Universidad Zaragoza
1542

Tesis Doctoral

Reuse-based and near-optimal replacement policies for high-performance shared caches in multicore processors

Autor

Javier Díaz Maag

Director/es

Pablo Ibáñez Marín
Teresa Monreal Arnal

Escuela de Doctorado
Programa de Doctorado en Ingeniería de Sistemas e Informática
2025

A Carlos Díaz Tomás y Gabriele Maag Nolte:

Como una flor guardada entre las páginas de un libro,

vuestra esencia perfuma cada línea de mi vida.

Papá y mamá,

dedicado a vosotros.

ABSTRACT

With the widespread adoption of chip multiprocessors, the Shared Last-Level Cache (SLLC) has become a critical architectural feature for sustaining performance by bridging the growing gap between fast processing cores and significantly slower main memory. Its management—particularly the replacement policy—plays a central role in overall system performance. However, effectively managing the SLLC remains challenging due to limited temporal locality in the reference streams, which renders traditional cache management policies inefficient.

This dissertation addresses these challenges by introducing two complementary lines of work: practical mechanisms that exploit reuse locality to improve content selection in real systems, and near-optimal replacement policies that, while not implementable, provide theoretical baselines to guide the design and evaluation of shared cache policies. Specifically, this dissertation presents four major contributions: (1) a reuse-based content selection mechanism for exclusive caches (ReD), (2) its adaptation to energy-efficient STT-RAM SLLCs, (3) an enhanced instruction-aware variant (ReD+), and (4) two near-optimal offline replacement policies (NOPT) tailored to shared caches.

Firstly, we introduce the Reuse Detector (ReD), an innovative content selection mechanism designed specifically for exclusive cache hierarchies. Positioned between each L2 private cache and the SLLC, ReD selectively inserts only those blocks exhibiting past reuse, effectively reducing cache misses and improving system performance. Compared to state-of-the-art alternatives such as Cache Hierarchy-Aware Replacement (CHAR), Reuse Cache, and the Evicted Address Filter (EAF), ReD achieves higher effectiveness, reducing SLLC misses per instruction and enhancing harmonic IPC by 10.1% and 9.5%, respectively, relative to a baseline without content selection.

Addressing the limitations of conventional SRAM technologies, this thesis explores the potential of STT-RAM technology for building energy-efficient SLLCs when coupled with an advanced replacement policy. We propose an adapted version of ReD to mitigate the primary drawbacks of STT-RAM, notably slow and energy-intensive write operations. In multiprogrammed workloads, this integration significantly reduces energy consumption in both the SLLC and main memory — achieving, on average, a 33% reduction in SLLC energy use, an additional 6% in main memory, and a 7% performance improvement over

a baseline STT-RAM configuration without ReD. Compared to DASCA, a leading alternative, ReD also delivers superior performance and energy savings.

Building upon the initial ReD mechanism, we propose ReD+, an enhanced content selection policy that further refines reuse detection by incorporating instruction-level behavior tracking via a dedicated hardware structure. ReD+ selectively inserts blocks into the SLLC based on refined criteria involving both block reuse history and reuse patterns of requesting instructions. Adapted to work on conventional non-inclusive SLLCs, evaluations demonstrate that ReD+ matches or exceeds the performance of state-of-the-art policies for this typical category of SLLCs. This mechanism achieved significant recognition at the 2nd Cache Replacement Championship, ranking third overall and second in multiprocessor-specific scoring.

Lastly, the dissertation presents two near-optimal offline replacement policies for shared caches. The first policy approximates the optimal miss rate by iteratively reconstructing the future global access stream that reaches the SLLC and applying it to a classical OPTb policy. This process consistently converges, across evaluations, to a stable result within 0.1% of a near-optimal miss rate. Our evaluation shows that existing online (theoretically implementable) policies achieve approximately 65% of the miss-rate reduction and around 75% of the IPC improvements provided by this near-optimal approach (vs. random replacement). Recognizing that minimizing miss rate is not the sole objective in shared caches, we also introduce a near-optimal policy explicitly focused on fairness across cores. The best online policy, our proposed ReD+, achieves about 60% of the fairness improvement demonstrated by our near-optimal solution.

Collectively, these contributions provide both theoretical insights and practical strategies for optimizing cache management in multicore processors, establishing a robust foundation for future research aimed at addressing existing challenges in computer architecture.

RESUMEN

Con la adopción generalizada de los multiprocesadores en chip, la Cache Compartida de Último Nivel (*Shared Last-Level Cache*, SLLC) se ha convertido en un componente arquitectónico crucial para mantener el rendimiento, al servir de puente entre los rápidos núcleos de procesamiento y una memoria principal significativamente más lenta. Su gestión — en particular, la política de reemplazo— desempeña un papel central en el rendimiento global del sistema. Sin embargo, gestionar eficazmente la SLLC representa aún un desafío debido a la escasa localidad temporal en los flujos de accesos, lo que vuelve ineficientes a las políticas tradicionales de gestión de cache.

Esta tesis aborda estos desafíos mediante la introducción de dos líneas de trabajo complementarias: mecanismos prácticos que aprovechan la localidad de reuso para mejorar la selección de contenido en sistemas reales, y modelos de reemplazo cuasi-óptimos que, aunque no son implementables, proporcionan referencias teóricas para orientar el diseño y la evaluación de políticas de cache compartida. En concreto, esta tesis presenta cuatro contribuciones principales: (1) un mecanismo de selección de contenido basado en reuso para caches exclusivas (ReD), (2) su adaptación a SLLCs de STT-RAM de bajo consumo, (3) una variante mejorada con correlación instrucción-bloque (ReD+), y (4) dos políticas de reemplazo cuasi-óptimas (NOPT) diseñadas específicamente para caches compartidas.

En primer lugar, presentamos el Detector de Reuso (ReD), un mecanismo innovador de selección de contenido diseñado específicamente para jerarquías de cache exclusivas. Situado entre cada cache privada L2 y la SLLC, ReD inserta selectivamente sólo aquellos bloques que han mostrado reuso previo, reduciendo eficazmente los fallos de cache y mejorando el rendimiento del sistema. En comparación con propuestas punteras como CHAR, Reuse Cache y EAF cache, ReD logra una mayor efectividad, reduciendo los fallos por instrucción en la SLLC y mejorando la media armónica de IPC en un 10,1 % y un 9,5 %, respectivamente, en relación con un esquema base sin selección de contenido.

Abordando las limitaciones de las tecnologías convencionales de SRAM, esta tesis explora el potencial de la tecnología STT-RAM para construir SLLCs energéticamente eficientes cuando se combina con una política de reemplazo avanzada. Proponemos una versión adaptada de ReD que mitiga las principales desventajas de la STT-RAM, en

particular las operaciones de escritura lentas y de alto consumo energético. En cargas de trabajo multiprogramadas, esta integración reduce significativamente el consumo de energía tanto en la SLLC como en la memoria principal: en promedio, se obtiene una reducción del 33 % en el consumo energético de la SLLC, un 6 % adicional en la memoria principal y una mejora del rendimiento del 7 % respecto a una configuración base de STT-RAM sin ReD. En comparación con DASCA, una de las mejores alternativas, ReD también ofrece un mejor rendimiento y mayor eficiencia energética.

Sobre la base del mecanismo ReD original, proponemos ReD+, una política mejorada de selección de contenido que perfecciona la detección de reuso mediante el seguimiento del comportamiento a nivel de instrucción, utilizando una estructura hardware dedicada. ReD+ inserta selectivamente bloques en la SLLC en función de criterios mejorados que combinan el historial de reuso de los bloques con los patrones de reuso de las instrucciones que los solicitan. Adaptado para funcionar con SLLCs convencionales no inclusivas, las evaluaciones muestran que ReD+ iguala o supera el rendimiento de las políticas más avanzadas para esta típica categoría de SLLCs. Este mecanismo obtuvo un reconocimiento destacado en la 2ª edición del Cache Replacement Championship (CRC2), al alcanzar el tercer puesto en la clasificación general y el segundo en la puntuación específica de sistemas multiprocesador.

Por último, esta tesis presenta dos políticas de reemplazo *offline* cuasi-óptimas para caches compartidas. La primera de ellas aproxima la tasa óptima de fallos mediante la reconstrucción iterativa del flujo global de accesos futuros a la SLLC, aplicándolo posteriormente a una política OPTb clásica. Este proceso converge de forma consistente, en todas las evaluaciones, a un resultado estable con una desviación inferior al 0,1 % respecto de una tasa de fallos cuasi-óptima. Nuestra evaluación muestra que las políticas *online* existentes (implementables en la práctica) logran aproximadamente el 65 % de la reducción de fallos y cerca del 75 % de la mejora en IPC alcanzadas por esta política cuasi-óptima (en comparación con una política de reemplazo *random*). Reconociendo que minimizar la tasa de fallos no es el único objetivo en caches compartidas, también introducimos una política cuasi-óptima orientada explícitamente a la equidad (*fairness*) entre núcleos. La mejor política *online*, nuestra propuesta ReD+, alcanza aproximadamente el 60 % de la mejora en equidad lograda por nuestra solución cuasi-óptima.

ACKNOWLEDGEMENTS

Mi más sincero agradecimiento a los sabios Pablo, Teresa, Víctor y José María por su constante ayuda a lo largo de todos estos años de trabajo. Muy especialmente a Pablo, cuya determinación, energía y aliento han sido fundamentales; sin su impulso, difícilmente estaría hoy escribiendo estas palabras.

Un profundo agradecimiento también a María, mi mujer, por su apoyo incondicional. No podría soñar con una mejor compañera de viaje, sin importar en qué aventura nos embarquemos. Y a mis hijos, Rubén, Sergio y Alba, cuyo entusiasmo y frescura me inspiran cada día.

PUBLICATIONS

Part of this dissertation includes results already published.

The list of publications is:

- **Javier Díaz**, Teresa Monreal, Pablo Ibáñez, José María Llabería, and Víctor Viñals. (2019). *ReD: A reuse detector for content selection in exclusive shared last-level caches*. Journal of Parallel and Distributed Computing, 125, 106-120.
- Roberto Rodríguez-Rodríguez, **Javier Díaz**, Fernando Castro, Pablo Ibáñez, Daniel Chaver, Víctor Viñals, Juan Carlos Sáez, Manuel Prieto-Matías, Luis Piñuel, Teresa Monreal and José María Llabería. (2018). *Reuse detector: Improving the management of STT-RAM SLLCs*. The Computer Journal, 61(6), 856-880.
- **Javier Díaz**, Pablo Ibáñez, Teresa Monreal, Víctor Viñals and José María Llabería. (2017). *ReD: a policy based on reuse detection for a demanding block selection in last-level caches*. The 2nd Cache Replacement Championship, at the International Symposium on Computer Architecture (ISCA).
- **Javier Díaz**, Pablo Ibáñez, Teresa Monreal, Víctor Viñals and José María Llabería. (2021). *Near-optimal replacement policies for shared caches in multicore processors*. The Journal of Supercomputing, 77, 11756-11785.

ABBREVIATIONS

APKC: Accesses per kilocycle

ART: Address reuse table

CHAR: Cache hierarchy-aware replacement (policy)

CMP: Chip multiprocessor

CRC2: Second cache replacement championship

DASCA: Dead write prediction assisted STT-RAM cache architecture

DBP: Dead block predictor

DDR3: Dual data rate version 3

DIP: Dual insertion (replacement) policy

DRAM: Dynamic random-access memory

DRRIP: Dynamic re-reference interval prediction (replacement policy)

EAF: Evicted address filter (replacement policy)

eDRAM: Embedded dynamic random-access memory

FIFO: First in, first out

FRD: Future reuse distance

hIPC: Harmonic instructions per cycle

I/D: Instruction / data (caches)

IPC: Instructions per cycle

L1: First-level (cache)

L2: Second-level (cache)

LLC: Last-level cache

LRF: Least recently filled (replacement policy)

LRU: Least recently used

MM: Main memory

MOESI: Modified, owned, exclusive, shared, and invalid (cache coherence protocol)

MOSI: Modified, owned, shared, and invalid (cache coherence protocol)

MPI: (Cache) misses per instruction

MPKI: (Cache) misses per kiloinstruction

MRU: Most recently used

MSHR: Miss status holding register

MTJ: Magnetic tunnel junction

NOPT: Near optimal (replacement policy)

NOPTb: Near optimal with bypass (replacement policy)

NRF: Not recently filled (replacement policy)

NRU: Not recently used (replacement policy)

NVM: Non-volatile memory

OAP: Obstruction-aware (cache management) policy

PC: Program counter

PCRT: Program counter reuse table

PHT: Pattern history table

QoS: Quality of service

RAM: Random access memory

ReD: Reuse detector

RRPV: Re-reference interval prediction value

SHiP: Signature-based hit prediction (replacement policy)

SLLC: Shared last-level cache.

SPEC: Standard Performance Evaluation Corporation

SRAM: Static random-access memory

SRRIP: Static re-reference interval prediction (replacement policy)

STT-RAM: Spin-Transfer Torque Random Access Memory

TC-AGE: Trip count and age (replacement policy)

TLB: Translation lookaside buffer

WPKI: Writes per kiloinstruction

WC: Write cache

WS: Weighted speedup

YAGS: Yet Another Global (branch prediction) Scheme

CONTENTS

Abstract ii
Resumen iv
Acknowledgements vi
Publications vii
Abbreviations viii
Contents xi
List of figures xvi
List of tables xix
1 Introduction 1
1.1 Motivation 1
1.2 Summary of contributions 4
1.3 Dissertation overview 8
1.4 Thesis Project framework 9
2 ReD: A Reuse Detector for content selection in exclusive Shared Last-Level Caches 10
2.1 Introduction 10
2.2 Motivation 13
2.2.1 Problem analysis 13
2.2.2 Detector design 14
2.3 Design and implementation of ReD 16
2.3.1 Baseline 16
2.3.2 Adding the Reuse Detector 16
2.3.3 ReD operation 18
2.3.4 Example 19
2.3.5 Implementation details 22
2.3.6 Hardware costs 22
2.4 ReD operation insight 24
2.5 Experimental setup 26
2.5.1 The experimental framework 26
2.5.2 Configuration of the baseline system 27
2.5.3 Configuration of the evaluated proposal 28

2.5.4 Performance metrics 29
2.6 Performance analysis of ReD 29
2.6.1 Results and comparison with other proposals 30
2.6.2 Alive and dead blocks 33
2.6.3 Per-application and per-mix performance 34
2.6.4 Content selection 35
2.6.5 Single-processor performance 35
2.6.6 Detector efficiency 36
2.6.7 Additional cache sizes 38
2.6.8 Alternative cache replacement policy: Least Recently Filled (LRF) 39
2.6.9 Additional performance metrics 40
2.7 Design space exploration 40
2.7.1 ReD capacity 41
2.7.2 ReD sector size 41
2.7.3 ReD tag size 43
2.8 Related work 44
2.8.1 Replacement policies 45
2.8.2 Content selection policies 46
2.9 Conclusions 46
3 ReD: Improving the management of STT-RAM Shared Last-Level Caches 48
3.1 Introduction 48
3.2 Background and motivation 49
3.2.1 Comparison of SRAM and STT-RAM technologies 50
3.2.2 The DASCA scheme 51
3.3 Design of ReD for an STT-RAM SLLC 52
3.3.1 Baseline system 52
3.3.2 Adding ReD 53
3.3.3 ReD operation 53
3.3.4 Implementation details 55
3.3.5 Comparison of ReD and DASCA 55
3.4 Experimental setup 57
3.4.1 The experimental framework 57
3.4.2 Configuration of the baseline system 58

3.4.3 Configuration of the evaluated proposal 59
3.4.4 Performance metrics 59
3.5 Evaluation 60
3.5.1 Evaluation in a single-processor system 60
3.5.2 Evaluation in a multiprocessor system 65
3.6 Related work 70
3.7 Conclusions 72
4 ReD+: A policy based on reuse detection for optimized demanding block selection in shared last-level caches 74
4.1 Introduction 74
4.1.1 Motivation. Problem analysis 74
4.1.2 Inclusion relationships in the cache hierarchy 76
4.1.3 The 2nd Cache Replacement Championship 76
4.2 ReD+ content selection policy 77
4.3 Implementation details 78
4.3.1 Address Reuse Table (ART) 78
4.3.2 Program Counter Reuse Table (PCRT) 79
4.3.3 Increasing the effectiveness of the ART 80
4.3.4 Other details 80
4.3.5 Storage costs 81
4.4 Evaluation 81
4.4.1 The experimental framework 81
4.4.2 Configuration of the baseline system 82
4.4.3 Results 83
4.5 Evaluation at CRC2 85
4.5.1 The experimental framework 85
4.5.2 Results 86
4.6 Conclusions 87
5 Near-optimal replacement policies for shared caches in multicore processors 89
5.1 Introduction 89
5.2 Background 91
5.2.1 Replacement policies for shared caches in multicore processors 91
5.2.2 Optimal policies 92
5.2.3 OPT and OPTb for a private cache 93

5.2.4 Why it is not possible to implement OPT for shared caches 94
5.2.5 Optimal replacement policies for shared caches 96
5.3 A near-optimal replacement algorithm to minimize miss rate 97
5.3.1 NOPTb-miss design 97
5.3.2 NOPTb-miss example 99
5.3.3 Computational complexity 101
5.4 Methodology 102
5.4.1 The experimental framework 102
5.4.2 Configuration of the baseline system 103
5.4.3 Metrics 104
5.4.4 Other replacement policies 105
5.4.5 Reproducibility 106
5.5 NOPTb-miss evaluation 106
5.5.1 Convergence analysis 107
5.5.2 How close is NOPTb-miss to the optimum? 109
5.5.3 Results per application 110
5.6 A near-optimal replacement algorithm to maximize fairness 111
5.6.1 NOPTb-fair design 111
5.6.2 NOPTb-fair example 113
5.7 NOPTb-fair evaluation 113
5.7.1 Results per application 114
5.7.2 How close is NOPTb-fair to the optimum? 114
5.7.3 Comparison between NOPTb-miss and NOPTb-fair at workload level 115
5.8 Comparison with state-of-the-art policies 116
5.8.1 MPKI and throughput comparison with state-of-the-art policies 116
5.8.2 Fairness comparison with state-of-the-art policies 117
5.8.3 Balancing miss-rate reduction and fairness in SLLC replacement policies 118
5.9 Sensitivity analysis 119
5.9.1 Convergence analysis of NOPTb-miss 120
5.9.2 MPKI and throughput 120
5.9.3 Fairness 121
5.10 Conclusions 121

6 Conclusions....123
6.1 Contributions....123
6.2 Future work....125
7 Conclusiones....127
7.1 Contribuciones....127
7.2 Trabajo futuro....130
References....132

LIST OF FIGURES

Figure 1.1: growth in processor performance over 40 years. 2
Figure 1.2: growth in clock rate of microprocessors in Figure 1.1. 3
Figure 2.1: fraction of blocks evicted from the SLLC cache, in an example mix 13
Figure 2.2: placement of the Reuse Detector. 17
Figure 2.3: Reuse Detector operation. 18
Figure 2.4: detail of the algorithms in operation. Left: block eviction from L2 cache (non-shared block). Right: request from a core to its L2 cache. 19
Figure 2.5: example illustrating the operation of ReD. 21
Figure 2.6: fraction of blocks evicted from L2 caches, in an example mix, categorized from the ReD standpoint according to the type of reuse 24
Figure 2.7. Left: fraction of the SLLC occupied by blocks of each program in the example mix. 26
Figure 2.8: normalized hIPC (left) and MPI reduction (right) compared to the base system with 8 MB, for five systems 30
Figure 2.9: average fraction of alive blocks present at any given moment in the SLLC 33
Figure 2.10: distribution of normalized IPC, for all applications in all workloads. 34
Figure 2.11: normalized harmonic IPC, for all 100 workloads. 35
Figure 2.12: fraction of incoming blocks bypassed by ReD at the SLLC 35
Figure 2.13: normalized IPC obtained by ReD, for single-processor workloads 36
Figure 2.14: number of blocks selected for SLLC insertion after coming from main memory, for all applications in the example workload. 37
Figure 2.15: accuracy of content selection mechanisms 37
Figure 2.16: SLLC MPI reduction with respect to the base system 37
Figure 2.17: fraction of overall detector space occupied by each application 38
Figure 2.18: normalized harmonic IPC (left) and MPI reduction (right), for different SLLC data sizes 39
Figure 2.19: normalized hIPC (left) and MPI reduction (right) obtained when adding ReD c to base systems with 4-bit LRF and 2-bit TC-AGE. 40
Figure 2.20: normalized IPC (left) and normalized weighted speedup (right), for four proposals: ReD, CHAR, Evicted Address Filter and Reuse Cache. 40
Figure 2.21: normalized hIPC (left) and reduction of SLLC misses per instruction (right), as a function of the ReD capacity per core. 41
Figure 2.22: ReD size per core in KB, as a function of capacity and sector size. 42

Figure 2.23: normalized hIPC, as a function of the ReD capacity per core, and for different sector sizes. 42

Figure 2.24. Left: average rate of detection errors in ReD due to tag compression. Centre: normalized hIPC. Right: SLLC MPI reduction. 43

Figure 3.1 STT-RAM memory cell structure (left), and STT-RAM equivalent circuit (right). 50

Figure 3.2: request from a core to its L2 cache. 54

Figure 3.3. Block eviction from an L2 private cache. 55

Figure 3.4. Number of writes to the STT-RAM SLLC in the single-processor system. 61

Figure 3.5: performance (IPC) in the single-processor system. 62

Figure 3.6. Energy consumption in the STT-RAM SLLC in the single-processor system. 63

Figure 3.7: breakdown of energy consumption in the SLLC into the static and dynamic contributions in the single-processor system. 63

Figure 3.8: number of writes to the STT-RAM SLLC in the CMP system. 66

Figure 3.9: performance in the CMP system, measured with the system IPC 66

Figure 3.10: energy consumption in the STT-RAM SLLC in the CMP system. 68

Figure 3.11: breakdown of energy consumption in the SLLC into the static and dynamic contributions for the baseline in the CMP system. 68

Figure 3.12: energy consumption in DRAM in the CMP system. 69

Figure 3.13: number of STT-RAM SLLC hits per kiloinstruction in the CMP system. 69

Figure 4.1: state of ReD+ internal tables after two initial requests, and a first-reuse request. 78

Figure 4.2: entry of the Address Reuse Table without (a) and with (b) PC sampling, respectively. 79

Figure 4.3: performance results. Speedup vs LRU for ReD+ and SRRIP. From top to bottom: c1) single core without prefetching, c2) single core with data prefetching, c3) four cores without prefetching, and c4) four cores with data prefetching. 84

Figure 4.4: SLLC bypass rate with ReD+ using configuration *c1* (single core without prefetching). 85

Figure 4.5: CRC2 results. 87

Figure 5.1: an example of the OPTb algorithm naively applied to a shared cache. 95

Figure 5.2: schematic diagram of the simulation of iteration *i* of NOPTb-miss. 98

Figure 5.3: an example of how NOPTb-miss works with sequences. 100

Figure 5.4: mean SLLC miss rate for step 1 of NOPTb-miss (labelled as iteration 0) and several iterations of step 3. 107

Figure 5.5: maximum relative difference in miss rate in iteration 4 of NOPTb-miss when starting with MISSES, RANDOM and SRRIP as replacement policies.....108

Figure 5.6. Bottom: mean number of accesses to the SLLC per kilocycle (APKC) for the SPEC CPU applications. Top: MPKI at the SLLC for NOPTb-miss.....110

Figure 5.7: an example of the victim selection procedure of NOPTb-fair.....113

Figure 5.8. Bottom: mean number of accesses to the SLLC per kilocycle (APKC) for all applications in our workload set. Top: MPKI at the SLLC for NOPTb-fair for the same applications. 114

Figure 5.9: (a) difference in normalized SLLC MPKI reduction and (b) difference in normalized IPC between NOPTb-miss and NOPTb-fair for all workloads. 115

Figure 5.10: (a) MPKI reduction in the SLLC and (b) normalized IPC, relative to random replacement. 117

Figure 5.11: unfairness for various SLLC replacement policies. Lower values indicate greater fairness. 118

Figure 5.12: mean SLLC miss rate for step 1 of NOPTb-miss (called iteration 0) and several iterations of step 3, using an octa-core setup.....120

Figure 5.13: (a) MPKI reduction in the SLLC, and (b) normalized IPC, relative to random replacement, using an octa-core setup. 120

Figure 5.14: unfairness for various SLLC replacement policies, using an octa-core setup. 121

LIST OF TABLES

Table 2.1. L1, L2 and LLC: average MPKI at each cache level of the base system. IPC: average multiprocessor IPC. 27

Table 2.2: processor parameters 27

Table 2.3: memory hierarchy parameters 28

Table 2.4: ReD evaluation configuration 28

Table 2.5: classification of previous work based on the reuse locality property, according to our taxonomy 45

Table 3.1: area, latency and energy consumption for 22nm SRAM and STT-RAM caches with 1MB size. 51

Table 3.2: benchmark characterization according to the number of SLLC writes per kiloinstruction (WPKI). 58

Table 3.3: CPU and memory hierarchy specification. 59

Table 3.3: ReD evaluation configuration 59

Table 4.1: ReD+ hardware cost, per core 81

Table 4.2: memory hierarchy parameters 82

Table 5.1: memory hierarchy parameters 104

1 INTRODUCTION

This chapter presents the motivations for exploring replacement algorithms in shared last-level caches. It also outlines the key contributions, provides an overview of this dissertation, and describes the project framework that supported its development.

1.1 MOTIVATION

Over the past decades, computer performance has advanced exponentially, largely driven by Moore's Law (Moore, 1965) and Dennard Scaling (Dennard et al., 1974).

Initially, processor performance improvements were closely linked to increasing frequencies. However, at the start of the 21st century, the expected gains in computational power and energy efficiency from simple transistor scaling began to diminish. The physical limitations of packing more transistors into a confined space while maintaining high switching speeds made further frequency increases unfeasible. Once the limits of power dissipation were reached, thermal constraints became a major concern (see Figure 1.1 and Figure 1.2).

As a result, processor designers had to shift toward innovative solutions beyond simple transistor scaling. The industry prioritized multiprocessor architectures at lower frequencies to mitigate excessive power consumption and sustain performance improvements. Nowadays, chip multiprocessor (CMP) systems dominate the market in high-performance servers, desktop systems, embedded systems, and mobile devices (Baer, 2009).

The ever-increasing processing power of CMPs can only be fully utilized if data flows to and from Dynamic Random-Access Memory (DRAM) at a rate that matches the demand, ensuring both sufficient bandwidth and low latency. However, as they are manufactured using different processes, disparities have emerged in the evolution of their performance. To mitigate the latency gap, cache memories became a central component in computer architecture. These small, fast storage structures accelerate memory access by exploiting regularities in the access stream, such as temporal locality (recently accessed data tends to be accessed again), spatial locality (nearby data is likely to be accessed

soon), and reuse locality (data that is accessed multiple times is likely to be accessed again).

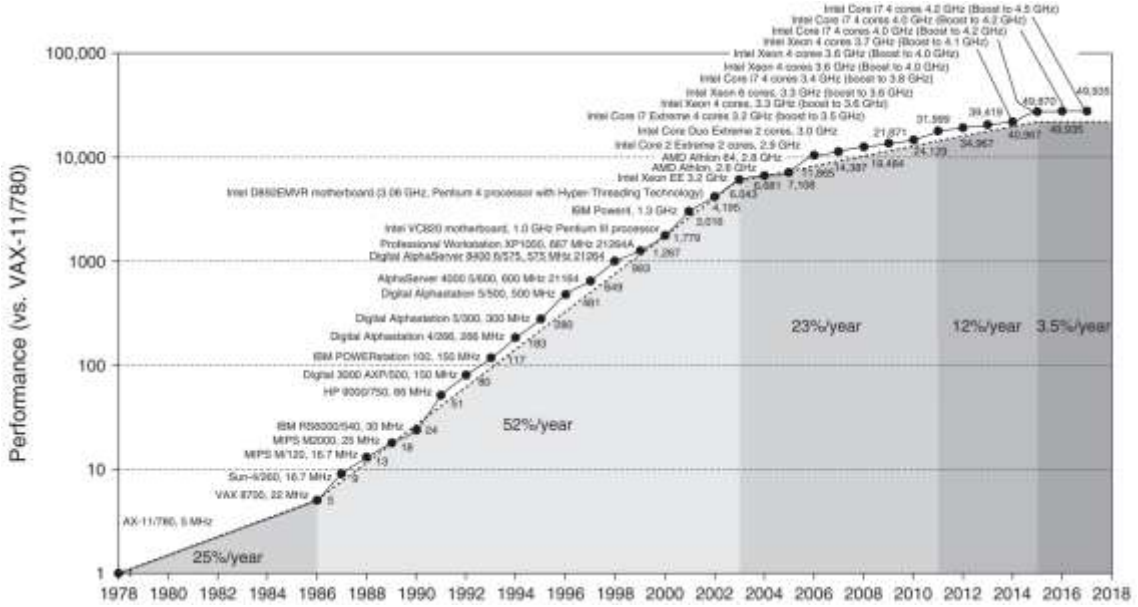

Figure 1.1: growth in processor performance over 40 years. This chart plots program performance relative to the VAX 11/780, as measured by the SPEC integer (intbase) benchmarks. Prior to the mid-1980s, growth in processor performance was largely technology-driven and averaged about 22% per year. The increase in growth to about 52% starting in 1986 is attributable to more advanced architectural ideas. In 2003 the limits of power due to the end of Dennard scaling and the available instruction-level parallelism slowed uniprocessor performance to 23% per year until 2011. From 2011 to 2015, the annual improvement was less than 12%. Since 2015, with the end of Moore's Law, improvement has been just 3.5% per year. Extracted from Hennessy & Patterson (2019).

In the mid-90s, differences between processor and DRAM speed where recognized as exponential and predicted to become the primary constraint on the rate of improvement in computer performance, a challenge referred to as the "memory wall" (Wulf & McKee, 1995). Addressing the memory wall challenge became a crucial objective for computer architects, driving innovations in memory technologies, cache structures, and management strategies to achieve more efficient and well-balanced computing systems. The industry adopted the multi-level memory hierarchy (Wilkes, 1965), which introduces several layers of cache between each processor and main memory (MM), as the leading design to alleviate the impact of the memory wall.

Nowadays, most CMPs include a multilevel memory hierarchy, ending with a last-level cache that is shared across several processing cores (LLC or SLLC) (Balasubramonian et al., 2011). Looking specifically at the current top 10 supercomputers (Strohmaier et al., 2024), all of them include general-purpose CMPs that have a multilevel cache and an

SLLC: AMD 4th Gen EPYC (Bhargava & Troester, 2024), AMD 3rd Gen EPYC (Evers et al., 2022), Intel 4th generation Xeon Scalable (Nassif et al., 2022), ARM A64FX (Yoshida, 2018), ARM Neoverse V2 (Bruce, 2023), and Intel 3rd generation Xeon Scalable (Papazian, 2020).

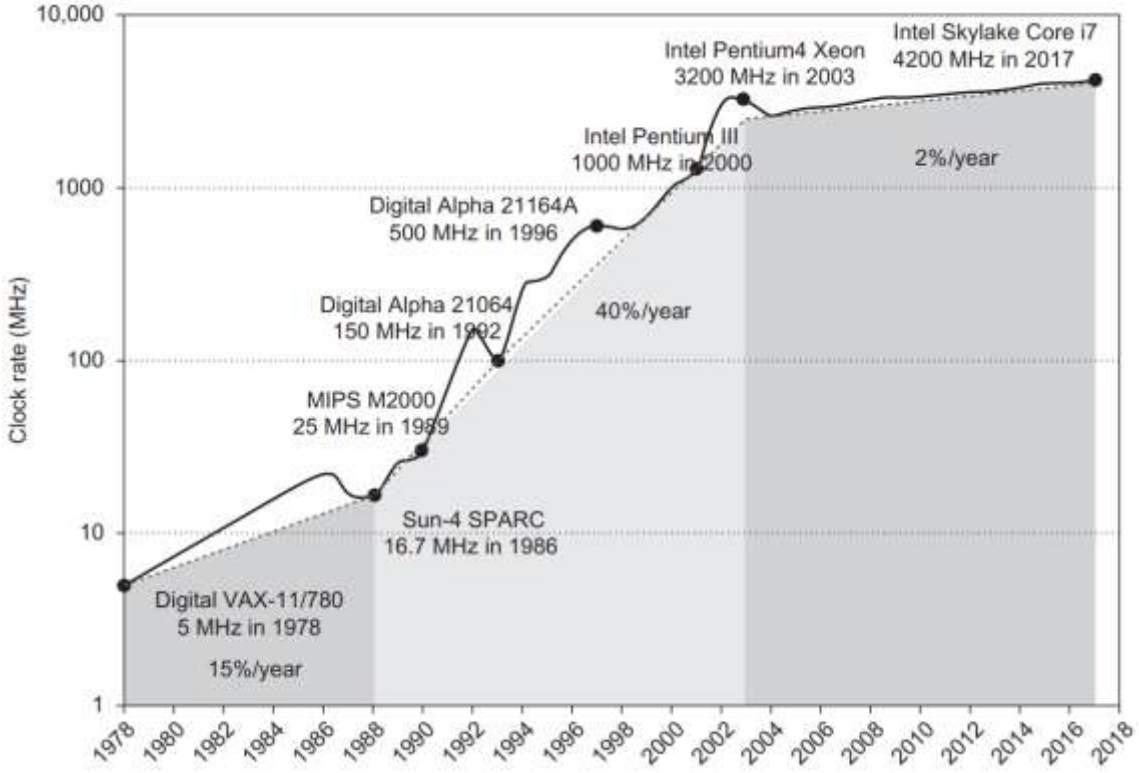

Figure 1.2: growth in clock rate of microprocessors in Figure 1.1. Between 1978 and 1986, the clock rate improved less than 15% per year while performance improved by 22% per year. Between 1986 and 2003, a period of 52% performance improvement per year, clock rates shot up almost 40% per year. Since then, the clock rate has been nearly flat, growing at less than 2% per year. Extracted from Hennessy & Patterson (2019).

The SLLC is critical in terms of cost and performance. In cost, because it occupies nearly 50% of the chip area. In performance, because it is the last resource before accessing DRAM, located outside the chip, which is much slower. Therefore, understanding its behavior and developing innovative mechanisms to enhance its efficiency are key objectives for modern computer architects.

The contents of a cache are managed by its own replacement policy, implemented in hardware (Smith, 1982). In its broadest sense the replacement policy is responsible for determining the identity of the incoming and outgoing cache blocks, to minimize the number of cache misses. In the event of a miss, the replacement algorithm first decides

whether the block that misses is loaded into the cache or bypassed (Mittal, 2016). Secondly, when a block is loaded, the replacement algorithm will choose a victim block to be evicted. In almost all replacement policies, the bypass decision and victim block selection are based on stay or replacement priorities for each block, which the algorithm manages to estimate future block usefulness.

Although replacement algorithms have been improving for more than 50 years, ongoing changes in capacity, system architecture (single-core, multi-core, multi-threaded) and memory technology (Static Random-Access Memory or SRAM, Embedded Dynamic Random-Access Memory or eDRAM, Spin-Transfer Torque Random-Access Memory or STT-RAM, etc.), among other factors, still encourage the development of optimizations or radically different proposals (Jain & Lin, 2019).

The replacement policy is important because it is largely responsible for the performance of the cache. In shared caches, which serve several competing cores or threads, it is also partially responsible for fairness and quality of service (QoS). The importance of the replacement policy is testified to by the organization of a cache replacement championship twice in recent years, driven by the community of computer architects (JILP, 2010; CRC2, 2017). These championships have been focused on policies for SLLCs.

1.2 SUMMARY OF CONTRIBUTIONS

This thesis aims to enhance the efficiency and performance of SLLCs by designing and studying innovative replacement policies. These policies are evaluated in a simulated environment, and their underlying mechanisms analyzed to understand how they achieve their objectives.

The contributions of this dissertation can be categorized as follows, with each category discussed in detail in a separate chapter:

Improving content selection for exclusive SLLCs. Previous publications reveal that the stream of references reaching the SLLC of a multiprocessor chip shows little temporal locality. However, it shows reuse locality, i.e., blocks referenced more than once are more likely to be referenced soon. This leads to an inefficient use of the cache using conventional management. There are several proposals addressing this problem for

inclusive caches; however, there are few that focus on exclusive caches (Jouppi & Wilton, 1994). In this regard, the specific contributions of this part are the following:

- An analysis of previous mechanisms that exploit reuse locality for an SLLC, highlighting three primary areas for improvement: low prediction accuracy, limitations in the size of reuse tracking mechanisms, and susceptibility to thrashing.
- A presentation of a new content selection mechanism, the Reuse Detector (ReD), specifically designed for exclusive caches to overcome the limitations of previous proposals. It utilizes an Address Reuse Table (ART) placed between each second-level cache (L2) and the SLLC to identify blocks evicted from L2 that have not experienced reuse, and prevent their insertion into the SLLC by bypassing them. Avoiding the insertion of many useless blocks allows the most reused ones to remain longer in the SLLC. This content selection mechanism is compatible with any promotion or victim selection policy.
- A comprehensive analysis and evaluation of the proposed mechanism, in terms of performance, cost and the trade-offs involved in its implementation. For a fair comparison, state-of-the-art solutions are included in the evaluation.

Improving the management of STT-RAM SLLCs[1]. Various limitations associated with SRAM technology are prompting the exploration of alternative memory technologies for the development of on-chip SLLCs. Currently, STT-RAM technology is emerging as a leading candidate, due to its superior energy efficiency, reduced die footprint, and enhanced scalability. Nevertheless, STT-RAM presents several challenges, including slow and energy-intensive write operations, which must be overcome to facilitate its integration into next-generation SLLC architectures. We address these challenges through improved management. The key contributions in this area are as follows:

- The development of a version of ReD aimed at minimizing both SLLC writes and invalidations, that is, adapted to avoid the negative aspects of STT-RAM

[1] This part of the work was originally presented by its first author, Roberto Rodríguez Rodríguez, in his own thesis. In this dissertation, the corresponding chapter highlights my specific contributions: the design of ReD, the modifications required to address the challenges posed by STT-RAMs, and the architectural differences with respect to the state-of-the-art competing proposal. A summary of Roberto's work is also included to provide appropriate context and show the outcomes.

SLLCs. In this version, the SLLC has to operate in exclusion with the private caches upon the block's initial request but transitions to an inclusive policy following a subsequent access resulting in an SLLC hit.

- An evaluation of the performance enhancements and energy savings facilitated by this modified ReD when implemented in a STT-RAM SLLC, across both single-processor and multiprocessor configurations.
- A comparative analysis of the design and performance relative to Dead Write Prediction Assisted STT-RAM Cache Architecture (DASCA) (Ahn et al., 2014), a state-of-the-art competing mechanism addressing similar challenges in STT-RAMs.

Leveraging the correlation between a block's reuse and the instruction issuing its first request to enhance content selection effectiveness. The specific contributions of this part are the following:

- A critical analysis of the shortcomings in the previous design of the Reuse Detector, identifying two key areas for enhancement: the occurrence of a compulsory miss on the second SLLC access of any block, and the suboptimal utilization of the ART in applications with a large working set.
- The introduction of an enhanced version of the Reuse Detector, named ReD+, which addresses the previously identified shortcomings while preserving the original benefits. This model incorporates a Reuse Table correlated with the Program Counter (PC), designed to identify which instructions initiate requests for blocks that are subsequently reused. Blocks fetched by these instructions are retained in the SLLC upon their initial request, rather than waiting for a confirmed reuse. Moreover, these blocks are not recorded in the Address Reuse Table, thereby enhancing the detection of the reuse of other blocks.
- Using content selection to enhance the performance of conventional non-inclusive SLLCs. The issues caused by the low temporal locality of the access stream in the SLLC extend beyond exclusive caches. Traditional non-inclusive caches also struggle with efficiency when managed conventionally. Adopting a selective block selection policy that leverages reuse locality could significantly improve efficiency across these cache configurations.

- This proposal, presented at the 2nd Cache Replacement Championship (CRC2), achieved third place in the overall ranking among 15 submissions.

Proposal and study of near-optimal replacement policies for SLLCs in multicore processors. Several decades ago, a replacement policy that minimizes miss rate, known as MIN (Belady, 1966) or OPT (Mattson et al., 1970), was proposed for private caches. This policy operates on the principle of knowing the future access sequence that the cache will receive, and must therefore be implemented offline. However, an equivalent policy for shared caches, which are accessed by several processors or cores, does not exist, because replacement decisions in shared environments modify these future sequences. We propose approaching the optimal execution and the optimal global access sequence iteratively. The specific contributions in this area are the following:

- We propose a near-optimal policy to minimize the miss rate for shared caches, named NOPTb-miss. It is an iterative algorithm that is applied to consecutive runs of the same workload. Our experiments show that, after several iterations, the miss rate for each workload converges to a minimum. To the best of our knowledge, this study is the first to investigate the theoretical minimum miss rate for shared caches and to offer a successful approach.
- Observing that an optimal shared cache that minimizes miss rate does not benefit all programs in the workload equally, we propose a near-optimal policy to maximize fairness among threads, named NOPTb-fair. Our experiments show that this algorithm achieves the highest level of fairness among all considered replacement policies.
- We compare the performance of our near-optimal offline proposals with current top-performing online policies, to gain insights into the current state of the art. We show that the best policies achieve around 65% of the MPKI reduction obtained by NOPTb-miss and 75% of the throughput improvement (vs. random). Comparing fairness, the best state-of-the-art policy, our proposed ReD+, achieves 60% of the improvement seen with NOPTB-fair, and the second-best only 45%.
- Additionally, by jointly analyzing performance and fairness, we show that designing a replacement policy for shared caches with the dual objective of

reducing miss rate and promoting inter-core fairness yields higher overall performance.

1.3 DISSERTATION OVERVIEW

This dissertation is organized as follows:

Chapter 2 presents the Reuse Detector (ReD), a novel mechanism for content selection in exclusive shared last-level caches. It describes its design and explores the trade-offs involved, gives insight into its operation and cost, evaluates its results, and compares them against other relevant proposals. It also includes a review of the state of the art in the matter.

Chapter 3 presents a summary of the application of the Reuse Detector mechanism to managing SLLCs based on STT-RAM technology. This chapter summarizes the necessary design modifications to the Reuse Detector to address the specific challenges of this case and presents a summary of its results, comparing them against a state-of-the-art proposal for this technology.

Chapter 4 presents ReD+, an evolution of the Reuse Detector that includes additional mechanisms to improve performance. It details the design modifications and their cost implications, and evaluates the results. Finally, it presents a summary of the outcomes of the 2nd Cache Replacement Championship (CRC2), where this proposal was submitted.

Chapter 5 explores the design of near-optimal replacement policies for SLLCs. It proposes one policy aimed at minimizing the miss rate and another focused on maximizing fairness between cores. It details their designs, presents their performance results, compares them with state-of-the-art proposals, and explains how they can provide insights into the design of future policies.

Chapter 6 presents the conclusions, summarizing the contributions of this dissertation and outlining potential directions for future work.

1.4 THESIS PROJECT FRAMEWORK

This thesis has been developed at the Grupo de Arquitectura de Computadores de la Universidad de Zaragoza (gaZ), which is part of the Departamento de Informática e Ingeniería de Sistemas (DIIS) and is supported by the Instituto de Investigación en Ingeniería de Aragón (I3A).

The thesis and its research work has been funded by PID2022-136454NB-C22 (“Arquitectura y programación de computadores escalables de alto rendimiento y bajo consumo III”), PID2023-146511NB-I00, PID2019-105660RB-C21, PID2019-107255GB-C22, TIN2016-76635-C2-1-R and TIN2015-65316-P from Agencia Estatal de Investigación (AEI) from Spain and European Regional Development Fund (ERDF); Consolider NoE TIN2014-52608-REDC and TIN2013-46957-C2-1-P from Spain; gaZ: T58_17R research group, gaZ: T58_20R research group, and gaZ: T58_23R research group from Aragon Government and European Social Fund (ESF); 2014-2020 "Construyendo Europa desde Aragón" from European Regional Development Fund; and HiPEAC Collaboration Grant, European Network of Excellence on High Performance and Embedded Architecture and Compilation from European Horizon 2020, H2020-EU.2.1.1. The funders had no role in the study, design, implementation, data collection, analysis, or preparation of the dissertation.

2 ReD: A REUSE DETECTOR FOR CONTENT SELECTION IN EXCLUSIVE SHARED LAST-LEVEL CACHES

The reference stream reaching a chip multiprocessor SLLC shows poor temporal locality, making conventional cache management policies inefficient. Few proposals address this problem for exclusive caches. In this chapter, we propose the Reuse Detector (ReD), a new content selection mechanism for exclusive hierarchies that leverages reuse locality at the SLLC, a property that states that blocks referenced more than once are more likely to be accessed in the near future. Placed between each L2 private cache and the SLLC, ReD prevents the insertion of blocks without reuse into the SLLC. It is designed to overcome problems affecting similar recent mechanisms (low accuracy, reduced visibility window and detector thrashing).

ReD improves performance over other state-of-the-art proposals: Cache Hierarchy-Aware Replacement (CHAR), Reuse Cache and Evicted Address Filter (EAF). Compared with the baseline system with no content selection, it reduces the SLLC misses per instruction (MPI) by 10.1% and increases harmonic instructions per cycle (hIPC) by 9.5%.

2.1 INTRODUCTION

Several studies show that conventional SLLC designs are inefficient because they waste a large portion of the cache. This is because they hold many dead blocks, i.e., blocks that are never re-accessed before their eviction. Frequently, those blocks are already dead when they enter the SLLC (Khan et al., 2010; Qureshi et al., 2007; Gaur et al., 2011). This occurs in multilevel hierarchies because private caches, often encompassing two levels (L1 and L2), exploit most of the temporal locality, which is effectively filtered out before reaching the SLLC (Jaleel et al., 2010a; Jaleel et al., 2010b). To address this drawback and increase the SLLC hit rate, several proposals suggest new SLLC insertion and replacement policies. Most of the work refers to inclusive or non-inclusive caches, and only a small group (Gaur et al., 2011; Chaudhuri et al., 2012) focuses on exclusive SLLCs (Jouppi & Wilton, 1994).

An exclusive SLLC acts as a victim cache of the private caches, storing their evicted blocks. Some recent AMD and Intel CMPs use exclusive or partially exclusive SLLCs (Conway et al., 2010; Clark, 2016; Kanter, 2017). The aggregate on-chip capacity of private caches increases with the number of cores, thus making exclusive hierarchies more appealing than inclusive ones. Over the next decades, we can expect many-core designs with more cores within the chip, and SLLCs not much larger than the current ones (Lotfi-Kamran et al., 2012). Therefore, using an inclusive cache will be even more inefficient and, unless there are drastic changes in the basic design of the memory hierarchy, the usefulness of exclusive SLLCs will grow in the future (Jaleel et al., 2015). This chapter focuses on enhancing the efficiency and performance of an exclusive SLLC in a chip multiprocessor.

The insertion policy of a replacement algorithm determines the position of incoming blocks in the replacement list. Several studies show the inefficiency of conventional insertions policies for SLLCs and propose inserting new blocks either with an intermediate priority (Jaleel et al., 2010b), or with the lowest priority within their cache set (Michaud, 2010; Qureshi et al., 2007). Alternatively, some incoming blocks could be selected to not be stored (be bypassed) in the SLLC, if their computed priority is lower than the minimum in the set (Gao & Wilkerson, 2010; Khan et al., 2010). Throughout this dissertation, we use the expressions “content selection policy” or “block selection policy” to refer to the part of the insertion policy that decides whether a new block has to be stored in the SLLC or bypassed. Exclusive caches offer the opportunity to implement a cache bypass mechanism with low complexity, in contrast to inclusive hierarchies. Bypassing a block evicted from a private cache means writing it directly into memory if dirty or discarding it if clean. Bypassed blocks do not affect the state of the SLLC.

Our proposal is a content selection mechanism that implements a new policy to select which blocks enter the SLLC and which ones bypass it. Specifically, we propose to take advantage of the reuse locality existing in the stream of requests to the SLLC. A block is said to have reuse locality if it has been referenced at least twice. A block with reuse locality is more likely to be accessed in the near future (Albericio et al., 2013a). Our mechanism prevents the insertion of many useless blocks in the SLLC. It is also an efficient solution to reduce traffic from private caches to the SLLC, which is one of the drawbacks of exclusive designs.

After an in-depth analysis of previous proposals that exploit reuse locality, we have identified three aspects where there is still room for improvement:

- Most of them predict reuse by linking it with a cache block feature, such as the instruction that brought the block into the SLLC or the memory area the block belongs to. The accuracy of these predictors is usually low.
- Most proposals detect reuse by keeping track of past accesses in a store embedded into the SLLC. In such proposals, the size of the SLLC restricts the number of detected blocks. They effectively lengthen the life of the blocks flagged as reused in the SLLC, but they are not able to detect further blocks.
- As far as we know, global thrashing may appear in all of them, since the reuse detection mechanism is shared among all the threads running on the CMP. A thread bringing too many blocks in the on-chip hierarchy can prematurely replace existing data from other applications, worsening their reuse detection.

The aim of our proposal is to fill up these gaps. To achieve this, we monitor blocks evicted from private L2 caches, by means of a specialized mechanism that remembers addresses of the recently evicted blocks. This mechanism, called hereafter the Reuse Detector or ReD, detects which blocks of those evicted from L2 do not have reuse, and avoids inserting them into the SLLC. Clean blocks are discarded, while dirty blocks are sent directly to main memory. ReD is a separate private hardware near each L2 cache, sized and organized regardless of the SLLC configuration.

We evaluate ReD using a set of multiprogrammed workloads running on a chip multiprocessor with eight cores and a three-level cache hierarchy. Results show that ReD enhances performance, above other recent proposals such as CHAR (Chaudhuri et al., 2012), Reuse Cache (Albericio et al., 2013b), and EAF (Seshadri et al., 2012).

The chapter is structured as follows. Section 2.2 explains the motivation. Section 2.3 describes ReD in detail. Section 2.4 gives insight into the ReD operation. Section 2.5 details the methodology used, including the experimental environment and the configuration of the simulated systems. Section 2.6 presents and analyzes results, and compares them against other relevant proposals. Section 2.7 explores the trade-offs in the design of ReD. Section 2.8 reviews the state of the art in the matter. Finally, in Section 2.9 we summarize our conclusions.

2.2 MOTIVATION

2.2.1 Problem analysis

Several studies have shown that, in a memory hierarchy, most of the blocks have already received all accesses when they are evicted from the caches close to the processor. Caches that are further away from the processor are used inefficiently because the stream of references that reaches them has very little temporal locality. Instead, these references show reuse locality. The reuse locality property has been empirically proved in several works (Gaur et al., 2011; Chaudhuri et al., 2012; Albericio et al., 2013a; Albericio et al., 2013b). It can be stated as follows: lines accessed at least twice tend to be reused many times in the near future.

We have conducted an experiment to quantify the number of blocks with reuse and the amount of reuse. Figure 2.1 plots a classification of the blocks evicted by an exclusive SLLC depending on the number of SLLC accesses that each block registered during their stay in the on-chip caches. Each block is classified according to whether it has received a single access (U), two accesses (R, reuse), or more than two accesses (M, multiple reuse). The average number of reuses for each M block is shown on top of the bars. The figure shows the distribution for eight applications running together in an example mix. The labels on the X axis indicate the names of the applications. Details about the baseline system are provided in Section 2.3.1, and the experimental setup is described in Section 2.5.

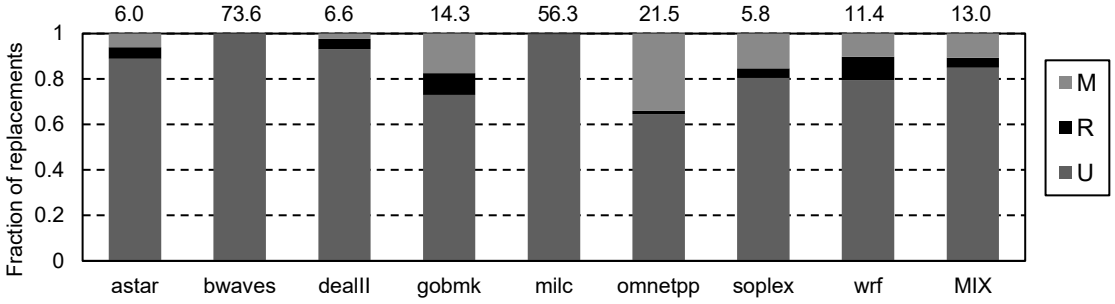

Figure 2.1: fraction of blocks evicted from the SLLC cache, in an example mix, according to the number of accesses received before its eviction: (U) one access, (R) two accesses, and (M) three or more accesses. The labels on the X axis indicate the name of the applications. Figures on top of each bar show the average number of reuses for an M block.

On average, 85% of the blocks do not receive any hit in the SLLC (U). These blocks could bypass the SLLC without loss of performance. Blocks with only one reuse (R) are

4% of the total, and those with more reuses (M) are 11%. For each block classified as M, there are 13.0 reuses on average. A content selection policy that only stored blocks with reuse (at least two accesses, R + M) would keep the small set of blocks that produces most hits. Furthermore, this policy would prevent the storage in the SLLC of the large set of U blocks, reducing the likelihood of M blocks being replaced. Our proposal is a mechanism that detects the second use of a block to classify it as reused, and only stores these reused blocks in the SLLC.

2.2.2 Detector design

We have analyzed previous mechanisms designed to classify blocks as reused for an SLLC and have identified three aspects where they could be improved:

Prediction accuracy. Most works predict reuse by linking it with some cache block feature, such as the instruction that brought the block into the SLLC or the memory area the block belongs to (Qureshi et al., 2007; Jaleel et al., 2010b; Wu et al., 2011; Gaur et al., 2011; Chaudhuri et al., 2012; Li et al., 2012). The accuracy of these predictors is limited. As an example, a mechanism that associates SLLC reuse with PC would only be accurate if most blocks brought by each PC present the same behavior, whether having reuse in the SLLC or not. However, accuracy would drop if some blocks brought by a PC have reuse in the SLLC, but others do not. Our proposal relies on detection instead of prediction of the reuse locality. Our mechanism keeps the addresses of all blocks evicted from the private caches during a certain time window (reuse detection window). The eviction from private caches of a block whose address is already stored is a true indicator of reuse locality, and thus the block is tagged as such. In Section 2.6 we provide results to test the accuracy of our proposal against CHAR, a state-of-the-art predictor (Chaudhuri et al., 2012).

Thread-global detector hardware. All the previous proposals have in common an important constraint: their classification hardware is shared among all threads running on the CMP. A thread delivering lots of misses will cause premature evictions of addresses previously inserted into the detector by other threads, restricting the detector's ability to discover more reuse for those other threads. In other words, a thread missing a lot in its private caches shrinks the reuse detection window of the remainder applications. In fact, these mechanisms reproduce in the detector the thrashing problem they try to avoid in the SLLC. To overcome this, we propose implementing reuse detectors that are private to

each processing core. Each detector is placed next to the private L2 cache, and remembers the addresses of all blocks evicted only by its associated L2 cache. In Section 2.6 we provide results to compare the amount of reuse detected by our proposal versus a global detector.

Size of the detector. A larger detector size allows it to remember more blocks for longer, thereby increasing the opportunity to classify more blocks as reused. Most previously proposed techniques track reuse patterns using the SLLC (Gaur et al., 2011; Chaudhuri et al., 2012; Albericio et al., 2013a; Gao & Wilkerson, 2010; Khan et al., 2012; Wu et al., 2011; Li et al., 2012). In these proposals, the SLLC size defines the size of the detector and, consequently, this detector is not able to discover more reuse than an LRU-managed SLLC of the same size would do (Mattson et al., 1970). In other words, blocks categorized as reused do not increase in number. In fact, increasing the lifespan of reused blocks indirectly shortens the life for those blocks that have not yet shown reuse, due to the capacity limit. This leads to a reduction in the detection window size relative to a cache with an LRU replacement policy. Our proposal aims to increase the number of blocks detected as reused. This requires a larger detection window. To achieve this, we include an additional store that is able to remember more block addresses than the SLLC can keep. In Section 2.6 we provide results that show the amount of reuse detected by our proposal with detection windows of different sizes.

ReD is an efficient content selection mechanism that detects with high accuracy when a block has been reused, and only stores these reused blocks in the SLLC. This removal of unused blocks enables a SLLC keeping more blocks with reuse and for longer time. Compared with previous proposals that also exploit reuse locality, ReD is more accurate detecting reused blocks, permits a greater visibility window, and does not suffer from global thrashing. In fact, among all those proposals, ReD is the only one that manages to have on average more alive than dead blocks in the SLLC. The use of a private and separate store makes the SLLC replacement policy not adversely affect the detector efficiency, so ReD can be implemented in an SLLC managed with any replacement policy. ReD is designed for exclusive SLLCs and chip multiprocessor systems, turning out to be a bypass mechanism that is simple and easy to implement.

2.3 DESIGN AND IMPLEMENTATION OF ReD

2.3.1 Baseline

The baseline system is a three-level cache hierarchy consisting of an SLLC whose contents are managed in exclusion with respect to the contents of two-level private caches, which are inclusive. Coherence is kept by means of a directory that holds, for each block in the hierarchy, both its status and precise location, which can be one or several private caches or the SLLC.

Blocks coming from main memory are sent directly to the requesting L2 cache. Eventually, when a block is evicted from L2 it is sent to the SLLC. From here on, either the block is requested again from any L2 cache, then being sent and invalidated in the SLLC, or it is replaced by another block that needs room for insertion. If a block placed in an L2 cache is requested by another L2 cache, the directory detects this situation, and the block is retrieved from the former to be delivered to the latter. Shared blocks are inserted in the SLLC only when the last copy is evicted from the L2 caches.

It is possible to implement ReD with any SLLC replacement policy. We select Trip Count and Age (TC-AGE) for our baseline design because this policy has proved to be very efficient in exclusive SLLCs (Gaur et al., 2011). It is equivalent to Static Re-Reference Interval Prediction (SRRIP) for inclusive caches (Jaleel et al., 2010b). It uses two bits to store the age of each cache line. The age is assigned when the block is inserted into the SLLC: if the block has previously received a hit in the SLLC, it is inserted with age 3, otherwise it is tagged with age 1. Each block in the private L2 cache stores one additional trip count bit to remember if it has had a hit in the SLLC (the TC bit). This bit is also sent to the SLLC with the block when it is evicted from the L2 cache. TC-AGE selects a random victim among those blocks in the younger group (age 0). If there is no block with age 0 in the cache set, the age of all blocks is decremented, and the victim selection restarts. In summary, TC-AGE assigns older age, and therefore less likelihood of replacement, to blocks that have been reused.

2.3.2 Adding the Reuse Detector

We propose placing our Reuse Detector next to every L2 cache, in the path from each L2 cache to the SLLC. ReD receives the addresses of every block evicted from the

corresponding L2 cache, see Figure 2.2. Being located outside the critical path from the SLLC to L2 caches, ReD does not affect the SLLC read latency. Instead, it slightly increases the time that a block evicted from an L2 cache takes to be sent to the SLLC or main memory.

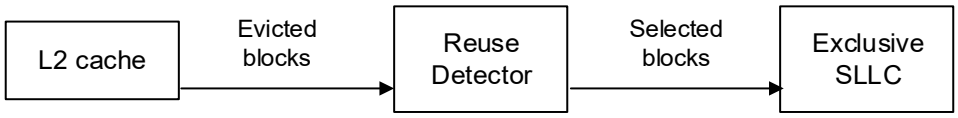

Figure 2.2: placement of the Reuse Detector. Every private L2 cache has one ReD.

When a block is evicted from the L2 cache, ReD decides between sending the block to the SLLC and bypassing it. The decision is driven by the block reuse history: if a block has a single use, it is bypassed. If it has one or more reuses, it is stored in the SLLC.

A block is classified as reused if it satisfies one of these conditions:

- The detector remembers the block address. ReD includes a buffer called the Address Reuse Table (ART), which stores addresses of blocks coming from L2 evictions. By checking if an address is present in the ART, ReD detects whether it is the first time the block is evicted from L2 or it has already experienced a previous eviction. A first eviction implies no reuse detected, whereas subsequent evictions indicate reuse.
- The block was supplied to the private cache either by the SLLC or by another private cache. We add a bit to each L2 cache block, called the Reuse bit, to record this condition.

The first condition is employed to detect the initial reuse of a block, whereas both conditions are active when identifying subsequent reuses. Thus, the primary objective of the ART is detecting the first reuse of a block. Secondarily, it detects subsequent reuses if the block has been evicted from both private caches and the SLLC.

The ART is structured as a set associative buffer, and its capacity, associativity, and replacement policy are design parameters. We define ReD capacity as the number of tracked evicted blocks times block size. For instance, a ReD able to track 1024 blocks of 64B has a capacity of 64KB. The ReD capacity is a metric that allows us to measure its tracking potential relative to the SLLC size. Capacity is a key parameter, since an effective

reuse detection requires storing a significant number of addresses between consecutive L2 evictions of a given block.

Neither the SLLC nor the directory require structural changes to be adapted to the new mechanism. To consider a possible bypass action, the coherence protocol and control logic will need to be adapted. In addition, our mechanism requires to add the Reuse bit to each L2 cache block.

2.3.3 ReD operation

Figure 2.3 shows a diagram illustrating the operation of ReD. On L2 evictions the Reuse bit is first checked. If the evicted block came from the SLLC or another private cache, it is stored again in the SLLC, without looking up the ART (1). Otherwise, if the evicted block came from main memory, its address is looked up in the ART (2). A miss means no reuse, so the block is bypassed, but the address is added to the ART (3). A hit means reuse, so the block is sent to the SLLC (4). Bypassed blocks send a control message to update the directory (6). Then, clean blocks are discarded, and dirty blocks are written to DRAM (7).

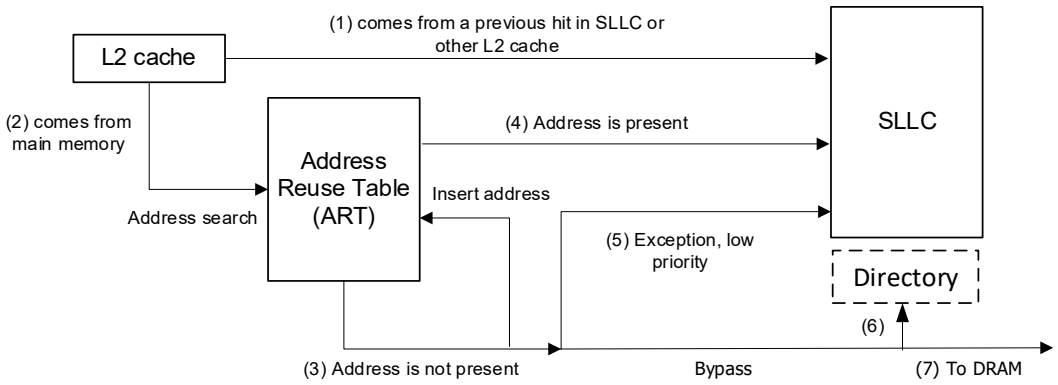

Figure 2.3: Reuse Detector operation.

As an exception, a small fraction of bypassed blocks is sent to the SLLC with “low insertion priority” (5). This means the SLLC will store them only if there are free ways in the corresponding set; moreover, those blocks will be inserted with the highest replacement priority. This exceptional filling policy comes from the observation that exclusive SLLCs experiencing at the same time many hits and bypasses may present many empty ways. Experimentation has shown us that diverting to the SLLC one of every 32 bypassed blocks takes advantage of the free space and increases performance.

Figure 2.4 shows two block diagrams with the algorithms in operation. On the left we show the steps followed when a block is evicted from an L2 cache, and on the right, those followed when a core requests a block, detailing the management of the Reuse bit.

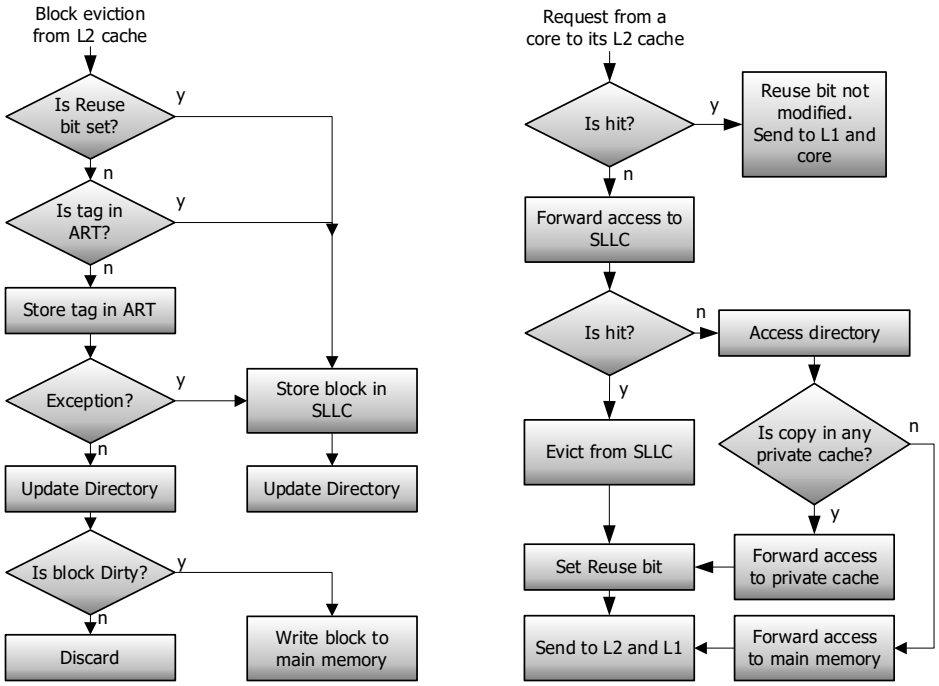

Figure 2.4: detail of the algorithms in operation. Left: block eviction from L2 cache (non-shared block). Right: request from a core to its L2 cache.

2.3.4 Example

To clarify the operation of ReD, this subsection presents a straightforward example illustrating how five memory blocks (A, B, C, D, and E) move through the various cache levels under a specific access pattern. In this example, we consider a multiprocessor system with two processing cores ($Core_0$ and $Core_1$) with private first-level caches ($L1_0$ and $L1_1$), an SLLC, and the corresponding ARTs between both cache levels. We assume a simplified configuration in which: (1) the L1 caches are direct-mapped, the ARTs are 2-way set-associative and the SLLC is 4-way set-associative; (2) all memory blocks map to the same L1 frame and to the same ART and SLLC set; and (3) all caches and ARTs are initially empty. Figure 2.5 details the access sequence in our example and shows the contents of the memory hierarchy after each access. Note that, for each block *X* in the private caches, we indicate the dirty and reuse bits in the form $X_{d,r}$, whereas for blocks in the SLLC, only the dirty bit is shown, as X_d.

The access sequence and the corresponding memory hierarchy behavior are as follows:

1. $Core_0$ *requests a word within block A for reading*. The access misses in $L1_0$ and is forwarded to the SLLC. Since the access also misses in the SLLC and block A is not present in any other private cache, the request is forwarded to DRAM. As shown in Figure 2.4, block A is then inserted into $L1_0$ with its reuse bit unset, and the requested word is delivered to $Core_0$.
2. $Core_1$ *requests a word within block B for reading*. The access misses in both $L1_1$ and the SLLC, and since the block is not present in any other private cache, the request is forwarded to DRAM. As shown in Figure 2.4, block B is inserted into $L1_1$ with its reuse bit unset, and the requested word is returned to $Core_1$.
3. $Core_1$ *requests a word within block C for writing*. The access misses in both $L1_1$ and the SLLC, and since the block is not present in any other private cache, the request is forwarded to DRAM. Block C is inserted into $L1_1$, and its dirty bit is set after the word is written. Block C replaces block B. Since block B's reuse bit was unset and its tag was not present in ART_1, the tag is stored in ART_1, as shown in Figure 2.4. Given that block B is clean and not reused, it is not inserted into the SLLC but just discarded.
4. $Core_1$ *requests a word within block B for reading*. The access misses in both $L1_1$ and the SLLC, and since the block is not present in any other private cache, the request is forwarded to DRAM. Block B is inserted into $L1_1$, replacing block C. Since block C's reuse bit was unset and its tag was not present in ART_1, the tag is stored in ART_1. Given that block C is dirty and not reused, it is not inserted into the SLLC but directly updated in DRAM (noted as C').
5. $Core_1$ *requests a word within block D for reading*. The access misses in both $L1_1$ and the SLLC, and since the block is not present in any other private cache, the request is forwarded to DRAM. Block D is inserted into $L1_1$, replacing block B. Since block B's reuse bit was unset but its tag was present in ART_1, block B is inserted into the SLLC.
6. $Core_1$ *requests a word within block B for reading*. The access misses in $L1_1$ and hits in the SLLC. Block B is evicted from the SLLC and inserted into $L1_1$ with its reuse bit set, as it comes from the SLLC. This insertion replaces block D. Since block D's reuse bit was unset and its tag was not present in ART_1, the tag is stored in ART_1 replacing

the tag of block B. Given that block D is clean and not reused, it is not inserted into the SLLC but discarded.

7. $Core_1$ *requests a word within block E for reading*. The access misses in both $L1_1$ and the SLLC, and since the block is not present in any other private cache, the request is forwarded to DRAM. Block E is inserted into $L1_1$, replacing block B. Since block B's reuse bit was set, it is inserted into the SLLC, as shown in Figure 2.4.
8. $Core_1$ *requests a word within block A for reading*. The access misses in both $L1_1$ and the SLLC. However, the coherency mechanism detects that the block is present in $L1_0$, so the request is forwarded there. As shown in Figure 2.4, the block is inserted into $L1_1$ and the reuse bit is set, as this access is recognized as a reuse.

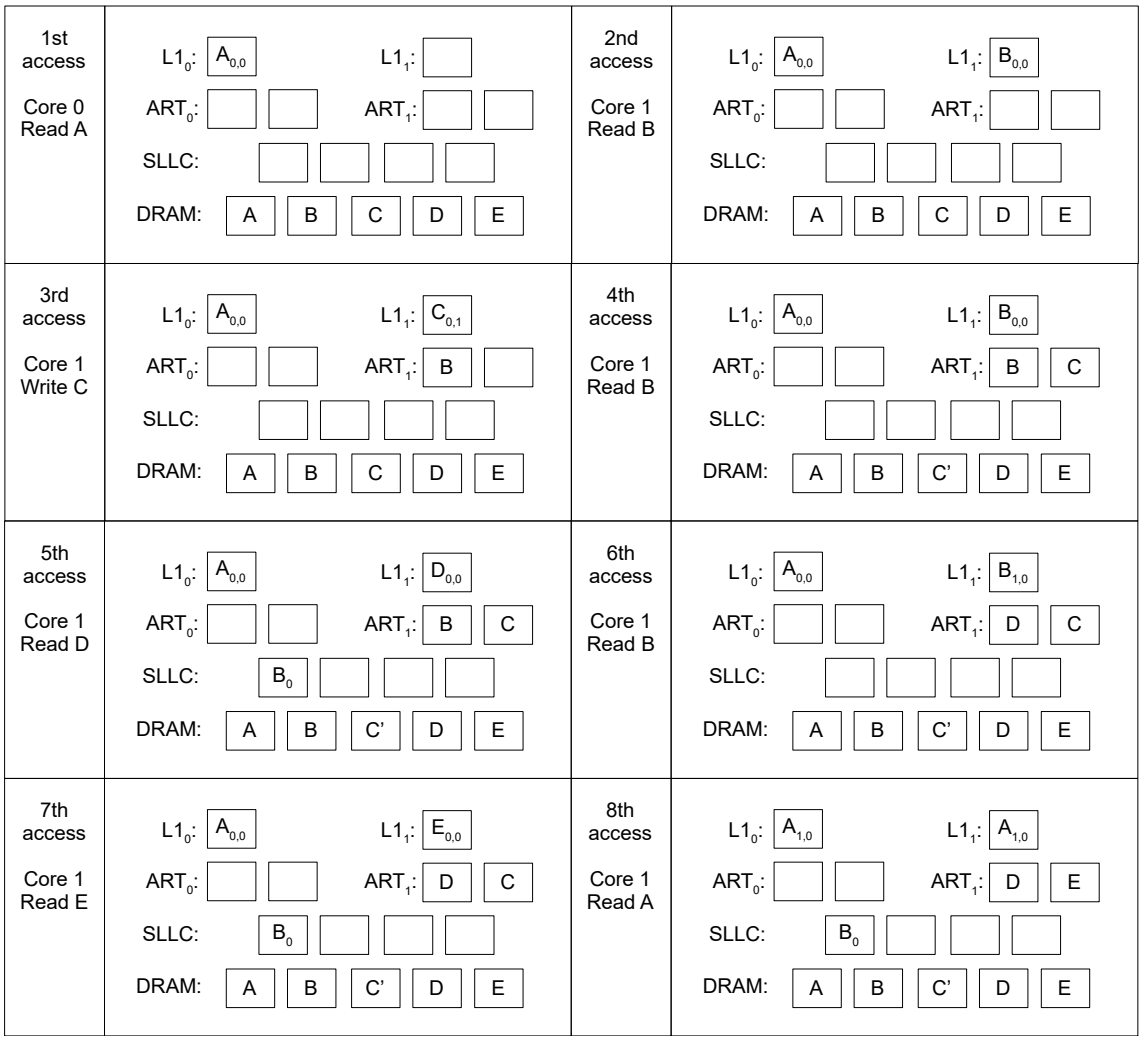

Figure 2.5: example illustrating the operation of ReD.

2.3.5 Implementation details

We implement the ART as a set associative cache, with entries containing tags for addresses, valid bits, and some bits for the replacement policy. We will use a 16-way ART; higher associativities lead to hardly any performance improvements.

Our experiments indicate that using a First-In First-Out (FIFO) replacement policy to manage the ART performs adequately. FIFO replacement means that the age of an address relates to its insertion (first use) and not to its last access. This is in line with the ART main goal of finding out the first reuse of a block.

When ReD is implemented on an SLLC with TC-AGE replacement, only one bit is needed in the L2 cache to map the Reuse bit and the TC bit. So, in this case, our mechanism does not have any overhead in the L2 caches. From the TC-AGE perspective, this bit maintains the same meaning: it remembers whether the block came from the SLLC or main memory. That is, when blocks are sent to the SLLC, it is reset when ReD discovers a first reuse (second time a block is evicted from L2); it is set in the subsequent L2 evictions (a block has already received at least three accesses). Therefore, TC-AGE is driven to give higher replacement priority in the SLLC to the blocks having one reuse, and less to those having multiple reuses.

2.3.6 Hardware costs

In this section, we calculate the total number of bits required to implement the ART of ReD, that we attach to each L2 cache.

The hardware cost of ReD depends primarily on its capacity. By increasing capacity, ReD can track blocks that have been evicted from the L2 cache longer ago, that is, it can detect more distant reuses. This is beneficial until it gets to a point where it detects more blocks with reuse than those the SLLC can effectively store, and performance starts to decline. The optimal performance for our baseline system with eight cores and an 8 MB SLLC (see details in Section 2.5.2) is obtained with a ReD capacity of 2 MB (see study in Section 2.7.1).

For a given capacity, the cost depends on how block addresses are stored in the ART. A naive implementation that stores all individual block addresses and includes the whole

tag would require a significant area. For example, for a ReD capacity of 2 MB, and assuming a physical address width of 40 bits, it would require 2K 16-way sets, with 24 bits (23 tag bits and 1 valid bit) per entry, and four FIFO bits per set. The total size for the eight cores would be 776 KB, a 9.5% of an 8 MB SLLC.

To reduce the ART area, we propose storing sector tags and compressing them. A sector is a set of consecutive blocks aligned to the sector size, a power of two. As our design requires per-block reuse tracking, every sector tag needs as many valid bits as the number of blocks a sector has. For example, a ReD sector size of four blocks, requires entries with four valid bits. As some blocks of a sector may not be referenced at all, the performance for a given ReD capacity decreases when the sector size increases. Therefore, the right sector size is a trade-off between area and performance.

Compression aims to shorten the tag size while maintaining good ability to distinguish between sectors. To compress we propose the following bit folding: let t and c be the number of bits of the entire and compressed tags, respectively. The t bits are split into consecutive pieces of size c, filling with zeroes if the last piece does not consist of c bits. Then, the compressed tag results from an XOR operation to all pieces. False positives may appear by using compression, as several sectors share the same compressed tag. Therefore, it might happen that blocks without reuse get inserted into the SLLC. Such wrong insertion is not a functional error, but can hurt performance. So, the right number of bits is also a trade-off between area and performance.

After trading off performance and cost (see details in Section 2.7), the chosen configuration has a capacity of 2MB, sector size of two blocks and 10-bit tags. This balanced configuration is the one used in our experiments unless stated otherwise. It requires 12 bits (10 tag bits and 2 valid bits) per entry, and four FIFO bits per set. The number of entries for each ReD is 16K, which means 24.5 KB per core. The total size for the eight cores is 196 KB, a 2.3% of an 8 MB SLLC. This is a 74.7% reduction compared to the initial size without cost optimizations.

The Reuse bits in the L2 caches do not require additional area if ReD is implemented on top of our baseline design, because they are the same TC bits used by TC-AGE. If ReD is implemented using an alternative replacement policy, 4 KB should be added (1 bit for each of the 4K entries in our eight 256KB L2 caches).

2.4 ReD OPERATION INSIGHT

In this section we use an example workload to analyze in depth the ReD operation, and how it is able to reduce the SLLC miss rate.

We plot how ReD classifies the blocks it receives, into five classes: first use (U), first reuse (R), multiple reuse detected only by ReD (MD)[2], multiple reuse detected only because the block comes from the SLLC or another private cache (MC), and multiple reuse detected by both mechanisms (MA). Figure 2.6 shows the distribution for the eight applications of an example mix.

A L2 eviction of a block classified as U causes an SLLC bypass, while an eviction of a block classified as any other class causes the insertion of the block into the SLLC. Evictions of blocks classified as U, R or MD denote that the block comes originally from main memory, while blocks classified as MC and MA denote a previous hit in the SLLC or in another private cache.

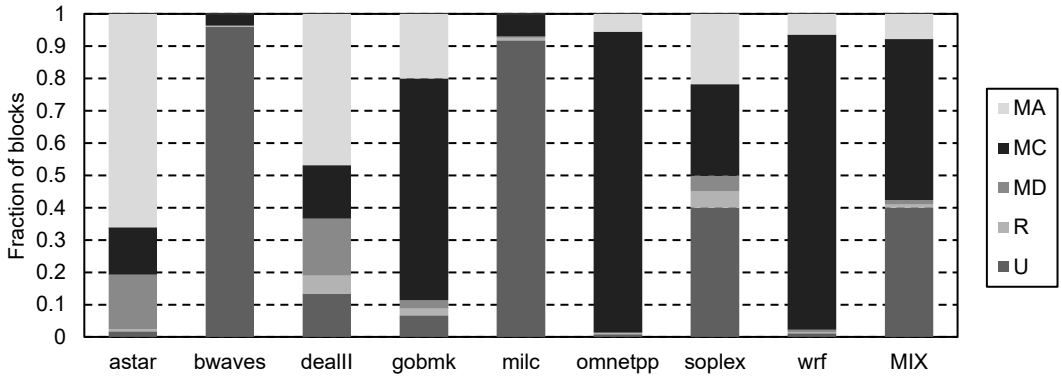

Figure 2.6: fraction of blocks evicted from L2 caches, in an example mix, categorized from the ReD standpoint according to the type of reuse in: (U) first use, (R) first reuse, (MD) multiple reuse detected only by ReD, (MC) multiple reuse detected only because the block comes from the SLLC or another private cache, and (MA) multiple reuse detected by any of them.

As shown, the amount of bypass varies from one program to another. In *bwaves* and *milc*, more than 91% of blocks evicted from L2 show a single use, and are bypassed. This is consistent with the measurements presented later in Table 2.1, which show that the

[2] Although the proposed ReD hardware cannot distinguish between the R and MD classes, they were separated in the figure to illustrate that both reuse detection mechanisms are complementary.

SLLC miss rates for these programs are very high. At the opposite extreme, in *astar*, *omnetpp* and *wrf* less than 2% of the blocks evicted from the private caches do not show reuse, so there is almost no bypass for these applications. The rest of the programs present intermediate figures: *dealII*, *gobmk* and *soplex*, with bypass levels of 13%, 7% and 40%, respectively.

The number of blocks that are sent to the SLLC after detecting their first reuse (R class) varies between 0.2% and 5.5% for *omnetpp* and *dealII*, respectively, with an average of 1.0%. These few blocks showing a first reuse are accessed later multiple times (MD, MC, and MA classes). On average, for each block classified as R (first reuse), ReD detects 61 additional reuses.

Blocks classified as MD have been previously detected by ReD, classified as reused and stored in the SLLC, but they have been prematurely evicted from there. As ReD has a detection window that is larger than the SLLC, it can detect this situation and re-insert them into the SLLC. This occurs on average 13% of the times a multiple reuse is detected.

The bypass of blocks without reuse in *bwaves*, *milc*, *dealII* and *soplex* allows the SLLC to better preserve the useful blocks from these programs, because they will not be evicted as often. Moreover, other programs of the mix will also benefit from this.

Figure 2.7 shows the average fraction of the SLLC occupied by each program, for the baseline system and for ReD (left). It also shows the SLLC misses per instruction (MPI) reduction of each application with respect to the baseline (right). *Bwaves* and *milc* take much less SLLC space with ReD. However, the block selection done by ReD does not harm their performance, since both maintain a miss rate similar to that of the baseline (0.9% worse in *bwaves*). For the rest of the programs, having more space in the SLLC and a better block selection mechanism allows them to keep more blocks with reuse and for a longer time, resulting in reductions in the SLLC MPI between 30.4% and 97.3% for *soplex* and *omnetpp*, respectively. For the whole mix[3], MPI is reduced by 24.0%, and the normalized hIPC is 1.28.

[3] If we order our 100 workloads, described in Section 2.5, from higher to lower normalized hIPC, this example mix is in position number 8.

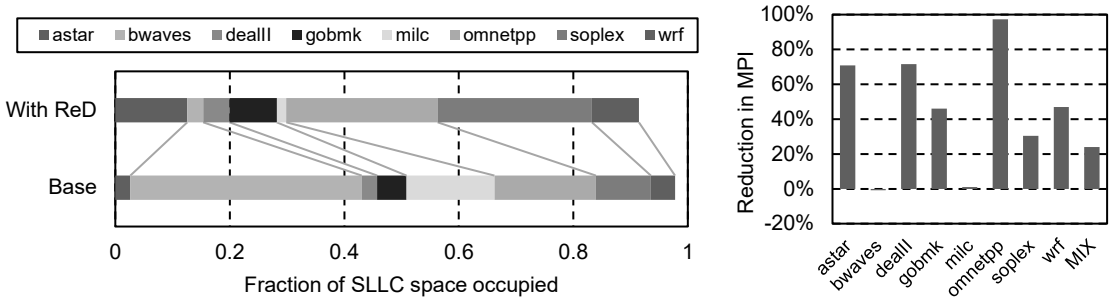

Figure 2.7. Left: fraction of the SLLC occupied by blocks of each program in the example mix. We take data every million cycles and show the average. Right: SLLC MPI reduction of ReD with respect to the base system.

2.5 EXPERIMENTAL SETUP

This section details the experimental framework and the configuration of the baseline system we use to evaluate the proposal.

2.5.1 The experimental framework

As a simulation engine we use the Simics full-system simulator (Magnusson et al., 2002), and the plugins Ruby and Opal from the GEMS Multifacet toolset (Martin et al., 2005) and DRAMSim2 from the University of Maryland College Park (Rosenfeld et al., 2011). Ruby is used to accurately model the memory hierarchy of the CMP system: caches, directory, coherence protocol, on-chip network, buffering, and blocking of components. Opal (also known as TFSim) is used to model in detail a superscalar out-of-order processor. DRAMSim2 is used to model a cycle-accurate DDR3 memory system.

Our Simics platform simulates SPARC cores managed by Solaris 10, and runs a multiprogrammed workload made of applications from the CPU 2006 suite of the Standard Performance Evaluation Corporation (SPEC) (Henning, 2006). For our system with 8 processors, we have generated a set of 100 mixes, composed by random combinations of 8 benchmarks each, taken from among all the 29 included in the SPEC CPU 2006 benchmark suite. Each program appears between 18 and 41 times, this representing an average of 27.6 times with a standard deviation of 6.1.

To identify initialization phases, we run until completion all the SPARC binaries, with the reference inputs, on a real machine. During the execution we use hardware counters to detect the end of the initialization phase of each benchmark. For every mix, we ensure

that no application was in its initialization phase by fast-forwarding the simulation until all the initialization phases are finished. Starting at this point, we first run 300 million cycles to warm up the memory system, and then collect data statistics for the next 700 million cycles.

The first three columns in Table 2.1 show the average number of misses per kilo-instruction (MPKI) in all three levels of the memory hierarchy. These figures are averages for each benchmark in all mixes in which appears, and when the eight benchmarks in each mix run together on the base system. The last column shows average multiprocessor instructions per cycle (IPC).

Benchmark	L1	L2	SLLC	IPC	Benchmark	L1	L2	SLLC	IPC
astar	7.5	1.1	0.7	1.17	libquantum	45.8	33.2	32.2	0.28
bwaves	24.5	21.1	20.1	0.66	mcf	64.9	36.0	18.9	0.18
bzip2	8.4	3.9	0.9	1.30	milc	24.6	23.5	22.0	0.23
cactusADM	20.8	11.4	4.9	0.64	namd	1.7	0.2	0.2	3.16
calculix	8.5	4.3	1.5	1.61	omnetpp	12.6	9.2	2.2	0.63
dealII	1.6	0.5	0.3	2.69	perlbench	10.2	1.8	0.8	1.37
gamess	6.7	1.0	0.6	2.60	povray	11.5	0.2	0.1	2.65
gcc	22	6.4	2.1	0.78	sjeng	6.9	0.8	0.5	1.28
gemsFDTD	42.7	29.7	22.8	0.45	soplex	8.9	7.1	3.1	0.63
gobmk	13.2	1.1	0.3	1.23	sphinx3	18.8	14.3	11.7	0.26
gromacs	11.7	3.0	1.2	1.60	tonto	6.7	1.3	0.5	2.18
hmmer	3.3	2.4	0.2	2.50	wrf	14.3	8.9	1.5	2.26
h264ref	4.2	1.4	0.7	1.36	xalancbmk	15.1	8.7	2.8	0.68
lbm	65.4	38.6	36.7	0.21	zeusmp	32.3	8.7	7.2	0.87
leslie3d	40.4	23.2	17.9	0.58					

Table 2.1. L1, L2 and SLLC: average MPKI at each cache level of the base system (exclusive SLLC with 8 MB and TC-AGE replacement policy). IPC: average multiprocessor IPC.

2.5.2 Configuration of the baseline system

We model a base system of eight superscalar processors with speculative out-of-order execution. Each processor has a 4-wide pipeline of 18 stages and 10 functional units. Branch prediction uses a Yet Another Global Scheme (YAGS) structure (Eden & Mudge, 1998) with a direction pattern history table (PHT) of 4K entries. Table 2.2 summarizes all the parameters of the simulated processor.

Base architecture	SPARC v9
Cores	8, 4-way superscalar, 4 GHz
Pipeline	18 stages: 4 fetch, 4 decode, 4 dispatch/read, 1 (>4) execute, 3 memory, 2 commit
ROB size	128 entries
Register Files	int: 160 (logical) + 128 (rename) FP: 64 (logical) + 128 (rename)
Functional units	4 int, 4 FP, 2 load/store
Branch prediction	YAGS cache structure with a PHT of 4,096 entries

Table 2.2: processor parameters

Each processor core has a two-level private cache hierarchy, being the exclusive third and last level cache shared among all the cores. The SLLC has a total size of 8MB, and is split into four banks that are cache line interleaved (64B).

A crossbar network connects the eight processors to the four SLLC banks. The Dual Data Rate version 3 (DDR3) memory system is accessed through two memory channels running at a frequency of 667 MHz (DDR3-1333). Table 2.3 shows all the details of the cache hierarchy we simulate.

Private cache L1 Instruction/Data (I/D)	32 KB, 4-way, LRU replacement, block size of 64 B, 3 cycles access latency
Private cache L2 unified	256 KB in inclusion with L1, 8-way, replacement LRU, block size of 64 B, 7 cycles access latency
Network	Crossbar, 80 bits bus width, 5 cycles latency
Shared cache L3 (SLLC)	8 MB exclusive (4 banks of 2 MB each), block interleaving, block size of 64 B. Each bank: 16-way, TC-AGE replacement with 2 bits, 10 cycles access latency, 32 demand miss status holding registers (MSHR)
DRAM	Device Micron 32M 8B x8, 2 channels, 2 ranks per channel, 8 devices per rank, 8 GB total.
DRAM bus	2 channels at 667 MHz, Double Data Rate (DDR3-1333), 8 B bus width, 4 DRAM cycles/line, 24 processor cycles/line

Table 2.3: memory hierarchy parameters

2.5.3 Configuration of the evaluated proposal

Table 2.4 summarizes the configuration of ReD used in the subsequent performance analysis. The reported values correspond to each individual ART. For more details, refer to Section 2.3.

ReD capacity	2MB
Associativity	16-way
Replacement	FIFO
Sector size	2 blocks
Tag size	10 bits
Number of sets	1,024
Number of entries	16,384

Table 2.4: ReD evaluation configuration

2.5.4 Performance metrics

Two performance metrics are mainly used: the harmonic mean of weighted IPCs (Luo et al., 2001; Eyerman & Eeckhout, 2014) normalized to that of the base system (normalized harmonic IPC or normalized hIPC) and the reduction in misses per instruction against the base system (MPI reduction). Unless stated otherwise, figures show the average of the results obtained for each of the 100 workloads.

For each mix, the normalized harmonic IPC for a proposal "PROP" is calculated as

$$normalized\ hIPC^{PROP} = \frac{H_t\left(\frac{IPC_t^{PROP\ MP}}{IPC_t^{BASE\ SP}}\right)}{H_t\left(\frac{IPC_t^{BASE\ MP}}{IPC_t^{BASE\ SP}}\right)} \quad (2.1)$$

where $IPC_t^{\mathrm{PROP}\ MP}$ is the IPC obtained using *PROP* for processor *t* when run in the multiprogrammed experiment, $IPC_t^{\mathrm{BASE}\ MP}$ is the IPC obtained using the base system for processor *t* when run in the multiprogrammed experiment, $IPC_t^{\mathrm{BASE}\ SP}$ is the IPC obtained using the base system for processor *t* when run alone on the system, and function *H* is the harmonic mean, defined as

$$H_t(x) = \frac{t}{\sum_t \frac{1}{x_t}} \quad (2.2)$$

The harmonic IPC metric is used because it incorporates a notion of fairness, in addition to performance. This is because the harmonic mean tends to be lower when there is much variance among the different weighted IPCs of each processor.

The reduction in misses per instruction is calculated as

$$1 - \frac{\sum_t M_t^{PROP}}{\sum_t M_t^{BASE}} \cdot \frac{\sum_t I_t^{BASE}}{\sum_t I_t^{PROP}} \quad (2.3)$$

where M_T is the number of SLLC misses counted during simulation for processor *t,* and I_T is the number of instructions executed by processor *t*.

2.6 PERFORMANCE ANALYSIS OF ReD

In this section we first present our performance results and compare ReD with state-of-the-art proposals. Next, we present data on the fraction of alive blocks in the SLLC achieved by each proposal. In Section 2.6.3 we analyze the IPC results of our proposal

broken down by application and mix. Next, we show the extent of content selection performed by ReD. In Section 2.6.5 we present results on single-processor workloads. Next, we analyze the efficiency of our detector and compare it to the other mechanisms. In Section 2.6.7 we analyze ReD performance using different SLLC sizes, and using an alternative replacement policy. Finally, we provide additional performance metrics.

2.6.1 Results and comparison with other proposals

Figure 2.8 plots normalized hIPC and MPI reduction against the baseline obtained by ReD. Compared to the baseline, it reduces MPI by 10.1% and increases harmonic IPC by 9.5%.

Hereunder we compare the performance of our mechanism with three other recent proposals, also shown in Figure 2.8: CHAR (Chaudhuri et al., 2012), Reuse Cache (Albericio et al., 2013b) and EAF (Seshadri et al., 2012). We also compare it with a base system with double the SLLC size, that is, 16 MB.

Comparison with CHAR. CHAR is a content selection proposal that bases the bypass decision on the access pattern that a block has at all levels of the memory hierarchy. CHAR was proposed both for inclusive and exclusive SLLCs. We use here the exclusive version.

ReD outperforms CHAR both in MPI reduction (10.1% vs. 4.3%) and normalized hIPC (1.095 vs. 1.070). CHAR uses a predictor to foresee which blocks will show or not reuse. In Section 2.6.6 we will show that the accuracy of predictor-based CHAR is lower than the accuracy of our detector-based ReD.

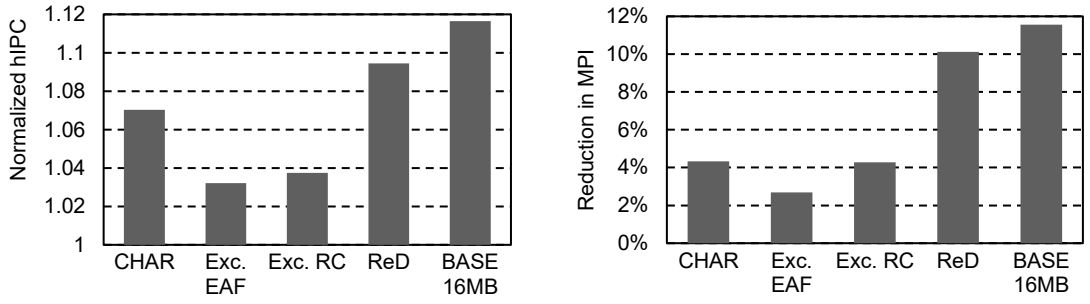

Figure 2.8: normalized hIPC (left) and MPI reduction (right) compared to the base system with 8 MB, for five systems: ReD (with the balanced configuration), CHAR, Evicted Address Filter, Reuse Cache (with RC-32/8 and NRR in tag array), and a base system with double the SLLC size (16 MB). All are implemented on an exclusive SLLC with TC-AGE in the data array.

Comparison with the Reuse Cache. The Reuse Cache is a content selection proposal for an SLLC whose tag and data arrays are decoupled, and that stores data only of those lines that have shown reuse. To be fair in the comparison, we have modelled a Reuse Cache in which the data array works in exclusion with the private L2 caches. Our exclusive Reuse Cache works as follows: each block in the L2 private caches includes a bit indicating whether it should be inserted into the SLLC when evicted from L2 (bypass / no bypass). On an SLLC miss (first block access), the block is sent from main memory to the L2 cache indicating “bypass”, and the tag is inserted into the SLLC tag array. This allows subsequent reuse to be detected. On a hit in the tag array of the SLLC that misses in the data array (second block access) the block is sent from main memory to L2 cache indicating “no bypass”. When the block is evicted again from L2, it is stored in the SLLC data array. In subsequent accesses, which hit both in tag and data arrays, the block is sent to the private L2 indicating “no bypass”, and is evicted from the SLLC data array. There are no changes in the SLLC tag and data arrays.

We use an exclusive Reuse Cache with a data array of 8MB and a tag array equivalent to 32MB. In our simulations, we found that this configuration offers the best performance among designs with 8MB of data. TC-AGE is used as replacement policy in the data array.

As shown in Figure 2.8, ReD outperforms the Exclusive Reuse Cache both in MPI reduction (10.1% vs. 4.5%) and normalized hIPC (1.095 vs. 1.038).

The Reuse Cache employs a global reuse detector, whose effectiveness is hindered by interference between the distinct applications. We analyze this effect in Section 2.6.6. In addition, the Reuse Cache embeds the reuse detector into the SLLC tag array. This increases the detector complexity, since each entry must keep the complete tag along with coherency information, limiting the design opportunities.

Comparison with the Evicted-Address Filter. The Evicted-Address Filter Cache is an SLLC that tracks the addresses of blocks that were recently evicted from the SLLC in a structure called the Evicted-Address Filter (EAF). Missed blocks whose addresses are present in the EAF are predicted to have high reuse, while the remaining blocks are predicted to have low reuse. This prediction determines their insertion priority: blocks predicted to have high reuse are inserted at the Most-Recently-Used (MRU) position, whereas those predicted to have low reuse follow a bimodal insertion policy — inserted at

MRU with probability 1/64, and at Least-Recently-Used (LRU) otherwise. The EAF is implemented using a Bloom filter (Bloom, 1970), which is cleared periodically.

Even though EAF is a replacement policy and not a content selection policy, we have included it in our comparison because it also attaches a reuse detection mechanism. The information ReD stores and acts upon is different, because it monitors blocks sent from L2 caches to the SLLC whereas EAF does it from the SLLC to main memory. EAF cannot be used to implement a content selection policy, because it uses a Bloom filter to store the reuse information. The filter is periodically cleared, which produces a loss of information that leads to temporarily classify all blocks as not reused. This is beneficial when the detector is used to adjust the replacement policy, as it is in the original publication and in our setup. Conversely, it makes it unsuitable for use as a content selection mechanism, as it would lead to not inserting any new blocks into the SLLC after the reset, for any application, until the filter is adequately refilled.

To be fair in the comparison, we have modelled an EAF Cache in which the data array works in exclusion with the private L2 caches. The L2 caches are also extended to store the Reuse bit, which is sent to the SLLC on eviction. At the time the block enters into the SLLC, that is, when it is evicted from an L2 cache, the Reuse bit is checked first. If it is set, the block is inserted at the MRU position. If not, the EAF is checked, applying the described policy. We store the SLLC MRU information using 2 bits per block, in line with the 2 bits that we use for TC-AGE in our other models. Our experimental results show that for the exclusive version a larger Bloom filter is required. We obtain the best results with a filter 25% larger than in the original publication. This is consistent with the increase in distinct blocks in the whole cache subsystem due to the move from inclusion to exclusion, from 8 to 10 MB (eight cores with 256 KB of L2 cache each).

As shown in Figure 2.8, ReD outperforms the exclusive EAF Cache both in MPI reduction (10.1% vs 2.7%) and normalized hIPC (1.095 vs. 1.032).

Comparison with a double-sized base system. Figure 2.8 plots normalized hIPC and MPI reduction against the baseline obtained by ReD with an 8 MB SLLC, and by a base system with a 16 MB SLLC (BASE 16MB).

ReD with an 8 MB SLLC achieves 87% of the MPI reduction (10.1% vs 11.6%) and 81% of the increase in normalized hIPC (1.095 vs 1.117) of the double-sized base system, with only a 2.3% increase in SLLC space.

2.6.2 Alive and dead blocks

In this section we present the average number of alive blocks that the SLLC stores at any given time. We define a block in the SLLC as alive at a given time if it receives a hit in the future before its eviction. Conversely, a block is defined as dead at a given time if it does not receive an additional hit before its eviction. Dead blocks waste storage.

Figure 2.9 plots these results for CHAR, exclusive EAF, exclusive Reuse Cache and ReD. Additionally, we include the baseline configuration (labelled TC-AGE), and Not Recently Filled (NRF) as the most basic replacement policy (NRF is analogous to Not Recently Used or NRU in inclusive caches). For each workload, we take measures every million cycles and calculate the average. We show the average over all our workloads.

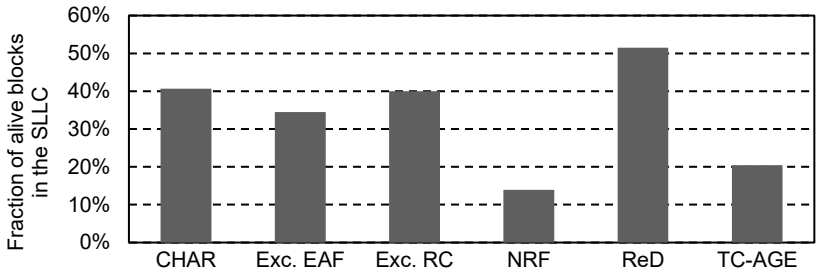

Figure 2.9: average fraction of alive blocks present at any given moment in the SLLC, for different cache management proposals, using an 8 MB SLLC.

As Figure 2.9 shows, when using the basic 1-bit NRF policy only 14.0% of the blocks are alive on average. Our baseline configuration (2-bit TC-AGE) increases that figure up to 20.4%. All other proposals, implemented on top of TC-AGE, improve the management of the SLLC, increasing the fraction of alive blocks. ReD achieves the best results with 51.5% of alive blocks, being the only proposal that manages to have on average more alive than dead blocks.

Comparing Figure 2.9 with previous Figure 2.8 (right), we realize that increasing fractions of alive blocks correlates to a higher reduction in misses per instruction, albeit it is not proportional. As ReD prioritizes multiple-reused blocks in the SLLC (see Section 2.3.5), it takes more advantage of the alive fraction, thus leading to a higher miss rate reduction.

2.6.3 Per-application and per-mix performance

As explained previously, application performance depends both on the application itself and on the other applications running in the workload. Figure 2.10 shows box-and-whisker plots with the distribution of speed-ups (normalized IPC) by application, with respect to the baseline system, for all instances of the applications that are running in our 100 workloads. Five values are plotted, namely minimum, first quartile, median, third quartile, and maximum.

Out of all 29 applications, 5 show improved performance in all workloads they appear in (*astar*, *bzip2*, *hmmer*, *tonto* and *xalancbmk*), with medians as high as 1.41 for *xalancbmk*. Another 11 show improved performance starting with the first quartile (*bwaves*, *gamess*, *gobmk*, *gromacs*, *h264ref*, *mcf*, *omnetpp*, *sjeng*, *soplex*, *sphinx3*, wrf*),* although in some mixes they show reduction. In 8 of them (*cactusADM*, *dealII*, *gcc*, *libquantum*, *milc*, *namd*, *perlbench*, *povray*), the median shows improvement but the first quartile shows reduction. The 5 remaining applications (*GemsFDTD*, *calculix*, *lbm*, *leslie3d*, and *zeusmp*) show less performance in the median.

Performance results also vary by workload, depending on the applications it includes. Figure 2.11 plots the normalized harmonic IPC for all the workloads, relative to the baseline. Out of the 100 mixes, 94 show speed-up improvements of up to 1.70, the worst having a value of 0.98.

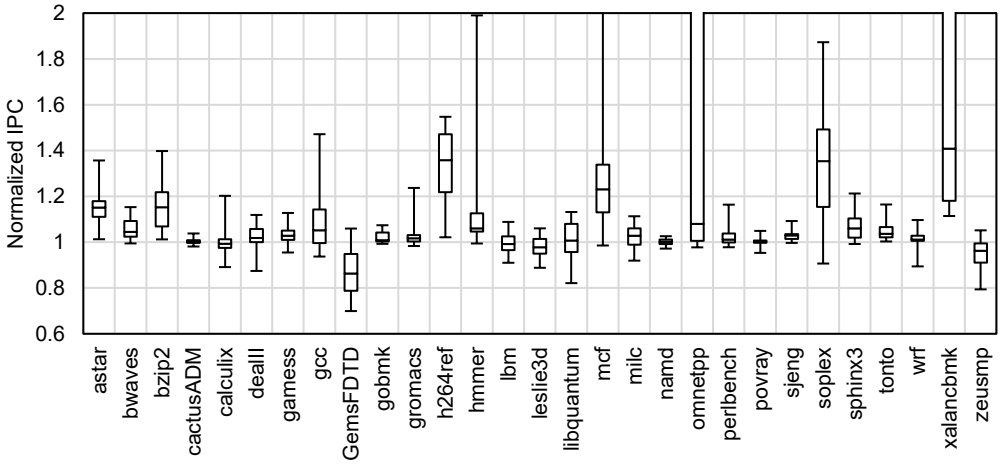

Figure 2.10: distribution of normalized IPC, compared to the base system, for all applications in all workloads.

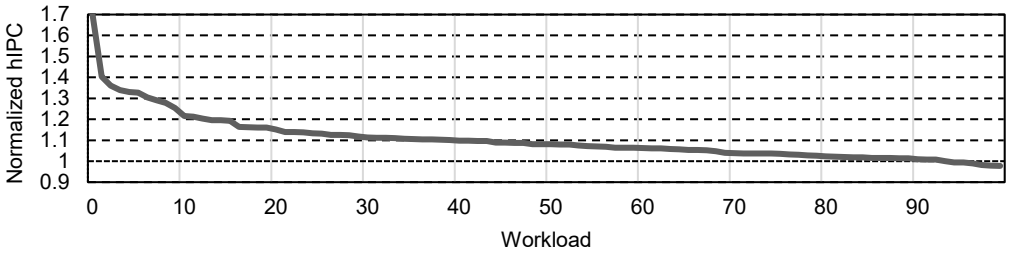

Figure 2.11: normalized harmonic IPC, compared to the base system, for all 100 workloads.

2.6.4 Content selection

In this section, we present the extent of content selection performed by ReD. Figure 2.12 plots, for each workload, the fraction of blocks evicted from the L2s that are bypassed. Workloads are ordered in descending normalized hIPC, following the same order as in Figure 2.11. The average bypass fraction is 0.49, the maximum is 0.85 and the minimum is 0.21.

As shown by the trend line, there is very low correlation between the bypass fraction and the performance gain of each mix. This is expected due to the heterogeneity in mix composition and the varying SLLC utilization patterns of the constituent programs.

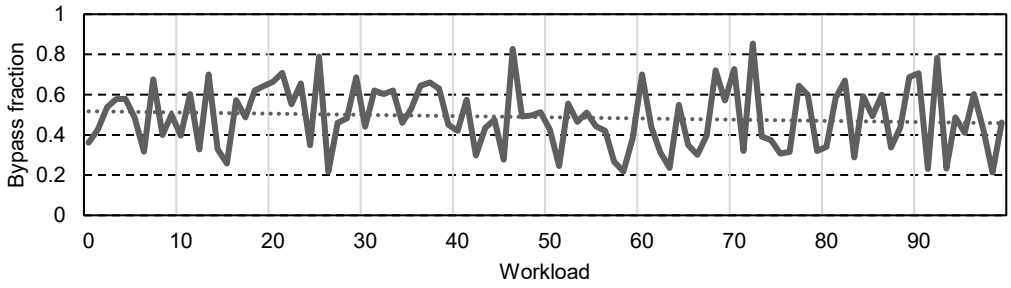

Figure 2.12: fraction of incoming blocks bypassed by ReD at the SLLC. Workloads are ordered in descending normalized harmonic IPC.

2.6.5 Single-processor performance

In this section we show the performance of ReD for single-processor workloads. For these experiments we use a 1 MB LLC, the same per-processor amount as in our multiprocessor simulations. All other parameters are the same. We show results for all benchmarks that have MPKI > 2 at the LLC.

Figure 2.13 plots normalized IPC obtained by ReD compared to the base system. Although ReD is specifically designed for chip multiprocessor systems, it still provides performance enhancements for 9 of these 14 sequential workloads, up to 12.8% speedup for *xalancbmk*. It decreases the performance of the other 5, up to 1.8% for *omnetpp*.

It is interesting to compare these results with those shown in Figure 2.10 for multi-processor workloads. O*mnetpp* shows there positive normalized IPC in most of the workloads it is present in (starting with the first quartile). As shown in Figure 2.7, the content selection made by ReD changes the distribution of space in the SLLC, and is often able to assign more space to the alive blocks of *omnetpp*, surpassing the 1 MB that we use in this section. With increasing space, the reused working set fits in the SLLC, and *omnetpp* turns to show IPC improvements in most multiprocessor benchmarks.

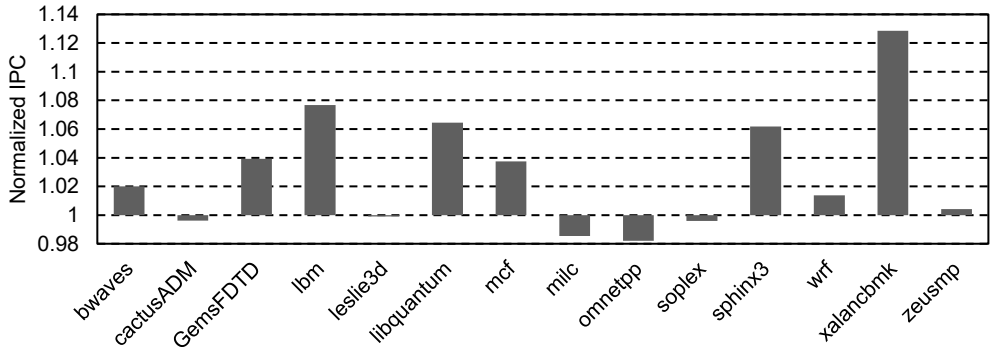

Figure 2.13: normalized IPC obtained by ReD, compared to the base system, for single-processor workloads that have LLC MPKI >2, on an exclusive LLC with 1 MB in the data array

2.6.6 Detector efficiency

In this section we analyze ReD efficiency in terms of the number of blocks selected for SLLC insertion, and their usefulness. We show figures about the number of blocks selected by the distinct detectors/predictors, and the subsequent reuse of these blocks.

Figure 2.14 plots the number of blocks in our example workload that, after coming from main memory to L2, are selected for SLLC insertion (“new blocks”). We show figures for ReD, CHAR, and a detector named “Shared ReD”. This detector is similar to ReD, but it uses a single address buffer that is shared among all cores instead of multiple private ones. Figure 2.15 shows the accuracy of each content selection mechanism. This is defined as the percentage of new blocks that are accessed at least once after being sent

to the SLLC. Figure 2.16 plots the MPI reduction of each mechanism versus our base system.

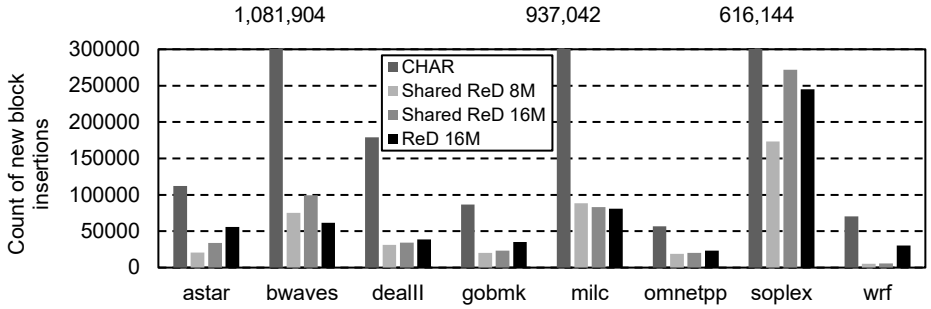

Figure 2.14: number of blocks selected for SLLC insertion after coming from main memory, for all applications in the example workload. From left to right: CHAR, shared ReD (8 MB and 16 MB) and ReD (16 MB, 2MB per core)

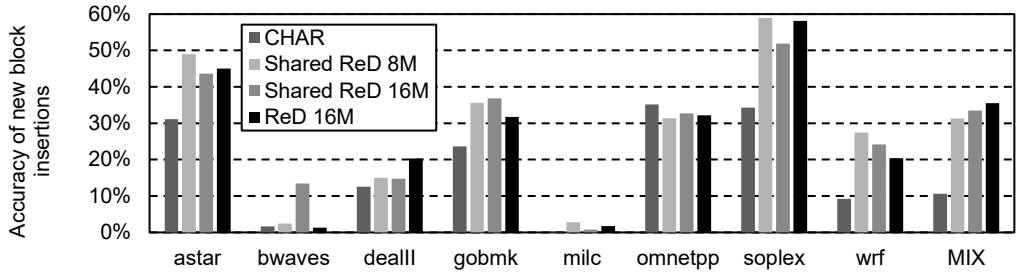

Figure 2.15: accuracy of content selection mechanisms: percentage of new blocks used at least once after being sent to the SLLC, in the example workload. From left to right: CHAR, shared ReD (8MB and 16 MB) and ReD (16 MB, 2MB per core)

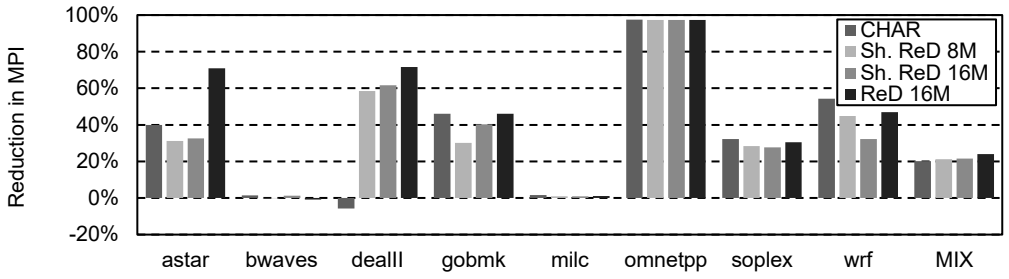

Figure 2.16: SLLC MPI reduction with respect to the base system. From left to right: CHAR, shared ReD (8MB and 16 MB) and ReD (16 MB, 2MB per core)

As shown in Figure 2.14 and Figure 2.15, predictor-based CHAR is less selective than our detector-based ReD. CHAR inserts many more blocks into the SLLC but its accuracy is low. For example, for *bwaves* CHAR inserts into the SLLC about 18 times the blocks inserted by ReD. However, both have similar SLLC miss ratio because only 2% of the blocks inserted by CHAR are used before being evicted. The behavior is similar for *milc* (12x and 0.1%). To store blocks for these two applications, CHAR evicts useful blocks from

other cores. Consequently, ReD is much better than CHAR at reducing MPI for two other applications of this mix (*astar* and *dealII*). For the whole mix the MPI reduction for CHAR is 20% vs 24% for ReD 16M (Figure 2.16). These differences, or even larger ones, appear consistently across our workloads, leading to the average 6% difference shown in Figure 2.8.

Shared ReD with a capacity of 16M overall inserts 32% more blocks than Shared ReD with 8 MB. Additionally, the accuracy increases by 2%. This does not directly translate into a much higher MPI reduction in this particular workload (only 0.3%), but on average (across our 100 workloads) it leads to a 1% reduction in MPI.

ReD 16M outperforms Shared-ReD 16M in 5 applications of the mix, and obtains similar performance in *bwaves*, *milc* and *omnetpp*. Figure 2.17 plots the average fraction of the ART occupied by each program using these two mechanisms. Only two applications, *bwaves* and *milc*, occupy 76% of the shared detector, stealing capacity from the other six applications. The private detector in ReD *protects* all applications from this thrashing, which leads to a fair distribution of the detection window, and ultimately better performance of the workload.

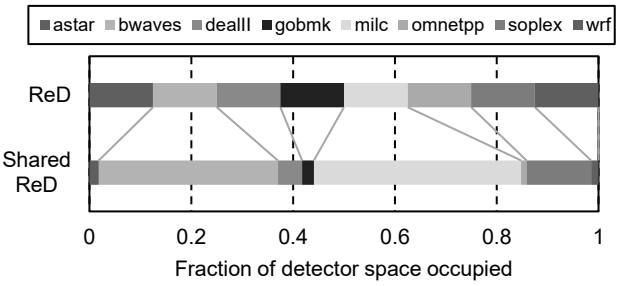

Figure 2.17: fraction of overall detector space occupied by each application on the example mix, for ReD (16 MB, 2MB per core) and Shared ReD 16MB.

2.6.7 Additional cache sizes

In this section we show the performance of ReD using distinct SLLC cache sizes, and compare it with the selected three proposals.

Figure 2.18 plots normalized hIPC and MPI reduction against the base system obtained by CHAR, exclusive EAF, exclusive Reuse Cache and ReD for varying SLLC sizes. ReD outperforms all other proposals in both metrics across all SLLC sizes considered.

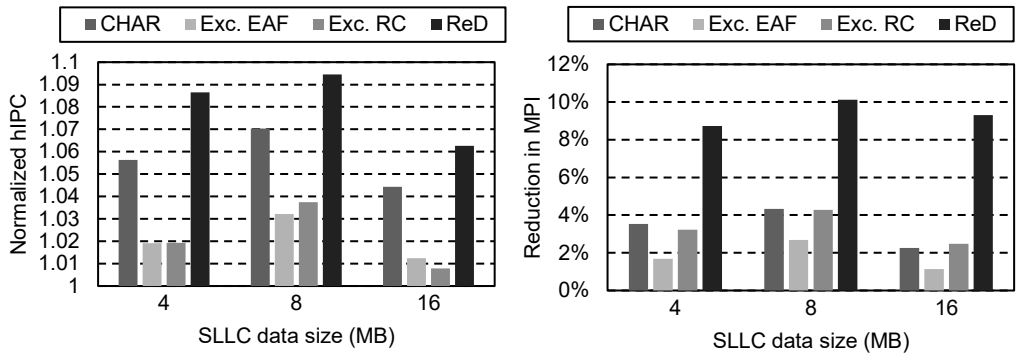

Figure 2.18: normalized harmonic IPC (left) and MPI reduction (right) compared to the base system, for different SLLC data sizes, for four systems: ReD (with a balanced configuration for each size), CHAR, Evicted Address Filter Cache and Reuse Cache (RC-16/4, RC-32/8 and RC-64/16). All are implemented on an exclusive SLLC with TC-AGE in the data array.

2.6.8 Alternative cache replacement policy: Least Recently Filled (LRF)

Content selection and replacement policies are usually aligned, as they share the same objectives. Therefore, we use TC-AGE as replacement policy: ReD selects reused blocks to be stored in the SLLC, and TC-AGE aims to retain the most reused blocks as long as possible. However, content selection and replacement policies are orthogonal. The former chooses which blocks enter the SLLC and the latter which blocks are evicted to make room for others. Therefore, ReD can be implemented on an SLLC managed with any replacement policy.

As ReD has the detector store decoupled from the cache, the replacement policy cannot adversely affect the detector's efficiency. In other mechanisms where the SLLC itself serves as detector store, there is a clear dependence between detection and replacement. A poor replacement algorithm can adversely affect the detector.

Next, we analyze the impact of our proposal on an exclusive SLLC with a 4-bit Least-Recently-Filled (LRF) replacement policy, similar to LRU in inclusive caches. Figure 2.19 plots normalized hIPC and MPI reduction obtained by ReD when using LRF and TC-AGE, compared to a base system with the same replacement policy but without ReD. Adding ReD leads to better MPI reductions and higher normalized hIPC with LRF than with TC-AGE. This is not a surprising result, as the former is not as efficient as the latter and leaves more room for improvement.

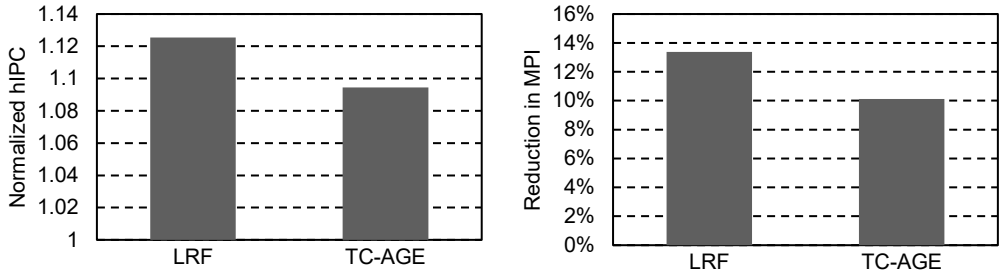

Figure 2.19: normalized hIPC (left) and MPI reduction (right) obtained when adding the ReD content selection mechanism to base systems with 4-bit LRF (Least-Recently-Filled) and 2-bit TC-AGE as replacement policies. Both are implemented on an exclusive SLLC with 8 MB in the data array.

2.6.9 Additional performance metrics

In this section we show our results using two alternative performance metrics. Figure 2.20 plots normalized IPC (speedup) and normalized weighted speedup (WS) (Snavely & Tullsen, 2000) for CHAR, exclusive EAF, exclusive Reuse Cache and ReD, using the same configurations as in Section 2.6.1.

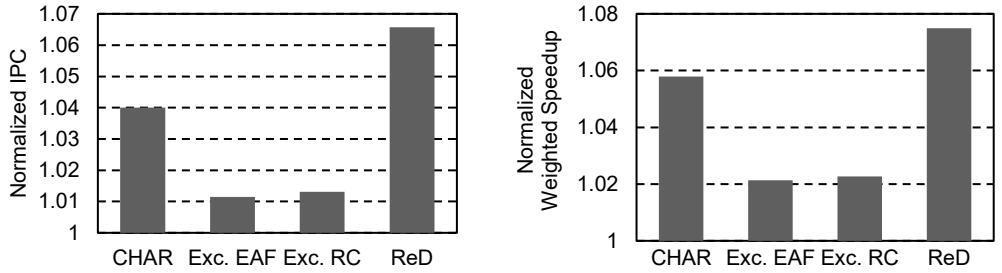

Figure 2.20: normalized IPC (left) and normalized weighted speedup (right) compared to the base system with 8 MB, for four proposals: ReD, CHAR, Evicted Address Filter and Reuse Cache. All are implemented on an exclusive SLLC with TC-AGE in the data array.

The results exhibit the same trends observed in Figure 2.8 (left), based on normalized harmonic IPC.

2.7 DESIGN SPACE EXPLORATION

The following three subsections evaluate the performance-cost trade-offs of ReD capacity, sector size and tag compression, searching for a balanced configuration.

2.7.1 ReD capacity

Here we study how the results vary depending on the ReD capacity (see Section 2.3.2). Figure 2.21 shows normalized hIPC and reduction in MPI, with respect to the baseline, as a function of the ReD capacity per core. For this experiment, the ReD sector size is one block and it stores the entire tag. We show average values across the 100 mixes described in Section 2.5.

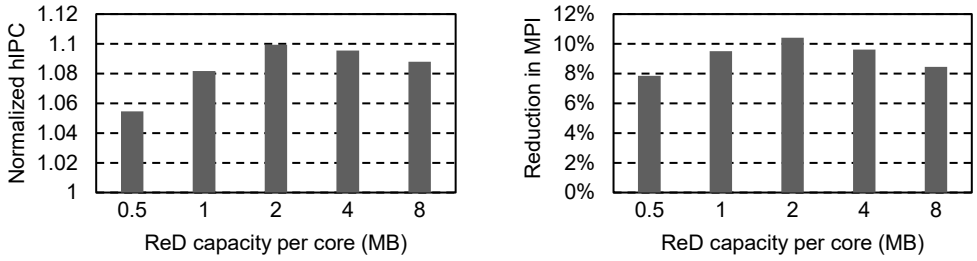

Figure 2.21: normalized hIPC (left) and reduction of SLLC misses per instruction (right) with respect to the base system, as a function of the ReD capacity per core.

By increasing capacity, ReD can track blocks that have been evicted from the L2 cache longer ago, that is, it can detect more distant reuses. The optimal configuration is achieved with a capacity of 2 MB per core, which presents a hIPC increase of 9.9%, and reduces MPI by 10.4%. A ReD with capacity larger than 2 MB detects more blocks with reuse than those the SLLC can effectively store, leading to a performance decrease compared to the 2 MB ReD.

2.7.2 ReD sector size

Increasing sector size decreases the ReD hardware cost (see Section 2.3.6). We call ReD size the hardware cost of a given configuration measured in bytes. Figure 2.22 shows ReD size as a function of capacity and sector size, when using tags with 10 bits.

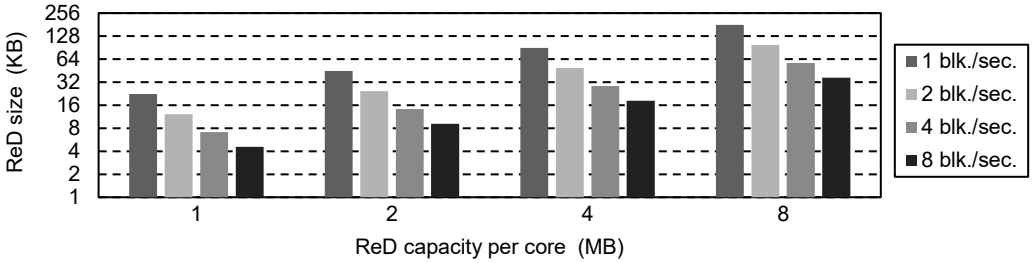

Figure 2.22: ReD size per core in KB, as a function of capacity and sector size. The sector size is the number of blocks associated with each ReD tag. We consider tags of 10 bits.

For a given ReD capacity, doubling the sector size allows to halve the number of ART sets. Therefore, a ReD with bigger sector size requires less ReD size. This is because the storage saved by reducing the number of entries is greater than the storage needed to add valid bits for each block of a sector.

On the other hand, Figure 2.23 shows how performance varies when the ReD sector size increases.

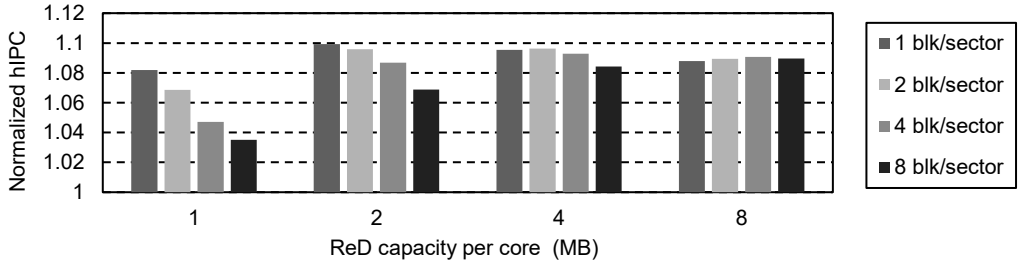

Figure 2.23: normalized hIPC with respect to the base system, as a function of the ReD capacity per core, and for different sector sizes.

We have defined the reuse detection window as the set of block addresses that ReD remembers of an executing thread (ReD window for short). Note that it could be different from the ReD capacity because some threads may not use the full capacity of the detector. If we increase the sector size while maintaining the ReD capacity, the ReD window decreases because sometimes the thread will not reference all the blocks of a sector. Therefore, ReD will detect less reuse, leading to a performance degradation for all the ReD capacities except for 8 MB, where ReD already detected more blocks with reuse than those the SLLC is able to store.

The best configuration in terms of performance, with 2MB of capacity and sectors with one block, requires a ReD size of 45 KB per core. However, other configurations have better performance/cost ratios: the one with 2 MB of capacity and sectors with 2 blocks shows 0.35% lower hIPC and a ReD size of 24.5 KB, 46% lower.

2.7.3 ReD tag size

As explained in Section 2.3.6, the ART can store compressed tags to reduce the amount of storage required. Figure 2.24 shows on the left, depending on the tag size, the average error rate in reuse detection due to tag compression. These errors are false positives: a false reuse is detected because the compressed tag that is being searched matches with that of a different sector previously registered. Inserting not-reused blocks into the SLLC reduces the effectiveness of the mechanism. The error rate is less than 1% with a tag of only 12 bits.

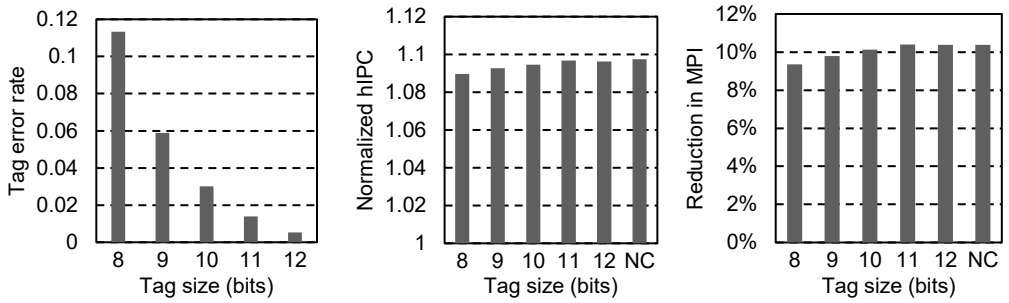

Figure 2.24. Left: average rate of detection errors in ReD due to tag compression. Centre: normalized hIPC with respect to the base system. Right: SLLC MPI reduction with respect to the base system. The NC bar represents the tag with no compression.

Figure 2.24 shows also, in the centre and on the right, normalized hIPC and MPI reduction obtained for our selected configuration (2MB of capacity and a sector of 2 blocks) as a function of tag size. The performance loss is almost negligible for a tag size of 10 bits: normalized hIPC decreases 0.29% while MPI increases 0.26% compared to the configuration with uncompressed tags. This justifies compression in order to reduce the amount of storage required.

This is the balanced configuration we have selected: 2 MB ReD capacity, sector size of two blocks and 10-bit tags. It has a ReD size of 24.5 KB.

2.8 RELATED WORK

Any cache content management mechanism is based on a model that forecasts whether a block will be used in the immediate future. State-of-the-art mechanisms for the SLLC can be broadly classified into two groups: those that rely on the last touch to each block — such as PC-based approaches by Khan et al. (2010), PC-sequence-based methods by Lai et al. (2001), and counter-based strategies by Kharbutli & Solihin (2008) — and those that exploit reuse locality, including techniques proposed by Qureshi et al. (2007), Gaur et al. (2011), Jaleel et al. (2010b), Chaudhuri et al. (2012), Albericio et al. (2013a, 2013b), Gao & Wilkerson (2010), Khan et al. (2012), Wu et al. (2011), Li et al. (2012), Seshadri et al. (2012), and Gupta et al. (2013).

In a similar way, Faldu & Grot (2016) classify management strategies into Dead Block Predictors (DBPs) and insertion policies. DBPs try to predict whether a block has reached the end of its useful lifetime on chip. Insertion policies try to predict when a block is dead on arrival (it will not see any reuse in the cache after its insertion). The paper concludes that DBPs are less accurate than insertion policies.

We focus on proposals relying on the reuse locality property of the SLLC blocks, which are “insertion policies” in Faldu & Grot (2016) taxonomy. We can classify them according to three characteristics:

- Replacement / content selection: In some proposals, the replacement algorithm gives higher priority to stay in the cache to those blocks showing reuse. On the other hand, some works leverage reuse locality to select for insertion only those blocks classified as reused, bypassing the rest.
- Detection / prediction: To classify blocks, some authors suggest reuse detection mechanisms, while others propose mechanisms to predict a reuse behavior before it appears.
- Address store: Both detection and prediction mechanisms need to remember past block addresses to identify the second access to a block. Detection mechanisms associate reuse to the block receiving a second access while prediction mechanisms associate reuse to a signature of the block receiving a second access. The structure that holds past accesses can be implemented in

several ways, either embedded into the SLLC itself or as an independent structure.

Table 2.5 contains a sample of previous work classified according to this taxonomy.

		Addresses in the SLLC	In a different store
Replacement	Detection	Gao & Wilkerson 2010 Gaur et al., 2011 (TC-AGE) Khan et al., 2012 Albericio et al., 2013a	Seshadri et al., 2012 Gupta et al., 2013
	Prediction	Qureshi et al., 2007 Jaleel et al., 2010b Wu et al., 2011	
Content selection	Detection	Albericio et al., 2013b	**Our proposal (ReD)**
	Prediction	Gaur et al., 2011 (Bypass + TC-AGE) Chaudhuri et al., 2012 Li et al., 2012	

Table 2.5: classification of previous work based on the reuse locality property, according to our taxonomy

2.8.1 Replacement policies

Both detection and prediction have been used to guide replacement algorithms, and most of them keep the reuse window in the SLLC itself.

Replacement mechanisms based on detection label a block as not reused when it comes from main memory (its first use in the reuse window that the SLLC is able to recall). Subsequent SLLC hits (second and later touches to the block) will flag the block as reused. Two proposals include an added store to remember these blocks. Seshadri et al. (2012) use a Bloom filter (Bloom, 1970) and Gupta et al. (2013) use a bypass buffer.

Replacement mechanisms based on prediction try to figure out whether a block will be reused before it really is, and flag the block as reused just after its first touch. Prediction policies categorize blocks according to certain features — signatures in Wu et al. (2011) — and study the reuse characteristics of any block in each category. Wu et al. (2011) analyze distinct types of signatures: memory region, program counter, or instruction sequence. As an example, the PC signature policy acts by classifying blocks according to the PC of the memory instruction responsible for bringing them in the chip. It identifies the reuse behavior of the blocks that each instruction loads (mainly categorizing them as reused or not reused), and assigns the same category to all blocks that the same instruction will bring in the future.

The Dual Insertion Policy (DIP) and Dynamic Re-Reference Interval Prediction (DRRIP) mechanisms are also predictors (Qureshi et al., 2007; Jaleel et al., 2010b). Using set-

dueling techniques, these mechanisms analyze the reuse behavior of the entire application and apply it to all its blocks. All blocks are categorized into a single category, the one of their own application.

2.8.2 Content selection policies

Except the Reuse Cache proposed by Albericio et al. (2013b), all other content selection policies include some sort of prediction: Li et al. (2012) uses the PC signature policy, Gaur et al. (2011) the trip count and use count of blocks, and Chaudhuri et al. (2012) the behavior of blocks during their stay in private caches and their coherence status. For each class, an algorithm analyzes its SLLC behavior and extends it to all future blocks belonging to the same category.

Comparing our proposal with others using prediction, they tend to be more complex, and often require the transfer of data between cache levels or to send the PC to the cache subsystem. Additionally, predictors show lower accuracy than detectors.

On the other hand, all previous content selection techniques track reuse (and reuse patterns) using the SLLC. Therefore, the SLLC size defines and limits the size of the reuse detection window.

Finally, all these proposals have in common an important constraint: their reuse detector is shared among all threads running on the CMP. A single thread can thrash the detector, shrinking the reuse detection window of the remainder applications. To overcome this, we propose implementing reuse detectors that are private to every processing core.

2.9 CONCLUSIONS

Previous publications reveal that the stream of references reaching the SLLC of a CMP shows little temporal locality. However, it shows reuse locality, i.e., blocks referenced more than once are more likely to be referenced in the near future. This leads to an inefficient use of the cache if conventional management is performed. There are several proposals addressing this problem for inclusive caches, but few that focus on exclusive ones.

This chapter proposes a novel content selection mechanism for exclusive SLLCs that leverages the reuse locality embedded in the SLLC request stream. We propose adding a Reuse Detector, placed in between each L2 cache and the SLLC, to discover which of the

L2 evicted blocks have not experienced reuse and avoid their insertion in the SLLC, bypassing them. We analyze problems affecting similar recent mechanisms (low accuracy, reduced visibility window and thrashing in the detector) and design ReD to overcome them as much as possible. We evaluate the proposal in a multicore chip with eight processors that executes a multiprogrammed workload. Properly designed, ReD prevents the insertion of many useless blocks in the SLLC, and helps keeping the most reused. Experimental results show that this allows for enhancing SLLC performance beyond other recent proposals. Specifically, ReD reduces the SLLC misses per instruction by 10.1% with respect to a base cache with TC-AGE replacement and no content selection, while CHAR and exclusive-cache versions of the Reuse Cache and the EAF cache reduce MPI by 4.3%, 4.5% and 2.7% respectively.

3 ReD: IMPROVING THE MANAGEMENT OF STT-RAM SHARED LAST-LEVEL CACHES[4]

The limitations of SRAM have prompted exploration of alternative memory technologies for implementing on-chip SLLCs. Among these, STT-RAM has emerged as a leading candidate due to its superior energy efficiency, reduced die area, and greater scalability. However, its relatively slow and energy-intensive write operations remain a significant drawback. This chapter addresses these issues by adapting the Reuse Detector (ReD), introduced in the previous chapter. ReD identifies blocks with no reuse potential and prevents their insertion into the SLLC, thereby reducing the frequency of write operations and hence the energy consumption in STT-RAM.

Experimental results on a system with eight cores running multiprogrammed workloads demonstrate that our mechanism reduces SLLC energy consumption by an average of 33%, cuts main memory energy usage by 6%, and improves performance by 7% compared to a baseline STT-RAM SLLC without ReD. Moreover, our proposal surpasses DASCA, the state-of-the-art STT-RAM SLLC management scheme, achieving 5% greater SLLC energy savings, 3% higher performance, and an additional 3% reduction in DRAM energy consumption.

3.1 INTRODUCTION

Current technologies used to implement SLLCs primarily rely on SRAM or embedded DRAM. However, both are power-hungry, particularly at the large capacities demanded by the growing number of processor cores. A promising approach to mitigate this issue involves adopting emerging Non-Volatile Memory (NVM) technologies. Among these, STT-RAM stands out as the most compelling candidate. It significantly reduces static power consumption and offers greater density than SRAM, enabling larger cache

[4] This part of the work was originally presented by its first author, Roberto Rodríguez Rodríguez, in his own thesis. In this dissertation, the chapter highlights my specific contributions: the design of ReD, the modifications required to address the challenges posed by STT-RAMs, and the architectural differences with respect to the state-of-the-art competing proposal. A summary of Roberto's work is also included to provide appropriate context and show the outcomes.

capacities within the same area. In addition, STT-RAM provides more efficient read operations in terms of both latency and energy. Despite these advantages, certain drawbacks hinder its widespread use as an SLLC in next-generation CMPs: specifically, its write operations are slower and more energy-intensive than those of SRAM. These limitations can lead to performance degradation and may even offset the energy gains obtained from reduced static power.

This chapter improves both the performance and energy efficiency of an STT-RAM SLLC by reducing the number of write operations it performs. To this end, we propose adapting and integrating ReD, which evaluates each block evicted from the private cache levels to determine whether it has been reused at the SLLC. If reuse is detected, the block is either inserted into the SLLC or used to update an existing entry. Otherwise, it bypasses the SLLC and is sent directly to main memory.

We evaluate our proposal using single- and multiprogrammed workloads, and experimental results show that ReD effectively prevents the insertion of low-utility blocks into the STT-RAM SLLC, thereby increasing the likelihood of retaining blocks that exhibit reuse. As a result, the number of slow and energy-intensive write operations to the STT-RAM SLLC is reduced, leading to lower energy consumption and improved system performance.

The rest of the chapter is organized as follows: Section 3.2 explains our motivation and some necessary background. Section 3.3 presents our proposal to improve the STT-RAM SLLC management. Sections 3.4 and 3.5 detail the experimental framework used and the obtained results, respectively. Section 3.6 recaps some related work and finally, Section 3.7 concludes the chapter.

3.2 BACKGROUND AND MOTIVATION

In this section, we first motivate the need for a new SLLC management scheme by outlining the main limitations of SRAM technology and conventional management strategies. We then briefly describe the DASCA scheme (Ahn et al., 2014), which represents the closest approach to our proposal and stands as the current state of the art in STT-RAM SLLC management.

3.2.1 Comparison of SRAM and STT-RAM technologies

Various emerging technologies are currently being considered as potential replacements for SRAM in constructing SLLCs. Among them, STT-RAM is the most promising candidate to overcome key limitations of SRAM, such as high energy consumption and read latency.

The primary difference between STT-RAM and SRAM lies in their storage mechanisms: STT-RAM stores information using a Magnetic Tunnel Junction (MTJ), whereas SRAM relies on electric charges. An MTJ consists of two ferromagnetic layers — referred to as the free layer and the reference layer — and a dielectric tunnel barrier in between (Figure 3.1, left). The reference layer maintains a fixed magnetic orientation, while the free layer can switch its orientation when a current is applied across the MTJ. If the magnetic orientations of the two layers differ, the MTJ exhibits high resistance, representing a logical '1'; if the orientations are aligned, the resistance is low, representing a logical '0'.

A read operation is performed by applying a small voltage across the MTJ and sensing the resulting current. Refer to Figure 3.1 (right), where the STT-RAM cell functions as a variable resistor. Writing to an MTJ involves applying a larger voltage for a defined duration, known as the *write pulse width*, to alter the magnetic direction of the free layer.

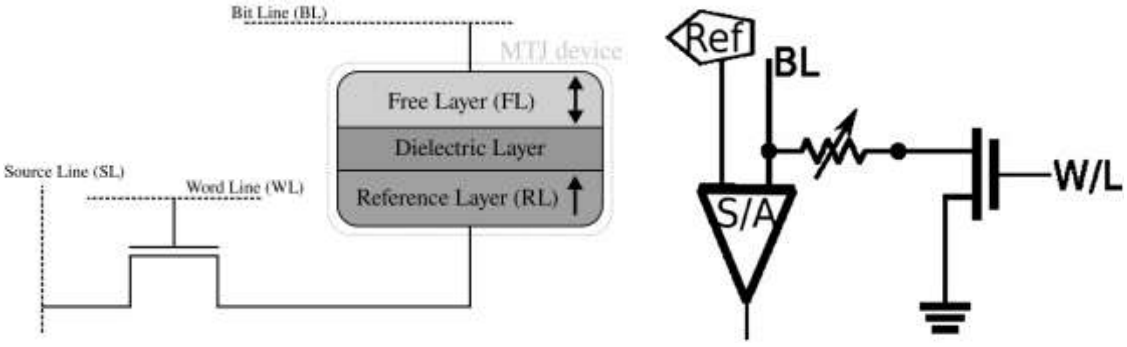

Figure 3.1 STT-RAM memory cell structure (left), and STT-RAM equivalent circuit (right). Extracted from Escuin (2024).

To illustrate the trade-offs between SRAM and STT-RAM for SLLC design, Table 3.1 summarizes the key characteristics of a 1-bank, 1MB cache implemented in 22nm technology. The SRAM and STT-RAM models were obtained using CACTI 6.5 (Hewlett-Packard, 2013) and NVSim (Dong et al., 2012), respectively. The STT-RAM cache offers a smaller die footprint and greater efficiency in read operations compared to its SRAM counterpart. More notably, it consumes nearly sixty times less static power. However, STT-

RAM also presents a critical drawback that must be addressed: significantly poorer write performance, both in terms of latency and energy consumption.

Parameter	SRAM	STT-RAM	Ratio SRAM / STT-RAM
Area (mm^2)	0.94	0.35	2.68
Read latency (ns)	8.75	5.61	1.56
Write latency (ns)	8.75	16.50	0.53
Read energy (nJ)	0.56	0.32	1.75
Write energy (nJ)	0.56	1.31	0.43
Leakage power (mW)	190.58	3.09	61.67

Table 3.1: area, latency and energy consumption for 22nm SRAM and STT-RAM caches with 1MB size.

3.2.2 The DASCA scheme

The *Dead Write Prediction Assisted STT-RAM Cache Architecture* (DASCA), proposed by Ahn et al. (2014), aims to reduce write energy by predicting and bypassing *dead writes,* which are writes to data in the SLLC that are not referenced again during the lifetime of the corresponding cache blocks. The authors introduce a theoretical model that classifies dead writes into three categories: dead-on-arrival fills, dead-value fills, and closing writes, each representing a different form of redundant write. Building on this model, DASCA employs a dead write predictor derived from a state-of-the-art dead-block predictor (Khan et al., 2010). Consequently, a write to the SLLC is bypassed only if it is predicted not to result in additional cache misses.

DASCA adds a dedicated field to each private cache line to store the program counter (PC) of the instruction that last wrote to the block. This field is updated only on write operations. In addition, a PC-signature table (prediction table) is incorporated into the design to guide dead write predictions, and is updated following the mechanism described in Table 2 of their paper. Specifically, the design samples a subset of cache sets and tracks PC information only for those sets. Predictions are made using the signature table, which consists of saturating counters similar to those employed in bimodal branch predictors. These counters are indexed by the signatures recorded in the sampler entries. In this way, the PC-based predictor establishes a correlation between dead blocks and memory instruction addresses (signatures), allowing distinct signatures to be used depending on the type of dead write being predicted.

3.3 DESIGN OF ReD FOR AN STT-RAM SLLC

In this section, we first describe the baseline system we build upon. We then present the proposed design that extends it.

3.3.1 Baseline system

The baseline multiprocessor system features a memory hierarchy composed of two private cache levels (L1 and L2) and an SLLC. All caches follow a write-back, write-allocate policy and use LRU replacement. The L1 and L2 caches are inclusive, whereas the SLLC is non-inclusive.

The baseline management policy for this memory hierarchy operates as follows: when a block is fetched from main memory, it is inserted into the private cache levels (L1 and L2) of the requesting core, but not into the SLLC. While the block resides in the private caches of that core, it may be requested by other cores. In such cases, the block is transferred from the source core's private L2 cache to the L1–L2 caches of the requesting core, as specified by the directory-based coherence protocol.

When a block is evicted from an L2 cache, the SLLC is checked. If the block is not present — either because it was never inserted or because it was previously evicted by the SLLC's replacement policy — it is inserted into the SLLC. Consequently, in our memory hierarchy, insertions into the SLLC originate exclusively from L2 caches, not directly from main memory, resembling the behavior of an exclusive cache policy. However, to avoid as much writing in the SLLC as possible, our baseline differs from a strictly exclusive policy in one key aspect: on an SLLC hit, the block is copied to the requesting core's private caches but is also retained in the SLLC. This way, a subsequent eviction from the L2 cache overwrites the existing copy in the SLLC only if the block has been modified in the private caches (i.e., it is dirty).

Consequently, for each specific block the SLLC works in exclusion with the private caches upon its initial request, but transitions to an inclusive policy following a subsequent access that results in a hit.

3.3.2 Adding ReD

As discussed in the previous chapter, several studies have shown that a significant fraction of the blocks inserted or updated in the SLLC are ultimately useless, as they correspond to dead blocks. These blocks are detrimental for two reasons: they displace other blocks that may still hold future reuse potential, and they increase the number of write operations to the SLLC. The latter is particularly undesirable in the context of NVMs, where write operations are both costly and energy-intensive, as outlined in previous sections.

In the previous chapter, we demonstrated that ReD can approximately halve the number of insertions into an SRAM-based SLLC under an exclusive memory hierarchy. Building on this, in the present chapter we apply ReD to reduce the number of dead blocks inserted or updated in an STT-RAM SLLC. This reduction enhances the efficiency of the SLLC, leading to improvements in overall system performance and reductions in energy consumption.

To achieve this, we adapt and extend the use of ReD to a configuration with a different inclusion/exclusion behavior, where the SLLC operates partially in exclusion and partially in inclusion with respect to the private levels. In the previous chapter, the exclusive policy dictated that, upon an SLLC hit, the block was moved to the private caches and removed from the SLLC. In the current design, the block is also inserted into the private levels upon an SLLC hit, but unlike before, it is retained in the SLLC until it is evicted by the replacement policy.

3.3.3 ReD operation

Figure 3.2 illustrates a flow diagram representing the process of a block request originating from a core. Since the primary objective is to minimize writes to the SLLC, the management approach differs from the one described in the previous chapter; specifically, blocks are not evicted (invalidated) in the SLLC when they receive a hit, to avoid writing to the cache.

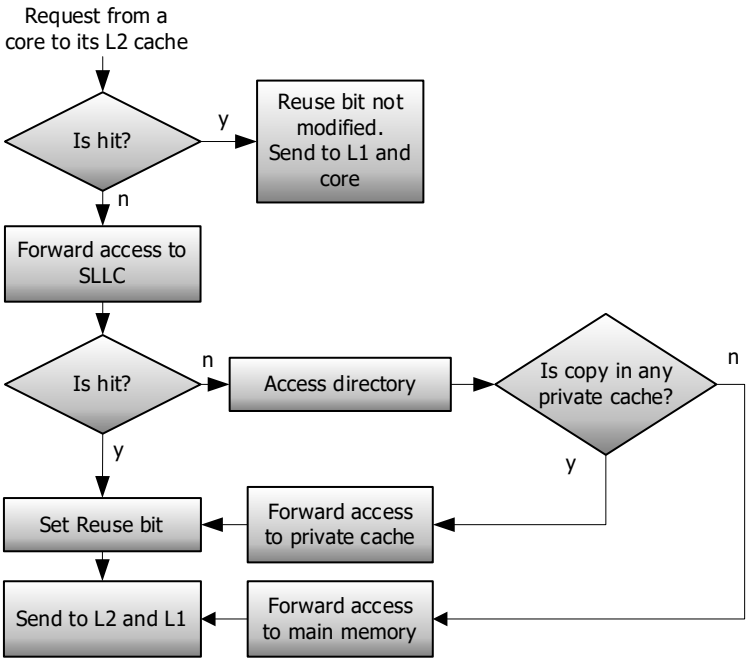

Figure 3.2: request from a core to its L2 cache.

Figure 3.3 shows how this approach modifies the management of block evictions from an L2 cache. When a block is evicted, its reuse bit is first examined. If this reuse bit is set, the block is inserted into the SLLC, provided it is not already present. Moreover, if the block already exists in the SLLC and is marked dirty, it is updated. If the reuse bit is not set but the block's tag is found in the ART, the block is handled identically to the previous scenario. In all other cases, the block is bypassed. Unlike our proposal for exclusive caches, no exception mechanism is necessary here, since there are no empty ways created by evictions following SLLC hits.

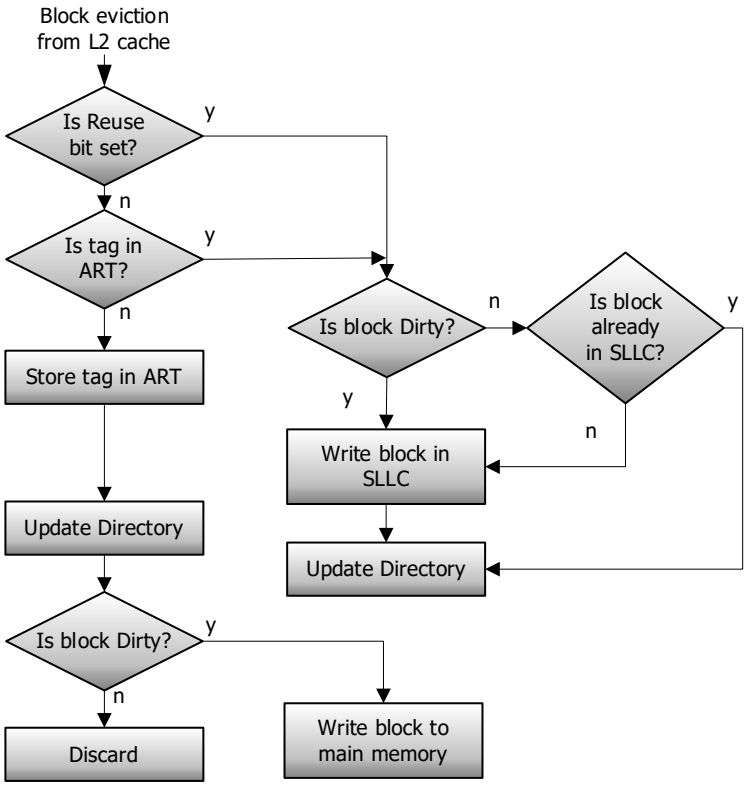

Figure 3.3. Block eviction from an L2 private cache.

3.3.4 Implementation details

Since our objective in this chapter is to strictly control the content stored in the SLLC in order to minimize write operations, we reduce the ReD capacity used in the previous chapter by half, setting it to 1 MB. This reduced capacity aims to increase the bypass fraction while still preserving good performance.

As in the previous chapter, we track sectors in the ART and compress the tags to minimize the hardware overhead associated with ReD. With 8K entries in the ART, the additional storage required per core is 14 KB. Therefore, for system with eight processors, the total storage overhead amounts to 112 KB, which corresponds to approximately 1.3% of an 8 MB SLLC.

3.3.5 Comparison of ReD and DASCA

In this section, we briefly outline the main differences between the ReD mechanism and the DASCA scheme. Regarding their operation, DASCA aims to predict dead writes in the SLLC. The authors of DASCA categorize dead writes into three types: dead-on-arrival fills,

dead-value fills, and closing writes. Dead-on-arrival fills refer to blocks that are accessed only once during their lifetime in the SLLC, specifically on the read miss that triggers their insertion. Dead-value fills occur when a block, after being inserted into the SLLC due to a read miss, receives a writeback from lower levels before any intervening read, effectively rendering the original data unused. Closing writes correspond to writeback for blocks that will not be accessed again prior to their eviction. DASCA relies on a PC-based predictor to identify these dead writes, which requires recording the PC signature of each instruction that accesses the SLLC.

In contrast, ReD predicts dead blocks based on their reuse behavior. Specifically, our approach uses the addresses of accessed data rather than instruction addresses, as in DASCA. Furthermore, while ReD stores these data addresses directly, DASCA employs a PC-signature table trained using an auxiliary sampler cache that operates in parallel with the conventional cache hierarchy.

Focusing on the specific implementation of the DASCA scheme used in our evaluation, it is important to note that our approach employs a memory hierarchy in which L1 and L2 caches are inclusive, while the SLLC is partially exclusive and partially non-inclusive (see Section 3.3.1). In contrast, the original DASCA evaluation was conducted on a two-level cache hierarchy, assuming non-inclusive caches by default. That is, their baseline setup differs from ours. To enable a fair comparison between ReD and DASCA, we re-implement DASCA using the same three-level cache hierarchy adopted for our evaluation of ReD.

As a result, the only high-level deviation from the original DASCA design concerns one of the three dead write classifications: dead-value fills — blocks that receive a writeback from lower-level caches right after being filled into the SLLC (after a read miss), before any read operation in between. This means the first SLLC write operation was useless, as data is overwritten consecutively. This write-after-write case cannot occur in our configuration. Specifically, in our setup, all insertions into the SLLC originate from L2 evictions. The first fill to the SLLC never happens directly, it happens when the block is evicted, in this case dirty, from L2. Even if the processor subsequently issues a write request for a word of such block, and results in a write hit in the SLLC, the rest of the block needs to be read from the SLLC to copy it into the private caches. Therefore, a useless SLLC write never happens.

Therefore, we consider this evaluation fair, as it applies DASCA under the same conditions as our proposal. In doing so, we ensure that ReD does not gain an unfair

advantage by structurally avoiding dead-value fills through a different content management policy between cache levels.

3.4 EXPERIMENTAL SETUP

3.4.1 The experimental framework

For our experiments, we use the *gem5* simulator (Binkert et al., 2011), a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture.

Within *gem5*, we configure an O3 (out-of-order) processor type CPU model to enable detailed simulation and improve accuracy. We employ the Ruby memory system with the MOESI_CMP_directory coherence protocol, which implements the Modified, Owned, Exclusive, Shared, and Invalid (MOESI) policy. We focus on a MOESI-based protocol because coherence models with an owned state (e.g., MOESI and MOSI) have been shown to reduce the number of writes to the SLLC, as reported by Chang et al. (2014). The on-chip network is modeled as a crossbar using Garnet (Agarwal et al., 2009), a detailed interconnection network model inside *gem5.* DRAM main memory is modeled using DRAMSim2 (Rosenfeld et al., 2011).

We simulate both single- and multiprocessor scenarios. Our experiments use workloads drawn from the SPEC CPU 2006 benchmark suite (Henning, 2006). Note that results from 4 out of the 29 benchmarks are excluded from the evaluation due to constraints in the experimental framework.

For the single-processor evaluations, we use reference inputs and simulate 1 billion instructions from the checkpoints selected using PinPoints (Patil et al., 2004). For the multiprocessor experiments, we fast-forward 100 million instructions, perform a cache warm-up phase for 200 million instructions, and then collect results over a minimum of 500 million instructions per processor.

Multi-processor evaluations report results from 28 multiprogrammed mixes composed of SPEC CPU 2006 benchmarks, executed on a CMP system with eight processors. To construct the multiprogrammed mixes, we adopt the following methodology: first, we execute each benchmark in isolation on our baseline system, configured with a 1 MB SLLC, and measure the number of write operations it generates to the SLLC. For each

benchmark, we compute the *number of writes to the SLLC per 1000 instructions* (WPKI). Based on this metric, we classify benchmarks into three categories: *high*, *medium* and *low*. Specifically, benchmarks with WPKI greater than 8 are assigned to the *high* category, those between 1 and 8 into the *medium* category, and benchmarks with WPKI less than 1 are placed in the *low* category. Table 3.2 shows the results of this categorization.

Category	Benchmarks
High	lbm, mcf, libquantum, bwaves, milc, cactusADM, zeusmp, leslie3d
Medium	hpcg, soplex, gcc, wrf, astar, hmmer, xalancbmk, gobmk, perlbench
Low	gromacs, calculix, h264ref, tonto, omnetpp, namd, sphinx3, GemsFDTD

Table 3.2: benchmark characterization according to the number of SLLC writes per kiloinstruction (WPKI).

Based on this classification, we build a set of 28 multiprogrammed mixes. First, we randomly generate three groups of four mixes each composed exclusively of applications from a single WPKI category. These are referred to as *H0-H3*, *M0-M3* and *L0-L3*, corresponding to the high, medium, and low WPKI categories, respectively. Next, we create an additional 16 mixes by combining applications from different categories in a balanced and homogeneous manner. The workload names encode the WPKI categories of the constituent applications. For example, *HL2* denotes the third mix composed of eight applications evenly split between the high and low WPKI categories.

3.4.2 Configuration of the baseline system

The main characteristics of both the processor and the memory hierarchy are summarized in Table 3.3. The SLLC read and write latencies are adjusted to reflect the characteristics of STT-RAM technology. Both latency and energy consumption values are obtained from NVSim (Dong et al., 2012) for a 1MB single-bank cache and are presented in Table 3.1. When scaling the SLLC to larger capacities, we multiply the leakage power by the number of cores.

Architecture	x86
CPUs	1 or 8, 2 GHz
Pipeline	8 Fetch, 8 decode, 8 rename, 8 issue/execute/writeback, 8 commit
Registers	Integer (256), floating point (256)
Buffers	Reorder buffer (192), instruction queue (64)
Branch predictor	TournamentBP
Functional units	IntALU = 6, IntMultDiv = 2, FPALU = 4, FPMultDiv = 2, SIMD-Unit = 4, RdWrPort = 4, 1fpPort = 1
Private Cache L1 D/I	32 KB, 8 ways, LRU replacement, block size 64B, access latency 2 cycles, SRAM
Private Cache L2 D/I	256 KB, 16 ways, LRU replacement, block size 64B, access latency 5 cycles, SRAM
Interconnection	Crossbar network, modeled using Garnet, latency 3 cycles
Shared Cache L3	1 bank/1MB per core, 16 ways, LRU replacement, block size 64B, R/W latency 6/17 cycles, STT-RAM
DRAM	2 Ranks, 8 banks, 4 KB page size, DDR3 1066 MHz
DRAM bus	2 channels with a bus of 8 bits

Table 3.3: CPU and memory hierarchy specification.

3.4.3 Configuration of the evaluated proposal

To evaluate ReD, we implement the policy within the cache hierarchy and introduce modifications to the coherence protocol. Table 3.4 summarizes the ReD configuration used in the subsequent performance analysis. The reported values correspond to each individual ART instance. For additional details, refer to Section 3.3.4.

ReD capacity	1MB
Associativity	16-way
Replacement	FIFO
Sector size	2 blocks
Tag size	10 bits
Number of sets	512
Number of entries	8,192

Table 3.4: ReD evaluation configuration

3.4.4 Performance metrics

Single-processor performance is reported using IPC, normalized to that of the baseline configuration. To evaluate performance in multiprogrammed workloads, we compute the system IPC, defined as the sum of the instructions per cycle committed by all processors in the chip ($\sum_{i=1}^{n} IPC_i$, where *n* is the number of processors), and normalize this value with respect to the baseline.

Regarding energy usage, we report the total energy consumption of both the SLLC and DRAM, normalized to that of the baseline. For the SLLC, we consider both dynamic and static contributions. The static contribution is computed based on NVSim (Dong et al., 2012), which provides the leakage energy for a 1 MB SLLC. To obtain the total static energy, we multiply the leakage value by the execution time and by the number of cores. The dynamic contribution is obtained by multiplying the number of SLLC accesses by the per-access dynamic energy, also reported by NVSim. The total dynamic energy consumption is calculated as follows:

$$\begin{aligned}\text{DynamicEnergy} \quad &= H_{\text{SLLC}} * HE_{\text{SLLC}} + W_{\text{SLLC}} * WE_{\text{SLLC}} \\ &\quad + M_{\text{SLLC}} * ME_{\text{SLLC}}\end{aligned} \tag{3.1}$$

where H_{SLLC}, W_{SLLC} and M_{SLLC} denote the number of hits, writes and misses in the SLLC. The terms HE_{SLLC}, WE_{SLLC} and ME_{SLLC} denote the corresponding energy consumption per hit, write and miss, respectively.

For the DRAM, the total energy consumption is obtained directly from the simulator.

In the multiprocessor results, we divide the number of SLLC writes as well as both the dynamic and static energy consumption by the total number of instructions executed, before normalizing. This adjustment accounts for variations in the number of simulated instructions per core across different configurations (see Section 3.4.1). In the single-processor scenario, this is not required, as all configurations execute the same number of instructions (1 billion) in every run.

3.5 EVALUATION

This section compares the effectiveness of ReD and DASCA in managing an STT-RAM SLLC, focusing on both performance and energy consumption in the SLLC and main memory. Section 3.5.1 presents the single-processor evaluation, while Section 3.5.2 discusses the multiprocessor scenario.

3.5.1 Evaluation in a single-processor system

We begin by presenting the number of writes to the SLLC generated by each evaluated proposal, along with the resulting performance. Next, we analyze the associated energy consumption in both the STT-RAM and main memory, following the model described in

Section 3.4. Finally, we discuss the observed results. All graphs in this section report individual values for each benchmark, along with the arithmetic or geometric mean, labelled as *AVG* and *GMEAN* respectively. They also report the same type of mean, computed over only the eight most write-intensive benchmarks (as identified in Table 3.2), labelled as *HIGH*.

Content selection. Figure 3.4 illustrates the number of writes to the STT-RAM SLLC generated by the DASCA scheme and our ReD proposal, normalized to a baseline system that does not employ any write reduction or filtering mechanism.

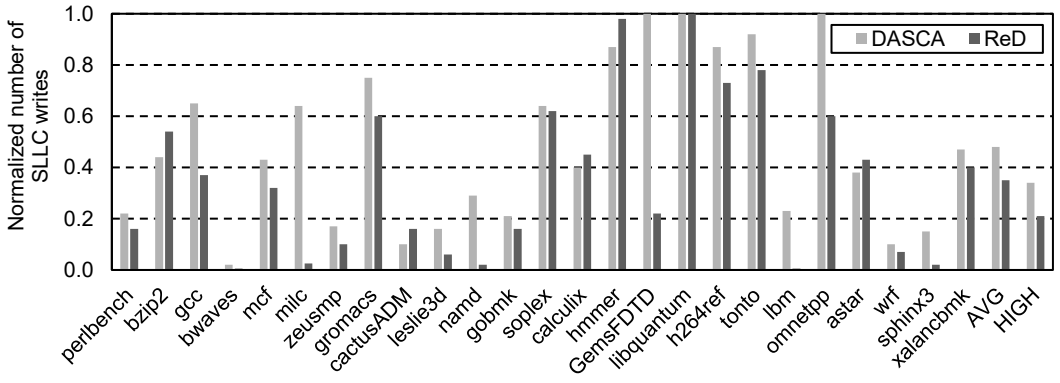

Figure 3.4. Number of writes to the STT-RAM SLLC normalized to the baseline.

As illustrated, our proposal significantly outperforms DASCA. Notably, in 20 out of 25 benchmarks evaluated, ReD demonstrates superior effectiveness in reducing write traffic to the STT-RAM SLLC. Overall, block-bypassing decisions guided by ReD reduce the number of SLLC writes by approximately 65% compared to the baseline system, whereas DASCA achieves only a 52% reduction. Furthermore, when focusing specifically on the eight programs with the highest traffic (*HIGH*), ReD decreases the number of SLLC writes by 80% relative to the baseline, compared to a 66% reduction achieved by DASCA.

Performance. In addition to energy efficiency improvements, performance remains a critical aspect. It is therefore essential to ensure that our approach does not degrade performance while reducing SLLC energy consumption. To further assess the benefits of ReD, Figure 3.5 presents the normalized IPC achieved by each scheme.

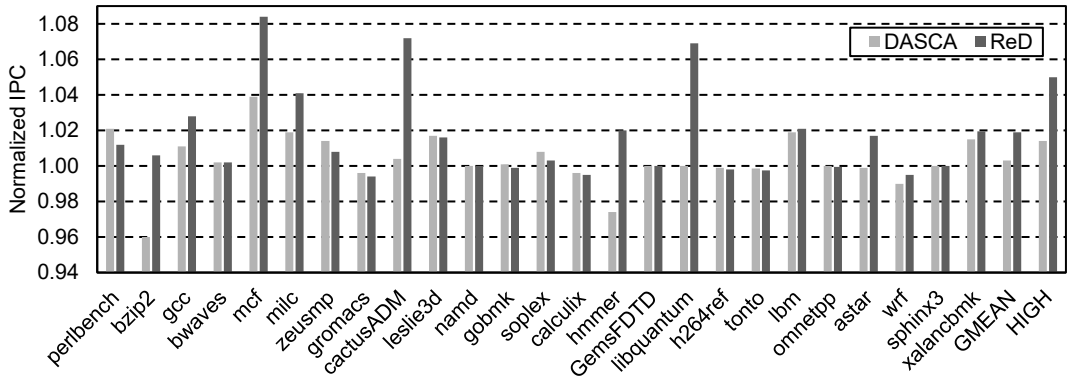

Figure 3.5: performance (IPC) normalized to the baseline.

Overall, our scheme performs moderately better than DASCA. ReD achieves a 1.9% performance improvement over the baseline, whereas DASCA yields only a 0.3% gain. When focusing on the write-intensive applications, ReD clearly outperforms DASCA, with performance improvements of 5.0% and 1.4%, respectively. These results suggest that our approach is particularly effective for applications that generate a high number of writes to the SLLC — a trend that will be further confirmed in the multiprocessor evaluation.

Energy savings. Figure 3.6 shows the total energy savings in the SLLC, accounting for both dynamic and static components. Overall, our proposal achieves a 34.5% energy reduction compared to the baseline, while DASCA achieves 29.5%. When considering only the write-intensive programs, the savings increase to 60% for ReD and 49% for DASCA.

Breaking down the total energy savings, ReD reduces dynamic energy by 50%, compared to 42% for DASCA. In the write-intensive subset, the reductions are 68% and 57%, respectively. For the static component, ReD achieves a 2.0% energy reduction, whereas DASCA reaches only 0.3% (5.0% and 1.4%, respectively, for the write-intensive applications).

It is important to note that avoiding SLLC writes reduces dynamic energy, while improvements in performance translate into lower static energy consumption. As illustrated in Figure 3.7, dynamic energy in the baseline system dominates for most of the applications evaluated, being significantly higher than the static contribution.

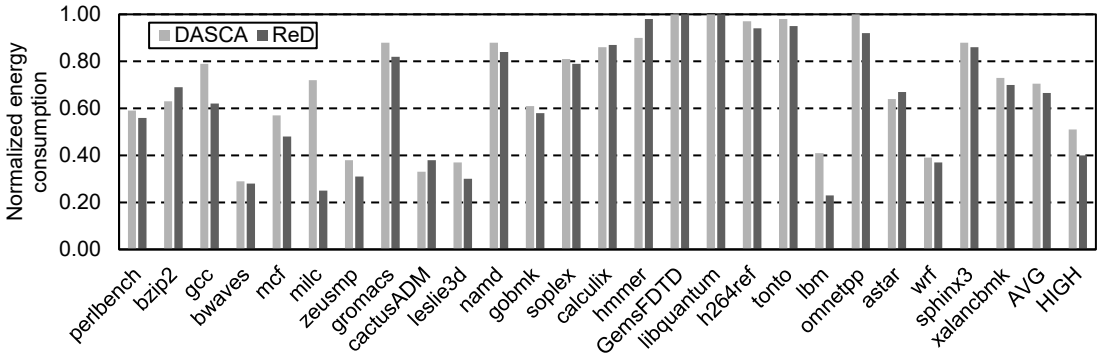

Figure 3.6. Energy consumption in the STT-RAM SLLC normalized to the baseline.

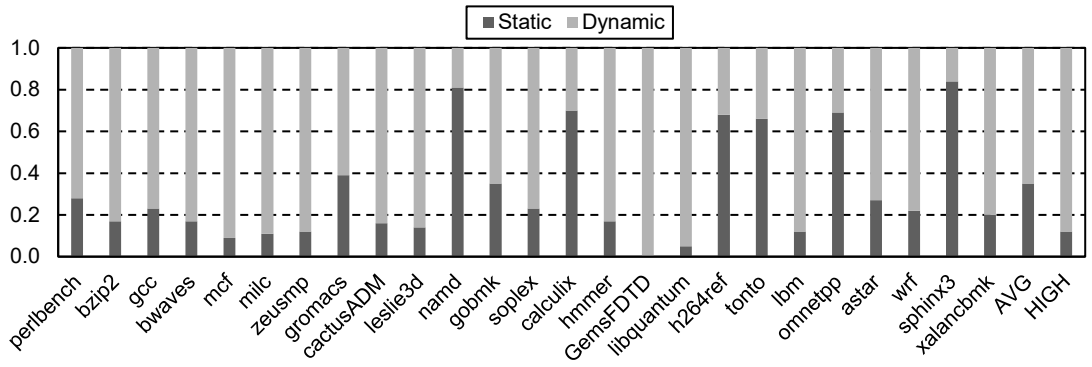

Figure 3.7: breakdown of energy consumption in the SLLC into the static and dynamic contributions for the baseline in the single-processor system.

Finally, we have also evaluated the impact of each scheme on energy consumption in DRAM main memory. For simplicity, we do not show the results for all individual applications. As expected, DRAM energy savings generally follow the same trend as performance improvements. Overall, our proposal reduces DRAM energy consumption by 2.0% compared to the baseline, whereas DASCA achieves a modest 0.2% reduction. In the case of write-intensive applications, the reductions are 4.7% and 1.1%, respectively.

Discussion. A closer inspection of individual benchmarks reveals several noteworthy cases that merit further analysis. Overall, the relative trend in the number of SLLC writes between ReD and DASCA is largely reflected in the energy consumption differences, although this relationship is modulated by the corresponding performance results. However, some exceptions can be observed in *namd*, *GemsFDTD* and *omnetpp.* In these benchmarks, ReD achieves a significantly larger reduction in the proportion of SLLC writes compared to DASCA, yet this reduction does not yield proportional energy savings or

performance gains. In fact, the improvements in both metrics remain modest. This can be attributed to the nature of these applications: they are three of the four benchmarks with the lowest WPKI values. Although the relative reduction in writes is significant, the absolute number of avoided writes is small. Consequently, the energy savings and performance gains in these cases remain modest.

Also, in applications such as *mcf*, *cactusADM* and *hmmer*, ReD achieves significantly higher IPC values than DASCA, despite both techniques exhibiting similar write reduction capabilities. For these three benchmarks, the results indicate that ReD achieves substantial improvements over both DASCA and the baseline in terms of SLLC hit rate. Notably, the number of SLLC hits in *cactusADM* and *mcf* increases by factors of 7.23× and 2× compared to the baseline, while DASCA achieves only 1.89× and 0.89×, respectively. Across all evaluated benchmarks, ReD increases the number of SLLC hits by approximately 31% compared to the baseline, and 106% when considering only the write-intensive ones. In contrast, DASCA yields only a 5% improvement overall and 31% in the high-WPKI subset. In summary, ReD is more effective at identifying blocks with high reuse, resulting in improved performance.

A third special case is observed in the *libquantum* application, whose behavior may initially appear counterintuitive. Neither ReD nor DASCA is able to significantly reduce the number of writes to the SLLC. However, *libquantum* running under ReD reports a 7% performance improvement over the baseline, while performance remains largely unchanged with DASCA. Additionally, as expected due to the low number of bypasses, the number of SLLC hits is approximately the same across all three configurations. The explanation for this performance improvement lies in SLLC bank contention caused by intense write activity. Of all benchmarks, *libquantum* experiences the highest level of stalling due to write contention. Although the write reduction achieved by ReD is limited, it is sufficient to reduce stall occurrences by approximately 8% compared to the baseline. In absolute terms, this translates into several million stalls avoided, which ultimately accounts for the performance improvement observed.

In contrast, although benchmarks such as *gromacs*, *calculix* and *wrf* exhibit moderate reductions in SLLC writes under both ReD and DASCA, they perform worse than the baseline. In these cases, the number of SLLC hits is lower than in the baseline, suggesting that the bypassing decisions made by both schemes are ineffective. As a result, the energy

savings from reduced write activity are partially offset by performance losses due to an increased number of SLLC misses. Additionally, although write operations are not on the critical path, the performance gains from avoiding long write latencies may be mitigated if contention arises at the SLLC banks for the writes that are still performed.

3.5.2 Evaluation in a multiprocessor system

This section presents and analyzes the key results obtained from evaluating ReD and DASCA on a CMP system with eight processors and an 8 MB SLLC.

In addition to the individual results of the 28 evaluated mixes, we report the mean values for several groups of mixes. For normalized IPC, we use the geometric mean, while for all other metrics, the arithmetic mean is reported. The groups and their corresponding labels are defined as follows:

- *AVG*: all 28 mixes
- *HIGH*: the four mixes composed exclusively of benchmarks in the high WPKI category (H*i* mixes)
- *H + HM*: the four H*i* mixes, plus the four mixes combining high and medium WPKI benchmarks (HM*i* mixes).
- *H + HM + HML*: the H*i* mixes, the HM*i* mixes, and the four mixes combining high, medium, and low WPKI benchmarks (HML*i* mixes)
- *SomeH*: all 16 mixes that include at least one application from the high WPKI category.

Content selection. Figure 3.8 illustrates the number of writes to the STT-RAM SLLC generated by DASCA and ReD. Results for both schemes are normalized to the baseline.

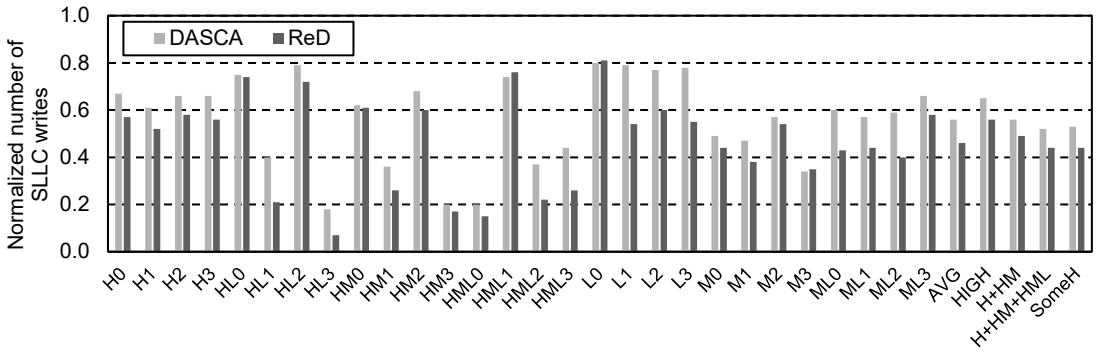

Figure 3.8: number of writes to the STT-RAM SLLC normalized to the baseline in the CMP system.

These results show that ReD is consistently more effective than DASCA in reducing write operations to the SLLC. In all but two of the evaluated workloads, ReD outperforms DASCA. On average, ReD reduces the number of writes by 54% compared to the baseline, while DASCA achieves a 44% reduction. For the write-intensive mixes, the reductions are 44% and 35% respectively.

When comparing the level of SLLC bypassing achieved by ReD in these multiprogrammed workloads to the 49% reported in the previous chapter (see Section 2.6.4), we observe a moderately higher rate. Although the two scenarios are not directly comparable due to differences in simulation environment, workload composition and cache architecture, this outcome is nonetheless expected. In the STT-RAM design, the ReD capacity was intentionally reduced by half to further decrease the number of write operations to the SLLC.

Performance. Figure 3.9 illustrates the IPC delivered by each evaluated policy, normalized to the baseline.

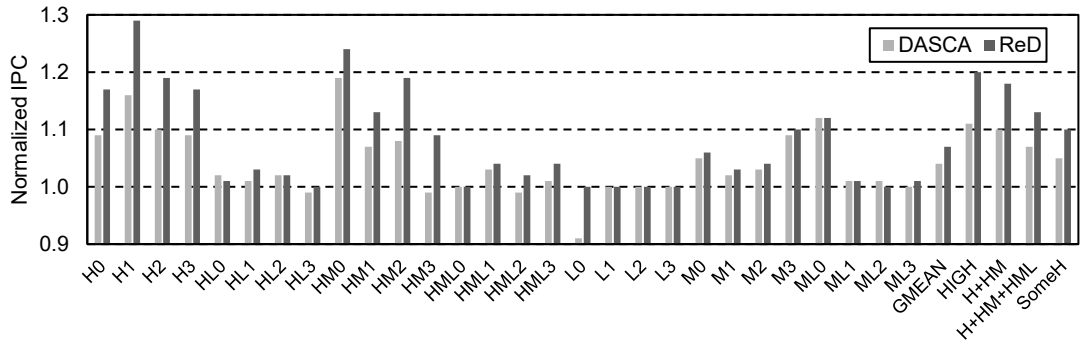

Figure 3.9: performance in the CMP system, measured with the system IPC normalized to the baseline

As shown, ReD moderately outperforms DASCA. This highlights a key advantage of ReD, as it not only reduces the number of writes to the SLLC more effectively than DASCA, but also delivers higher performance. This improvement, as will be discussed later, contributes to greater energy savings in both the SLLC and main memory. Overall, ReD improves performance by 7% compared to the baseline, while DASCA achieves a 4% improvement.

A closer examination of individual mixes reveals that ReD particularly outperforms DASCA in the H*i* mixes, composed of write-intensive benchmarks. In these cases, ReD improves performance by 20% relative to the baseline, while DASCA yields an 11% improvement. As shown in Figure 3.9, ReD also significantly surpasses DASCA and the baseline in the *SomeH* mixes, which include at least one high-WPKI benchmark.

Energy savings. Figure 3.10 illustrates the energy savings in the SLLC. As in the single-processor scenario, the graph exhibits a trend similar to that observed in write reduction (Figure 3.8), with ReD consistently outperforming DASCA, but modulated with the performance results. As shown in Figure 3.11, the dynamic component of SLLC energy consumption is higher than the static one, except in mixes composed exclusively of low-WPKI benchmarks. Consequently, the ability to reduce SLLC write activity (which affects the dynamic portion) has a greater impact on total energy consumption than performance improvements, which influence the static portion. Overall, ReD achieves a 32.5% reduction in SLLC energy consumption compared to the baseline, while DASCA yields a 27% reduction. For write-intensive mixes, ReD and DASCA reduce SLLC energy by 34% and 27.5%, respectively.

Breaking down the savings, ReD reduces the static energy component by 6% on average (15% for the write-intensive mixes), while DASCA achieves a 3.6% reduction (9.5% in the *high* mixes). In terms of dynamic energy, ReD reports average savings of 43% (36% in the write-intensive mixes), whereas DASCA achieves 36% (30% in the *high* mixes). Finally, mixes composed of low-WPKI benchmarks show the lowest energy savings across all configurations. This is consistent with their modest write reduction and the relatively high contribution of static energy to total SLLC consumption in such scenarios.

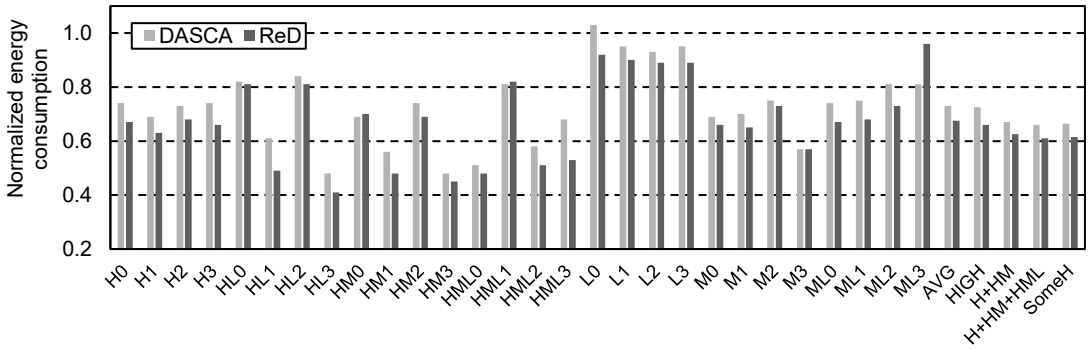

Figure 3.10: energy consumption in the STT-RAM SLLC normalized to the baseline in the CMP system.

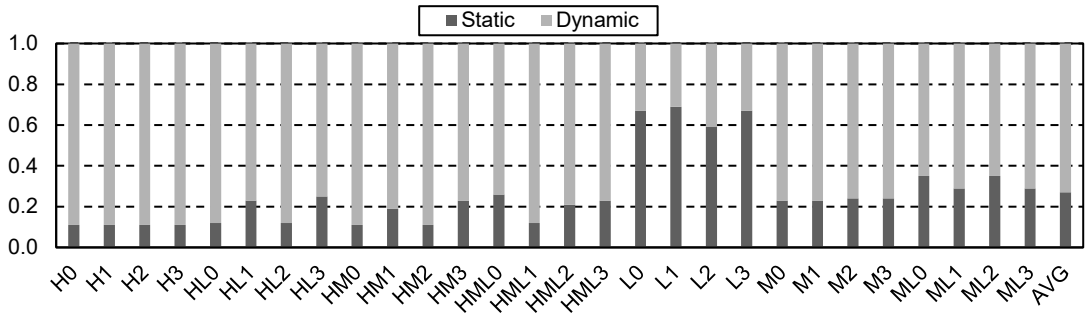

Figure 3.11: breakdown of energy consumption in the SLLC into the static and dynamic contributions for the baseline in the CMP system.

Figure 3.12 additionally illustrates the energy savings achieved in main memory. Unlike in the STT-RAM SLLC, leakage power constitutes a significantly larger portion of total energy consumption in DRAM. As a result, the observed trends largely mirror those of the IPC graph, but in inverse relation: higher performance leads to lower DRAM energy usage due to shorter execution time. On average, ReD reduces main memory energy consumption by 6% compared to the baseline (3% for the write-intensive mixes). DASCA achieves a 3% reduction on average but consumes 6% more energy in the high-WPKI mixes. This energy overhead may appear counterintuitive, especially considering that DASCA reduces SLLC writes by 35% and delivers a performance improvement exceeding 10%. However, the explanation lies in the increase in SLLC misses under DASCA for this subset, which leads to more frequent DRAM accesses, as shown by our study.

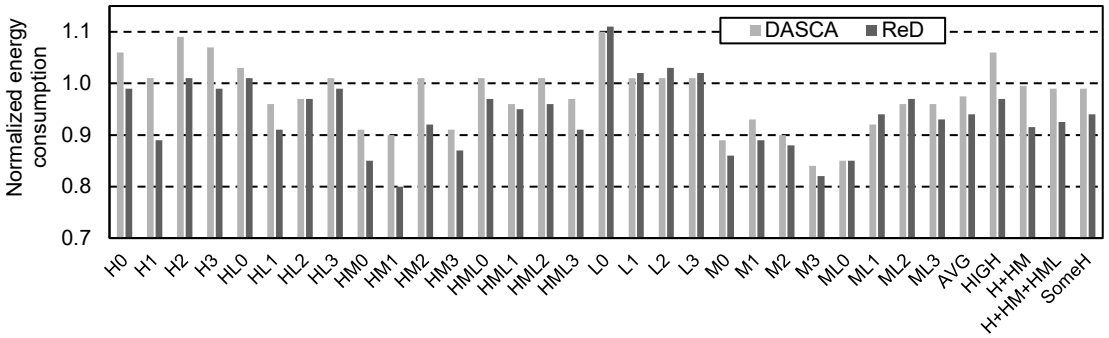

Figure 3.12: energy consumption in DRAM normalized to the baseline in the CMP system.

Discussion. The superior performance of ReD is attributable to several factors, the most important of which is the high accuracy achieved in content selection. Figure 3.13 shows the number of SLLC hits per kilo instruction experienced by each mix, normalized to the baseline. As the figure illustrates, ReD achieves a higher number of hits than DASCA in all but one mix, confirming that it is more effective at selecting highly reused blocks.

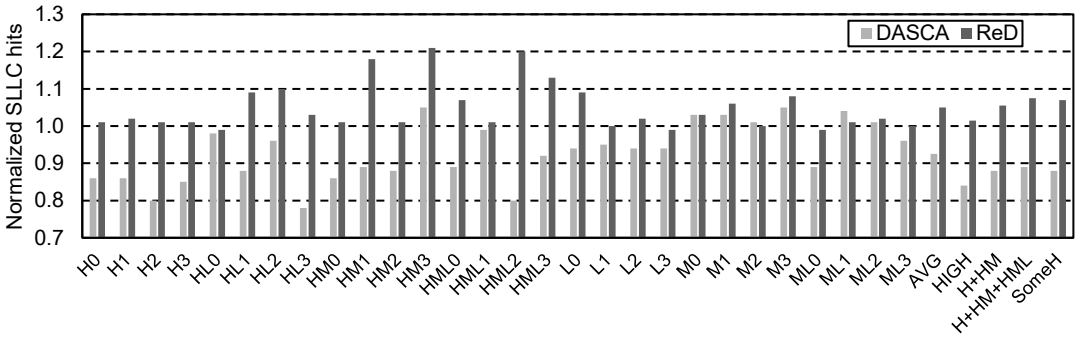

Figure 3.13: number of STT-RAM SLLC hits per kiloinstruction normalized to the baseline in the CMP system.

In addition to the improvement in hit rate, several other metrics also help explain ReD's superior performance. These include SLLC misses, DRAM reads and writes, row buffer read hit rate, and SLLC bank contention. Our analysis shows that ReD consistently outperforms DASCA across all of these metrics, both when considering the full set of mixes and when focusing specifically on the write-intensive ones.

3.6 RELATED WORK

In recent years, various researchers have proposed solutions to address the challenges of energy consumption and performance in STT-RAM SLLCs. These efforts primarily focus on either reducing the number of writes or lowering the energy consumed per write.

A substantial body of work has focused on reducing write traffic to the STT-RAM. Wang et al. (2013) propose an obstruction-aware cache management (OAP) mechanism, which periodically monitors the cache to detect specific applications that obstruct the SLLC. These are applications that use the SLLC but do not obtain any performance benefit from it, instead causing degradation to the rest of the workload. Upon detection, OAP forwards the data to the next cache level or main memory as appropriate.

Rasquinha (2011) introduces two techniques aimed at reducing write activity and improving energy efficiency of a last level (L2) STT-RAM cache. The first involves the insertion of a small cache between L1 and the LLC, called Write-Cache (WC). It stores only dirty lines evicted from L1 and is mutually exclusive with the LLC. On a cache access, both the LLC and WC are probed in parallel. Write misses are allocated in WC and load misses are directed to the LLC. By absorbing the majority of L1 writebacks, WC substantially decreases write traffic to the LLC.

Yazdanshenas et al. (2014) present a coding scheme based on the concept of value locality to reduce switching activity. By replacing frequently observed data patterns with limited weight codes, their method achieves fewer and more uniform writes.

Jung et al. (2013) exploit the observation that a significant portion of the data written to a cache consists of zero-valued bytes or words. Their design augments the tag array with fine-grained ‘all-zero’ flags at byte and word levels. Before writing, the data value is checked. If the zero-valued bytes or words are detected, the system sets the corresponding flags and only writes the non-zero bytes or words. On reads, the original content is reconstructed by combining the stored non-zero data with the flag information.

Park et al. (2012) propose logically dividing each cache line into several partial lines and maintaining a per-segment history bit in L1 to track which parts have changed. When a dirty L1 block is written back to the LLC, only the modified segments are updated, thereby reducing write volume.

Mao et al. (2013) address the write pressure introduced by prefetching. One of their techniques prioritizes different types of SLLC requests — load, store, prefetch or write back — based on their criticality, ensuring that more critical requests are served first. In multiprocessor systems, excessive requests from cache-intensive programs can block those from cache-unintensive ones, leading to starvation. To mitigate this, they propose a second mechanism that raises the priority of requests from cache-unintensive programs so they are promptly served.

Finally, Chang et al. (2014) analyze the impact of cache coherence protocols on the number of writes in STT-RAM SLLCs. They show that protocols with an owned state, such as MOESI and MOSI, contribute to reduce them.

Another line of research focuses primarily on improving the performance of STT-RAM caches. Jog et al. (2012) propose a cache revive technique to calculate the retention time. Some cache blocks retain data even after the retention time has expired. The retention time is selected to minimize the number of unrefreshed cache blocks.

Guo et al. (2010) propose the use of STT-RAM for designing combinational logic, register files and on-chip storage structures, including instruction and data L1 caches, translation lookaside buffers (TLBs), and L2 caches. To hide the write latency inherent to STT-RAM, they introduce a sub-bank buffering technique, which enables writes to complete locally within each sub-bank while allowing reads from other locations within the array to proceed without obstruction. They show that, by carefully designing the pipeline, the STT-RAM design can significantly reduce leakage power while maintaining a performance level close to that of an SRAM-based design.

Sun et al. (2011) propose an STT-RAM cache design for lower-level caches in which different cache ways are designed with varying retention periods. For instance, in a 16-way cache, way 0 is built using a fast STT-RAM design with a low retention period, while the remaining 15 ways use a slower STT-RAM design with higher retention period. Their technique employs hardware to detect whether a block is read- or write-intensive. Write-intensive blocks are primarily allocated to way 0, whereas read-intensive blocks are allocated to the other ways. Additionally, to avoid refreshing dying blocks in way 0, their design incorporates data migration, relocating such blocks to banks with longer retention periods.

Finally, Sun et al. (2009) propose a write-buffer design to address the long write latency of a last level (L2) STT-RAM cache. In their design, the LLC may receive requests from both L1 and the write buffer. Since STT-RAM exhibits lower read latency than write latency, and because read operations are more performance-critical, the buffer adopts a read-preemptive management policy that gives higher priority to read requests over writes. Additionally, the authors propose a hybrid cache design combining SRAM and STT-RAM, with the goal of redirecting the most write-intensive blocks to the SRAM portion.

We compare ReD exclusively against DASCA for three main reasons. First, DASCA is more recent than all the aforementioned techniques and is the most closely related to our work, as both mechanisms aim to reduce the energy consumption of an STT-RAM SLLC by bypassing write operations predicted to be dead or unlikely to exhibit reuse. Second, some of the previously discussed proposals, such as the one by Wang et al. (2013), are already evaluated in the DASCA paper and are clearly outperformed by it. Third, several other approaches (Yazdanshenas et al., 2014; Jung et al., 2013; Park et al., 2012; Mao et al., 2013; Jog et al., 2012; Sun et al., 2009; Sun et al., 2011), although addressing the same general problem, are entirely orthogonal to ours. ReD could, in fact, be built on top of them, making a direct comparison inappropriate. For instance, Yazdanshenas et al. (2014) address the STT-RAM write energy problem at the circuit level by identifying frequently stored values in the SLLC (value locality) and encoding these patterns to reduce the number of writes and balance cell wear. Unlike our proposal, this approach operates at the bit level.

3.7 CONCLUSIONS

In this chapter, we addressed two major limitations of conventional SRAM SLLCs: their high energy consumption and inefficient content management. To overcome these issues, we proposed using an STT-RAM SLLC whose contents are selected by our ReD mechanism, adapted to this cache architecture. ReD improves the management of the STT-RAM SLLC in two complementary ways. First, it bypasses to main memory a significant portion of incoming blocks, reducing the number of costly write operations. Second, it increases the SLLC hit rate, leading to moderate performance gains. Additionally, main memory energy consumption is also reduced as a result.

Our approach outperforms other techniques aimed at reducing energy consumption in STT-RAM SLLCs, including the state-of-the-art DASCA scheme. DASCA delivers inferior results due to its lower effectiveness in reducing write operations and its less accurate block selection strategy. As a result, it achieves smaller reductions in energy consumption and more limited performance gains. In contrast, ReD more accurately identifies reusable blocks, leading to substantial improvements. On average, ReD reduces SLLC energy consumption by 33%, achieves an additional 6% reduction in main memory energy, and improves performance by 7% compared to an STT-RAM SLLC baseline without ReD. More notably, when compared directly with DASCA, ReD provides an average of 5% greater SLLC energy savings, 3% higher performance, and a further 3% reduction in DRAM energy consumption.

4 ReD+: A POLICY BASED ON REUSE DETECTION FOR OPTIMIZED BLOCK SELECTION IN SHARED LAST-LEVEL CACHES

In this chapter, we propose a novel content selection policy for SLLCs that, like the previously proposed ReD policy on which it is based, leverages reuse detection to determine whether a block coming from main memory should be inserted into the SLLC. This enhanced policy, named ReD+, optimizes the block selection mechanism of ReD to achieve improved accuracy and performance: blocks bypass the SLLC unless their expected reuse behavior suggests likely reuse, inferred either from recent reuse history or from the behavior of the associated requesting instructions. As in the original ReD, blocks are inserted into the SLLC upon their second request within a limited time window. Additionally, ReD+ permits certain blocks to enter the SLLC upon their first request if the associated requesting instruction has previously demonstrated a pattern of requesting highly reused blocks. In addition to the ART, which tracks reuse for specific block addresses, ReD+ introduces a PC Reuse Table (PCRT) that measures and correlates reuse with its requesting program counters, and regulates the insertion rate into the ART to prevent thrashing.

In this chapter, we also adapt ReD+ to work with conventional non-inclusive SLLCs, showing that it achieves performance comparable to that of leading state-of-the-art replacement policies for such architectures. This mechanism was presented under the generic name "ReD" at the 2nd Cache Replacement Championship (CRC2). It achieved third place by overall score among 15 submissions, and second in shared-cache (multiprocessor) score.

4.1 INTRODUCTION

4.1.1 Motivation. Problem analysis

In previous chapters, we proposed the Reuse Detector (ReD), a novel content selection mechanism that leverages an Address Reuse Table to identify blocks that have not experienced prior reuse and prevents their insertion into the SLLC by bypassing them.

Avoiding the insertion of many useless blocks in the SLLC helps maintain more frequently reused blocks for extended periods. We explore the design and evaluate the results of applying this block selection policy to exclusive and STT-RAM-based SLLCs.

The foundational design of ReD presents a significant limitation related to the reuse detection mechanism: blocks with reuse will experience two last-level cache misses before finally being inserted into the SLLC. Although our previous results indicate that the increased efficiency provided by ReD more than compensates for this drawback, thus improving overall performance, additional performance gains could still be realized by preventing the second cache miss.

To avoid the second miss, we propose to exploit the correlation between the reuse pattern of the blocks and the instructions that request them for the first time. A similar correlation has been pointed out and exploited in a previous study (Wu et al., 2011). In our policy, we focus on the instructions that request blocks with a high reuse probability. SLLC misses coming from such instructions will always trigger SLLC block storage, avoiding the second miss.

An additional problem in ReD is detector thrashing within the ARTs. As explained in Chapter 2, ReD prevents global thrashing by implementing private ARTs per core. In this way, a thread experiencing many misses in its private caches does not reduce the reuse detection window for threads running on other cores. However, the private ART of the affected core may still experience thrashing, significantly reducing its reuse detection window.

To mitigate this problem, we leverage the previously stored information to detect correlations between the reuse patterns of blocks and the instructions that initially request them, with the goal of reducing the insertion rate of new addresses into the ART. Specifically, the insertion rate is lowered if the associated requesting instruction has already shown a correlation with either high or very low block reuse. In the first case, requested blocks are classified as reused regardless of their status in the ART. In the second case, blocks are predicted to have no future reuse, making its recording into the ART unnecessary.

4.1.2 Inclusion relationships in the cache hierarchy

Based on our previous results, we believe that the block selection policy is the key component of the SLLC replacement policy. An improved content selection could deliver significant performance gains comparable to those of any state-of-the-art full replacement policy — not only in specific scenarios, as discussed in Chapters 2 and 3, but also in generic conventional SLLCs.

Specifically, in Chapter 2 we have presented ReD for exclusive SLLCs. In Chapter 3, we adapt it to function optimally with STT-RAM SLLCs using a mixed approach: for each specific block, the SLLC operates in exclusion with the private caches upon the block's initial request but transitions to an inclusive policy following a subsequent access resulting in an SLLC hit. In both scenarios, the mechanism acts upon every L2 eviction; consequently, each ART is placed between its corresponding L2 and the SLLC.

In this chapter, we further adapt ReD+ to operate within a conventional non-inclusive hierarchy. When retrieved from DRAM, blocks may or may not be stored in the SLLC depending on the content selection decision. Here, the mechanism acts on fills from main memory and is positioned adjacent to the SLLC.

4.1.3 The 2nd Cache Replacement Championship

To demonstrate that an improved content selection mechanism can deliver performance gains comparable to state-of-the-art policies in conventional SLLCs, we presented our ReD+ proposal (under the generic name "ReD") at the most recent 2nd Cache Replacement Championship (CRC2, 2017), which exclusively targeted non-inclusive SLLC policies. It achieved third place by overall score among 15 submissions, with an average performance score only 0.2% lower than the leading policy. In the multiprocessor score – the more relevant metric given the shared nature of the cache – it ranked second.

The structure of the rest of this chapter is as follows: Section 4.2 describes the operation of ReD+. Section 4.3 discusses implementation details and provides storage costs. Section 4.4 presents our internal evaluation results. Section 4.5 summarizes the independent results obtained and published by CRC2. Finally, Section 4.6 concludes the chapter.

4.2 ReD+ CONTENT SELECTION POLICY

As in ReD, the primary reuse detection is performed by per-core Address Reuse Tables (ARTs). In this non-inclusive adaptation, they are all placed adjacent to the SLLC instead of between each L2 and the SLLC, and store addresses of requests that have recently missed in the SLLC instead of blocks evicted from L2.

The goal of the ART remains the same: to detect whether blocks requested by its associated core are requested again within a time window. A request that misses both in the SLLC and the ART is marked as a candidate for bypass in the SLLC, and its address is stored in the ART. We refer to this as an initial request. A request that misses in the SLLC but hits in the ART is stored in the SLLC, as the ART hit provides a clear indication of reuse. We call this a first-reuse request. Subsequent requests are expected to hit in the SLLC.

When relying solely on the previous mechanism, as ReD does, a block with reuse experiences two SLLC misses: both the initial and the first-reuse requests miss in the SLLC. To avoid this second miss, ReD+ introduces a secondary mechanism, designed to predict reuse patterns at the time of a block's initial request. This mechanism is based on the hypothesis that a block's reuse behavior correlates with the PC of the instruction issuing the initial request, referred to as the trigger instruction. ReD+ records the reuse behavior of blocks brought in by initial requests in a table indexed by the PC of the trigger instruction, enabling it to estimate the reuse probability of a new block based on previously recorded values associated with its trigger instruction. We call this table the Program Counter - Reuse Table (PCRT). The PCRT stores two counters per entry: *#reused* and *#notreused*. To manage these counters, certain sampled sets of the ART are expanded to store the PC of the trigger instruction along with its block address.

When a first-reuse request hits in the ART, the retrieved trigger PC is used to increment the corresponding *#reused* counter in the PCRT. The requested block is fetched from main memory, sent to the L2 cache, stored in the SLLC, and its corresponding ART entry is invalidated. When a valid (and thus not reused) block is evicted from the ART, the associated trigger PC is used to increment the corresponding *#notreused* counter. The reuse probability for the trigger instruction can be computed simply as

$$\text{Reuse probability} = \frac{\#\text{reused}}{\#\text{reused}+\#\text{notreused}} \qquad (4.1)$$

Further details are provided in Section 4.3.2.

Requests marked by the ART as candidates for bypass are checked against the PCRT. If the trigger instruction has a high reuse probability, the bypass mark is disregarded, and the block is inserted into the SLLC to avoid a miss on the expected first-reuse request from the L2 cache.

To facilitate visualization of the entire mechanism, Figure 4.1 provides a schematic representation of the ART and the PCRT, illustrating their states after three requests.

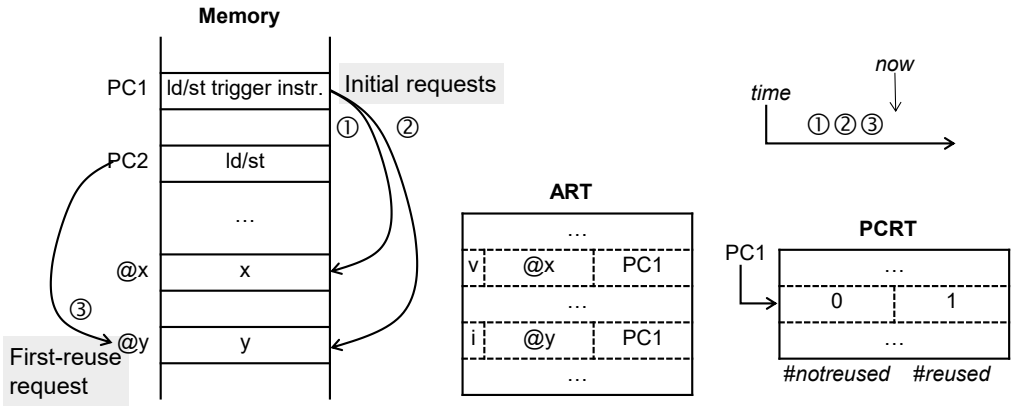

Figure 4.1: state of ReD+ internal tables after two initial requests ①②, and a first-reuse request ③. It is assumed that the ART set shown uses PC sampling.

4.3 IMPLEMENTATION DETAILS

We configure the ART and PCRT to meet the limited storage budget defined by CRC2, a constraint established to ensure a fair comparison among all submitted replacement policies. This budget is defined as 32 KB per core, representing 1.56% of an SLLC that consists of 2MB per core.

4.3.1 Address Reuse Table (ART)

The ART is organized as a set-associative buffer with 16 ways and 512 sets. As in ReD, we employ a FIFO replacement policy (requiring 4 bits per set) and, to reduce hardware costs, the ART is organized into sectors. Each entry, or sector, tracks four consecutive blocks, requiring four valid bits per entry to distinguish among them, as illustrated in Figure

4.2(a). ReD+ also uses partial address tags in the ART. The partial tag size is set to 11 bits, a value determined in our experiments to offer a good trade-off between size and performance. With this configuration, ReD+ capacity is 2MB per core.

We sample 1/4 of the ART sets to collect information for the PCRT. In each entry of these sampled ART sets, we store the PCs of the trigger instructions corresponding to the four blocks within the sector. We store only the amount of bits required to index the PCRT, not the whole PC, as illustrated in Figure 4.2(b).

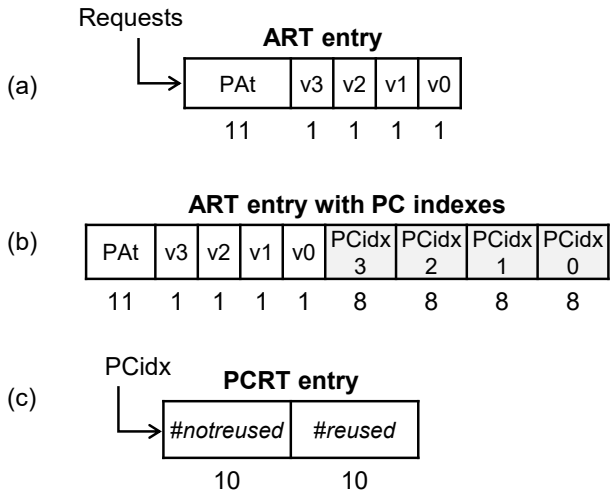

Figure 4.2: entry of the Address Reuse Table without (a) and with (b) PC sampling, respectively. Entry of the Program Counter - Reuse Table (c).

4.3.2 Program Counter Reuse Table (PCRT)

The PCRT is tagless and contains 256 entries, a value determined experimentally to provide a good trade-off between size and performance. This relatively small, tagless design is sufficient for ReD+, as the PCRT serves only as a secondary mechanism. For example, if two aliased PCs show markedly different behaviours — one with high reuse and the other with low reuse — and the PCRT categorizes their combined reuse probability as low, not all initial requests will be sent to the SLLC. Nevertheless, the ART will still correctly handle the first-reuse requests.

The PCRT is indexed using 8 bits (bits 2–9) of the trigger PC. Each PCRT entry contains two 10-bit counters (*#reused* and *#notreused*), as shown in Figure 4.2(c). When either counter reaches its maximum, both counters in the entry are halved.

The minimum reuse probability that forces all initial requests to be sent to the SLLC is set to 1/4. This value was determined experimentally and corresponds to a *#notreused* / *#reused* ratio of 3.

4.3.3 Increasing the effectiveness of the ART

The PCRT also allows identifying initial requests that are not worth keeping in the ART. Specifically, we utilize information stored in the PCRT to reduce the insertion rate of addresses in the ART in the following two cases:

- Addresses requested by a trigger instruction with very low reuse probability (less than 1/64).
- Addresses requested by a trigger instruction with high reuse probability (more than 1/4). Since ReD+ already stores all blocks requested by instructions in this category in the SLLC, keeping all their initial requests in the ART is unnecessary.

Reducing the insertion rate of these addresses allows the ART to retain other, more useful addresses for longer periods, thereby enhancing its effectiveness.

The reduced insertion rate is set to 1 out of every 8 insertions. It is not advisable to reduce this rate to zero, as a non-zero insertion rate ensures that the ART can store at least a portion of a thrashing working set. Moreover, ReD+ must occasionally insert addresses along with their associated PCs into the ART to monitor changes in trigger-instruction behavior over time.

4.3.4 Other details

ReD+ can be combined with any other SLLC replacement policy, either by adding it as a block selection policy or by substituting for the one used in the base policy. The base replacement policy considered in the results section and used for the CRC2 submission is 2-bit SRRIP (Jaleel et al., 2010b). This policy is applied upon insertion only when ReD+ decides not to bypass a block.

Write-back requests are ignored by ReD+ and SRRIP. If these requests miss, they are always allocated in the SLLC, but with the lowest priority. The simulation infrastructure does not permit bypassing them.

4.3.5 Storage costs

Table 4.1 summarizes the storage costs per core of ReD+ (ART and PCRT), plus the costs of SRRIP. The total cost remains within the 32 KB per-core budget imposed by CRC2.

ART	Parameters	512 sets, 16 ways, 4 blocks/sector
	# bits / entry	11 tag, 4 valid
	# bits / set	4 (FIFO replacement)
	Cost	512 * (16 * 15 + 4) = 124928 bits = 15616 bytes
ART sampled sets	Parameters	128 sets, 16 ways, 4 blocks/sector
	# bits / entry	4 * 8 bits PC
	Cost	128 * 16 * 32 = 65536 bits = 8192 bytes
PCRT	Parameters	256 entries
	# bits / entry	2 * 10
	Cost	256 * 20 = 5120 bits = 640 bytes
SRRIP	Parameters	2048 sets, 16 ways
	# bits / entry	2
	Cost	2048 * 16 * 2 = 65536 bits = 8192 bytes
Total cost: 15616 + 8192 + 640 + 8192 = 32640 bytes (31.875 KB)		

Table 4.1: ReD+ hardware cost, per core

4.4 EVALUATION

4.4.1 The experimental framework

For our evaluation, we use a simulation framework based on ChampSim (ChampSim, 2021), which is designed for detailed memory subsystem research and was also employed in CRC2. It is a cycle-accurate trace-based simulator that models an out-of-order multicore processor and a memory hierarchy with three levels of cache, the last of which is shared.

We consider the four configurations defined in the championship: single core without prefetching (*c1*), single core with data prefetching (*c2*), four cores without prefetching (*c3*) and four cores with data prefetching (*c4*).

For single-core configurations, we use 45 traces collected from various phases of execution from the 29 applications of the SPEC CPU 2006 benchmark suite. For multi-core configurations, we create 80 mixes by randomly combining these 45 traces.

Single-core configurations run 200 million instructions for warm-up followed by 800 million instructions for data collection. Multicore configurations execute at least 1 billion instructions per core. If the end of a trace is reached during simulation, execution continues by restarting the trace from the beginning.

4.4.2 Configuration of the baseline system

We model a base system of four superscalar processor cores with speculative out-of-order execution. Each processor has a 6-wide pipeline. A maximum of two loads and a maximum of one store can be issued every cycle. The reorder buffer has 256 entries and no scheduling restrictions. Branch prediction uses a simple *gshare* predictor (McFarling, 1993). Hardware prefetchers are disabled. All instructions have one-cycle latency except for memory accesses.

Each processing core has a two-level private cache hierarchy, the third and last level cache being shared by all the cores. The SLLC has a size of 2 MB per core and is non-inclusive. Write-back bypassing for dirty blocks is not allowed, but read bypassing is (its usage depends on the policy).

Main memory is partially modelled including data bus contention, bank contention and write-to-read bus turnaround delays. The memory read queue is out-of-order and uses a modified Open Row First-Ready First-Come-First-Serve (FR-FCFS) policy. Table 4.2 lists all the details of the memory hierarchy.

ITLB / DTLB	4 KB,16-set, 8-way
2nd level TLB unified	96 KB, 128-set, 12-way
Private cache L1 I/D	32 KB, 8-way, LRU replacement, block size of 64 B, 4 cycles of access latency
Private cache L2 unified	256 KB, includes all L1 contents, 8-way, LRU replacement, block size of 64 B, 8 cycles of access latency
Shared cache L3 (SLLC)	8 MB (2 MB for single-core), non-inclusive, 16-way, block size of 64 B, 20 cycles of access latency, 32 demand MSHR
DRAM	DRAM core access latency: 13.5 ns on row hit, 40.5 ns on row miss Two 64-bit DRAM channels (one for single-core)

Table 4.2: memory hierarchy parameters

All previously mentioned parameters mirror the setup used in CRC2 and are representative of current CMP architectures.

4.4.3 Results

Figure 4.3 shows results obtained with our proposed policy, alongside those for SRRIP as a reference. For the single-core configurations (*c1* and *c2*), we plot the speedup relative to LRU, whereas for multicore configurations (*c3* and *c4*), we plot the average speedup across all occurrences of the trace in all mixes, relative to the performance obtained with LRU. We only plot results for traces that achieve more than a 2% speedup when increasing SLLC capacity from 2 MB to 8 MB with the LRU replacement algorithm in a single-core configuration.

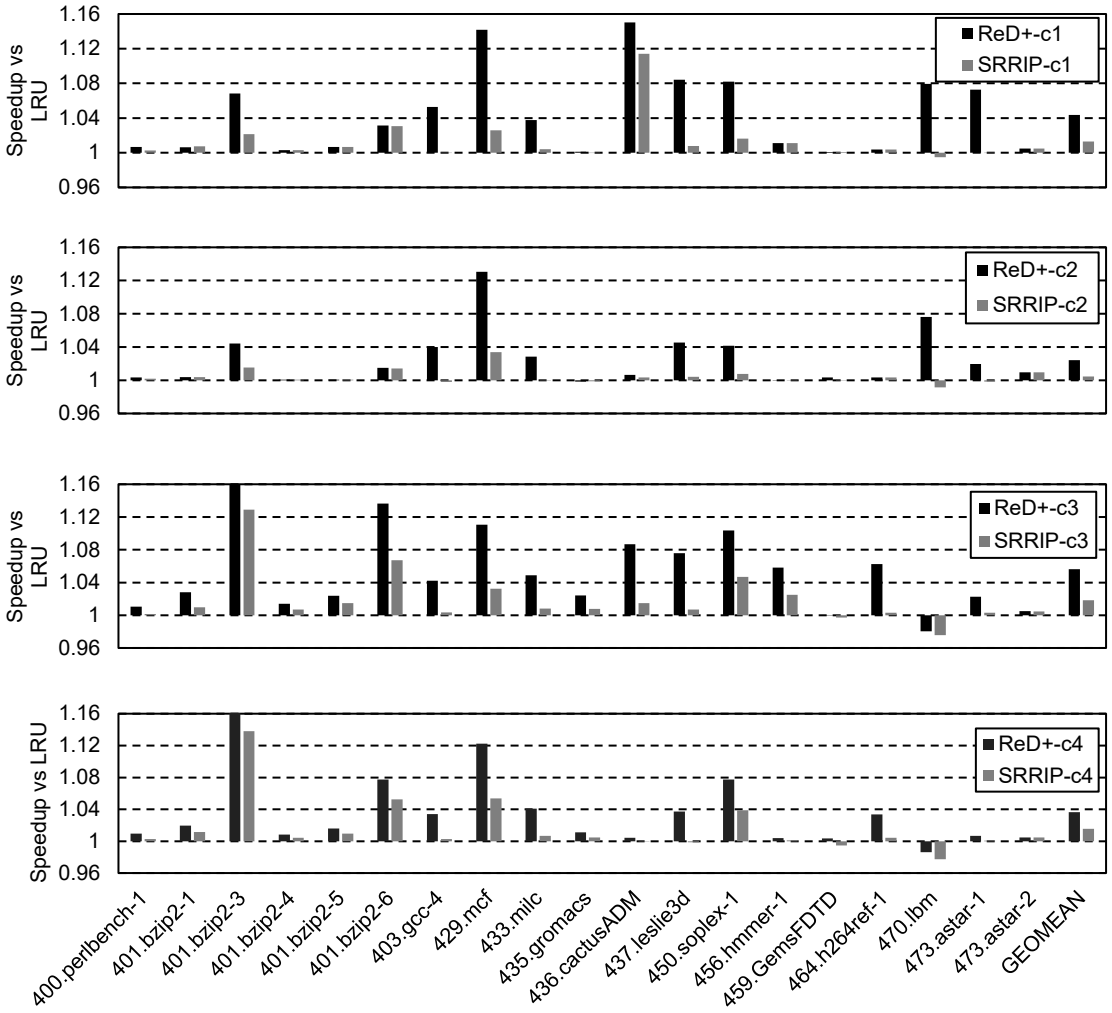

Figure 4.3: performance results. Speedup vs LRU for ReD+ and SRRIP. Results for all SPEC CPU 2006 benchmarks that show more than a 2% improvement in IPC between a 2MB and an 8MB LRU-managed LLC, and its geometric mean. From top to bottom: c1) single core without prefetching, c2) single core with data prefetching, c3) four cores without prefetching, and c4) four cores with data prefetching.

The geometric mean of speedups over all selected traces is 4.4% using configuration *c1*, 2.4% using *c2*, 5.6% using *c3* and 3.6% using *c4*. For the remaining 26 traces not plotted, the geometric mean of speedups is 0.1% in *c1*, 0.2% in *c2*, 1.5% in *c3* and 1.4% in *c4*.

Figure 4.4 shows the bypass rate using configuration *c1*. The average of the plotted traces is 32.8%, reaching a maximum of 82.1% in *429.mcf*.

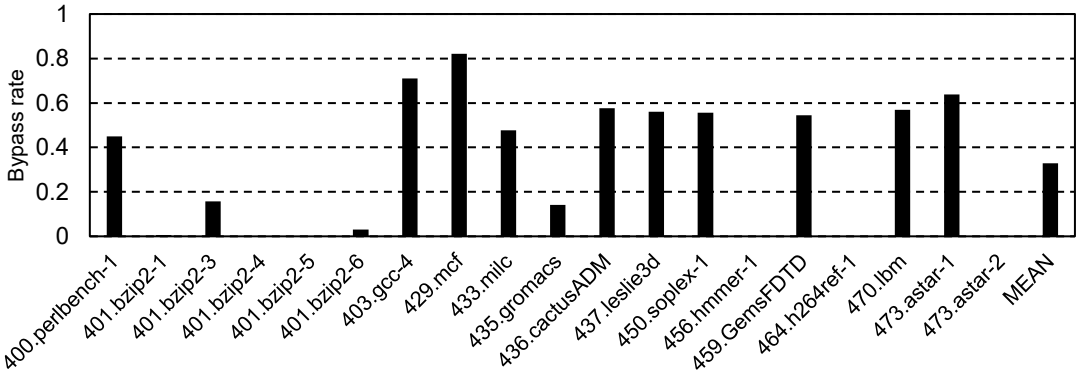

Figure 4.4: SLLC bypass rate with ReD+ for all SPEC CPU 2006 traces that show more than 2% improvement in IPC between a 2MB and an 8MB LRU-managed SLLC, using configuration *c1* (single core without prefetching).

4.5 EVALUATION AT CRC2

In this section we summarize the results obtained and published by the CRC2 organization and evaluation committee (CRC2, 2017). We focus on SHiP (Wu et al., 2011) – used by CRC2 as a reference policy for comparison –, ReD+, and the remaining policies within the top 5 by overall score: Hawkeye (Jain & Lin, 2016; Jain & Lin, 2017), SHiP++ (Young et al., 2017), Multiperspective Reuse Prediction (Jiménez & Teran, 2017), and Less is More (Wang et al., 2017). This summary enables an independent comparison of ReD+ against state-of-the-art replacement policies for conventional SLLCs.

4.5.1 The experimental framework

Both the simulation framework and the configuration of the baseline system are identical to those used in our internal evaluation. Refer to Section 4.4 for further details.

CRC2 simulates the same four configurations (*c1*-*c4*) described in the previous section but uses a different set of traces derived from the SPEC CPU 2006 benchmark suite (CRC2, 2021). Out of the 26 benchmark applications, the 20 with an SLLC MPKI greater than one are selected to ensure sufficient cache stress. For each selected program, the reference input is used, and the highest-weighted SimPoint (Perelman et al., 2003; Hamerly et al., 2005) comprising 1 billion instructions is traced. For multicore simulations, 100 multi-programmed workloads are created by randomly combining these 20 single-core traces.

Single-core configurations run 200 million instructions for warm-up followed by 1 billion instructions for data collection. Multicore configurations execute at least 1 billion instructions per core. As in our internal evaluation, if the end of a trace is reached during simulation, execution continues by restarting the trace from the beginning.

CRC2 also simulates two additional configurations with four cores using CloudSuite (Ferdman et al., 2012). Configuration *c5* runs with prefetching disabled, while configuration *c6* has prefetching enabled. Traces are generated by running CloudSuite in an execution-driven full-system simulator, intercepting all instructions and memory references within the simulator to build the trace files. Traces are collected at six sampling points per benchmark, with each sample containing at least 100 million instructions.

The simulation runs at least 100 million instructions per core. If the end of a trace is reached during simulation, the trace is restarting from the beginning and the execution continues.

Single-core simulation results are scored using the geometric mean of IPC speedups relative to the baseline LRU policy. Multicore results are scored using the geometric mean of weighted IPC speedup (Luo et al., 2001; Eyerman & Eeckhout, 2014) relative to the baseline LRU policy.

4.5.2 Results

Figure 4.5 presents the results for the six configurations and all the considered policies, along with the overall weighted score used by the organizers to rank the policies in the competition. This overall score assigns 50% weight to the SPEC CPU configurations (12.5% each) and 50% to the CloudSuite configurations (25% each).

ReD+ ranks third in overall weighted score, 0.2% below Hawkeye and 0.1% below SHiP++. Considering the average of the multicore scores — which excludes single-core results due to their limited relevance for shared cache policies — ReD+ ranks second, after Hawkeye.

Comparing ReD+ with Hawkeye, two aspects are important to contextualize these results. First, the implementation complexity of ReD+ — including the ART and the PCRT — is much lower than Hawkeye's OPTgen. Second, our proposal achieves these results by focusing solely on content selection, relying on a relatively simple SRRIP policy for insertion, promotion, and eviction. Future improvements in these aspects can be integrated

with ReD+ content selection to achieve even higher performance. In contrast, Hawkeye defines specific policies for insertion, promotion and eviction.

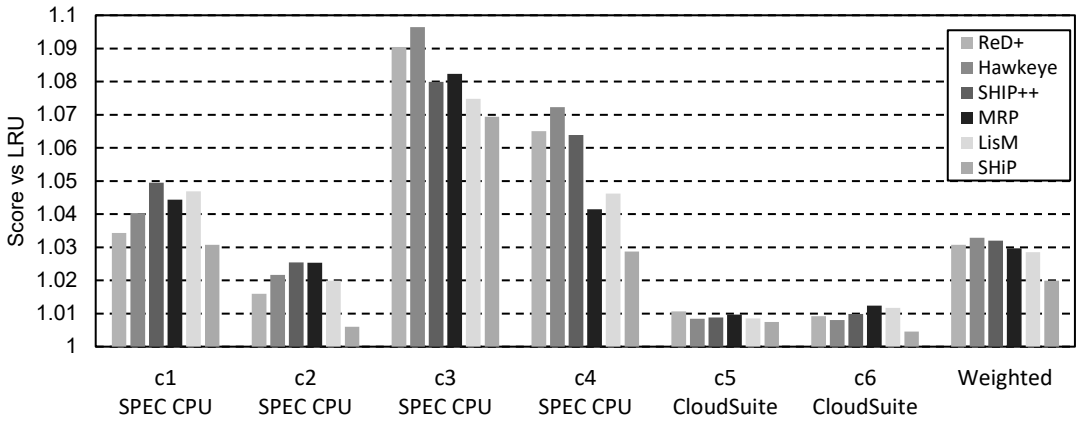

Figure 4.5: CRC2 results. Score for top benchmarks for the different configurations: *c1*–SPEC CPU 2006 single core without prefetching, *c2*–SPEC CPU 2006 single core with prefetching, *c3*–SPEC CPU 2006 four cores without prefetching, *c4*–SPEC CPU 2006 four cores with prefetching, *c5*–CloudSuite four cores without prefetching, *c6*–CloudSuite four cores with prefetching. On the right, weighted score used to rank results in the competition.

4.6 CONCLUSIONS

Based on our previous work, we design ReD+, a block selection policy that combines, in a synergistic way, two different approaches to computing the reuse likelihood of a block that misses the SLLC: a) the detection of a recent-past use of the block as an indicator of future reuse, and b) the past reuse behavior of blocks fetched by the same instruction that requests the block.

As in ReD, we design a block reuse detector, separate from the SLLC, that tracks addresses that have recently missed in the SLLC. Additionally, we include a PC-indexed store that monitors the reuse of blocks requested by each instruction and, in some cases, can predict reuse behavior. Although a similar table was used in a previous study (Wu et al., 2011), our approach differs in two key aspects. First, we train it using the reuse observed by the address detector, rather than the SLLC. Second, in ReD+, it serves as a secondary mechanism that acts only in specific scenarios: to avoid the miss on the first-reuse request and to reduce the number of insertions into the address detector. Both mechanisms are implemented in private per-core tables, to ensure a fair distribution of resources and to avoid potential thrashing caused by a single thread.

Based on our previous results with ReD, we believe that the block selection policy (that is, the decision of whether to bypass the cache or not), is the key component of the SLLC replacement policy. Enhancing the content selection mechanism can lead to substantial performance improvements that match the gains of state-of-the-art full replacement policies. These improvements are achievable not only in specialized scenarios, but also in conventional SLLCs.

We submitted our ReD+ proposal under the generic name “ReD” in the latest Cache Replacement Championship (CRC2), which exclusively evaluates replacement policies for conventional non-inclusive SLLCs. Among the 15 competing submissions, ReD+ secured third place by overall score, achieving an average performance within 0.2% of the top-ranked policy, and second in shared-cache (multiprocessor) score.

5 NEAR-OPTIMAL REPLACEMENT POLICIES FOR SHARED CACHES IN MULTICORE PROCESSORS

An optimal replacement policy that minimizes the miss rate in a private cache was proposed several decades ago. It requires knowing the future access sequence the cache will receive, and therefore can only be implemented offline in a simulated environment. No equivalent exists for shared caches because replacement decisions can alter this future sequence. We present a novel near-optimal policy for minimizing the miss rate in a shared cache that approaches the optimal execution iteratively. During each iteration, the future access sequence is reconstructed on every miss by interleaving the future per-core sequences taken from the previous iteration. This single sequence is then used to feed a classical private-cache optimum replacement policy. Our evaluation on an SLLC shows that this iterative proposal converges to a near-optimal miss rate that is independent of the initial conditions, within a margin of 0.1%. The best state-of-the-art policies that do not require future knowledge (i.e., "online" policies) achieve approximately *65% of the miss rate reduction and 75% of the IPC improvement obtained by our near-optimal proposal, relative to random replacement.*

In a shared cache, miss rate minimization does not imply the optimization of other metrics. Therefore, we also propose a new near-optimal policy to maximize fairness between cores. The best state-of-the-art online policy, our previously described ReD+, achieves 60% of the improvement in fairness seen with our near-optimal policy. Our proposals are useful both for setting upper performance bounds and inspiring implementable mechanisms for shared caches.

5.1 INTRODUCTION

Ideally, a cache replacement algorithm would ensure that every cache request results in a hit, allowing the requested block to be serviced as soon as possible. However, this is evidently not feasible in practice. The OPT cache replacement algorithm minimizes the number of misses for a specific cache organization (size, associativity, etc.) and is therefore considered the optimal policy (Mattson et al., 1970).

It is worth developing an optimal policy for several reasons:

- To determine the performance gap between proposed replacement policies and the optimum, this being the room for improvement for new policies.
- To identify and analyze specific desirable behaviours of the optimal policy, to inspire real-life implementations that mimic or approach optimal algorithms.

Unfortunately, OPT is only applicable to private caches, those that serve a single core, program, or thread. It cannot be applied to shared caches. This is because it assumes that the future sequence of cache accesses is independent of the replacement algorithm, depending solely on the processor and the memory hierarchy between it and the cache level where the algorithm operates. This assumption does not hold for shared caches.

Therefore, there is no known optimal policy that minimizes misses for shared caches. The maximum performance that can be achieved by improving the replacement policy in a shared cache is unknown.

Our goal in this chapter is to propose new algorithms for shared caches that are approximations to theoretical schemes designed to optimize certain specific metrics. We refer to this family of algorithms as near-optimal replacement algorithms (NOPTs). These algorithms can be used in the design phase when developing new replacement policies for shared caches, similarly to how OPT has been used for private caches since it was proposed.

Specifically, our contributions are the following:

- We first propose an NOPT to minimize the miss rate for shared caches. It is an iterative algorithm that is applied to consecutive runs of the same workload. Our experiments show that, after several runs, the miss rate for each of our workloads converges to a minimum. To the best of our knowledge, this is the first study that seeks to determine the theoretical minimum miss rate for shared caches and provides a successful approach.
- Secondly, having noticed that an optimal shared cache that minimizes miss rate does not benefit all programs in the workload equally, we propose an NOPT to maximize fairness between threads. Our experiments show that, with this algorithm, we obtain the best fairness of all replacement policies considered.

- Additionally, we compare the performance of current state-of-the-art policies with that of our proposals. We show that the best previous policies achieve 65% of the miss rate reduction obtained by applying our NOPT designed to minimize miss rate (vs. random), and 75% of the increase in instructions executed per cycle (IPC). Further, our results are maintained when doubling the number of cores in the system. Regarding fairness, the best state-of-the-art policy, our proposed ReD+, achieves 60% of the improvement obtained with our near-optimal policy, and the second best only 45%. The gaps increase when doubling the number of cores.

The chapter is structured as follows. Section 5.2 explains the background and motivation. Section 5.3 describes in detail the NOPT proposed to minimize miss rate. Section 5.4 details the methodology used in our evaluations, including the experimental environment and the configuration of the simulated systems. Section 5.5 evaluates the NOPT to minimize miss rate. Section 5.6 describes our NOPT proposed to maximize fairness between cores and section 5.7 evaluates this algorithm. Section 5.8 presents results for state-of-the-art online policies and compares them to our near-optimal proposals. Section 5.9 presents an analysis on how sensible our proposals are to the most important variable in a CMP, the number of cores. Finally, in section 5.10, we summarize our conclusions.

5.2 BACKGROUND

5.2.1 Replacement policies for shared caches in multicore processors

Cache replacement algorithms for processors have been evolving for more than 50 years. Most of the published work has focused on SLLCs (Jain & Lin, 2019), as the benefits of more effective algorithms are greater in these caches: it is difficult to manage SLLC contents because both temporal and spatial localities are diminished in the stream of references observed by the SLLC (Jaleel et al., 2010; Qureshi et al., 2007).

These replacement policies are based on various mechanisms. Some prioritize blocks based on their recency of access, such as LRU, SRRIP (Jaleel et al., 2010b), and PDP (Duong et al., 2012). Other use the frequency of accesses, like LFU, LFRU (Lee et al., 2001) and SAR (Lim et al., 2010). Re-reference distance is the basis for Timekeeping (Hu

et al., 2002), EHC (Vakil-Ghahani et al., 2018) and Leeway (Faldu & Grot, 2017). A mixed approach is used in Modified LRU (Wong & Baer, 2000), DIP (Qureshi et al., 2007), DRRIP (Jaleel et al., 2010b), SBAR (Qureshi et al., 2006), EVA (Beckmann & Sanchez, 2017) and ACR (Warrier et al., 2013). Recent state-of-the-art policies classify blocks as either cache-averse or cache-friendly, and manage each class differently, such as DBCP (Khan et al., 2010), AIP/LvP (Kharbutli & Solihin, 2008), CHAR (Chaudhuri et al., 2012), the Reuse Cache (Albericio et al., 2013b), SHiP (Wu et al., 2011), Hawkeye (Jain & Lin, 2016), and our ReD and ReD+ policies, presented in previous chapters.

5.2.2 Optimal policies

The studies of Belady (1966) and Mattson et al. (1970) are the first to address the issue of obtaining the absolute minimum number of cache misses produced by a known sequence of references, and they do so from different perspectives. The first proposed approach, MIN, determines whether a reference causes a miss or a hit in a cache managed with an optimal algorithm by analyzing the past sequence of references, while the second one, OPT, decides the block to be replaced to obtain the minimum miss rate by analyzing the future sequence of references. A few years later, Belady and Palermo proved that MIN and OPT provide identical results (Belady & Palermo, 1974). The optimal algorithm with bypass (OPTb), called pass-through initially, is presented more than a decade later (McFarling, 1991). OPTb is the optimal algorithm when the cache is not required to store an incoming block.

In a recent paper, Michaud (2016) develops new ways to describe the OPT and OPTb algorithms that help to improve our understanding of them and mathematically demonstrates some of their properties.

OPT and OPTb require knowledge of future memory accesses and are therefore not implementable in real systems. However, they can be evaluated offline using a simulator. Such policies are referred to as *offline policies*. In contrast, those that do not rely on future knowledge and can, in principle, be implemented in real systems are known as *online policies*.

Several papers about cache replacement policies have used OPT in recent years. Some of them use it as a reference when presenting results (Qureshi et al., 2006; Wong & Baer, 2000; Qureshi et al., 2007; Khan et al., 2010; Chaudhuri et al., 2012; Qureshi et

al., 2005). But, even when designing a policy for shared caches, its usage is limited to single-processor comparisons. Authors of a recent study explicitly state their desire to compare with optimum results for SLLCs, but rule it out as impossible (Jain & Lin, 2018).

Other authors not only use OPT for presenting results, but also study and seek to mimic its behavior in new replacement policy proposals (Kharbutli & Solihin, 2008; Jain & Lin, 2016; Lin & Reinhardt, 2002; Jeong & Dubois, 2006; Rajan & Govindarajan, 2007; Gaur et al., 2011). Again, they sometimes use single-processor OPT to guide shared cache designs, because there is no multi-core alternative.

Similarly, other offline policies have been designed to represent the ideal application of a specific mechanism (Chaudhuri et al., 2012; Liu & Yeung, 2009). These are often called “oracle” policies. They are used to guide the development of implementable online policies based on the same principles.

5.2.3 OPT and OPTb for a private cache

In a set-associative cache, OPT selects the block in the cache set that is going to be referenced the furthest into the future as a victim, and it always inserts the missing block in the cache (Mattson et al., 1970). Suppose that a 4-way set of a cache contains blocks A, B, C and D, and that the future sequence of references to blocks in this set is FACEDABFC. The OPT algorithm would choose to replace block B to make room for block F. This is because blocks A, C and D are going to be referenced before block B.

The authors of the aforementioned study demonstrated that this algorithm minimizes the miss rate in the private cache, and thereby, optimizes conventional performance metrics such as execution time and IPC. Though it cannot be implemented on a real cache because it requires knowledge of the future sequence of address references, it can be implemented offline in a simulated environment, when the sequence is known from a previous, identical run.

OPTb is the optimal policy when the cache allows bypassing. In this architecture, a requested block does not need to be inserted into the cache, it can be *bypassed*. It can be used for example in non-inclusive L2 or L3 caches (Zahran et al., 2007). OPTb is similar to OPT but it also considers the requested block as a potential victim. If it is re-referenced the furthest into the future, the incoming block is not inserted into the cache and no block is evicted. In our previous example, OPTb would bypass the incoming block F instead of

evicting B, because all the blocks present in the cache are going to be referenced before the next reference for block F.

OPT and OPTb are implemented in two phases. In the first phase, the program is simulated, and the sequence of access requests made to the cache is recorded and stored in a file. In the second phase, the simulation is run again, but this time, replacement decisions are taken using OPT/OPTb, which look for the future references in the file generated in the first phase.

Evidently, replacement decisions in the second phase may differ from those taken in the first phase. A change in a replacement decision may modify the service time for a future request due to the difference in hit and miss times, and therefore, alter the instruction execution speed of the core. This, in turn, may change the timing of future accesses to the cache. In a private cache, however, this change will not modify the sequence itself: it will still include the same block references in the same order (assuming the core executes instructions in program order).

In other words, the access sequence received by a private cache does not depend on its own behavior. Therefore, during the first phase of OPT and OPTb, the cache can use any organization and replacement policy. In fact, simulating program execution and its interaction with the memory hierarchy between the core and the studied cache would be enough.

5.2.4 Why it is not possible to implement OPT and OPTb for shared caches

A shared cache receives several streams of accesses, one per core or thread sharing the cache. The interleaving of these per-core sequences forms the global access sequence the cache receives. To form this global sequence, it is necessary to know not only the individual sequences but also their timings. As in a private cache, the replacement policy of a shared cache cannot modify the content or the order of references in each individual stream, but it can alter the relative timing between them, because hits and misses have different service times. This, in turn, can alter the stream interleaving, and therefore the global sequence itself.

Figure 5.1 illustrates, with an example, how this affects the execution of OPTb on a system with a shared cache. A cache set contains blocks A, B, X and Y. Two cores are going to access this cache set. The future access sequences are shown as S_0 and S_1

respectively. We execute phase 1 of the OPTb algorithm with two different replacement policies α and β. They result in different timings due to the different replacement decisions, and form two different global streams $S_{\alpha G}$ and $S_{\beta G}$. When running the simulation again in phase 2 of the OPTb algorithm, core 0 first accesses block A, and a replacement decision is required. When working with global stream $S_{\alpha G}$, OPTb evicts block B, while when working with global stream $S_{\beta G}$, the first evicted block is Y. Clearly, both decisions cannot be optimal simultaneously.

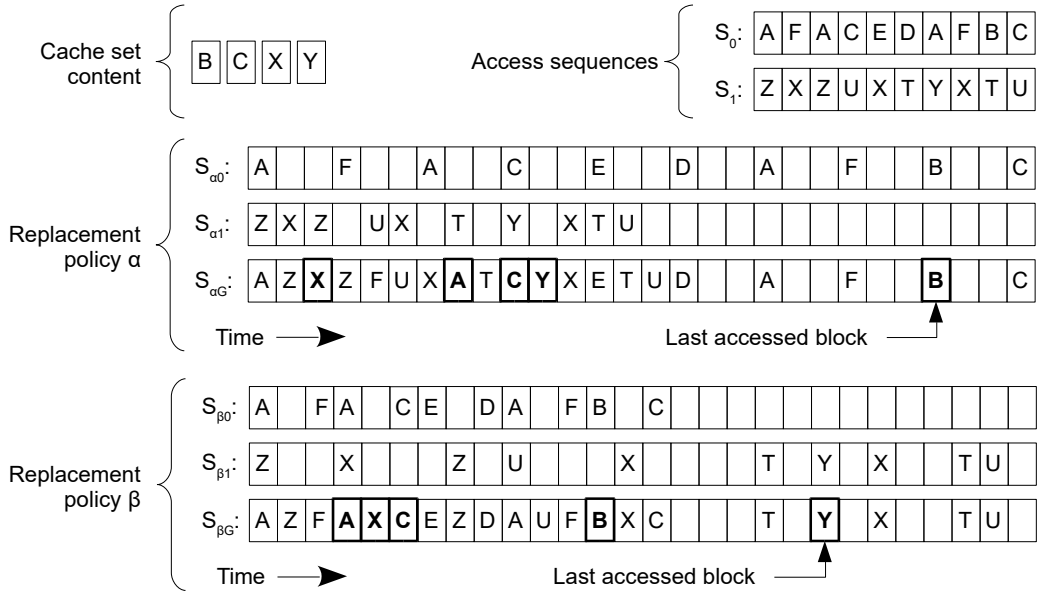

Figure 5.1: an example of the OPTb algorithm naively applied to a shared cache. Top: cache set contents and access sequences for a 4-way cache in a dual-core system. Middle: local and global streams resulting from running phase 1 of the OPTb algorithm with replacement policy α, in which B is the last accessed block from the initial cache set contents. Bottom: local and global streams resulting from replacement policy β, in which Y is the last accessed block. The difference in access timings between policy α and β appear because of the different replacement decisions. Highlighted boxes indicate first accesses to blocks considered for replacement or bypass. The last accessed block found in phase 2 of the OPTb algorithm is not the same in both cases.

OPT and OPTb require the global sequence they work with to be the same as the future stream the cache will receive. That is, the global access sequence that is captured in phase 1 must be the same as that obtained when applying the optimal algorithm in phase 2. This can only occur if, during phase 1, we already know the timings that the optimal replacement policy would generate. As they depend on block service times, which depend on whether accesses are hits or misses, and in turn, on the replacement policy, we would need to know the optimal replacement decisions during phase 1. This is impossible. Therefore, OPT and OPTb cannot be used as optimal replacement policies for shared caches.

5.2.5 Optimal replacement policies for shared caches

There are some specific characteristics of shared caches, not present in private caches, that need to be considered when developing an optimal policy for them.

First, as the global sequence the shared cache receives depends on timings, and those depend on the whole processor and memory organization, optimal replacement decisions for shared caches also depend on the timing behavior of the rest of the system. Therefore, the optimal cache replacement policy may differ from one computer organizations to another, even when the workload and the physical characteristics of the shared cache are the same.

Additionally, even the definition of optimum is not unique for a shared cache. Minimizing the miss rate of the cache does not automatically imply an optimal result in other metrics such as throughput (instructions executed in the whole system per unit of time), weighted speedup (Snavely & Tullsen, 2000) or fairness. Therefore, it is possible to propose various optimization goals for a shared cache.

In this chapter, we propose a family of algorithms — called NOPTs — that approximate optimal behavior for specific metrics in shared caches. Our proposals require knowledge of the future and therefore are offline policies, like OPT and OPTb. We apply them to SLLCs, which are currently the most common type of shared cache in CMPs[5].

The optimal algorithm we seek to approximate is OPTb. We decided to work with this variant because it obtains the minimum miss rate for a given cache organization and size. As mentioned, it requires the cache to be able to bypass incoming blocks. It is straightforward to adapt our proposals to a cache that does not allow bypassing by just taking OPT as the basis instead of OPTb.

[5] Another common use of shared caches appears in simultaneous multithreading processors, where a single core executes several threads in parallel (Eggers et al., 1997).

5.3 A NEAR-OPTIMAL REPLACEMENT ALGORITHM TO MINIMIZE MISS RATE

In this section, we propose a replacement algorithm whose goal is to achieve a miss rate close to the optimum for an SLLC. We call it NOPTb-miss, standing for near-optimal replacement algorithm with bypass to minimize miss rate.

5.3.1 NOPTb-miss design

Our proposed method for achieving a result close to the optimum in miss rate is to iteratively approach the optimal execution and the optimal global access sequence. Our hypothesis is that the global sequence obtained in an iteration with NOPTb-miss is closer to what an optimal algorithm would produce than the one obtained in the previous iteration.

We propose the following procedure:

1. Simulate the workload, using any replacement policy in the SLLC.
2. For each execution core, store its individual access stream to the SLLC in files. Include the timing of the accesses with this initial replacement policy, that is, the cycle when each request arrives at the SLLC.
3. Simulate the workload using OPTb, feeding it with a global access sequence that is dynamically constructed using the individual sequences generated and stored in the previous execution.
4. For each execution core, store its individual access stream to the SLLC in files, including the new access timings.
5. If the exit condition is not met, return to step 3 and iterate.

The files generated in steps 2 and 4 store, separated by the core and SLLC set, the list of accesses to the SLLC and the time (in cycles) between two consecutive accesses.

Figure 5.2 is a schematic diagram of the simulation of iteration *i* of NOPTb-miss (steps 3 and 4) using a cache shared by two cores.

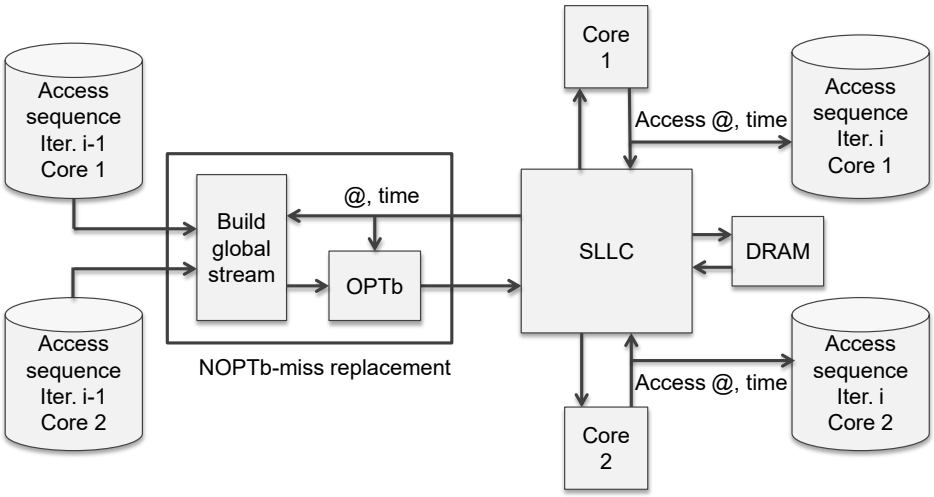

Figure 5.2: schematic diagram of the simulation of iteration *i* of NOPTb-miss.

During iteration *i*, the simulation executes the applications running in all cores, including its private caches, and generates requests made to the SLLC. In the event of a miss, the SLLC uses the NOPTb-miss replacement policy. Whenever a replacement is required, the policy reads the files created in iteration *i-1* to build the global access sequence expected in the immediate future for the affected set. It does so by interleaving the access streams of all cores, using the timing information included[6]. The replacement decision is taken using OPTb, fed by this future global access sequence, calculated for that specific time.

It is important to highlight that it is not enough to store and use a static global access sequence. Each execution changes the relative speed between cores, due to the different optimization possibilities of the various access patterns of the simulated applications. These variations accumulate over time and would quickly invalidate the information stored in a static global access sequence.

NOPTb-miss dynamically adapts to these changes. For each set, it maintains pointers to the individual access sequences indicating the last access the SLLC has received from each core. These pointers show the current execution point of each core. When a core generates more accesses per cycle than in the previous iteration due to a better miss rate, its pointers advance faster than before, meaning that the computation for that core is

[6] If two accesses from different cores arrive to the SLLC at the same time, the access from the lower-ordered core is taken first.

relatively more advanced. The global access sequence is dynamically built interleaving the individual access streams starting from the pointers.

Using this mechanism, NOPTb-miss adapts to past variations in core execution speeds and does not allow such variations to accumulate as the execution advances. A valid future global access sequence is fed into OPTb to take its replacement decisions; however, these decisions are sometimes sub-optimal because the timings used to interleave the individual streams come from a previous run. We expect the count of optimal decisions to increase as we iterate, as timings incrementally approach those obtained with the static global access sequence that would correspond to the optimal replacement.

The exit condition in step 5 depends on the goals set to achieve with NOPTb-miss and the precision required for the solution. We suggest iterating until the reduction in miss rate between iteration *i* and *i-1* is lower than a defined threshold. In section 5.5 we give details about the exit condition applied in our evaluation.

5.3.2 NOPTb-miss example

In this section, we present an example to illustrate how NOPTb-miss works in a cache shared by two cores. Figure 5.3 shows, at the top, the access sequences stored during iteration 1 for both cores and a specific cache set. They include the block address (“A”, “B”, etc.) and the timing of each access, relative to the previous one. For simplicity, we assume all accesses in the example have missed in the SLLC in iteration 1.

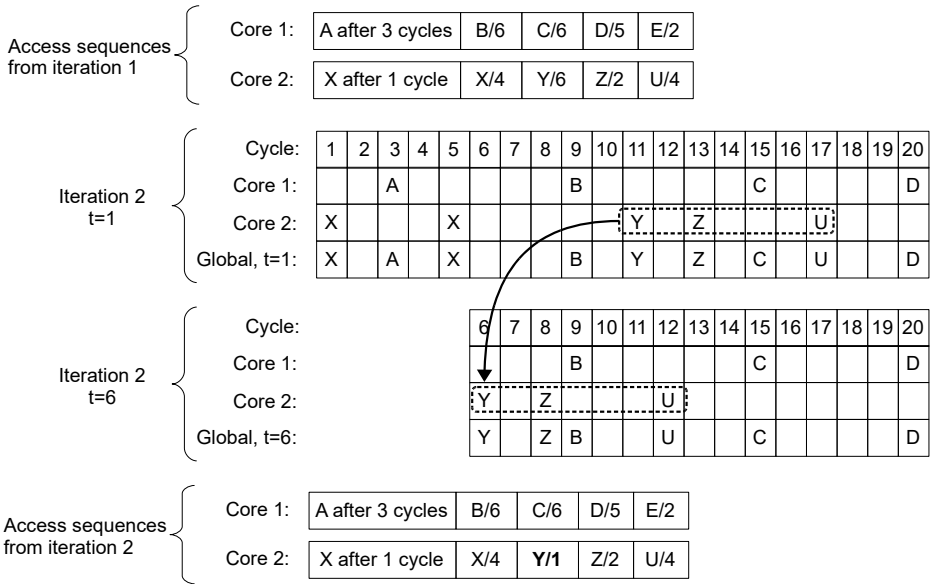

Figure 5.3: an example of how NOPTb-miss works with sequences.

During the simulation of iteration 2, at time t=1, the access to block X occurs. We assume that this access misses in the cache, and therefore NOPTb-miss is used to decide the replacement. The future global sequence is generated by interleaving the individual access sequences for the corresponding set (see “Iteration 2, t=1” in the figure). OPTb is applied to decide the replaced block based on this sequence (see “Global, t=1”). We suppose that in this case, OPTb decides to store X in the cache, a decision that in the future will convert the second access to X to a hit. As this first access to X misses as in the previous iteration, there are no changes in its service time. Therefore, future accesses will arrive as planned.

At time t=3, an access to A arrives. We assume it also misses. NOPTb-miss regenerates the global sequence from this cycle, which will coincide with the one generated at t=1 but starting in cycle 3, since there have been no changes with respect to the previous iteration.

At time t=5, the second access to X arrives, which in this case is a hit. NOPTb-miss does not apply since no replacement is required. The service time of this access to X is, however, less than that of the previous simulation, because it is served from the SLLC and not from DRAM. Therefore, core 2 experiences less delay and we assume it will be able to launch the next access to Y earlier than in the previous simulation.

At time t=6, an access to Y arrives, five cycles before the previous iteration. We assume it misses in the cache. NOPTb-miss regenerates the global sequence based on the current state (see “Iteration 2, t=6” in the figure). Note that the order of accesses in the global sequence generated at time t=6 (see “Global, t=6”) differs from that generated at time t=1 (see “Global, t=1”). This is because the speed with which the individual sequence of each core is traversed changes from one iteration to another, when accesses change from misses to hits or vice versa.

During the simulation of iteration 2, new SLLC access sequences are generated and stored in new files. Figure 5.3 shows, at the bottom, that in core 2 the distance between the second access to X and the access to Y has changed. These new sequences will be used in iteration 3.

5.3.3 Computational complexity

The computational time (*T)* required to execute NOPTb-miss depends mainly on the number of accesses of the workload that is being simulated. The order of magnitude of this figure for our simulated workloads (see section 5.4) ranges from tens of millions to hundreds of millions. It also depends on the number of cores and the cache associativity, but they are several orders of magnitude smaller and do not change unless the architecture changes. The number of iterations required is also a factor, but empirically we have found that it is a small figure (see section 5.5). We therefore consider the number of accesses of the workload as the input size of our algorithm (*n*).

The part of the algorithm that requires more computation is the construction of the global access sequence that happens on every miss. This task finishes when the first access for every block present in the set and for the incoming block are found in the future access sequences. The best-case scenario is when all accesses are found immediately. As this inexpensive task needs to be repeated on every miss, and the number of misses is proportional to the number of accesses, in this case

$$T_{best}(n) \in O(n) \tag{5.1}$$

The worst-case scenario happens when all accesses appear at the end of the individual sequences, forcing the construction of the whole global sequence on every miss. In this case, as we need to repeatedly build a list of *n*/2 items on average, we have

$$T_{worst}(n) \in O(n^2) \tag{5.2}$$

Although there is no known optimal algorithm for our scenario, it is a problem that can theoretically be solved by brute-force searching. On every miss, either one of the blocks present in the set is evicted or the incoming block is bypassed. The brute-force search would need to consider all situations, branching the simulation for every possible decision. At the end of all branches, the miss rate for every branch would be calculated and the minimum taken, which corresponds to the optimal solution. As branches are created on every miss, we have

$$T_{brute}(n) \in O(c^n) \tag{5.3}$$

where *c* is the associativity of the cache plus one, therefore greater than one.

Comparing (5.1) and (5.2) with (5.3), we can see that NOPTb-miss can be executed in quadratic time whereas the brute-force optimal solution requires exponential time.

5.4 METHODOLOGY

In this section, we describe the methodology used in the rest of the chapter for evaluation purposes. First, we present the experimental framework, in section 5.4.1, and details of our baseline system, in section 5.4.2 . Next, in section 5.4.3, we briefly describe the metrics used in our results, and in section 5.4.4 we describe other replacement policies that we use in our evaluation. Last, in section 5.4.5, we offer information to facilitate the reproduction of our results.

5.4.1 The experimental framework

For our evaluation, we use a simulation framework based on the ChampSim simulator, designed for detailed memory subsystem research. This is the same as that used in the most recent Cache Replacement Championship (CRC2, 2017).

Note that our proposals do not require this specific framework to work. They can be run on any cycle-accurate simulation engine, either based on the simulated execution of workloads or the reading of executed instructions from program traces.

We execute a multiprogrammed workload set composed of applications from the SPEC CPU 2006 suite (Henning, 2006) on a system with four cores. These are the same applications used in CRC2. The championship defines a “de facto” standard for the evaluation of cache replacement policies that we are reusing.

We have generated a set of 100 mixes, formed by random combinations of 4 benchmarks each, taken from all those in the SPEC CPU 2006 benchmark suite. Each program appears between 8 and 22 times, this corresponding to a mean of 13.8 times with a standard deviation of 4.0.

Traces for each benchmark are also taken from CRC2. For each individual program, the reference input is taken and the highest-weighted SimPoint (Perelman et al., 2003; Hamerly et al., 2005) of 1 billion instructions is traced. If during the simulation the end of a trace is reached, its execution continues from the beginning.

We first run at least 200 million instructions of each instruction trace to warm up the memory system, and then collect data for the next 800 million instructions. At the end of the data collection phase of a thread, the execution continues to put pressure on the shared resources, until data for all cores are collected.

Access streams and timings are captured during the whole execution for all cores. In each execution with NOPTb-miss, the simulation is extended beyond the data collection phase by up to 1000 million instructions, so that the access stream captured is long enough for all cores to not reach the end of it during the next execution.

5.4.2 Configuration of the baseline system

We model a base system that is the same used to evaluate ReD+ in the previous chapter. It consists of four superscalar processor cores with speculative out-of-order execution. Each processor has a 6-wide pipeline. A maximum of two loads and a maximum of one store can be issued every cycle. The reorder buffer has 256 entries and no scheduling restrictions. Branch prediction uses a simple gshare predictor (McFarling, 1993). Hardware prefetchers are disabled. All instructions have one-cycle latency except for memory accesses.

Each processor core has a two-level private cache hierarchy, the third and last level cache being shared by all the cores. The SLLC has a size of 2 MB per core and is non-inclusive. Write-back bypassing for dirty blocks is not allowed, but read bypassing is (its usage depending on the policy).

Main memory is partially modelled including data bus contention, bank contention and write-to-read bus turnaround delays. The memory read queue is out-of-order and uses a

modified Open Row First-Ready First-Come-First-Serve (FR-FCFS) policy. Table 5.1 lists all the details of the memory hierarchy.

ITLB / DTLB	4 KB,16-set, 8-way
2nd level TLB unified	96 KB, 128-set, 12-way
Private cache L1 I/D	32 KB, 8-way, LRU replacement, block size of 64 B, 4 cycles of access latency
Private cache L2 unified	256 KB, includes all L1 contents, 8-way, LRU replacement, block size of 64 B, 8 cycles of access latency
Shared cache L3 (SLLC)	8 MB (2 MB for single-core), non-inclusive, 16-way, block size of 64 B, 20 cycles of access latency, 32 demand MSHR
DRAM	DRAM core access latency: 13.5 ns on row hit, 40.5 ns on row miss Two 64-bit DRAM channels (one for single-core)

Table 5.1: memory hierarchy parameters

Some metrics are normalized to single-core execution results. In such cases, we run each of the benchmarks in the same simulation environment, changing the core count to a single core and reducing the SLLC size to a quarter of its original size.

5.4.3 Metrics

To evaluate performance, we report SLLC misses per kilo instruction (MPKI) and number of instructions executed by all cores per cycle (system IPC). The former is a performance metric for the SLLC alone and the latter is for the whole system. Unless stated otherwise, figures show the (arithmetic) mean of the results obtained for each of the 100 workloads.

To evaluate fairness, we use the M_1 (un)fairness metric defined by Kim et al. (2004). This is a pure (un)fairness metric that is independent of performance level, unlike mixed metrics such as harmonic IPC (Luo et al., 2001). For each mix, the unfairness is calculated as

$$M_1 = \sum_i \sum_j \left(\left| \frac{Miss\_shr_i}{Miss\_ded_i} - \frac{Miss\_shr_j}{Miss\_ded_j} \right| \right) \qquad (5.4)$$

for any pair i and j of cores executing in the workload (six combinations for our four cores). $Miss\_shr$ is the miss count in the shared cache scenario and $Miss\_ded$ is the miss count in the dedicated scenario, in which a program runs in a single-core setup with a

proportionally smaller SLLC (a quarter of the original size). Notice that M_1=0 means complete fairness.

5.4.4 Other replacement policies

Throughout the evaluation we use other replacement policies, either as part of the discussion or in our comparisons:

- The RANDOM policy evicts a block randomly selected among those stored in the set.
- The MISSES policy forces all blocks coming from DRAM to bypass the SLLC, and all write-backs to be allocated to a fixed way of the cache. Its goal is to force as many misses in the SLLC as possible.
- The SRRIP policy (Jaleel et al., 2010b) utilizes the concept of a re-reference interval prediction value (RRPV) for a cache block. The version we use stores this value in two bits. On insertion, a block gets assigned a *long* re-reference interval, with RRPV=2. If the block receives a hit, it is promoted to a *near-immediate* re-reference interval, with RRPV=0. In case an eviction is required, the lowest-order block with *distant* re-reference interval, RRPV=3, is selected. If there is no block like that in the cache set, the RRPV of all blocks is incremented, and the victim selection restarts. In summary, SRRIP assigns lower RRPV, and therefore less likelihood of replacement, to blocks that have been recently re-referenced.
- The LRU policy (Least Recently Used) maintains the stack of blocks in a set ordered by its recency of usage. It selects the least recently requested block for eviction.
- The SHiP++ policy (Young et al., 2017) is an evolution of SHiP-PC (Wu et al., 2011). SHiP-PC tries to predict the reuse characteristics of a line based on a signature of the line, which is defined by the PC of the instruction that caused the miss. A data structure called the Signature Hit Predictor stores SLLC hit/miss data correlated to this signature. SHiP leverages the RRPV concept of SRRIP but it assigns values on insertion in a different way: a block is inserted with *long* re-reference interval only if other blocks associated with its same

signature have previously hit in the SLLC, and with *distant* interval if not. The policy mirrors SRRIP in other areas. SHiP++ extends SHiP by improving the management of the Signature Hit Predictor and making the policy prefetch and writeback-aware.

- The Hawkeye policy (Jain & Lin, 2016; Jain & Lin, 2017) learns from Belady's MIN algorithm (Belady, 1966) by applying it to past cache accesses to inform future cache replacement decisions. It introduces a new method of efficiently simulating Belady's behavior on past accesses and stores the expected hit or miss behavior of a block correlated with the PC of the instruction that requested the access. Thereby PCs are classified as cache-friendly or cache-averse and this information is used to assign the replacement priority in the SLLC.
- Our ReD+ policy, presented in Chapter 4.

5.4.5 Reproducibility

All resources required to reproduce the evaluation in this chapter are available in a public repository (Díaz, 2021). These include:

- The version of the ChampSim simulator we use (ChampSim, 2021). The original version is continuously evolving.
- The composition of our workload set.
- Pseudo-code for NOPTb-miss.
- The source code for NOPTb-miss, NOPTb-fair and the other policies with which they are compared.
- Sample execution scripts.

Additionally, the traces we use are also publicly available (CRC2, 2021).

5.5 NOPTB-MISS EVALUATION

In this section, we first study NOPTb-miss convergence, showing the evolution of the SLLC miss rate as more iterations are run. Next, we analyze how near our proposal is to the optimum. Finally, we present results for each application in our workload.

5.5.1 Convergence analysis

In step 1, NOPTb-miss can model an SLLC with any replacement policy. To validate our design and hypothesis in different circumstances, we select three different initial SLLC policies: SRRIP (Jaleel et al., 2010b), RANDOM and a policy that misses in the SLLC as much as possible, which we call “MISSES”. These three initial policies are used in three isolated sets of experiments.

Our proposal is based on the hypothesis that the global sequence obtained in an iteration with NOPTb-miss is closer to what an optimal algorithm would produce than the one obtained in the previous iteration. This implies that the miss rate is reduced in each iteration. This hypothesis can be verified by measuring the miss rate, which should decrease asymptotically in each iteration, heading towards the optimum value. This value is expected to be independent of the starting replacement policy.

Figure 5.4 shows the mean SLLC miss rate for successive iterations of the simulation of NOPTb-miss, for all three initial replacement policies. Iteration 0 corresponds to step 1 of the algorithm, while the rest correspond to step 3. The chart on the right is a zoomed view of the chart on the left, excluding iteration 0 from the X-axis and narrowing the Y-axis range. Both charts display the same data for iterations 1 through 4. As shown on the left, miss rates converge rapidly, as soon as the first iteration, to a very similar value regardless of the replacement policy used in iteration 0. On the right, we can see how successive iterations improve the miss rate convergently. The miss rates in iteration 4 differ by less than 0.1% between the policies.

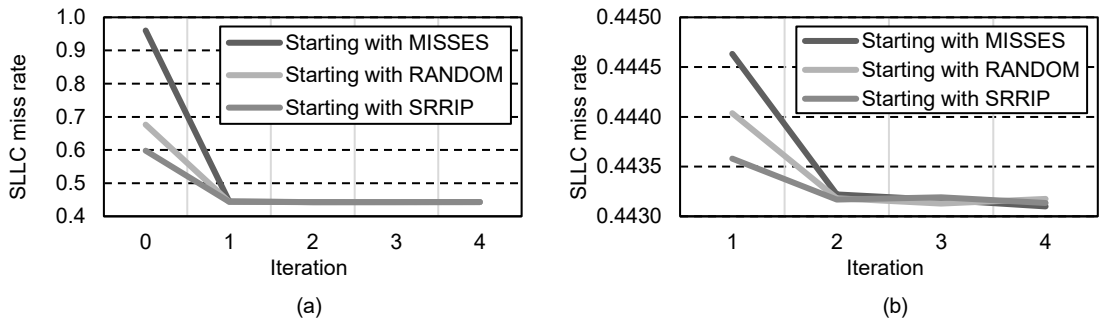

Figure 5.4: mean SLLC miss rate for step 1 of NOPTb-miss (labelled as iteration 0) and several iterations of step 3. The two graphs show the same values for iterations 1-4 but using different scales; (a) shows all iterations while (b) is a zoom showing the iterations in which the convergence can be seen most clearly.

The aforementioned values are means of the rates with the different 100 workloads. Figure 5.5 shows, for each workload, the maximum difference in miss rate obtained in iteration 4 between any two of the three experiments. In the worst case, the miss rates differ by 0.50%. In 96 of the 100 workloads, the difference is less than 0.1%.

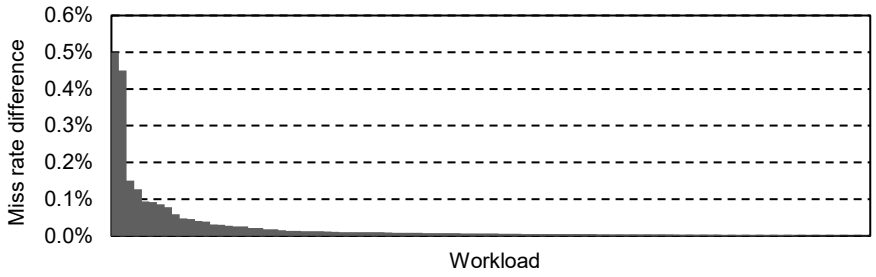

Figure 5.5: maximum relative difference in miss rate in iteration 4 of NOPTb-miss when starting with MISSES, RANDOM and SRRIP as replacement policies in step 1 (iteration 0).

In our experiments, we have stopped our simulations in iteration 4 since the results are very similar to those obtained in iteration 3. In 55% of workloads and experiments, the miss rate is lower than in iteration 3, and in the rest it is higher. In iteration 4 and later (not shown), miss rates no longer exhibit systematic improvement, but fluctuate slightly. According to our analysis, the main reason for this is that the simulation framework models an out-of-order processor. The unordered execution sometimes results in accesses from the same core in successive iterations not occurring in strictly the same order. The most common case is that two consecutive accesses exchange their order in the sequence. These accesses must be very close in time. Since the out-of-order model may slightly alter the order of closely spaced instructions between iterations, the individual access sequences are not completely reliable, making the OPTb policy generate sub-optimal replacements, which cause small variations in the result.

In summary, NOPTb-miss converges quickly, after a few iterations, to a miss rate that differs less than 0.1% for all the various initial replacement policies selected for step 1. Where not stated otherwise, in the rest of this chapter, we present the results obtained in iteration 4 when using SRRIP as the initial policy.

Although our experimental results show that the algorithm converges for all 100 mixes in our workload, a more thorough theoretical analysis would be required to mathematically

guarantee convergence in all cases. Such an analysis, while interesting, falls outside the scope of the present work.

5.5.2 How close is NOPTb-miss to the optimum?

Since there is no optimal policy for miss rate for shared caches, we believe it is not possible to mathematically demonstrate that NOPTb-miss provides the same results that such a policy would produce. That said, there is strong evidence that NOPTb-miss obtains results that, within the indicated margins, are indistinguishable from such a theoretical policy.

On the one hand, as we mentioned at the beginning, a theoretical way to minimize the miss rate in the SLLC would be to apply OPTb using the global sequence resulting from the execution of the workload using OPTb as a replacement algorithm. From this point of view, the optimal algorithm can be understood as the one that generates results and a global sequence such that, when applying OPTb again, these results are not modified. As we have seen, after several iterations of NOPTb-miss the results show very little variation, which in our opinion indicates that it is very close to the optimum.

On the other hand, it could be argued that with NOPTb-miss, in some cases, the algorithm could converge to a local minimum and fail to escape it through further iterations. We believe that the convergence towards a very similar miss rate starting with three very different replacement policies makes this very unlikely. In addition, we have carried out experiments introducing variations in the timing of the access sequences. With this, the goal is to introduce disturbances in input data in different directions, to force the algorithm out of a hypothetical local minimum. Specifically, for a subset of the workloads, we run multiple experiments in iteration 5, where the times between consecutive addresses in three of the cores during the whole simulation were multiplied by 0.5, 1, 3 or 9, while those of the other core were left unchanged. All 175 combinations were tested for each workload in the subset. The results in MPKI and IPC worsened for that iteration, as replacement decisions were misinformed and therefore not optimal, but quickly returned in iteration 6 to values within the same ranges as those obtained before the disturbance. These results provide strong evidence that the algorithm does not converge to local minima.

5.5.3 Results per application

In this section, we compare the results obtained by each of the applications when using NOPTb-miss as the replacement policy. Figure 5.6 shows, on the top chart, the normalized MPKI obtained with NOPTb-miss for all individual benchmarks. Values are normalized to the MPKI obtained when executing each application in a single-core system using OPTb as the replacement policy, with an SLLC reduced in size proportionally (from 8 MB for four cores to 2 MB for a single core). The vertical bar shows the range of results for all instances where the application appears in our workload set and the horizontal bar crossing the vertical one indicates the mean. Applications are ranked by the number of requests for accesses per kilocycle (APKC) they make to the SLLC, shown on the bottom chart.

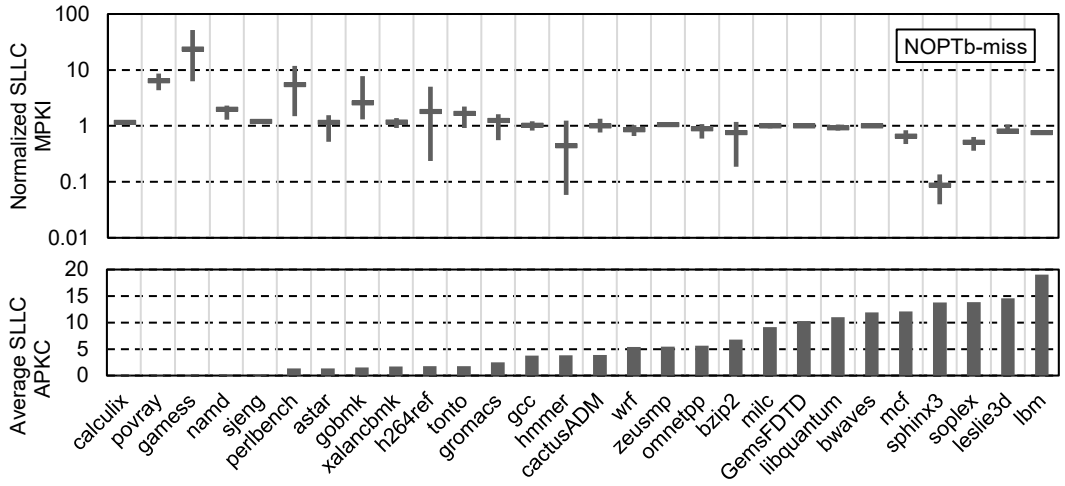

Figure 5.6. Bottom: mean number of accesses to the SLLC per kilocycle (APKC) for the SPEC CPU applications that compose our workload set, ranked by value. Top: MPKI at the SLLC for NOPTb-miss, normalized to the value obtained when executing each application in a single-core system with a proportionally reduced SLLC size and OPTb. The vertical bar shows the range of results for all instances where the application appears in our workload set. The horizontal bar shows the mean.

The figure shows that the applications on the right, those with the highest rate of access to the SLLC, generally achieve MPKI reductions when running alongside others in a shared cache. Conversely, the applications on the left, those with lower access rates, experience an increase in miss rate compared to that in their optimal solo execution. This result can be explained by the characteristics of the policy. To minimize miss rate, NOPTb-miss attempts to reconstruct the global sequence of references as it will occur in the future. Cores that launch more requests per cycle place relatively more blocks in that global sequence than those that launch fewer. As an optimal global miss rate is achieved

retaining blocks accessed soonest in the future, and those are mostly from applications with high access rate, cores that access the cache less frequently have less priority, this meaning that NOPTb-miss does not favor them.

A fair replacement for a shared cache should provide similar MPKI reduction to all applications. Therefore, we can conclude that NOPTb-miss does not treat applications fairly. Specifically, it seems that NOPTb-miss at least partially sacrifices fair service between cores to minimize the global miss rate of the shared cache.

5.6 A NEAR-OPTIMAL REPLACEMENT ALGORITHM TO MAXIMIZE FAIRNESS

As mentioned before, there is no single definition of an optimal replacement policy in a shared cache because the optimization of the total number of cache misses does not imply the optimization of other metrics such as global throughput (total number of instructions executed) or fairness between cores.

We can, therefore, propose the design of a NOPT with a different objective, seeking to approximate a different optimum. Considering the analysis presented in the previous section, our new objective is to treat all cores fairly, improving fairness among them. We call our proposed algorithm NOPTb-fair, standing for near-optimal replacement algorithm with bypass to maximize fairness.

Fairness itself is a desirable property but it is not a common main goal in real-world design. With this design and analysis, we aim to explore how fairness affects other metrics such as throughput, to show the potential shortcomings that may appear with a policy only looking to minimize miss rate and to ultimately help achieve a better balance between throughput and fairness in real-life policies.

5.6.1 NOPTb-fair design

NOPTb-miss decides the replacement based on the time it will take to reference each block in the global sequence, regardless of which core will reference it. Nonetheless, for a particular core, only the blocks that it will reference in the future and their order of reference are relevant. That is, its own sequence of future references is the relevant one, regardless of what the other cores do.

Let us suppose that, at some point, the first references that appear in the future sequences of cores C1 and C2 are for blocks B1 and B2 respectively. The most important block for core C1 is B1, and, similarly, that for C2 is B2, and this does not depend on whether B1 is accessed later than B2 or vice versa. Our proposal is based on respecting these individual priorities of each core.

We define the future reuse distance (FRD) of a block B, used by a particular core, as the number of accesses that appear in the individual access sequence of that core from the current time until the next reference to block B. FRD indicates the relative importance of each block to the core that uses it: the lower the FRD, the more important it is to keep the block in the SLLC for the core that uses it. Our proposal for NOPTb-fair replaces the block with the largest FRD of any of the cores. Therefore, it determines the priority of each block from the point of view of the core that uses it, instead of from that of the shared cache, and replaces the least important of them. It is interesting to note that OPTb for a single core also replaces the block with the largest FRD.

NOPTb-fair follows this procedure:

1. Simulate the workload, using any replacement policy in the SLLC.
2. For each execution core, store its individual access stream to the SLLC in files. Timing information is not required.
3. Simulate the workload using a replacement policy that evicts the block with the largest FRD, also considering the possibility of a bypass. This policy is fed with the individual access sequences captured in the previous step. It also uses pointers in each sequence to track the last access received in the SLLC from each core.

The result does not depend on the policy used in the initial step — within the limitations caused by the simulation of out-of-order processors (see Section 5.5.1) — and no iterative process is required. This is because the SLLC replacement policy affects the timing, but not the list or order of accesses of each core, which is the only information used by NOPTb-fair.

A complementary view on NOPTb-fair appears when we realize that its results are equivalent to applying OPTb on a global sequence constructed with the (unrealistic) assumption that the SLLC will receive the same number of accesses per cycle from each core, instead of using the actual expected timings as NOPTb-miss does.

5.6.2 NOPTb-fair example

In this section, we illustrate the operation of NOPTb-fair with an example of a cache shared by two cores. Figure 5.7 shows, for a specific time, the content of a cache set and the sequences of future accesses of the two cores to that cache set. These sequences have been obtained in a previous simulation and have no temporal information. Therefore, they could have been generated using any replacement algorithm in the SLLC.

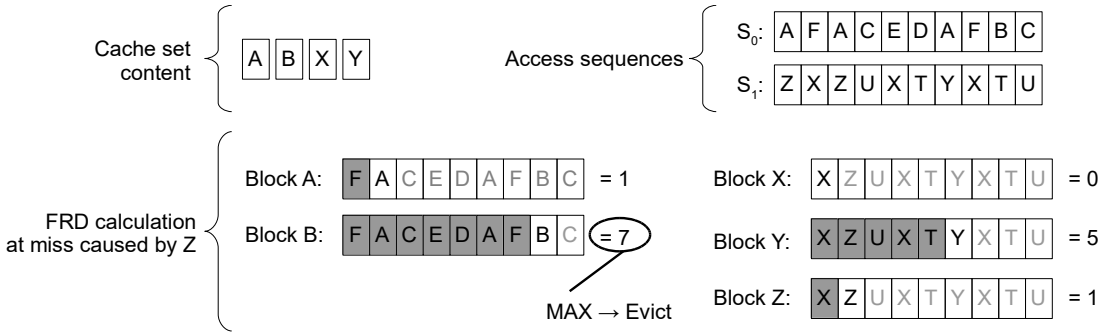

Figure 5.7: an example of the victim selection procedure of NOPTb-fair. Top: cache set contents and access sequences for a 4-way cache in a dual-core system. Bottom: FRD calculation at miss caused by Z, assuming A hits before. Each FRD calculation searches only in the sequence of the affected core. Block B has the largest FRD (7), and is therefore selected for eviction.

Suppose that the first access to reach the SLLC is to block A from core 0. It is a hit and therefore there is no replacement action. Then, the access to Z from core 1 arrives at the SLLC, this causing a miss and therefore requiring a replacement action. At that time, NOPTb-fair calculates FRD for each of the blocks in the set and the new block Z. FRD for block B is 7 since core 0 will access blocks F, A, C, E, D, A and F before block B. Similarly, the FRD is 1 for A (block F), 0 for X, 5 for Y (blocks X, Z, U, X and T), and finally, 1 for Z (block X). The largest FRD among the five candidate blocks is that of block B, which will therefore be evicted to make room for Z.

5.7 NOPTB-FAIR EVALUATION

In this section, we first present the per-application performance results for NOPTb-fair. Next, we analyze how near our proposal is to the optimum. Finally, we compare results for NOPTb-fair and NOPTb-miss per workload.

5.7.1 Results per application

Figure 5.8 shows the normalized MPKI obtained with NOPTb-fair for all individual benchmarks. The chart organization is the same as in Figure 5.6 (above). Comparing the figures, we can see that for NOPTb-fair there is no obvious correlation between the normalized SLLC MPKI and the SLLC APKC. With NOPTb-fair, only two applications (*perlbench* and *gobmk*) show a significantly higher mean MPKI than the single-core optimal execution, and the increase is much smaller than that observed with NOPTb-miss. These results confirm that NOPTb-fair achieves a substantially fairer outcome than NOPTb-miss, as evidenced by the lower variation in MPKI across applications.

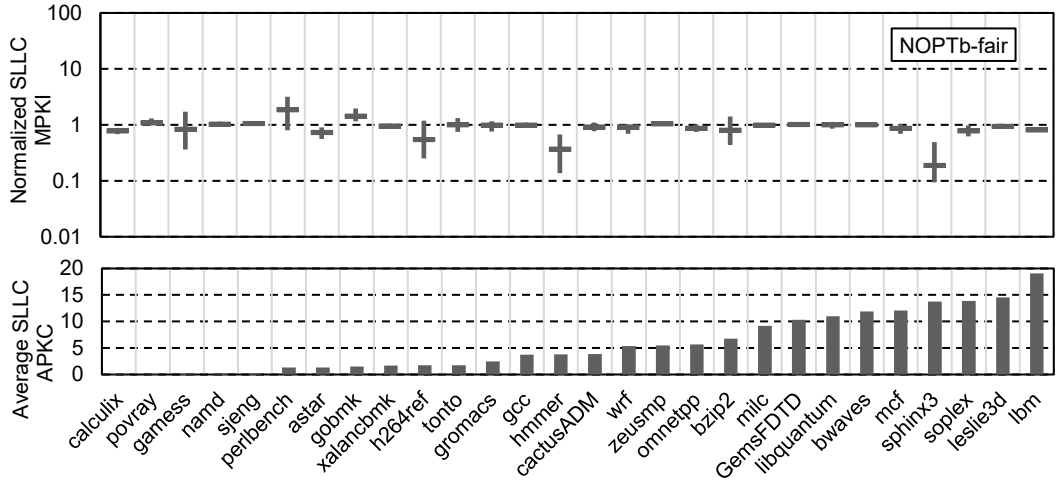

Figure 5.8. Bottom: mean number of accesses to the SLLC per kilocycle (APKC) for all applications in our workload set, ranked by value. Top: MPKI at the SLLC for NOPTb-fair for the same applications, normalized to the value obtained when executing each application in a single-core system with a proportionally reduced SLLC size. The vertical bar shows the range of results for all instances where the application appears in our workload set. The horizontal bar shows the mean.

5.7.2 How close is NOPTb-fair to the optimum?

To the best of our knowledge, there is no previous proposal for a replacement algorithm that optimizes fairness between cores in an SLLC. Therefore, we cannot compare the NOPTb-fair design or results against other optimal or near-optimal proposals.

In Section 5.8, we compare NOPTb-fair fairness results with those from other replacement policies. NOPTb-fair obtains a better fairness metric than any other policy. Beyond this, and unlike the case of NOPTb-miss, we are not able to provide strong evidence that its results are optimal or near optimal. We consider NOPTb-fair the best-known approximation to the theoretical optimal fairness policy to date.

5.7.3 Comparison between NOPTb-miss and NOPTb-fair at workload level

Having two policies whose optimization goals are different allows us to compare them and study how the goals affect the final performance of the system. In this section, we compare NOPTb-miss and NOPTb-fair, and analyze their performance for each workload.

Figure 5.9 shows a comparison of NOPTb-miss and NOPTb-fair at workload level. The chart on the left shows the differences in normalized SLLC MPKI between the two policies for each of the workloads executed, ranked by this difference in descending order. Positive values mean that NOPTb-miss misses less often (performs better) than NOPTb-fair. As expected, NOPTb-miss reduces the SLLC MPKI more than NOPTb-fair for all workloads in our set.

The chart on the right shows the differences in normalized IPC, with workloads ranked by this difference in descending order. Positive values mean that NOPTb-miss increases IPC more (performs better) than NOPTb-fair. We can see that the higher reductions in MPKI observed with NOPTb-miss do not always translate into a better throughput: 34 out of the 100 multiprogrammed workloads show better normalized IPC for NOPTb-fair than for NOPTb-miss.

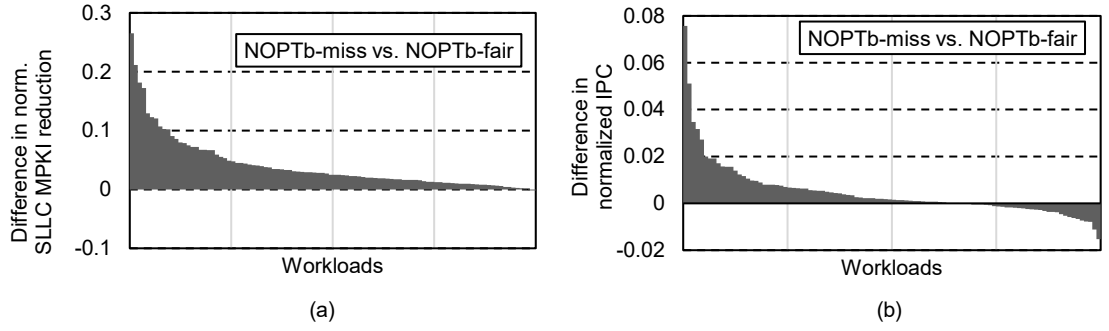

Figure 5.9: (a) difference in normalized SLLC MPKI reduction and (b) difference in normalized IPC between NOPTb-miss and NOPTb-fair for all workloads. A positive value indicates that NOPTb-miss achieves a better result in the corresponding metric. Workloads are ranked by value in descendent order (i.e., the workload order differs between the two charts).

It seems counterintuitive that the policy optimized for fairness should achieve better IPC than the one that yields a better miss rate, as is observed in several workloads. Although we have already noted in our introduction that, in shared caches, optimizing miss rate does

not imply optimizing other performance metrics, it is not obvious why greater fairness can, in some cases, result in higher system throughput.

Applications with fewer accesses to the SLLC tend to have higher IPC than applications that have more accesses. The former either require less data or rely more heavily on private caches, which are faster than the SLLC and significantly faster than main memory. Although the penalty for an SLLC miss is similar in all cases when measured in cycles, the equivalent number of instructions that cannot be executed during the miss penalty is higher in applications with low SLLC APKC. In other words, eliminating a miss in a low-APKC/high-IPC application increases the number of instructions executed per cycle more than eliminating a miss in a high-APKC/low-IPC application.

As NOPTb-miss favors applications with higher APKC and lower IPC, it reduces miss rate but does not increase throughput by the same proportion. This mechanism explains why, in certain workloads, the balanced behavior of NOPTb-fair achieves better global IPC than NOPTb-miss.

5.8 COMPARISON WITH STATE-OF-THE-ART POLICIES

One of the utilities of having an optimal or near-optimal policy is to compare it against the best existing online policies, which do not require knowledge of the future. In Sections 5.8.1 and 5.8.2, we compare the results of various online replacement policies, in terms of performance and fairness respectively, against the two offline near-optimal algorithms presented previously. Following this analysis, Section 5.8.3 proposes new design guidelines for future replacement algorithms.

5.8.1 MPKI and throughput comparison with state-of-the-art policies

The left chart of Figure 5.10 shows the percentage of MPKI reduction achieved by several replacement policies when taking random replacement as the baseline. We consider two basic policies, LRU and SRRIP (Jaleel et al., 2010b), along with the three best ones from the CRC2 competition: ReD+, SHiP++ (Wu et al., 2011; Young et al., 2017), and Hawkeye (Jain & Lin 2016; Jain & Lin 2017). Additionally, we include NOPTb-miss and NOPTb-fair.

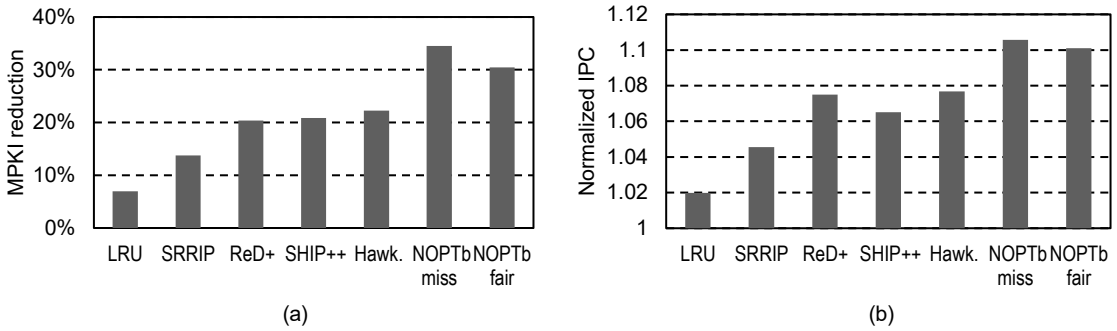

Figure 5.10: (a) MPKI reduction in the SLLC and (b) normalized IPC, relative to random replacement.

On average, NOPTb-miss results in an MPKI 35% lower than that obtained with random replacement. The winning algorithms from the CRC2 provide an MPKI between 20 and 22% lower than that obtained with random replacement. The differences between them are small, despite the significant differences in their design approaches. The reduction obtained by the three CRC2 winners corresponds to around 65% of the reduction obtained by NOPTb-miss.

Figure 5.10 also shows, on the right, the normalized IPC (versus random) with those same policies. The result is similar to that obtained for MPKI. On average, with NOPTb-miss, the IPC is 10.6% higher than with random replacement, while the winning algorithms of the last replacement competition achieved an IPC between 6.5 and 7.7% higher than that observed with random replacement.

This means that the best policies have already achieved around 75% of the maximum improvement possible in IPC. This average figure should not discourage further research on cache replacement since, on the one hand, significant improvements may be obtained for some applications and, on the other, the relative importance of cache memory tends to increase over time (Wulf & McKee, 1995).

5.8.2 Fairness comparison with state-of-the-art policies

In this section, we present the results of state-of-the-art policies, together with those of NOPTb-miss and NOPTb-fair, focusing on fairness. Figure 5.11 shows the unfairness score (see Section 5.4.3) for the same set of SLLC replacement policies as before.

Figure 5.11: unfairness for various SLLC replacement policies. Lower values indicate greater fairness.

As expected, NOPTb-fair achieves the lowest and therefore best value, specifically, an unfairness 85% lower than that achieved with NOPTb-miss.

Among the online policies, Random and LRU result in the worst fairness. This is because they do not use any mechanism to prevent an application with a high access rate from taking up excessive space in the SLLC. The worst case occurs with applications that heavily access with a scanning or thrashing pattern (Jaleel et al., 2010b), as they do not benefit from the SLLC and hinder others from doing so.

The next policies, in the ranking by unfairness, are SRRIP and SHiP++, which show intermediate values. Both have mechanisms to avoid problems with scanning or thrashing patterns; specifically, they do not allow blocks to have high priorities unless they detect reuse in the SLLC. Nonetheless, they do not prevent some applications that access heavily from taking up excessive space. Next in the ranking is Hawkeye which, imitating OPT, avoids thrashing and scanning, and respects the space of each application if that proves useful in reducing the overall miss rate.

The best result among online policies is obtained by ReD+. On the one hand, it protects against scanning and thrashing because it bypasses blocks without reuse. On the other, it protects applications with a relatively lower access rate because it uses a private reuse detector for each core, separated from the SLLC.

5.8.3 Balancing miss-rate reduction and fairness in SLLC replacement policies

Any design that attempts to minimize miss rate may resort to favoring specific applications or application types that are “easy targets”, thereby resulting in unfair outcomes. As we have seen in Section 5.7.3, a fair replacement policy like NOPTb-fair manages to convert the reductions in miss ratios to IPC increases better than an unfair

policy like NOPTb-miss. On the other hand, Figure 5.11 shows that state-of-the-art replacement policies achieve even worse results in terms of fairness than NOPTb-miss These findings suggest that fairness also affects throughput for state-of-the-art policies. If this is the case, the same effect of an advantage in IPC associated with better fairness should be also noticeable comparing two of these policies.

Figure 5.10 shows that MPKI reduction is 20.4% for ReD+ and 20.9% for SHiP++, higher for SHiP++, while the normalized IPC is 1.075 for ReD+ and 1.065 for SHiP++, higher for ReD+. This aligns with the higher unfairness obtained by SHiP++ shown in Figure 5.11: 22.5 with SHiP++ and 10.6 with ReD+. Similar trends are seen comparing other policies, like Hawkeye with SHiP++ and with ReD+.

We conclude that, in shared caches running heterogeneous workloads, higher system IPC values are obtained when considering fairness together with miss rate reduction when designing the replacement policy. As inter-core fairness is a concept that has no meaning in single-core systems, this conclusion suggests a replacement policy designed for private caches should not be transferred directly to a shared cache, even if the private cache is a last-level cache. Such an approach might result in a policy with relatively high unfairness between cores, that benefits only a subset of applications and fails to produce an average throughput improvement commensurate with the observed miss rate reduction. A quantitative analysis of the policy's fairness and per-application performance, using NOPTb-fair as a reference, may provide hints on how to improve the design to avoid such problems.

5.9 SENSITIVITY ANALYSIS

As these algorithms target shared caches in multicore processors, the core count becomes a critical parameter of the setup. In this section, we evaluate the sensitivity of these algorithms to the number of cores in the system by using eight cores, twice the number used before (octa-core vs quad-core). The size of the SLLC has also been doubled to 16 MB. Our workload consists of 10 mixes, a smaller set due to limited computational resources.

First, we analyze the convergence of NOPTb-miss using this new setup. Then, we present performance results for NOPTb-miss and NOPTb-fair and compare them with online policies. Finally, we report fairness results.

5.9.1 Convergence analysis of NOPTb-miss

Figure 5.12 shows the mean SLLC miss rate after several iterations of NOPTb-miss, with the octa-core setup. Comparing these results with the quad-core results in Figure 5.4, we see that the convergence is slightly slower in iteration 1, but results are similar after iteration 3 (<0.1% variation).

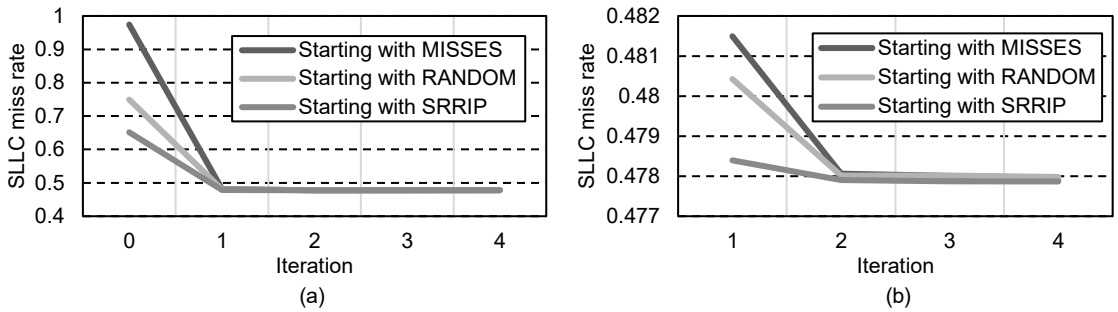

Figure 5.12: mean SLLC miss rate for step 1 of NOPTb-miss (called iteration 0) and several iterations of step 3, using an octa-core setup. The two charts show the same values for iterations 1-4 but using different scales; (a) shows all iterations while (b) is a zoom showing the iterations in which the convergence can be seen most clearly.

We conclude that the number of cores does not significantly affect the convergence of NOPTb-miss.

5.9.2 MPKI and throughput

Figure 5.13 shows the percentage of MPKI reduction and the normalized IPC achieved by our selected set of replacement policies when taking random replacement as the baseline, using the octa-core setup.

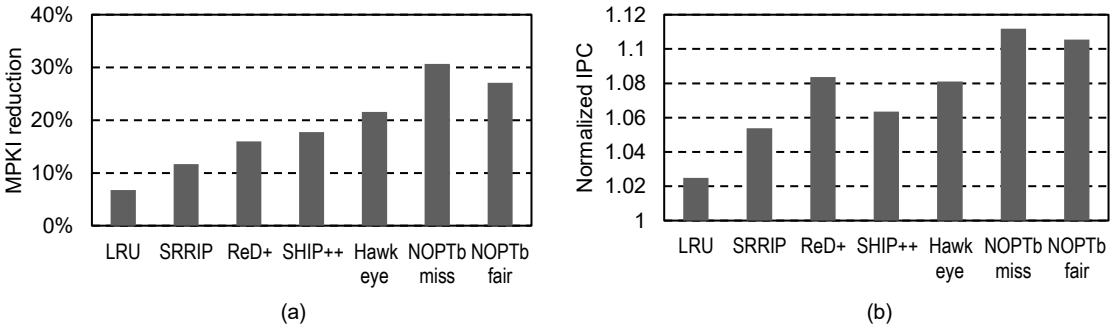

Figure 5.13: (a) MPKI reduction in the SLLC, and (b) normalized IPC, relative to random replacement, using an octa-core setup.

NOPTb-miss also outperforms all other policies in both metrics with eight cores, as expected. Comparing these results with the quad-core results in Figure 5.10, we can see that the gap with respect to the best online policies is similar. The best online policy is Hawkeye in terms of MPKI reduction, as with four cores. Nonetheless, when considering global IPC, the best online policy with eight cores is ReD+.

5.9.3 Fairness

Figure 5.14 shows the unfairness score for the same set of SLLC replacement policies as before, using the octa-core setup.

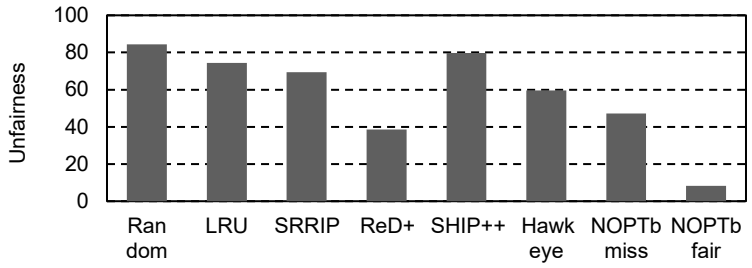

Figure 5.14: unfairness for various SLLC replacement policies, using an octa-core setup. A lower score means better fairness.

NOPTb-fair also achieves the best fairness with eight cores, as expected. Comparing these results with the quad-core results in Figure 5.11, we can see that the gap with respect to the best online policies is larger, and the differences between online policies are also larger, especially between ReD+ and Hawkeye. The greater fairness advantage achieved by ReD+ explains its superior IPC performance with eight cores, outperforming Hawkeye despite having a higher MPKI. This is consistent with our analysis in Section 5.8.3.

5.10 CONCLUSIONS

The replacement algorithm is a key piece in the design of cache memories. An optimal algorithm that minimizes the miss rate in a private cache has been available for several decades. This algorithm can be run offline on a simulator to provide reference values and insights into design problems. There is no equivalent algorithm, however, for shared caches, which are accessed by several processors or cores.

This chapter presents a novel near-optimal offline algorithm for minimizing the miss rate in a shared cache. We propose approaching the optimal execution and the optimal global access sequence iteratively. During each execution, the future access sequence that the shared cache will receive is reconstructed on every miss, interleaving the future individual per-core access sequences including their expected timings, as obtained from the previous execution. This reconstructed future access sequence feeds an OPTb policy, which decides to bypass the requested block or evict the block to be accessed further into the future. We evaluate NOPTb-miss running on an SLLC and show that it converges after a few iterations to a near-optimal miss rate that is independent of the initial conditions, within a margin of 0.1%.

In a shared cache, the optimization of the total number of cache misses does not imply the optimization of other metrics like fairness or throughput. We show that, running a multiprogrammed workload, NOPTb-miss benefits applications that access the SLLC more than others and is therefore not optimal in that regard. We then propose NOPTb-fair, a new near-optimal offline algorithm whose goal is to maximize fairness while maintaining a low miss rate. NOPTb-fair improves fairness by replacing the block with the largest Future Reuse Distance across all cores, defined as the number of accesses until its next use by its requesting core. NOPTb-fair achieves the best fairness among all replacement policies considered. Due to its better fairness, it even achieves a better overall throughput than NOPTb-miss in 34% of the workloads, despite always having a worse miss rate.

Our near-optimal proposals can be used to gain insights into state-of-the-art online policies. Comparing performance results, we show that the best online policies achieve around 65% of the MPKI reduction obtained by NOPTb-miss and 75% of the throughput improvement (vs. random). We demonstrate that our results are maintained when doubling the number of cores in the system. Comparing fairness, the best state-of-the-art policy, our previously proposed ReD+, achieves 60% of the improvement seen with our near-optimal policy, and the second-best only 45%. The gaps increase when doubling the number of cores. Analyzing performance and fairness together we show that, when designing a replacement policy for shared caches, higher system IPC values are obtained when seeking to achieve fairness in addition to miss rate reduction.

6 CONCLUSIONS

6.1 CONTRIBUTIONS

The research presented throughout this dissertation addresses important challenges in managing Shared Last-Level Caches (SLLCs) in Chip Multiprocessors (CMPs). The inherent inefficiencies in conventional cache management mechanisms, primarily stemming from limited temporal locality specifically within the stream of accesses reaching the SLLC and insufficient exploitation of reuse locality, have been thoroughly analyzed and mitigated through the proposals detailed in previous chapters.

Our initial contribution introduced ReD, an innovative content selection mechanism specifically designed for exclusive SLLCs. ReD effectively leverages reuse locality by employing a dedicated Reuse Detector positioned between each L2 cache and the SLLC. This component monitors the blocks evicted from L2 and determines whether they have experienced recent reuse. If a block shows no reuse, it is considered unlikely to be reused in the near future and is therefore bypassed—preventing its insertion into the SLLC. By filtering out non-reusable blocks before they reach the shared cache, ReD preserves valuable cache space for data with higher likelihood of reuse, significantly improving cache utilization and overall performance.

Through evaluation with multiprogrammed workloads on an eight-core processor, ReD demonstrated significant performance enhancements, achieving a 10.1% reduction in misses per instruction (MPI) compared to the baseline. Furthermore, ReD substantially outperformed contemporary strategies such as CHAR, Reuse Cache, and EAF cache, underscoring the effectiveness of our reuse-detection approach.

Expanding on these insights, we investigated power consumption challenges associated with conventional SRAM-based SLLCs, proposing the integration of Spin-Transfer Torque RAM (STT-RAM) technology coupled with our ReD mechanism, modified to adapt to this scenario. An SLLC based on STT-RAM features significantly lower static energy consumption, a smaller area footprint, and slightly lower dynamic read energy, but roughly doubles both the energy and latency of write operations. Its integration with ReD notably decreased cache writes, by selectively bypassing unnecessary block insertions,

resulting in an average energy saving of 33% in the STT-RAM SLLC and additional memory subsystem energy reductions. Moreover, the improved hit rates translated into meaningful performance enhancements, confirming ReD's dual benefit of power efficiency and performance improvement. Compared against the state-of-the-art DASCA scheme, ReD demonstrated superior accuracy in identifying reusable blocks, delivering further performance improvements and higher energy savings across both SLLC and main memory.

Recognizing the pivotal role of the block selection policy in cache performance, we developed ReD+, a refined version of our previous ReD mechanism. ReD+ synergistically integrates two complementary reuse-based proposals: the reuse detection mechanism from ReD, which tracks the reuse status of each block, and an instruction-based historical reuse predictor. The combination avoids the compulsory second miss of every block present in ReD and enhances its reuse detection effectiveness in case of a thrashing core, thus significantly improving the SLLC miss rate. This hybrid strategy further optimized content selection, achieving strong results demonstrated by its performance in the latest Cache Replacement Championship (CRC2). Specifically, ReD+ achieved third place overall, second in shared-cache scenarios, and exhibited performance within 0.2% of the best competing submission, reaffirming its robustness and versatility in diverse operating conditions.

Complementing our online policy improvements, we also introduced novel near-optimal offline replacement policies, NOPTb-miss and NOPTb-fair. The goal of NOPTb-miss is to minimize the miss rate in an SLLC, extending the principle of the classical OPTb algorithm — traditionally applied to private caches — to the more complex context of shared caches. To achieve this, NOPTb-miss adopts an iterative approach in which each execution refines the future global access sequence seen by the SLLC. This sequence, constructed by interleaving per-core future accesses obtained from the previous iteration, is fed to an OPTb policy to guide bypass and eviction decisions. Through successive iterations, the algorithm quickly converges to a near-optimal miss rate that is robust to initial conditions, providing a strong lower bound for online policies.

Meanwhile, NOPTb-fair extends beyond miss rate reduction, explicitly optimizing fairness among cores while maintaining excellent miss-rate reduction. It improves fairness by evicting the block with the largest Future Reuse Distance (FRD) across all cores, where

FRD is defined as the number of accesses before a block is reused by its requesting core. Our evaluation shows that fairness-oriented strategies frequently deliver better overall throughput, confirming that jointly optimizing for fairness and miss rate leads to superior system performance and underscoring the significance of comprehensive design objectives in shared-cache architectures.

These policies provide key references by approximating optimal replacement policies, thus serving as valuable tools for evaluating and guiding the development of practical, state-of-the-art cache management policies. The best state-of-the-art online policies achieve around 65% of the miss rate reduction and around 75% of the IPC improvement obtained by NOPTb-miss (vs random replacement). The best state-of-the-art online policy, our proposed ReD+, achieves 60% of the improvement in fairness seen with NOPTb-fair.

Collectively, the contributions of this dissertation demonstrate that reuse-based content selection, particularly when combined with advanced prediction techniques, significantly improves shared cache performance and efficiency in multicore processors. These benefits are consistently observed across a variety of scenarios and cache configurations. Furthermore, the development of near-optimal offline replacement policies not only establishes theoretical bounds for cache management strategies but also offers valuable insights to guide the design of practical, high-performance policies.

6.2 FUTURE WORK

The practical and theoretical advancements outlined in the previous section establish a robust foundation for future research, offering directions for the continued evolution of CMP cache optimization. Future work may explore the following directions:

Reuse-based replacement policies. ReD+ can be further improved by incorporating additional information or addressing certain limitations:

- Prefetch vs. demand requests: Currently, ReD+ does not distinguish between prefetch and demand requests, despite their fundamentally different lifecycles. Tailoring the management strategy based on request type could enhance the accuracy of content selection.
- Static vs dynamic thresholds: ReD+ relies on predefined static thresholds to classify blocks as having “very low” or “high” reuse probability. These values,

chosen through simulation, remain fixed regardless of the running workload. However, what qualifies as high or low reuse varies significantly depending on the characteristics of the running application. Introducing a dynamic mechanism to adjust these thresholds at runtime could make the policy more responsive to workload characteristics and, in turn, enhance its overall performance.

Near-optimal replacement policies. Future research in this area could explore the following directions:

- Execution time optimization in the presence of prefetching: Both our proposed near-optimal policies and the prior work reviewed in this dissertation have been studied under the assumption that all requests to the SLLC are of equal importance. However, prefetch requests typically have less critical impact, as they may lack accuracy or timeliness. Consequently, optimizing execution time may require differentiating between request types and managing them accordingly. Further investigation is needed to better understand this challenge and to develop strategies that account for the differing impact of prefetch and demand requests.
- Improving SLLC replacement policies through fairness considerations: Our proposed NOPTb-fair policy shows that targeting fairness in shared caches, alongside miss rate reduction, can lead to higher overall system IPC. In contrast, many state-of-the-art policies focus exclusively on minimizing miss rate. The principles introduced in this dissertation can serve as a foundation or guide for extending such policies with fairness-oriented mechanisms in a synergistic manner, thereby improving their overall effectiveness.

Taken together, the contributions presented in this dissertation not only advance the state of the art in shared cache management, but also lay a foundation upon which to build future solutions — fairer, more efficient, and better suited to the current and future needs of computer architecture.

7 CONCLUSIONES

7.1 CONTRIBUCIONES

La investigación presentada a lo largo de esta tesis aborda desafíos importantes en la gestión de las memorias cache compartidas de último nivel (SLLC, por sus siglas en inglés) en procesadores multinúcleo (CMPs). Las ineficiencias inherentes a los mecanismos convencionales de gestión de cache — derivadas principalmente de la limitada localidad temporal dentro del flujo de accesos que llega a la SLLC y del aprovechamiento insuficiente de la localidad de reuso — han sido analizadas en profundidad y mitigadas mediante las propuestas detalladas en los capítulos anteriores.

Nuestra contribución inicial presentó el Detector de Reuso (ReD), un innovador mecanismo de selección de contenido diseñado específicamente para SLLCs exclusivas — cuyo contenido se gestiona en exclusión con las caches privadas. ReD aprovecha eficazmente la localidad de reuso mediante el uso de un detector de reuso dedicado (ReD), ubicado entre cada cache L2 y la SLLC. Este componente supervisa los bloques expulsados de L2 y determina si han experimentado reuso reciente. Si un bloque no muestra reuso, se considera poco probable que vuelva a ser reutilizado en un futuro próximo y, por lo tanto, se omite — evitando su inserción en la SLLC. Al filtrar los bloques antes de que lleguen a la cache compartida, ReD preserva su valioso espacio para líneas con mayor probabilidad de reuso, mejorando significativamente la utilización de la cache y el rendimiento global.

Mediante una evaluación con cargas de trabajo multiprogramadas en un procesador de ocho núcleos, ReD ha demostrado mejoras significativas en el rendimiento, logrando una reducción del 10,1 % en los fallos por instrucción (MPI) en comparación con el esquema tomado como base. Además, ReD superó ampliamente a estrategias contemporáneas como CHAR, Reuse Cache y EAF cache, lo que pone de relieve la eficacia de nuestro enfoque basado en la detección de reuso.

En una segunda fase, investigamos los desafíos de consumo energético asociados a las SLLCs convencionales basadas en SRAM, proponiendo la integración de la tecnología Spin-Transfer Torque RAM (STT-RAM) junto con nuestro mecanismo ReD, modificado

para adaptarse a este escenario. Una SLLC basada en STT-RAM presenta un consumo estático de energía muy inferior, menor superficie y algo menor consumo dinámico en lecturas, pero aproximadamente duplica el consumo y la latencia de las escrituras. Su integración con ReD redujo notablemente las escrituras en cache, al omitir selectivamente la inserción de bloques innecesarios. Esto se tradujo en un ahorro energético medio del 33 % en la SLLC basada en STT-RAM, además de reducciones adicionales en el consumo energético del subsistema de memoria. Asimismo, las mejores tasas de acierto se tradujeron en notables mejoras de rendimiento, confirmando el doble beneficio de ReD en eficiencia energética y rendimiento. En comparación con la política DASCA, ReD demostró una mayor precisión en la identificación de bloques a mantener en la cache, ofreciendo mejoras adicionales en rendimiento y mayores ahorros energéticos tanto en la SLLC como en la memoria principal.

Reconociendo el papel fundamental de la política de selección de contenido en el rendimiento de la cache, desarrollamos ReD+, una versión perfeccionada de nuestro anterior mecanismo ReD. ReD+ integra de manera sinérgica dos propuestas complementarias basadas en el reuso: el mecanismo de detección de reuso de ReD, que detecta el estado de reuso de cada bloque, y un predictor histórico de reuso basado en la instrucción que dispara la petición de cada bloque. Esta combinación evita el segundo fallo obligatorio de cada bloque que se producía en ReD y mejora la eficacia de detección de reuso en situaciones con procesadores que generan un número excesivo de peticiones (*thrashing)*, lo que permite una reducción significativa de la tasa de fallos en la SLLC. Esta estrategia híbrida optimizó aún más la selección de contenido, logrando resultados destacables, como se evidenció en su desempeño en la última edición del Cache Replacement Championship (CRC2). Concretamente, ReD+ obtuvo el tercer puesto a nivel general y el segundo en simulaciones donde la cache es compartida, y mostró un rendimiento a tan solo un 0,2 % del mejor participante, reafirmando así su solidez y versatilidad en condiciones operativas diversas.

Como complemento a nuestras mejoras en las políticas “en línea”, es decir, aquellas que pueden implementarse teóricamente en un procesador real, también introdujimos nuevas políticas de reemplazo “fuera de línea” cuasi-óptimas, que sólo pueden ejecutarse en un simulador al requerir información futura. El objetivo de nuestra nueva política NOPTb-miss es minimizar la tasa de fallos en una SLLC, extendiendo el principio del algoritmo clásico OPTb — tradicionalmente aplicado a caches privadas — al contexto más

complejo de las caches compartidas. Para ello, NOPTb-miss adopta un enfoque iterativo en el que cada ejecución refina la secuencia futura global de accesos vista por la SLLC. Dicha secuencia, construida entrelazando los accesos futuros por núcleo obtenidos en la iteración anterior, se introduce en una política OPTb que guía las decisiones de omisión (*bypass*) e inserción. A través de iteraciones sucesivas, el algoritmo converge rápidamente hacia una tasa de fallos cuasi-óptima, robusta frente a las condiciones iniciales, proporcionando así una cota inferior sólida para las políticas “en línea”.

Por su parte, NOPTb-fair va más allá de la reducción de la tasa de fallos, al optimizar explícitamente la equidad (*fairness*) entre núcleos sin comprometer la eficacia en dicha reducción. Esta política mejora la equidad seleccionando para su expulsión el bloque con la mayor Distancia de Reuso Futura (FRD, por sus siglas en inglés) entre todos los núcleos, donde la FRD se define como el número de accesos que deben transcurrir antes de que un bloque sea reutilizado por el núcleo que lo solicitó originalmente. Nuestra evaluación muestra que las estrategias orientadas a la equidad con frecuencia logran un mayor rendimiento global, lo que confirma que una optimización conjunta de la equidad y la tasa de fallos conduce a un mejor desempeño del sistema completo, y subraya la importancia de adoptar objetivos de diseño integrales en las arquitecturas con cache compartida.

Estas políticas constituyen referencias fundamentales al aproximarse a las políticas de reemplazo óptimas, y se convierten así en herramientas valiosas para evaluar y orientar el desarrollo de políticas prácticas y de vanguardia en la gestión de cache. Las mejores políticas “en línea” del estado del arte alcanzan aproximadamente el 65 % de la reducción en la tasa de fallos y alrededor del 75 % de la mejora en IPC obtenidas por NOPTb-miss (frente al reemplazo aleatorio). En cuanto a la equidad, la mejor política “en línea” del estado del arte alcanza un 60 % de la mejora lograda con NOPTb-fair.

En conjunto, las contribuciones de esta tesis demuestran que la selección de contenido basada en reuso, especialmente cuando se combina con técnicas avanzadas de predicción, mejora de forma significativa el rendimiento y la eficiencia de las caches compartidas en procesadores multinúcleo. Estos beneficios se observan de manera consistente en una amplia variedad de escenarios y configuraciones de cache. Además, el desarrollo de políticas de reemplazo fuera de línea cuasi-óptimas no solo establece

cotas teóricas para las políticas de gestión de cache, sino que también ofrece valiosas orientaciones para el diseño de políticas prácticas de alto rendimiento.

7.2 TRABAJO FUTURO

Los avances teóricos y prácticos expuestos en la sección anterior establecen una base sólida para investigaciones futuras, ofreciendo diversas líneas para la evolución continua de la optimización de caches en procesadores multinúcleo. Entre las posibles direcciones para el futuro trabajo se incluyen las siguientes:

Políticas de reemplazo basadas en el reuso. ReD+ puede mejorarse aún más incorporando información adicional o abordando ciertas limitaciones:

- Peticiones adelantadas (*prefetch)* frente a peticiones de demanda: Actualmente, ReD+ no distingue entre peticiones adelantadas y peticiones de demanda, a pesar de que presentan ciclos de vida fundamentalmente distintos. Adaptar la estrategia de gestión en función del tipo de petición podría mejorar la precisión en la selección de contenido.
- Umbrales estáticos frente a dinámicos: ReD+ se basa en umbrales estáticos predefinidos para clasificar los bloques como de “muy baja” o “alta” probabilidad de reuso. Estos valores, elegidos mediante simulación, permanecen fijos independientemente de la carga de trabajo en ejecución. Sin embargo, lo que se considera un alto o bajo reuso varía considerablemente en función de las características de la aplicación que se está ejecutando. Introducir un mecanismo dinámico que ajuste estos umbrales en tiempo de ejecución podría hacer que la política responda mejor a las particularidades de cada carga de trabajo y, en consecuencia, mejorar su rendimiento global.

Políticas de reemplazo cuasi-óptimas. La investigación futura en esta área podría explorar las siguientes direcciones:

- Optimización del tiempo de ejecución en presencia de peticiones adelantadas: Tanto nuestras políticas cuasi-óptimas propuestas como los trabajos previos revisados en esta tesis han sido estudiados bajo la suposición de que todas las peticiones a la SLLC tienen la misma importancia. Sin embargo, las peticiones adelantadas suelen tener un impacto menos crítico, ya que pueden carecer de

precisión o puntualidad. En consecuencia, optimizar el tiempo de ejecución puede requerir diferenciar entre tipos de peticiones y gestionarlas de forma diferenciada. Se requiere una investigación más profunda para comprender mejor este desafío y desarrollar estrategias que tengan en cuenta el impacto desigual de las peticiones por demanda y las adelantadas.

- Mejora de las políticas de reemplazo en la SLLC mediante criterios de equidad: Nuestra política NOPTb-fair demuestra que orientar el diseño hacia la equidad en las caches compartidas, además de reducir la tasa de fallos, puede conducir a un mayor IPC global del sistema. Sin embargo, muchas de las políticas más avanzadas en el estado del arte se centran exclusivamente en minimizar la tasa de fallos. Los principios introducidos en esta tesis pueden servir como base o guía para extender dichas políticas con mecanismos orientados a la equidad de manera sinérgica, mejorando así su eficacia global.

En conjunto, las contribuciones aquí presentadas no solo avanzan el estado del arte en la gestión de caches compartidas, sino que también establecen un marco sobre el que construir futuras soluciones más justas, eficientes y adaptadas a las necesidades actuales y futuras de la arquitectura de computadores.

REFERENCES

Agarwal, N., Krishna, T., Peh, L.-S., & Jha, N. K. (2009). Garnet: A Detailed On-chip Network Model Inside a Full-System Simulator. *2009 IEEE International Symposium on Performance Analysis of Systems and Software,* 33–42.

Ahn, J., Yoo, S. & Choi, K. (2014). DASCA: Dead Write Prediction Assisted STT-RAM Cache Architecture. *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 25–36.

Albericio, J., Ibáñez, P., Viñals, V., & Llabería, J. M. (2013a). Exploiting reuse locality on inclusive shared last-level caches. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4), 38, 1-19.

Albericio, J., Ibáñez, P., Viñals, V., & Llabería, J. M. (2013b). The reuse cache: downsizing the shared last-level cache. *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 310-321.

Baer, J. (2009). *Microprocessor architecture: from simple pipelines to chip multiprocessors*. Cambridge University Press.

Balasubramonian, R., Jouppi, N. P., & Muralimanohar, N. (2011). Multi-core cache hierarchies. *Synthesis Lectures on Computer Architecture* 6(3), 1-153.

Beckmann, N., & Sanchez, D. (2017). Maximizing cache performance under uncertainty. *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 109–120.

Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 78-101.

Belady, L. A., & Palermo, F. P. (1974). On-line measurement of paging behavior by the multivalued MIN algorithm. *IBM Journal of Research and Development,* 18, 2-19.

Bhargava R., & Troester, K. (2024). AMD Next-Generation “Zen 4” Core and 4th Gen AMD EPYC Server CPUs. *IEEE Micro*, 44(3), 8-17.

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish,

M., Hill, M. D., & Wood, D. A. (2011). The gem5 simulator. *SIGARCH Computer Architecture News* 39(2), 1–7.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 422-426.

Bruce, M. (2023). ARM Neoverse V2 platform: Leadership Performance and Power Efficiency for Next-Generation Cloud Computing, ML and HPC Workloads. *IEEE Hot Chips 35th Symposium (HCS)*.

ChampSim. (2021). *ChampSim code repository*. Retrieved Feb 28th, 2025. https://github.com/ChampSim/ChampSim

Chang, M.-T., Lu, S.-L., & Jacob, B. (2014). Impact of cache coherence protocols on the power consumption of STT-RAM based LLC. *The Memory Forum Workshop*.

Chaudhuri, M., Gaur, J., Bashyam, N., Subramoney, S., & Nuzman, J. (2012). Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 293-304.

Clark, M. (2016). A New, High Performance x86 Core Design from AMD. *IEEE Hot Chips 28th Symposium (HCS)*.

Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., & Hughes, B. (2010). Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2), 16-29.

CRC2. (2017). *The 2nd Cache replacement championship, at the International Symposium on Computer Architecture (ISCA)*. Retrieved Feb. 28th, 2025. http://crc2.ece.tamu.edu.

CRC2. (2021). *CRC2 trace repository*. Retrieved Feb. 28th, 2025. https://crc2.ece.tamu.edu/?page_id=41

Dennard, R. H., Gaensslen, F. H., Yu, H.-N., Rideout, V. L., Bassous, E., & LeBlanc, A. R. (1974). Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), 256-268.

Diaz, J. (2021). *NOPT code repository*. Retrieved Aug. 25th, 2024. https://github.com/jdmaag73/NOPT

Dong, X., Xu, C., Xie, Y., & Jouppi, N. P. (2012). NVSim: a circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7), 994-1007.

Duong, N., Zhao, D., Kim, T., Cammarota, R., Valero, M., & Veidenbaum, A. (2012). Improving cache management policies using dynamic reuse distances. *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 389-400.

Eden, A. N., & Mudge, T. (1998). The YAGS branch prediction scheme. *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 69-77.

Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., Stamm, R. L., & Tullsen, D. M. (1997). Simultaneous multithreading: a platform for next-generation processors. *IEEE Micro,* 17(5), 12-19.

Escuin, C. (2024). *Crafting Non-Volatile Memory (NVM) Hierarchies: Optimizing Performance, Reliability, and Energy Efficiency* [Doctoral dissertation, University of Zaragoza]. Zaguan repository. https://zaguan.unizar.es/record/135624

Evers, M., Barnes, L. & Clark, M. (2022). The AMD Next-Generation "Zen 3" Core. *IEEE Micro*, 42(3), 7-12.

Eyerman, S. & Eeckhout, L. (2014). Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance. *IEEE Computer Architecture Letters*, 13(2), 93-96.

Faldu, P. & Grot, B. (2016). LLC Dead Block Prediction Considered Not Useful. *13th Workshop on Duplicating, Deconstructing and Debunking (WDDD)*.

Faldu, P. & Grot, B. (2017). Reuse-Aware Management for Last-Level Caches. *The 2nd cache replacement championship, at the International Symposium on Computer Architecture (ISCA)*.

Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., & Falsafi, B. (2012). Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *ACM SIGARCH Computer Architecture News*, 40, 37-48.

Gao, H., & Wilkerson, C. (2010). A dueling segmented LRU replacement algorithm with adaptive bypassing. *Proceedings of the 1st JILP Workshop on Computer Architecture Competitions*.

Gaur, J., Chaudhuri, M., & Subramoney, S. (2011). Bypass and Insertion Algorithms for Exclusive Last-level Caches. *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 81-92.

Guo, X., Ipek, E., & Soyata, T. (2010). Resistive Computation: Avoiding the Power Wall with Low-leakage, STT-MRAM Based Computing. *ACM SIGARCH Computer Architecture News*, 371-382.

Gupta, S., Gao, H., & Zhou, H. (2013). Adaptive Cache Bypassing for Inclusive Last Level Caches. *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 1243-1253.

Hamerly, G., Perelman, E., Lau, J., & Calder, B. (2005). SimPoint 3.0: faster and more flexible program phase analysis. *Journal of Instruction-Level Parallelism* 7(4),1-28.

Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture. A Quantitative Approach, 6th edition*. Morgan Kaufmann / Elsevier.

Henning, J. L. (2006). SPEC CPU 2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4), 1-17.

Hewlett-Packard Labs. (2013) *CACTI, an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model*. Retrieved Aug. 25th, 2024. http://www.hpl.hp.com/research/cacti/

Hu, Z., Kaxiras, S., & Martonosi, M. (2002). Timekeeping in the memory system: Predicting and optimizing memory behavior. *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 209-220.

Jain, A., & Lin, C. (2016). Back to the future: leveraging Belady's algorithm for improved cache replacement. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 78-89.

Jain, A., & Lin., C. (2017). Hawkeye cache replacement: leveraging Belady's algorithm for improved cache replacement. *The 2nd cache replacement championship, at the International Symposium on Computer Architecture (ISCA)*.

Jain, A., & Lin., C. (2018). Rethinking Belady's algorithm to accommodate prefetching. *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 110-123.

Jain, A., & Lin., C. (2019). Cache replacement policies. *Synthesis Lectures on Computer Architecture* 14(1), 1-87.

Jaleel, A., Borch, E., Bhandaru, M., Steely Jr., S. C., & Emer, J. (2010a). Achieving Non-Inclusive Cache Performance with Inclusive Caches. Temporal Locality Aware (TLA) Cache Management Policies. *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 151-162.

Jaleel, A., Theobald, K. B., Steely Jr., S. C., & Emer, J. (2010b). High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). *Proceedings of the 37th International Symposium on Computer Architecture*, 60-71.

Jaleel, A., Nuzman, J., Moga, A., Steely Jr., S.C., & Emer, J. (2015). High Performing Cache Hierarchies for Server Workloads. Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches. *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 343-353.

Jeong, J., & Dubois, M. (2006). Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers,* 55(4), 353-365.

JILP. (2010). *Proceedings of the 1st JILP workshop on Computer Architecture Competitions*.

Jiménez, D. A., & Teran, E. (2017). Multiperspective reuse prediction. *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 436-448.

Jog, A., Mishra, A. K., Xu, C., Xie, Y., Narayanan, V., Iyer, R., & Das, C. R. (2012). Cache Revive: Architecting Volatile STT-RAM Caches for Enhanced Performance in CMPs. *Proceedings of the 49th Annual Design Automation Conference*, 243-252.

Jouppi, N. P., & Wilton, S. J. E. (1994). Trade-offs in Two-Level On-Chip Caching. *Proceedings of the 21st International Symposium on Computer Architecture*, 34-45.

Jung, J., Nakata, Y., Yoshimoto, M., & Kawaguchi, H. (2013). Energy-efficient spin-transfer torque RAM cache exploiting additional all-zero-data flags. *14th International Symposium Quality Electronic Design (ISQED)*, 216-222.

Kanter, D. (2017, July 17th). Skylake-SP Scales Server Systems. *Microprocessor Report.*

Khan, S., Tian, Y., & Jiménez, D. A. (2010). Sampling Dead Block Prediction for Last-Level Caches. *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 175-186.

Khan, S., Wang, Z., & Jimenez, D. (2012). Decoupled dynamic cache segmentation. *2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, 1-12.

Kharbutli, M., & Solihin, Y. (2008). Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers*, 57(4), 433-447.

Kim, S., Chandra, D., & Solihin, Y. (2004). Fair cache sharing and partitioning in a chip multiprocessor architecture. *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT 2004)*, 111-122.

Lai, A. C., Fide, C., & Falsafi, B. (2001). Dead-Block Prediction & Dead-Block Correlating Prefetchers. *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 144-154.

Lee, D., Choi, J., & Kim, J. H. (2001). LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers* 50(12),1352-1361.

Li, L., Tong, D., Xie, Z., Lu, J., & Cheng, X. (2012). Optimal bypass monitor for high performance last-level caches. *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 315-324.

Lim, H., Kim, J., & Chong, J. (2010). A cache replacement policy to reduce cache miss rate for multiprocessor architecture. *IEICE Electron Express* 7(12), 850-855.

Lin, W. F., & Reinhardt, S. (2002). Predicting last-touch references under optimal replacement. *Tech Rep CSE-TR-447–02, University of Michigan.*

Liu, W., Yeung, D. (2009). Using aggressor thread information to improve shared cache management for CMPs. *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, 372-383.

Lotfi-Kamran, P., Grot, B., Ferdman, M., Volos, S., Kocberber, O., Picorel, J., Adileh, A., Jevdjic, D., Idgunji, S., Ozer, E., & Falsafi, B. (2012). Scale-Out Processors. *ACM SIGARCH Computer Architecture News*, 40(3), 500-511.

Luo, K., Gummaraju, J., & Franklin, M. (2001). Balancing Throughput and Fairness in SMT Processors. *2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 164-171.

Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., & Werner, B. (2002). Simics: A full system simulation platform. *Computer*, 35(2), 50-58.

Mao, M., Li, H. H., Jones, A.K., & Chen, Y. (2013). Coordinating Prefetching and STT-RAM Based Last-level Cache Management for Multicore Systems. *Proceedings of the 23rd ACM International Conference Great Lakes Symposium on VLSI*, 55-60.

Martin, M., Sorin, D., Beckmann, B., Marty, M., Xu, M., Alameldeen, A., Moore, K., Hill, M., & Wood, D. (2005). Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4), 92-99.

Mattson, R. L., Gecsei, J., Slutz, D. R. & Traiger, I. L. (1970). Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 78-117.

McFarling, S. (1991). Program Analysis and optimization for machines with instruction cache. *Tech Rep No. CSL-TR-91–493, Stanford University.*

McFarling, S. (1993). Combining branch predictors. *Technical Report WRL-TN-36, Digital Western Research Laboratory*.

Michaud, P. (2010). The 3P and 4P cache replacement policies. *Proceedings of the 1st JILP Workshop on Computer Architecture Competitions*.

Michaud, P. (2016). Some mathematical facts about optimal cache replacement. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4), 50, 1-19.

Mittal, S. (2016). A survey of cache bypassing techniques. *Journal of Low Power Electronics and Applications*, 6(2), 5.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8), 114-117.

Nassif, N., Munch, A. O., Molnar, C. L., Pasdast, G., Iyer, S. V., Yang, Z., Mendoza, O., Huddart, M., Venkataraman, S., Kandula, S., Marom, R., Kern, A. M., Bowhill, B., Mulvihill, D. R., Nimmagadda, S., Kalidindi, V., Krause, J., Haq, M. M., Sharma, R., & Duda, K. (2022). Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, 44-46.

Papazian, I. E. (2020). New 3rd Gen Intel Xeon Scalable Processor (Codename: Ice Lake-SP). *IEEE Hot Chips 32nd Symposium (HCS).*

Park, S. P., Gupta, S., Mojumder, N., Raghunathan, A., & Roy, K. (2012). Future Cache Design Using STT MRAMs for Improved Energy Efficiency: Devices, Circuits and Architecture. *DAC Design Automation Conference 2012*, 492-497.

Patil, H., Cohn, R. S., Charney, M., Kapoor, R., Sun, A., & Karunanidhi, A. (2004). Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation. *37th International Symposium on Microarchitecture (MICRO-37'04)*, 81-92.

Perelman, E., Hamerly, G., Van Biesbrouck, M., Sherwood, T., & Calder, B. (2003). Using Simpoint for accurate and efficient simulation. *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (SIGMETRICS '03),* 318-319.

Qureshi, M. K., Moinuddin, K., Thompson, D., & Patt, T. N. (2005). The V-Way cache: demand-based associativity via global replacement. *32nd International Symposium on Computer Architecture (ISCA'05)*, 544-555.

Qureshi, M. K., Lynch, D. N., Mutlu, O., & Patt, Y. N. (2006). A case for MLP-aware cache replacement. *33rd International Symposium on Computer Architecture (ISCA'06)*, 167-178.

Qureshi, M., Jaleel, A., Patt, Y., Steely, S., & Emer, J. (2007). Adaptive insertion policies for high performance caching. *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07)*, 381-391.

Rajan, K., & Govindarajan, R. (2007). Emulating optimal replacement with a shepherd cache. *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 445-454.

Rasquinha, M. (2011, December). *An energy efficient cache design using spin torque transfer (STT) RAM*. [Master of Science, School of Electrical and Computer Engineering, Georgia Institute of Technology].

Rosenfeld, P., Cooper-Balis, E., & Jacob, B. (2011). DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1), 16-19.

Seshadri, V., Mutlu, O., Kozuch, M. A., & Mowry, T. C. (2012). The evicted-address filter: a unified mechanism to address both cache pollution and thrashing. *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 355-366.

Smith, A. J. (1982). Cache memories. *ACM Computer Surveys (CSUR),* 14(3), 473-530.

Snavely, A. & Tullsen, D. M. (2000). Symbiotic jobscheduling for a simultaneous multithreading processor. *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPLOS IX)*, 234-244.

Strohmaier, E., Dongarra, J., Simon, H., Meuer, M., & Meuer, H. (2024, November). *Top 500, the list, 64th edition*. Retrieved Feb. 28th, 2025. https://www.top500.org

Sun, G., Dong, X., Xie, Y., Li, J., & Chen, Y. (2009). A novel architecture of the 3D stacked MRAM L2 cache for CMPs. *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 239-249.

Sun, Z., Bi, X., Li, H. H., Wong, W.-F., Ong, Z.-L., Zhu, X., & Wu, W. (2011). Multi retention level STT-RAM cache designs with a dynamic refresh scheme. *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 329-338.

Vakil-Ghahani, A., Mahdizadeh-Shahri, S., Lotfi-Namin, M., Bakhshalipour, M., Lotfi-Kamran, P., & Sarbazi-Azad, H. (2018). Cache Replacement Policy Based on Expected Hit Count. *IEEE Computer Architecture Letters,* 17(1), 64-67.

Wang, K., Dong, X., & Xie, Y. (2013). OAP: An obstruction-aware cache management policy for STT-RAM last-level caches. *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 847-852.

Wang, J., Zhang, L., Panda, R. & John, L. K. (2017). Less is More: Leveraging Belady's Algorithm with Demand-based Learning. *The 2nd Cache replacement championship, at the International Symposium on Computer Architecture (ISCA)*.

Warrier, T., Anupama, V., & Mutyam, M. (2013). An Application-Aware Cache Replacement Policy for Last-Level Caches. *Architecture of Computing Systems – ARCS 2013. Lecture Notes in Computer Science*, 7767, 207-219.

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

Wong, W., & Baer, J. L. (2000). Modified LRU policies for improving second-level cache behavior. *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture. HPCA-6*, 49-60.

Wu, C. J., Jaleel, A., Hasenplaugh, W., Martonosi, M., Steely Jr., S. C., & Emer, J. (2011). SHiP: signature-based hit predictor for high performance caching. *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*, 430-441.

Wulf, W. A., & McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News* 23(1), 20-24.

Yazdanshenas, S., Ranjbar Pirbast, M., Fazeli, M., & Patooghy, A. (2014). Coding last level STT-RAM cache for high endurance and low power. *IEEE Computer Architecture Letters*, 13, 73-76.

Yoshida, T. (2018). Fujitsu high performance CPU for the post-k computer. *IEEE Hot Chips 30th Symposium (HCS)*.

Young, V., Chou, C., Jaleel, A., & Qureshi, M. (2017). SHiP++: enhancing signature-based hit predictor for improved cache performance. *The 2nd cache replacement championship, at the International Symposium on Computer Architecture (ISCA)*.

Zahran, M., Albayraktaroglu, K., & Franklin, M. (2007). Non-inclusion property in multi-level caches revisited. *International Journal of Computers and Their Applications*, 14(2), 99-108.