceedings of the 10th International Symposium on Computer Architecture (ISCA)*, pages 124–131, 1983.

[31] P. Frost Gorder. Multicore Processors for Science and Engineering. *IEEE Computer Science Engineering*, 9(2):3–7, 2007.

[32] Darryl Gove. CPU2006 Working Set Size. *SIGARCH Computer Architecture News*, 35(1):90–96, March 2007.

[33] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A Novel SIMD Architecture for the Cell Heterogeneous Chip-multiprocessor. In *Proceedings of the 17th Hot Chips*, 2005.

[34] S. Gupta, M. Hilbert, S. Hong, and R. Patti. Techniques for Producing 3D ICs with High-Density Interconnect. In *Proceedings of the 21st International VLSI Multilevel Interconnection Conference*, 2004.

[35] F. Hameed, L. Bauer, and J. Henkel. Dynamic Cache Management in Multi-Core Architectures through Run-time Adaptation. In *Proceedings of the 14th conference on Design, Automation and Test in Europe (DATE)*, pages 485–490, March 2012.

[36] F. Hameed, L. Bauer, and J. Henkel. Adaptive Cache Management for a Combined SRAM and DRAM Cache Hierarchy for Multi-Cores. In *Proceedings of the 15th conference on Design, Automation and Test in Europe (DATE)*, pages 77–82, March 2013.

[37] F. Hameed, L. Bauer, and J. Henkel. Reducing Inter-Core Cache Contention with an Adaptive Bank Mapping Policy in DRAM Cache. In *IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*, 2013.

[38] F. Hameed, L. Bauer, and J. Henkel. Simultaneously Optimizing DRAM Cache Hit Latency and Miss Rate via Novel Set Mapping Policies. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'13)*, 2013.

[39] F. Hameed, M.A. Al Faruque, and J. Henkel. Dynamic Thermal management in 3D Multi-core Architecture Through Run-Time Adaptation. In *Proceedings of the 14th conference on Design, Automation and Test in Europe (DATE)*, pages 299–304, March 2011.

[40] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and More Flexible Program Analysis. *Journal of Instruction Level Parallelism*, 7, 2005.

[41] J. Henkel, L. Bauer, J. Becker, O. Bringmann, U. Brinkschulte, S. Chakraborty, M. Engel, R. Ernst, H. Hartig, and L. Hedrich. Design and Architectures for Dependable Embedded Systems. In *IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'11)*, 2011.

[42] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends. In *Proceedings of the 50th Design Automation Conference (DAC'13)*, 2013.

[43] J. Henkel, T Ebi, H. Amrouch, and H. Khdr. Thermal Management for Dependable On-chip Systems. In *Asia and South Pacific Design Automation Conference (ASP-DAC'13)*, 2013.

[44] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. Invasive Manycore Architectures. In *Asia and South Pacific Design Automation Conference (ASP-DAC'12)*, pages 193–200, 2012.

[45] J.L. Hennessy and D.A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. 2006.

[46] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[47] M.D. Hill. A Case for Direct-mapped Caches. *IEEE Computer*, 21(12):25–40, December 1988.

[48] M.D. Hill and A. J. Smith. Experimental Evaluation of On-chip Microprocessor Cache Memories. In *Proceedings of the 11th International Symposium on Computer Architecture (ISCA)*, pages 158–166, 1984.

[49] M.D. Hill and A.J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[50] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting Set-associative Cache for High Performance and Low Energy Consumption. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '99)*, pages 273–275, 1999.

[51] A. Jaleel, W. Hasenplaugh, M.K. Qureshi, J. Sebot, S. Steely Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 208–219, 2008.

[52] Tokyo Japan, Elpida. Elpida completes development of Cu-TSV (Through Silicon Via) multi-layer 8-Gigabit DRAM. http://www.elpida.com/pdfs/pr/2009-08-27e.pdf, 2009.

[53] D. Jevdjic, S. Volos, and B. Falsafi. Die-stacked DRAM caches for Servers: Hit Ratio, Latency, or Bandwidth? Have it All with Footprint Cache. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 404–415, 2013.

[54] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms. In *Proceedings of the 16th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[55] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Integrating DRAM Caches For CMP Server Platforms. *IEEE Micro Magazine (Top Picks), IEEE Computer Society*, pages 99–108, 2011.

[56] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, 1994.

[57] N.P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 364–373, 1990.

[58] N.P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. *SIGARCH Computer Architecture News*, 18(2SI):364–373, May 1990.

[59] N.P. Jouppi and S.J.E Wilton. Tradeoffs in Two-level On-chip Caching. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 34–45, 1994.

[60] A. Kagi, J.R. Goodman, and D. Burger. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, pages 78–89, May 1996.

[61] U. Kang, H.-J. Chung, S. Heo, S.-H. Ahn, H. Lee, S.-H. Cha, J. Ahn, D. Kwon, J.H. Kim, J.-W. Lee, H.-S. Joo, W.-S. Kim, H.-K. Kim, E.-M. Lee, S.-R. Kim, K.-H. Ma, D.-H. Jang, N.-S. Kim, M.-S. Cho, S.-J. Oh, J.-B. Lee, T.-K. Jung, J.-H. Yoo, and C. Kim. 8 Gb 3-D DDR3 DRAM using Through-Silicon-Via Technology. In *IEEE Journal of Solid State Circuits*, volume 45, pages 111–119, January 2010.

[62] M. Kawano, S. Uchiyama, Y. Egawa, N. Takahashi, Y. Kurita, K. Soejima, M. Komuro, S. Matsui, K. Shibata, J. Yamada, M. Ishino, H. Ikeda, Y. Saeki, O. Kato, H. Kikuchi, and A. Mitsuhashi. A 3D packaging technology for 4 Gbit stacked DRAM with 3 Gbps data transfer. In *International Electron Devices Meeting*, pages 1–4, 2006.

[63] C.N. Keltcher, K.J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE MICRO*, 23(2):66–76, 2003.

[64] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache Replacement Based on Reuse-distance Prediction. In *Proceedings of the 25th International Symposium on Computer Design(ICCD)*, pages 245–250, 2007.

[65] S.M. Khan, D.A. Jiménez, D. Burger, and B. Falsafi. Using Dead Blocks as a Virtual Victim Cache. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 489–500, 2010.

[66] J.S. Kim, C.S. Oh, H. Lee, D. Lee, H.R. Hwang, S. Hwang, B. Na, J. Moon, J.G. Kim, H. Park, J.W. Ryu, K. Park, S.K. Kang, S.Y. Kim, H. Kim, J.M. Bang, H. Cho, M. Jang, C. Han, J.B. Lee, K. Kyung, J.S. Choi, and Y.H. Jun. A 1.2V 12.8GB/s 2Gb Mobile Wide-I/O DRAM with 4x128 I/Os using TSV-based Stacking. In *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 496–498, 2011.

[67] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, 2004.

[68] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory controllers. In *Proceedings of the 16th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 9–14, January 2010.

[69] Y. Kim, M. Papamichael, O. Mutlu, and M.H. Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior". In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 65–76, December 2010.

[70] D. Klein. The future of memory and storage: Closing the gaps. presented at the Microsoft Windows Hardware Engineering Conference, Los Angeles, CA, 2007.

[71] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th International Symposium on Computer Architecture (ISCA)*, pages 81–87, 1981.

[72] B.C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 2–13, June 2009.

[73] J. Lin, Q. Lu, X. Ding, Z. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of the 14th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 367–378, February 2008.

[74] G.H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 453–464, 2008.

[75] G.H. Loh. Extending the Effectiveness of 3D–stacked Dram Caches with an Adaptive Multi-Queue Policy. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 174–183, 2009.

[76] G.H. Loh. A Register-file Approach for Row Buffer Caches in Die-stacked DRAMs. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 351–361, 2011.

[77] G.H. Loh and M.D. Hill. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 454–464, 2011.

[78] G.H. Loh and M.D. Hill. Supporting Very Large DRAM Caches with Compound Access Scheduling and MissMaps. *IEEE Micro Magazine, Special Issue on Top Picks in Computer Architecture Conferences*, 32(3):70–78, 2012.

[79] G.H. Loh, S. Subramaniam, and Y. Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[80] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 164–171, 2001.

[81] N. Madan and R. Balasubramonian. Leveraging 3D Technology for Improved Reliability. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 223–235, 2007.

[82] C. Madriles, P. López, J.M. Codina, E. Gibert, F. Latorre, A. Martinez, R. Martinez, and A. Gonzalez. Boosting Single-thread Performance in Multi-core Systems through Fine-Grain Multi-Threading. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 474–483, June 2009.

[83] J. Meza, L. Jing, and O. Mutlu. A Case for Small Row Buffers in Non-volatile Main Memories. In *Proceedings of the 30th International Symposium on Computer Design(ICCD)*, pages 484–485, September 2012.

[84] A.K. Mishra, X. Dong, G. Sun, Y. Xie, N. Vijaykrishnan, and C.R. Das. Architecting On-chip Interconnects for Stacked 3D STT-RAM Caches in CMPs. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 69–80, 2011.

[85] G.E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[86] M. Moreto, F.J. Cazorla, A. Ramirez, and M. Valero. MLP-aware dynamic cache partitioning. In *Proceedings of the 3rd international conference on High performance embedded architectures and compilers (HiPEAC)*, pages 337–352, 2008.

[87] N. Muralimanohart and N. Balasubramonian, R.and Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 3–14, December 2007.

[88] O. Mutlu and T. Moscibroda. Stall-time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–160, December 2007.

[89] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 22–32, June 2008.

[90] E.J. O'Neil, P.E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297–306, 1993.

[91] V.S. Pai, P. Ranganathan, and S.V. Adve. The Impact of Instruction-level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 72–83, February.

[92] J. T. Pawlowski. Hybrid Memory Cube: Breakthrough DRAM Performance with a Fundamentally Re-Architected DRAM Subsystem. In *Proceedings of the 23rd Hot Chips*, 2011.

[93] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS Performance Evaluation Review*, 31(1):318–319, 2003.

[94] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 318–319, 2003.

[95] S. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 160–169, 1990.

[96] S. Przybylski, M. Horowitz, and J. Hennessy. Characteristics of Performance-Optimal Multi-Level Cache Hierarchies. In *Proceedings of the 16th International Symposium on Computer Architecture (ISCA)*, pages 114–121, 1989.

[97] M. K. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 45–54, February 2009.

[98] M. K. Qureshi, D Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, pages 167–178, June 2006.

[99] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-performance, Runtime mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–432, 2006.

[100] M.K. Qureshi, M.M. Franceschini, A. Jagmohan, and L.A. Lastras. PreSET: Improving Performance of Phase Change Memories by Exploiting Asymmetry in Write Times. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA)*, pages 380–391, 2012.

[101] M.K. Qureshi, M.M. Franceschini, and L.A. Lastras-Montano. Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing. In *Proceedings of the 16th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–11, Jan 2010.

[102] M.K. Qureshi and G.H. Loh. Fundamental Latency Trade-offs in Architecting DRAM Caches. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 235–246, 2012.

[103] M.K. Qureshi, V. Srinivasan, and J.A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 24–33, June 2009.

[104] M.K. Qureshi, D. Thompson, and Y.N. Patt. The V-Way Cache: Demand Based Associativity via Global Replacement. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 544–555, June 2005.

[105] V. Zyuban R. Kumar and D.M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 408–419, June 2005.

[106] J. M. Rabaey and S. Malik. Challenges and Solutions for Late- and Post-Silicon Design. *Design and Test*, 25(4):292–308, 2008.

[107] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. Memory Access Scheduling. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 128–138, June 2000.

[108] B.M. Rogers, A. Krishna., G.B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *SIGARCH Computer Architecture News*, 37(3):371–382, 2009.

[109] Samsung Semiconductor. Package Information. http://www.samsung.com/global/-business/semiconductor/support/package-info/overview, 2010.

[110] M. Shafique, S. Garg, D. Marculescu, and J. Henkel. The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives. In *Proceedings of the 51st Design Automation Conference (DAC'14)*, 2014.

[111] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, R. Hetherington, P. Jordan, M. Luttrel, O. Chistopher, B. Saha, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A Highly-Threaded, PowerEfficient, SPARC SOC. In *IEEE Asian Solid State Circuit Conference*, pages 22–25, 2007.

[112] J. Sim, G.H. Loh, H. Kim, M. O'Connor, and M. Thottethodi. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 247–257, 2012.

[113] J. Sim, G.H. Loh, V. Sridharan, and M. O'Connor. Resilient die-stacked DRAM caches. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, 2013.

[114] A. J. Smith. A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory. *IEEE Transaction on Software Engineering*, 4(2):121–130, March 1978.

[115] A. J. Smith. Line (Block) Size Choice for CPU Cache Memories. *IEEE Transaction on Computers*, 36(9):1063–1076, September 1987.

[116] A.J. Smith. Cache Memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[117] C.W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M.R. Stan. Relaxing Non-volatility for Fast and Energy-efficient STT-RAM Caches. In *Proceedings of the 17th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 50–61, February 2011.

[118] A. Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. *SIGARCH Computer Architecture News*, 28(5):234–244, 2000.

[119] G.S. Sohi. Instruction Issue Logic for High-performance, Interruptible, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, 1990.

[120] R. A. Sugumar and S.G. Abraham. Set-associative Cache Simulation Using Generalized Binomial Trees. *ACM Transaction on Computer System (TOCS)*, 13(1):32–56, February 1995.

[121] Z. Sun, X. Bi, H.H. Li, W-Fai Wong, Z-Liang Ong, X. Zhu, and W. Wu. Multi Retention Level STT-RAM Cache Designs with a Dynamic Refresh Scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '11, pages 329–338, 2011.

[122] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive Computing: An Overview. *Multiprocessor System-on-Chip – Hardware Design and Tool Integration, M. Hübner and J. Becker (Eds.), Springer*, 31(4):241–268, July 2011.

[123] S. Thoziyoor, J.Ho Ahn, A. Monchiero, J.B. Brockman, and N.P. Jouppi. A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 51–62, June 2008.

[124] S. Thoziyoor, J.H. Muralimanohart, R.and Ahn, and N. Jouppi. CACTI 5.1 HPL 2008/20, HP Labs, April 2008.

[125] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[126] J. Torrellas, M.S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[127] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS. In *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 98–589, February 2007.

[128] W. H. Wang, J.-L. Baer, and H. M. Levy. Organization and Performance of a Two-level Virtual-real Cache Hierarchy. *SIGARCH Computer Architecture News*, 17(3):140–148, April 1989.

[129] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, S. Weitzel, S.G. Chu, S. Islam, and V. Zyuban. The Implementation of Power7TM: A Highly Parallel and Scalable Multi-core High-end Server Processor. In *Proceedings of the International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 102–103, February 2010.

[130] R.T. White, F. Mueller, C.A. Healy, D.B. Whalley, and M.G. Harmon. Timing Analysis for Data Caches and Set-associative Caches. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.

[131] Maurice V. Wilkes. The Memory Gap and the Future of High Performance Memories. *SIGARCH Computer Architecture News*, 29(1):2–7, 2001.

[132] M.V. Wilkes. Slave Memories and Dynamic Storage Allocation. *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, 1965.

[133] D.H. Woo, N.H. Seong, D.L. Lewis, and H-H.S. Lee. An Optimized 3D-stacked Memory Architecture by Exploiting Excessive, High-density TSV Bandwidth. In *Proceedings of the 16th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[134] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid Cache Architecture with Disparate Memory Technologies. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 34–45, June 2009.

[135] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Design Exploration of Hybrid Caches with Disparate Memory Technologies. *ACM Transaction on Computer System (TOCS)*, 7(3):15:1–15:34, December 2010.

[136] W.A. Wulf and S.A. McKee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[137] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, June 2008.

[138] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 174–183, June 2009.

[139] Y. Xie, G.H. Loh, B. Black, and K. Bernstein. Design Space Exploration for 3D Architectures. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2(2):65–103, 2006.

[140] A. Zeng, R. Rose, and R.J. Gutmann. Memory Performance Prediction for High-Performance Microprocessors at Deep Submicrometer Technologies. *IEEE Transactions on Computer-aided Design Of Integrated Circuits and Systems*, 25(9):1705–1718, 2006.

[141] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-based Page Interleaving Scheme to Reduce Row-buffer Conflicts and Exploit Data Locality. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 32–41, 2000.

[142] Z. Zhang, Z. Zhu, and X. Zhang. Design and Optimization of Large Size and Low Overhead Off-Chip Caches. *IEEE Transactions on Computers*, 53(7):843–855, 2004.

[143] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring DRAM Cache Architecture for CMP Server Platforms. In *Proceedings of the 25th International Symposium on Computer Design(ICCD)*, pages 55–62, 2007.

[144] L. Zhao, R. Iyer, S. Makineni, J. Moses, R. Illikkal, and D. Newell. Performance, Area and Bandwidth Implications on Large-Scale CMP Cache Design. In *Proceedings of the Work. on Chip Multiprocessor Memory Systems and Interconnects*, 2007.

[145] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. Energy Reduction for STT-RAM Using Early Write Termination. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD'09)*, pages 264–268, 2009.

[146] Q. Zhu, B. Akin, H.E. Sumbul, F. Sadi, J.C. Hoe, L. Pileggi, and F. Franchetti. A 3D-stacked Logic-in-memory Accelerator for Application-specific Data Intensive Computing. In *Proceedings of the IEEE International conference on 3D Systems Integration (3DIC)*, pages 1–7, October 2013.