



Coherence-on-Demand and Hybrid Eviction Policies for Multiprocessor System-on-Chip Architectures

Akshay Sateesh Srivatsa

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:

Prof. Dr.-Ing. Ulf Schlichtmann

Prüfende der Dissertation:

1. Prof. Dr. sc.techn. Andreas Herkersdorf
2. Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat,
Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg

Die Dissertation wurde am 17.01.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 07.07.2022 angenommen.

Abstract

The increase in performance of modern computing systems can be briefly credited to two driving forces. One, the advancements in semi-conductor manufacturing technology, which have significantly increased the number of transistors on a chip by leveraging Dennard Scaling and Moore’s Law. Two, the advancements in transistor-technology agnostic optimization techniques, which have led to innovative computer (micro)architecture designs that efficiently utilize the transistors on a chip. However, with the end of Dennard Scaling and slowing of Moore’s Law, the “optimization-spotlight” is ever-more on enhancing the architecture of computing systems. Further, it is becoming increasingly clear that solely optimizing the processor architecture is not sufficient, and interfaces around it, like the memory subsystem also need to be considered jointly. This thesis proposes two computer architecture concepts, both directed towards optimizing the memory subsystem of large computing platforms.

The first contribution is an alternative hardware-based cache coherence methodology for ever-growing tile-based manycore architectures. The motivation is that a single application rarely utilizes all processing and memory resources of a manycore system. This questions the need for global coherence which consumes substantial hardware area overheads, leading to scalability issues. This thesis proposes a Region-based Cache Coherence (RBCC) concept that provides scalable and flexible inter-tile coherence for tile-based manycore systems. RBCC introduces the notion of *coherence regions*, which comprise a sub-cluster of tiles that are guaranteed to be cache coherent. By confining inter-tile coherence support to within coherence regions, RBCC significantly reduces the required book-keeping overheads compared to global coherence schemes (73% area reduction assuming a 64-tile system with a maximum coherence region size of 8-tiles). RBCC’s coherence regions are also designed to be flexible. They can be created, dissolved, re-sized or even re-located at run-time based on the application’s requirements, thereby establishing a *coherence-on-demand* environment. The performance of RBCC-enabled inter-tile coherence is evaluated against software-based alternatives for different workloads. Results show that workloads execute faster when using hardware-supported coherence (by up to 45%), compared to the software alternatives.

The second contribution focuses on efficient data management for cache memories. The need for quick memory accesses coupled with limited cache capacities highlight the importance of eviction policies. Modern applications tend to exhibit non-uniform memory access patterns, which are further amplified when several such applications are executing simultaneously. As a consequence, the caches experience highly irregular memory access patterns, making it nearly impossible for a single eviction policy to manage cache data efficiently. This thesis introduces a Hybrid Voting-based Eviction Policy (HyVE) that enhances cache data management by considering several eviction

Abstract

criteria. HyVE is a modular framework that fuses multiple standalone eviction policies together, and evaluates their individual decisions using voting theory. The concept of HyVE has been explored for Last Level Caches (LLCs) as well as sparse directory structures. For LLCs, results show that HyVE reduces the cache misses by up to 25% compared to its standalone counterparts. The performance of HyVE has also been evaluated against state-of-the-art (hybrid and learning-based) eviction policies, which show both improvements and deterioration. For sparse directories, results show that HyVE reduces the coherence traffic and execution time of workloads by up to 11% compared to the LRU policy.

The experimental evaluations of both research contributions, RBCC and HyVE have been performed on simulation as-well-as an FPGA-based prototyping platform.

Zusammenfassung

Den Leistungszuwachs in modernen Computersystemen kann zwei treibenden Kräften zugeschrieben werden. Einerseits, dem Fortschritt in der Halbleitertechnologie, welches zu einer höheren Anzahl an Transistoren auf einem Chip führte (Dennard Scaling und Moore's Law). Andererseits, transistor-technologieunabhängige Verbesserungen, welche zu innovativen Computerarchitekturentwürfen und eine bessere Nutzung der vorhandenen Transistoren führt. Allerdings, mit dem Ende von Dennard Scaling und der Verlangsamung von Moores Law, liegt der Fokus immer mehr an der Verbesserung der Architektur von Computersystemen. Optimierung der Prozessorarchitektur allein reicht allerdings nicht aus und Schnittstellen, wie das Speicher-Subsystem sollten ebenfalls berücksichtigt werden. Diese Thesis schlägt zwei Computerarchitektur-Konzepte vor, beide haben das Ziel das Speicher-Subsystem in großen Computerplattformen zu optimieren.

Der erste Beitrag ist ein alternative hardwarebasierte Cache-Kohärenz Methode für wachsende kachelbasierte Mehrkernsysteme. Die Motivation dahinter ist, dass eine einzige Applikation selten alle Rechen- und Speicherressourcen eines Mehrkernsystem ausnutzen kann. Dies stellt in Frage, ob eine globale Kohärenz benötigt wird, welche zu einem Hardwaremehraufwand und Skalierungsproblemen führt. Diese Arbeit schlägt eine Region-based Cache Coherence (RBCC) vor, welche eine skalierbare und flexible Kohärenz zwischen Kacheln in einem großen kachelbasierte Mehrkernsysteme ermöglicht. RBCC führt das Konzept von Kohärenzregionen ein, in denen Untergruppen von Kachel gebildet werden können, für welche Cache-Kohärenz garantiert werden kann. Durch beschränken der Inter-Kachelkohärenz auf innerhalb einer Kohärenzregion, verringert RBCC den Mehraufwand, welches eine globales Kohärenzschema aufweist (73% Flächen-Reduzierung in einem 64-Kachel System mit einer maximalen Kohärenzregion von 8-Kacheln). RBCC's Kohärenzregionen wurden entwickelt um flexibel zu sein. Je nach Anforderung der Applikation können sie während der Laufzeit erstellt, aufgelöst, deren Größe angepasst und verlagert werde. Somit wird eine *coherence-on-demand* Umgebung erstellt. Die Leistung von RBCC wird gegenüber softwarebasierten Alternativen für verschiedene Arbeitslasten ausgewertet. Die Ergebnisse zeigen, dass Arbeitslasten mit hardwareunterstützten Kohärenz bis zu 45% schneller sind, verglichen zu Software-Alternativen.

Der zweite Beitrag fokussiert sich auf effiziente Datenverwaltung für Cache-Speicher. Die Anforderung für schnelle Speicherzugriffe gekoppelt mit begrenzter Cachekapazität verdeutlicht die Wichtigkeit von Räumungsstrategien (eviction policies). Moderne Anwendungen tendieren zu uneinheitlich Speicherzugriffe, welche noch weiter verstärkt werden, wenn mehrere solche Anwendungen gleichzeitig ausgeführt werden. Dies hat zu Folge, dass Caches sehr irreguläre Speicherzugriffsmuster aufweisen, was es für eine

Zusammenfassung

einzigste Räumungsstrategie nahezu unmöglich macht die Daten im, Cache effizient zu verwalten. Diese Thesis führt eine Hybrid Voting-based Eviction Policy (HyVE) ein, welche die Datenverwaltung in Caches mit mehreren Räumungsstrategien erweitert. HyVE ist ein modulares Framework, welches verschiedene Räumungsstrategien kombiniert indem es dessen einzelnen Entscheidungen mit Hilfe von Wahltheorie (voting theory) evaluiert. Das Konzept von HyVE wurde für sowohl Last Level Caches (LLCs), als auch Spärliche Verzeichnis-Strukturen (sparse directory structures) erforscht. Die Ergebnisse für LLCs zeigen, dass HyVE die Anzahl an Cache um bis zu 25% verringert, im Vergleich zu einzelnen Räumungsstrategien. HyVE wurde auch zu anderen Räumungsstrategien auf neuem Stand der Technik verglichen und zeigt Verbesserungen und Verschlechterungen auf. Die Ergebnisse für sparse directories zeigen, dass der Kohärenzverkehr und die Laufzeit der Arbeitslast um bis zu 11% abnehmen im Vergleich zur Least Recently Used (LRU) Strategie.

Die experimentelle Evaluation beider Forschungsbeiträge, RBCC und HyVE wurden mit sowohl Simulationen, als auch mit Field Programmable Gate Array (FPGA)-basierten Prototyp.

Contents

Abstract	iii
Zusammenfassung	v
Contents	vii
List of Figures	xi
List of Tables	xv
Glossary	xvii
Acronyms	xix
1 Introduction	1
1.1 Background	1
1.1.1 The Walls of Computer Architecture	1
1.1.2 Cache Memories	2
1.1.3 The Evolution of Manycore Architectures	2
1.1.4 The Need for Cache Coherence	3
1.1.5 Cache Data Management	5
1.1.6 Resource-Aware Computing	5
1.2 Problem Statements	6
1.2.1 Scalable Cache Coherence	6
1.2.2 Optimizing Cache Data Management	7
1.3 Contributions	7
1.3.1 Region-based Cache Coherence (RBCC)	7
1.3.2 Hybrid Voting-based Eviction Policy (HyVE)	8
1.4 Organization	8
2 State of the Art	9
2.1 Cache Coherence	9
2.1.1 No Hardware Coherence Support	10
2.1.1.1 MPI-based Communication	10
2.1.1.2 Software-based Coherence Schemes	10
2.1.2 Hardware Coherence Support	11
2.1.2.1 Overcoming Scalability Limitations	12

CONTENTS

2.1.2.2	Alternatives to Global Coherence	12
2.1.3	How is RBCC Different?	14
2.2	Eviction Policies	15
2.2.1	Standalone Cache Eviction Policies	16
2.2.2	Hybrid Cache Eviction Policies	17
2.2.3	Learning-based Cache Eviction Policies	18
2.2.4	Eviction Policies for Sparse Directories	19
2.2.5	How is HyVE Different?	20
3	Region-based Cache Coherence (RBCC)	21
3.1	The RBCC Concept	21
3.1.1	Target Architecture	21
3.2	RBCC Features	22
3.2.1	Scalability	22
3.2.2	Flexibility	24
3.2.3	Coherence-on-Demand	26
3.2.4	Auxiliary Features	26
3.3	RBCC Design	27
3.3.1	The Coherence Region Manager (CRM)	27
3.3.2	Architectural Design	27
3.3.3	The CRM and its Sub-modules	28
3.4	RBCC Functionality	30
3.4.1	Coherence Region Configuration	30
3.4.2	Coherence-on-Demand: RBCC-malloc()	32
3.4.3	Coherence Operations	33
3.4.4	Coherence Barrier Mechanism	34
3.4.5	False Sharing Resolution	36
3.4.6	Auxiliary Functions	38
3.5	Concept Evaluation - High-Level Simulation	39
3.5.1	Simulation Framework	39
3.5.1.1	Extracting Traces from the Gem5 Simulator	40
3.5.1.2	Data Placement Strategies	41
3.5.2	Experimental Setup	41
3.5.3	Results and Analysis	42
3.6	Hardware Implementation and Evaluation - FPGA Prototype	44
3.6.1	Hardware Setup	44
3.6.2	FPGA Resource Utilization and Timing	44
3.6.3	Experimental Setup	48
3.6.4	Results and Analysis	49
3.6.4.1	RBCC mode versus MP mode	51
3.6.4.2	Run-time Re-configuration Analysis	55
3.6.4.3	RBCC-malloc() Analysis	58
3.7	Enabling Shared Memory Workloads	60
3.7.1	Two Methodologies for Shared Memory Programming	60

3.7.2	Experimental Setup and Evaluation	61
3.8	Additional Case-Study - RBCC and In-NoC Circuits (INCs)	62
3.8.1	Concept	62
3.8.2	Experimental Setup and Evaluation	63
4	Hybrid Voting-based Eviction Policy (HyVE)	65
4.1	The HyVE Concept	65
4.1.1	Voting Theory Background	66
4.2	HyVE: Features and Design	68
4.2.1	Rank Generation	68
4.2.2	Modular and Flexible Framework	68
4.2.3	Tie Handling	69
4.3	Case-Study 1: HyVE for Caches	70
4.3.1	Ingredients for HyVE	70
4.3.2	Exploring HyVE Flavours	71
4.4	Experimental Evaluation - HyVE for Caches	73
4.4.1	Simulation Framework	73
4.4.2	Experimental Setup	73
4.4.3	Analysing HyVE Flavours	74
4.4.4	Cache Size Sensitivity Analysis	79
4.4.5	Voting Methodology Analysis - Borda Count vs Condorcet Method	80
4.4.6	Comparison to State-of-the-art Policies	82
4.4.7	Take-away Points	84
4.4.8	Hardware Implementation	85
4.5	Case-Study 2: HyVE for Sparse Directories	87
4.5.1	Architecture-aware Eviction Policies	88
4.5.2	Building HyVE for Sparse Directories	89
4.6	Experimental Evaluation - HyVE for Sparse Directories	90
4.6.1	Target Architecture	90
4.6.2	Experimental Setup	90
4.6.3	Results and Analysis	91
4.6.4	Highlighting HyVE's Properties	96
4.6.4.1	Experimental Setup	98
4.6.4.2	Results and Analysis	99
5	Conclusion & Outlook	103
5.1	Conclusion	103
5.2	Outlook	104
	Bibliography	107
	A Results of all Explored HyVE Flavours	117
	B Analysing HyVE with the Condorcet Method for Sparse Directories	121

List of Figures

1.1	A simple example illustrating the need for cache coherence	4
1.2	An example of snoop-based coherence	5
1.3	An example of directory-based coherence	5
2.1	The speedup of several Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmarks executed using different input set sizes for increasing thread counts, taken from [1] © 2016 IEEE . .	13
2.2	The diversity of different memory access patterns for increasing cache sizes using the LRU policy, taken from [2] © 2008 IEEE	15
2.3	A taxonomy of cache eviction policies	16
2.4	The lifespan of a cache block with different phases, adapted from [3] . . .	16
2.5	An example illustrating the concept of Set-Dueling (SD), adapted from [4]	18
3.1	A heterogeneous DSM-based tiled manycore architecture used to evaluate RBCC. This architecture is also used in as part of the Invasive Computing (InvasIC) project	22
3.2	An example sparse directory structure	23
3.3	A visualization of the achievable directory area reductions when using RBCC compared to global coherence for different N and M_{max} combinations	24
3.4	The InvasIC tile-based manycore architecture with several coherence regions	25
3.5	An example memory-map with code and data sections	26
3.6	The internal block diagram of the CRM with its sub-modules	28
3.7	An example illustrating how RBCC-malloc() dynamically tailors coherence regions to only track actually shared applications' working-sets at run-time	32
3.8	An example illustrating the internal block diagram and operation of the coherence barrier mechanism	35
3.9	An example of the false sharing problem	37
3.10	An example of false sharing resolution	37
3.11	The normalized execution time of each benchmark with three DoPs (4, 8, 16) for the RBCC::FT, RBCC::MA and AiO system configuration	43
3.12	Logic utilization break-down of the CRM by functionality	45
3.13	Logic utilization for a $N = 16$ tile manycore system with increasing coherence region sizes ($2 \leq M_{max} \leq 16$) normalized to a Virtex-7 FPGA	46
3.14	BRAM utilization for a $N = 16$ tile manycore system with increasing coherence region sizes ($2 \leq M_{max} \leq 16$) normalized to a Virtex-7 FPGA	47
3.15	Directory re-configuration overheads for different reset steps	47

LIST OF FIGURES

3.16	A flow diagram depicting the feature extraction task	49
3.17	Clustered coherence region	50
3.18	Corner coherence region	50
3.19	Per-frame execution time of the donkeykong video clip, with and without BT	51
3.20	Per-frame execution time of the spaceinvaders video clip, with and without BT	52
3.21	Per-frame execution time of the pacman video clip, with and without BT	52
3.22	Per-frame execution time of the snake video clip, with and without BT	53
3.23	Average execution time of the <i>rbcc</i> mode and <i>mp</i> mode for all video clips with increasing BT	54
3.24	Expanding the coherence region using the donkeykong clip	56
3.25	Expanding the coherence region using the snake clip	56
3.26	Relocating the coherence region using the donkeykong clip	56
3.27	Relocating the coherence region using the snake clip	56
3.28	FIFO load reduction when using RBCC-malloc() for all video clips	59
3.29	The execution time of two SPLASH-2 benchmarks using both VSM- and CRM-based approaches to enable shared memory programming, for different DoPs	61
3.30	Different coherence region configurations on a 4x4 tile-based manycore system	63
3.31	The normalized average delay of inter-tile coherence messages for the <i>clustered-corner</i> and <i>pure-corner</i> configurations with and without INCs for all benchmarks. Adapted from [5]	64
4.1	An abstract example demonstrating the basic concept of HyVE using three eviction policies casting their votes on four candidates	65
4.2	Condorcet Method	67
4.3	An example illustrating how standalone eviction policies are incorporated within the HyVE framework using a 4 way cache structure	69
4.4	Target architecture	74
4.5	Normalized LLC misses for HyVE-a and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks	76
4.6	Normalized LLC misses for HyVE-b and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks	77
4.7	Normalized LLC misses for HyVE-C and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks	78
4.8	Normalized LLC misses for HyVE-d and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks	78

4.9	Normalized LLC misses for HyVE-e and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks	79
4.10	Normalized LLC misses with different cache sizes for all HyVE flavours	80
4.11	Normalized LLC misses of all HyVE flavours using the Condorcet Method	81
4.12	Eviction count plots for HyVE-a and HyVE-d using the Condorcet Method and Borda Count voting methodologies, for four representative benchmarks each	82
4.13	The normalized LLC misses of all benchmarks using the HyVE flavours, DRRIP and Hawkeye	83
4.14	The normalized IPC of all benchmarks using the HyVE flavours, DRRIP and Hawkeye	84
4.15	Breaking-down the FPGA resource utilization for each HyVE flavour	86
4.16	An example demonstrating the LNS eviction policy	88
4.17	An example demonstrating the SDF eviction policy	89
4.18	An example demonstrating the LRU policy	89
4.19	An example demonstrating HyVE with LRU, LNS and SDF as the constituent eviction policies	90
4.20	The execution time and additional evaluation metrics for the canneal benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)	92
4.21	The execution time and additional evaluation metrics for the swaptions benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)	92
4.22	The execution time and additional evaluation metrics for the fluidanimate benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)	93
4.23	The execution time and additional evaluation metrics for the fft benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)	93
4.24	The execution time and additional evaluation metrics for the lucb benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)	94
4.25	Victim distribution analysis of all benchmarks for the sparse directory configuration - 128 sets, 8 ways using Borda Count as the voting procedure	95
4.26	HyVE break-down	97
4.27	The three team configurations and their respective characteristics on a 4x4 tiled manycore architecture	98
4.28	The total eviction count for multiple test scenarios using the micro-benchmarks100	
4.29	The total number of <i>dirInvs</i> and the total number of <i>dirInv</i> hops for multiple test scenarios using the micro-benchmarks	101
4.30	The total execution cycles for multiple test scenarios using the micro-benchmarks, with and without the presence of BT	101

LIST OF FIGURES

4.31	The average <i>dirInvs</i> per eviction, and the average hops per <i>dirInv</i> for multiple test scenarios using the micro-benchmarks	102
A.1	Normalized LLC misses of the standalone eviction policies for all benchmarks	117
A.2	Normalized LLC misses of the 18 remaining HyVE flavours for all benchmarks	118
A.3	Normalized LLC misses of the already analysed 5 HyVE flavours for all benchmarks	119
B.1	The execution time of all benchmarks using the standalone eviction policies and HyVE with the Condorcet Method for different sparse directory configurations	122
B.2	Victim distribution analysis of all benchmarks for the sparse directory configuration - 128 sets, 8 ways using the Condorcet Method as the voting procedure	123

List of Tables

3.1	Revisiting coherence alternatives for large tile-based manycore architectures	21
3.2	An example of two CCT entries that establish a coherence region spanning two tiles	30
3.3	A summary different memory access operations and their coherence actions	35
3.4	An example trace file format	41
3.5	FPGA resource utilization of the CRM module in terms of LUTs, REGs, MUXs and BRAMs for a $N = 16$ tile manycore system with a coherence region size of $M_{max} = 8$	45
3.6	Latency of different CRM operations	46
4.1	Ranking Distribution	66
4.2	List of all explored HyVE flavours categorized into groups of two, three and four	72
4.3	Architecture configuration parameters for the sniper simulator	73
4.4	FPGA resource utilization (LUT, REG) and logic delay for all standalone eviction policies and HyVE flavours using Borda Count	86
4.5	Fundamental differences between a data cache and a sparse directory in the context of eviction decisions	87
4.6	Optimization attributes of the constituent eviction policies used for HyVE	89
4.7	FPGA resource utilization (LUT, REG) and logic delay for all standalone eviction policies and HyVE flavours using Borda Count	97
4.8	The test scenarios with combinations of different data-set sizes and their properties	99

Glossary

constituent eviction policy	An eviction policy used within the HyVE framework.
LEON	A high-performance RISC-based processor designed using the SPARC ISA.
manycore architecture	An MPSoC platform consisting of several dozens of cores. Used synonymously with the term manycore system.
manycore system	An MPSoC platform consisting of several dozens of cores. Used synonymously with the term manycore architecture.
multi-core architecture	An MPSoC platform consisting of 2-8 cores. Used synonymously with the term multi-core system.
multi-core system	An MPSoC platform consisting of 2-8 cores. Used synonymously with the term multi-core architecture.
standalone eviction policy	An eviction policy used independent of the HyVE framework.
TLM-2.0	A SystemC standard for inter-module communication known as Transaction Level Modelling.

Acronyms

<i>dirInv</i>	Directory-induced Invalidation.
AHB	Advanced High-performance Bus.
AiO	All-in-One.
AMBA	Advanced Microcontroller Bus Architecture.
ARC	Adaptive Replacement Cache.
BIP	Bimodal Insertion Policy.
BRAM	Block RAM.
BRRIIP	Bimodal Re-Reference Interval Prediction.
BT	Background Traffic.
CCT	Coherence Configuration Table.
CDR	Coherence Domain Restriction.
CPU	Central Processing Unit.
CRM	Coherence Region Manager.
DASH	Directory Architecture for Shared Memory.
DDC [™]	Dynamic Distributed Cache.
DIP	Dynamic Insertion Policy.
DLT	Directory Look-aside Table.
DMA	Direct Memory Access.
DoP	Degree of Parallelism.
DRAM	Dynamic Random Access Memory.
DRRIIP	Dynamic Re-Reference Interval Prediction.
DSM	Distributed-Shared Memory.
DSS	Dynamic Set Sampling.
FAU	Friedrich-Alexander-Universität Erlangen-Nürnberg.
FIFO	First In First Out.
FP	Frequency Priority.
FPGA	Field Programmable Gate Array.
FT	First Touch.
GUI	Graphical User Interface.

Acronyms

HP	Hit Priority.
HPC	High Performance Computing.
HyVE	Hybrid Voting-based Eviction Policy.
I/O	Input/Output.
IDT	Image Distribution Time.
ILP	Instruction Level Parallelism.
IMSuite	IIT Madras Benchmark Suite.
INC	In-NoC Circuit.
InvasIC	Invasive Computing.
IP	Intellectual Property.
IPC	Instructions-per-Cycle.
IPT	Image Processing Time.
ISA	Instruction Set Architecture.
ITIV	Institute for Information Processing Technologies.
KIT	Karlsruhe Institute of Technology.
L1	Level 1.
L2	Level 2.
L3	Level 3.
LFU	Least Frequently Used.
LIP	LRU Insertion Policy.
LIS	Lehrstuhl für Integrierte Systeme / Chair of Integrated Systems.
LLC	Last Level Cache.
LNS	Least Number of Sharers.
LRA	Least Recently Accessed.
LRU	Least Recently Used.
LUT	Look-Up Table.
MA	Most Accessed.
MESI	Modified-Exclusive-Shared-Invalid.
MESIF	Modified-Exclusive-Shared-Invalid-Forward.
MI	Modified-Invalid.
MMU	Memory Management Unit.
MOESI	Modified-Owner-Exclusive-Shared-Invalid.
MPI	Message-Passing Interface.
MPPA	Massively Parallel Processor Array.
MPSoC	Multi-Processor System-on-Chip.
MRU	Most Recently Used.
MSI	Modified-Shared-Invalid.
MUX	Multiplexer.

NINE	Non-Inclusive Non-Exclusive.
NoC	Network-on-Chip.
NPB	NAS Parallel Benchmarks.
NRU	Not Recently Used.
NUMA	Non-Uniform Memory Access.
OS	Operating System.
PARSEC	Princeton Application Repository for Shared-Memory Computers.
PGAS	Partitioned Global Address Space.
PSEL	Policy Selector.
RBCC	Region-based Cache Coherence.
REG	Register.
RISC	Reduced Instruction Set Computer.
RoI	Region of Interest.
RRIP	Re-Reference Interval Prediction.
RRPV	Re-Reference Prediction Value.
SCC	Single-Chip Cloud Computer.
SCD	Scalable Coherence Directory.
SD	Set-Dueling.
SDBP	Sampling Dead Block Prediction.
SDF	Shortest Distance First.
SHiP	Signature-based Hit Predictor.
SPLASH-2	Stanford Parallel Applications for Shared Memory.
SRAM	Static Random Access Memory.
SRRIP	Static Re-Reference Interval Prediction.
TLB	Translation Lookaside Buffer.
TLM	Tile Local Memory.
UMA	Uniform Memory Access.
VSM	Virtual Shared Memory.

1 Introduction

The insatiable hunger for faster, smarter and power-efficient electronic products are some of the key factors revolutionizing the computing industry. The computer architecture community has been significantly contributing towards this cause through continuous innovation. For decades, the transistor manufacturing technology has been evolving with the likes of Dennard Scaling [6] and Moore’s Law [7]. Owing to these factors, the number of transistors that can be accommodated on a given chip have been increasing, whilst maintaining the power consumption within acceptable limits. The performance of computing systems has also been enhanced using transistor-technology agnostic optimization techniques such as frequency scaling, exploiting the Instruction Level Parallelism (ILP), reducing the processor-memory performance gap, increasing data locality, etc. With the end of Dennard Scaling and slowing of Moore’s Law [8], the need for such transistor-technology agnostic optimization techniques are increasing even more. However, each of these optimization techniques come with certain limitations and challenges.

1.1 Background

1.1.1 The Walls of Computer Architecture

From a system level perspective, a modern computing system is a combination of a processing subsystem, a memory subsystem and different types of interconnects. The challenges or “walls” of computer architecture present themselves at all parts of the computing system.

Power Wall. Increasing the clock frequency of a processor increases its capacity to do work. An increase in the processor’s operating frequency also increases its power consumption, subsequently increasing its temperature. At a certain frequency threshold, a power wall is reached, beyond which cooling techniques fail to cope. Operating at/beyond such critical frequencies may reduce the lifespan of transistors and affect their functionality, ultimately making the processor non-operable. Therefore, the focus shifts towards investing the abundantly available transistors (from Moore’s Law) to design multiple (safely-clocked) processors that can work in parallel, leading towards a Multi-Processor System-on-Chip (MPSoC) architecture.

ILP Wall. There are many techniques which exploit ILP such as instruction pipelining, superscalar execution units and even out-of-order processing. These techniques undoubtedly improve the performance of an MPSoC, but have their limitations. Designing very deep pipelines has a negative impact on instruction latency. For superscalar systems,

1 Introduction

the performance benefits are limited by the amount of parallelism that can be extracted from an application.

Memory Wall. One of the main challenges in the memory subsystem of an MPSoC, is the memory wall. The speed of the processor subsystem has been increasing at a much higher rate compared to the memory subsystem. Rephrasing the popular proverb, “A computing system is only as fast as its slowest component”. Therefore, any mismatch between the processing and memory subsystems degrades the performance of the entire computing system. For example, if an application performs a lot of memory accesses, the processor is forced into an idle state, while waiting for slow memory loads/stores. To overcome this problem, layers of cache memories are used to minimize memory access latencies.

1.1.2 Cache Memories

Computing systems typically store data in the main memory, which is made up of Dynamic Random Access Memory (DRAM) cells. Accessing the DRAM usually results in high memory access latencies, which is hidden by using cache memories. Caches are composed of Static Random Access Memory (SRAM) cells which respond to data requests much faster than DRAM memories. But SRAM-based memories require more transistors to hold data compared to DRAM-based memories, making them expensive in terms of area. Therefore, as a trade-off, cache memories are designed and dimensioned to hold a limited amount of important data, which can vary dynamically throughout an application’s execution.

Caches hold data using two basic principles namely, temporal locality and spatial locality. Various structural designs of a cache have been extensively explored, resulting in *directly mapped* caches or *N-way associative* caches, with the latter being used in a majority of MPSoC systems. The write-policy of caches have also been explored, producing two commonly used policies namely *write-through* or *write-back*. On modern MPSoCs, it is common to see a memory hierarchy consisting of several caches between the processor and main memory. Typically a small Level 1 (L1) cache is used for quick data access. The L1 cache is usually private to each core. This is followed by a larger Level 2 (L2) cache, which may be shared between multiple cores. If required, an even larger Level 3 (L3) cache is included. The cache at the last level of a given cache hierarchy is usually referred to as the Last Level Cache (LLC).

1.1.3 The Evolution of Manycore Architectures

Interconnect. MPSoC systems consisting of a small number of processing units ($\approx 2-8$), also referred to as multi-core systems, are usually interconnected using a bus topology. With an increase in the number of processing units, the available bus bandwidth saturates quickly. This reduces the efficiency of inter-processor communication, rendering the bus interconnect impractical for large number of cores. A Network-on-Chip (NoC) is used to cope with such limitations, leading to tile-based manycore architectures that

can accommodate a large number of processing units (several dozens) efficiently. Some tiled manycore architectures use both, a bus-based interconnect within a tile, and a NoC-based interconnect between tiles.

Memory Subsystem. The transition to tiled manycore architectures introduces Non-Uniform Memory Access (NUMA) latencies, as the distance of the main memory is not constant for all processing units. Furthermore, a single dedicated tile with main memory causes access bottlenecks similar to that of the bus interconnect. To address this problem, memories are commonly split-up and distributed into all tiles, creating a distributed memory architecture. In modern tiled manycore systems, it is common to see distributed memory architectures where both the main memory (DRAM) and a Tile Local Memory (TLM) (SRAM) are distributed among different tiles.

Parallel Programming Models. The introduction of distributed memory architectures opens up different methodologies that can be used to program them. They can be loosely classified into two parallel programming models namely, *message-passing* and *shared memory*. The *message-passing* model assumes that each tile in the distributed memory architecture has a private address space and communicates with other tiles using explicit software messages. While this approach avoids bottlenecks to shared resources, it increases the programming effort that is required to manage inter-tile communication. By using the standardized Message-Passing Interface (MPI), this additional effort can be reduced. However, the programmer still has to orchestrate all inter-tile communication (added programming effort), and the communication messages themselves can be inefficient as they are performed by system software. Furthermore, all existing applications which have mostly been written assuming a shared memory programming model would need to be modified to follow the MPI model.

Alternatively, the *shared memory* programming model views the distributed memory architecture from a shared memory perspective, making it a Distributed-Shared Memory (DSM) architecture. Here all tiles have a common view of the entire address space and inherently communicate via load/stores to the shared memory. Applications can continue using the shared memory programming paradigm, only if the underlying architecture can guarantee a coherent shared memory view of the common address space. The *shared memory* programming model comes with its challenges, such as mandatory cache coherence support, which is challenging especially for large manycore systems.

1.1.4 The Need for Cache Coherence

In tile-based manycore architectures, multiple processing units operate collectively to speed-up a given application. This may involve operating on common data from main memory, which passes through the cache hierarchy. As a consequence, multiple copies of the same data may be present locally in different caches across the entire manycore system. Now, when a processor modifies shared data in its local cache, this information needs to be conveyed to all other caches which also possess a copy of the shared data.

1 Introduction

Figure 1.1 demonstrates the need for cache coherence using a simple example. Initially, two processing elements (P0, P1) read from a *variable* residing in main memory ① ②. Now, the *variable* with the value x is present in both caches. When P0 issues a memory store operation ③, it modifies the *variable* value to y . Assuming write-through caches, this new *variable* value is updated in the main memory, but within P1's private cache, the *variable* value is still outdated ④. Therefore, a coherence protocol is needed to ensure that the data within both caches are always kept up-to-date. This allows the processors to confidently operate on shared data.

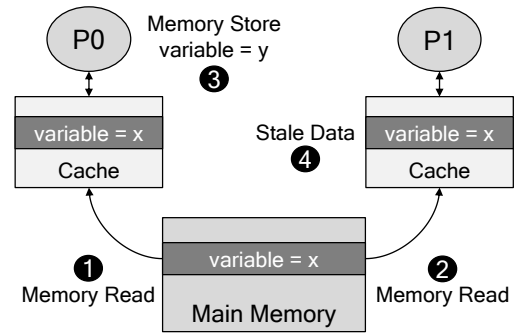


Figure 1.1: A simple example illustrating the need for cache coherence

Coherence Protocols. Caches are appended with additional bits which represent the state of a cache line¹. A rule-set is formulated to govern these states and their transitions, known as a coherence protocol. Extensive research has produced several coherence protocols such as Modified-Invalid (MI), Modified-Shared-Invalid (MSI), Modified-Exclusive-Shared-Invalid (MESI), Modified-Exclusive-Shared-Invalid-Forward (MESIF), Modified-Owner-Exclusive-Shared-Invalid (MOESI), etc., each representing the states of a cache line. More states undoubtedly optimize the coherence protocol, but significantly add to its design complexity. The coherence protocol is also influenced by properties like the cache's write policy (write-through or write-back) and how cache data is updated (cache-invalidate or cache-update). Further, the interconnect topology of the MPSoC also dictates the type of coherence support that can be offered.

Snoop-based Coherence. In a pure bus-based architecture, snooping coherence schemes are used to maintain cache coherence. The presence of a common bus medium makes all memory load/store requests transparent to the caches connected to the bus. Snoop-based coherence is relatively simple, and mostly used in small scale multi-core systems. Figure 1.2 illustrates a basic example of snoop-based coherence.

Directory-based Coherence. Snoop-based coherence cannot be applied for tiled many-core architectures, as the advantage of a common bus medium is lost. Therefore, directory-based coherence schemes are used. The directory is a hardware book-keeping database which holds the sharer information of all cached data. Therefore, all memory load/store requests explicitly communicate with the directory to keep data coherent. Directories use status bits and bit-vectors to support coherence on a cache line granularity. Figure 1.3 illustrates a basic example of directory-based coherence. In large tile-based manycore architectures, directories may be distributed among all tiles to avoid access bottlenecks, where each directory holds the sharer information for a predefined partition

¹Hardware-based cache coherence is commonly supported on a cache line granularity

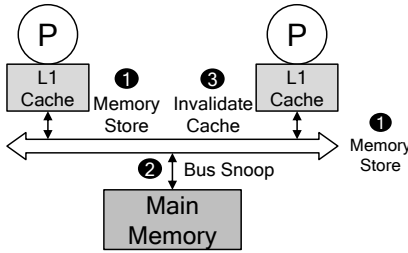


Figure 1.2: An example of snoop-based coherence

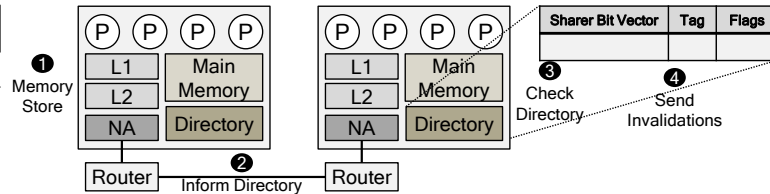


Figure 1.3: An example of directory-based coherence

of the main memory. Depending on how the memory address range is partitioned, the sharer information of a load/store request could reside in a different tile than the actual data. This introduces additional NoC hops, also known as home-node hops, that need to be traversed to access the sharer information. For the example in Figure 1.3, both the main memory and the directories are distributed among the tiles. If each directory is made responsible for the corresponding local main memory address range, additional home-node hops can be avoided. From a structural perspective, directories have been optimized for area using designs that limit the number of entries, similar to caches. Such optimized directory structures are known as *sparse directories* [9].

1.1.5 Cache Data Management

Cache structures offer quick memory access latencies, but are limited in capacity. Therefore, empty cache entries are seen as prime real-estate for application data. Depending on how data within a cache is managed, it can either have a positive or negative impact on an application's execution time. A cache replacement algorithm is used to maximize the efficiency of cache data management. Replacement algorithms have been extensively explored especially for cache memories, generating staple eviction policies such as LRU, Least Frequently Used (LFU), LRU Insertion Policy (LIP), First In First Out (FIFO) etc. By optimizing for eviction criteria such as recency, frequency, etc., these policies attempt to retain the most relevant data within the cache, for quick memory accesses.

1.1.6 Resource-Aware Computing

An increase in the number of processing and memory resources of modern manycore architectures reveals new challenges regarding its overall management. InvasIC [10] is a collaborative research project that investigates a resource-aware programming paradigm for manycore architectures. The InvasIC approach allows applications to request for hardware resources, which are granted by a run-time system based on availability. During the course of its execution, the InvasIC paradigm allows applications to dynamically expand, shrink, or even relocate to a different amount hardware resources, depending on the current status of the manycore system. After an application terminates, the

1 Introduction

run-time system retracts the hardware resources which can then be used by other applications. The goal of the InvasIC project is to efficiently manage the execution of several applications that are simultaneously requesting for hardware resources on a manycore architecture. In order to support these features, both software-based programming concepts and hardware-based architectural changes of the manycore system are required.

1.2 Problem Statements

This thesis tackles two research challenges, both of which optimize the memory subsystem of MPSoC platforms. The first challenge is to provide scalable hardware coherence support for large DSM-based tiled manycore architectures. The second challenge is to enhance cache data management for MPSoCs.

1.2.1 Scalable Cache Coherence

There has always been a debate on whether cache coherence is even needed for today's MPSoC systems. Authors of [11, 12, 13] argue that cache coherence, especially when supported by hardware mechanisms will not scale with increasing processor/tile counts, owing to its hardware overheads. Instead, they prefer MPI (no coherence) that moves away from the traditional shared memory programming paradigm or software-managed coherence. This thesis favours a cache coherent architecture due to its many advantages. One, it allows programmers to continue with the shared memory programming paradigm as opposed to MPI, where inter-tile communication has to be managed explicitly by the programmer. Secondly, most Operating Systems (OSs) (Linux, Windows) and even applications expect a cache coherent shared memory view of the underlying architecture. Therefore, a majority of leading chip manufacturers like Intel, AMD, ARM, etc. have mostly been designing fully cache coherent computing systems by default [14]. With a cache coherent system, software programmers can dedicate their time towards writing efficient code, without being bothered by the inconvenience of managing low-level application communication and synchronization.

Cache coherence can be supported either by software or hardware techniques. This thesis favours hardware-supported coherence methods as it exhibits better performance than software schemes [15, 16]. The paper *Why On-Chip Cache Coherence is Here to Stay* [14] reinforces these views on hardware-supported coherence methodologies. It carefully analyses the overheads of on-chip coherence and argues that with proper system design, hardware-supported coherence can be provided for the foreseeable future.

Of course, hardware-supported coherence comes with its challenges, especially with today's ever growing manycore architectures. Intra-tile coherence can be supported quite efficiently using snoop-based schemes, as the number of processing units within a tile does not increase drastically. But as the number of tiles increase, inter-tile coherence faces challenges such as design complexity, scalability and area overheads. This thesis proposes a scalable and flexible hardware-supported coherence methodology to overcome these challenges for DSM-based tiled manycore architectures.

1.2.2 Optimizing Cache Data Management

Standalone eviction policies such as LRU, LFU, LIP, FIFO etc. usually optimize for a single eviction criterion such as recency, frequency, etc. Therefore, they thrive when applications exhibit non-erratic data access patterns. Modern day applications are growing both in number and complexity, thereby increasing the diversity of data access patterns. Even within a single application, there are certain phases that exhibit contrasting memory access patterns. On manycore systems, multiple such applications are executed on the same platform (even simultaneously), further diversifying the memory access patterns. Such erratic access patterns can easily disorient an eviction algorithm, resulting in cache pollution/thrashing, subsequently degrading application performance.

This motivates the need for eviction algorithms that can handle varying data access patterns. The challenge is to provide a smart eviction policy which efficiently manages cache data by optimizing for multiple eviction criteria. This thesis proposes a hybrid eviction policy which is evaluated for LLCs in a multi-core system, as well as for sparse directories in a tile-based manycore architecture.

1.3 Contributions

The contributions of this thesis can be divided into two sub-topics. The first contribution is a scalable and flexible hardware-supported coherence methodology for DSM-based tiled manycore architectures. The second contribution is a hybrid eviction policy which enhances cache data management by combining several eviction criteria.

1.3.1 Region-based Cache Coherence (RBCC)

A majority of chips produced today offer hardware-supported coherence by default. They mostly offer global coherence, where all processing elements have a coherent view over the entire memory address space. Global coherence for tile-based manycore systems does not scale well with increasing core counts, as it greatly increases the memory overheads required by directory-based coherence schemes.

Taking a step back and questioning, “does a manycore system even need global coherence?” reveals insightful findings on the behaviour of multi-threaded applications. On manycore systems, a single multi-threaded application cannot efficiently utilize all available processing and memory resources [1]. In fact, manycore systems are designed to simultaneously execute several multi-threaded applications, each of which consume a certain share of processing and memory resources. These arguments deem global coherence unnecessary for large tile-based manycore architectures.

This thesis introduces RBCC, a concept that uses a divide-and-conquer approach to provide scalable and flexible coherence for tile-based manycore systems. The main idea of RBCC is to provide hardware-supported coherence for a limited number of tiles within a large manycore system. The subset of coherent tiles or *coherence regions* are configured based on the application’s requirements. Limiting coherence to a cluster of tiles has its advantages. One, it significantly reduces the book-keeping overheads required for

1 Introduction

directory-based inter-tile coherence. This sidesteps the scalability issues of hardware-supported coherence for tile-based manycore architectures. Two, by confining coherence to a subset of tiles, inter-tile communication and the corresponding coherence traffic are limited to within the coherence regions. Furthermore, the RBCC concept is designed to be dynamic and flexible. At run-time, the coherence regions can be re-configured or even relocated to span specific clusters of tiles, all controlled according to the application's requirements.

The contributions related to the RBCC concept, its design, implementation and evaluation were published at international conferences [17, 18, 5], a journal [19] and part of a book chapter [20].

1.3.2 Hybrid Voting-based Eviction Policy (HyVE)

Standalone eviction policies struggle to cope with cache pollution/thrashing when running several applications that exhibit non-uniform data access patterns. This thesis introduces HyVE, a novel framework that combines multiple standalone eviction policies together and uses concepts from the voting theory domain to decide on an eviction victim. HyVE uses multiple eviction criteria to optimize cache data management and has several advantages. One, almost any new or existing standalone eviction policy can be extended to be incorporated as part of HyVE. Two, by design, HyVE is a modular framework that can accommodate an arbitrary number of standalone eviction policies. This enables designers to build HyVE with the desired number and/or combination of constituent eviction policies that best suit the MPSoC platform and the applications. Three, by design, all constituent eviction policies within the HyVE framework can operate simultaneously. From a hardware implementation perspective, this allows HyVE's timing complexity to scale gracefully when accommodating several standalone eviction policies. This thesis evaluates HyVE using two case-studies: LLC evictions in a multi-core system and sparse directory replacement decisions in a DSM-based tiled manycore system.

The contributions related to the HyVE concept, its design, implementation and evaluation were published at international conferences [21, 22] and a journal [23]. Additionally, the HyVE concept has been submitted and published as an international patent [24].

1.4 Organization

The rest of this thesis describes the contributions in detail. Chapter 2 talks about other work in the field that are related to the contributions of this thesis. Chapter 3 describes the concept and features of RBCC, and how they have been realized through the Coherence Region Manager (CRM) module. The concept of RBCC is evaluated using a high-level simulation model, as well as on an FPGA prototype for different benchmarks. Chapter 4 describes the concept, features and implementation of the HyVE framework. HyVE is evaluated for Last Level Caches (LLCs) as well as for sparse directory structures using a simulator and on an FPGA prototype. Chapter 5 summarizes this thesis and talks about how RBCC and HyVE can be further improved.

2 State of the Art

2.1 Cache Coherence

Cache memories bridge the performance gap between the processor and memory subsystem. Manycore systems are usually designed with a cache hierarchy consisting of both private and shared cache memories. Incoherent cache data tend to corrupt the functionality of applications, which makes cache coherence support one of the vital features of a manycore system. The presence of cache coherence both influences, and is influenced by the following parameters:

- The interconnect topology of the manycore system,
- The parallel programming model of the manycore system

Early manycore architectures consisted of a small number of processing elements, where inter-processor communication could be efficiently conveyed using a shared bus medium. Bus-based manycore architecture exhibit Uniform Memory Access (UMA) latencies and are kept coherent using hardware-based snooping coherence schemes [25]. Snoop-based coherence is relatively easy to implement as it does not incur much area overheads, nor does it induce costly coherence-related messages. This enables application developers to use the shared memory model to easily program such manycore systems.

With the increase in number of processing and memory resources, modern manycore systems have evolved into distributed tile-based architectures. This evolution brings-in a NoC-based interconnect, triggering a transition from UMA to NUMA latencies. From a coherence perspective, the advantage of the common bus is medium lost, necessitating alternative coherence solutions using directory structures [26].

Compared to snoop-based schemes, directory-based coherence is much more complex to implement and comes with overheads such as additional directory look-up times and increased NoC traffic for coherence messages. As explained in Section 1.1.4, directories can be designed either as a unified structure or can be distributed among different tiles. The former approach could lead to directory access bottlenecks in large systems, and the latter might induce additional home-node hops. Further, hardware-supported directory-based coherence schemes consume significant area overheads that grow with the size of the manycore system. However, from a performance perspective, hardware-based directory coherence schemes are usually better than the software alternatives [14].

Therefore, depending on these trade-off factors, a tile-based manycore systems can be designed to either provide hardware coherence support or not. This is an important design choice, which in-turn influences the type of parallel programming model to be used:

2 State of the Art

- No hardware coherence support
 - Use an MPI-based programming model
 - Use software coherence schemes → shared memory programming model
- Hardware coherence support → shared memory programming model

2.1.1 No Hardware Coherence Support

If a manycore architecture does not support hardware coherence, there are yet two alternatives on how it can be programmed. Application developers can leverage MPI-based techniques for inter-tile communication and synchronization. Else, software coherence schemes can be used to continue with the shared memory programming model.

2.1.1.1 MPI-based Communication

Distributed memory manycore architectures are commonly used in the High Performance Computing (HPC) domain. Applications are parallelized over several computing nodes that are part of a large cluster configuration. Due to the lack of coherence support, application developers use explicit software messages to orchestrate inter-tile communication, at the cost of additional programming effort. These efforts can be partially reduced by making use of standard MPI libraries [27]. The Single-Chip Cloud Computer (SCC) developed by Intel [28] is one example that makes use of the MPI programming model. Kalray’s Massively Parallel Processor Array (MPPA) [29] is another example of a manycore architecture which uses MPI for inter-tile communication.

While MPI-based programming models limit hardware area overheads, they do come with certain challenges [30, 31]. The use of MPI-based programming models require explicit communication messages that are usually performed by system software, thereby adding to the application’s execution time. Further, the MPI model increases the programming effort for application developers, compared to the shared memory model. Since a majority of applications have been developed assuming a shared memory programming model, additional programming effort would be needed to port such legacy applications onto a non-coherent manycore architecture.

2.1.1.2 Software-based Coherence Schemes

Software enabled coherence mechanisms enable application developers to view the distributed memory manycore architecture from a shared memory programming perspective, i.e., as a DSM manycore architecture. Software coherence schemes act as an intermediate solution by offering a shared memory view of the underlying system, whilst avoiding the hardware area overheads and implementation effort. Software coherence schemes are implemented at different abstraction levels as stated in [32]. The works *mirage* [33] and *munin* [34] integrate software coherence support as part of the OS and run-time system respectively. The works *vote for peace* [35] and *IVY* [36] expose a

library-based interface to applications, which can be used to synchronize shared memory accesses. Languages such as *X10* [37] have been designed with specific constructs that orchestrate and synchronize memory accesses to shared data.

Software-based coherence schemes allow applications to continue with the shared memory programming paradigm, when an MPSoC does not provide hardware coherence support. However they lack the performance benefits of their hardware counterparts [14, 15, 16]. This can be seen in almost every flagship product by major commercial vendors like Intel, AMD, ARM, etc., who make use of hardware supported coherence mechanisms. It is important to note that this in no way rules out the usability of software coherence methodologies. For instance, a hybrid hardware-software coherence scheme would be beneficial, especially for large manycore architectures. Coherence within a cluster of tiles (coherence region) could be supported by hardware, while software coherence methodologies operate between the different coherence regions.

2.1.2 Hardware Coherence Support

Hardware coherence for tiled manycore systems was introduced by the Stanford Directory Architecture for Shared Memory (DASH) team [26]. They use a 2x2 tile-based design, where each tile contains 1 processing element and a two-level cache hierarchy. Every tile also has access to its own main memory and Input/Output (I/O) peripherals. The DASH multi-processor was primarily used as a platform to introduce and evaluate inter-tile coherence using a directory-based scheme. Therefore, a directory unit is present per tile, which is responsible to hold the sharer information of its corresponding main memory address space. This allowed applications developers to have a shared memory view over a distributed memory manycore architectures.

Tilera’s TILEPro64[™] [38, 39] is an example of a commercially deployed tile-based manycore architecture that supports hardware-based coherence. The architecture contains 64 tiles, each with 1 processing element, and a cache hierarchy of L1 and L2 caches. The 64 tiles are connected together using a 2D-mesh topology consisting of 5 independent networks. The TILEPro64 uses a Dynamic Distributed Cache (DDC[™]) mechanism to support inter-tile coherence, which allows each processor to have a shared memory view over all L2 caches. This effectively transforms the L2 cache within each tile into a system-wide shared cache. The inter-tile coherence messages are transported exclusively using one of the 5 independent networks. Interestingly, Tilera was bought by Mellanox Technologies, which was in-turn acquired by NVIDIA, to boost their presence in the HPC domain.

Cavium’s OCTEON[®] [40] is a commercial manycore architecture that supports hardware coherence. The OCTEON II contains 32 cores, where each core is equipped with a private L1 cache. The cores are connected together with a crossbar interconnect, where they can access a shared L2 cache. The L2 cache controller is responsible to ensure cache coherence for the entire system. Cavium is now part of Marvell Technologies, where the OCTEON multi-processor is used for network processing applications.

Another commercial example pushing for shared memory programming on distributed memory manycore architectures using hardware coherence schemes is Numascale [30, 31].

Numascale is an up-and-coming company that designs a unique NumaConnect™ node controller. This node controller allows processor manufacturers to “scale-up” several computing nodes in a tightly-coupled manner, as opposed to “scaling-out” computing nodes into distributed cluster machines.

However, as the size of manycore systems increase, the book-keeping area overheads required for directory-based coherence also grow [41]. Assuming that hardware coherence is supported globally with an exact representation of the sharer information (1 bit per sharer), the directory area overheads increase quadratically with increasing processor counts [41]. Depending on the available hardware real-estate of the chip, this may lead to scalability issues especially for large manycore systems.

2.1.2.1 Overcoming Scalability Limitations

Research on reducing the hardware area overheads for directory-based coherence mostly focus on two directory optimization techniques namely *directory-width reduction* and *directory-height reduction*. The first approach narrows the directory width by using limited pointer or coarse vector schemes to represent the sharer information [9]. While these approaches reduce the width of the directory, they come with certain consequences. Limited pointer schemes require costly linked-list traversals that add to the latency of coherence management. Coarse vector schemes hold inexact sharing information, forcing them to conservatively send-out more coherence messages than required. The second approach shrinks the directory height by using a sparse design. This approach significantly reduces the required hardware area overheads by limiting the number of directory entries. This introduces a capacity problem similar to data caches, thus requiring a replacement algorithm, where the penalty for each eviction is Directory-induced Invalidations (*dirInvs*). There are also several alternative directory designs [42, 43, 44, 45, 46] that optimize for area and eviction policies. These are discussed in detail within the context of eviction policies for sparse directories in Section 2.2.4.

2.1.2.2 Alternatives to Global Coherence

Exploring alternatives for global coherence schemes is motivated by two arguments:

1. To avoid directory scalability limitations altogether,
2. Multi-threaded applications do not efficiently utilize all processing and memory resources of a large manycore system ¹.

The first argument is orthogonal to the *directory-height* and *directory-width* reduction solutions that were discussed previously. The second argument is supported by the work in [1], which reports the speedup of the PARSEC benchmarks for increasing thread counts. Figure 2.1 shows that most benchmarks do not benefit for thread counts approximately > 16 to 48 ². In fact, some benchmarks even suffer from performance

¹Embarrassingly parallel workloads are exceptions, which are considered to be a minority

²Note that the speedup saturates much faster for smaller input set sizes

2.1 Cache Coherence

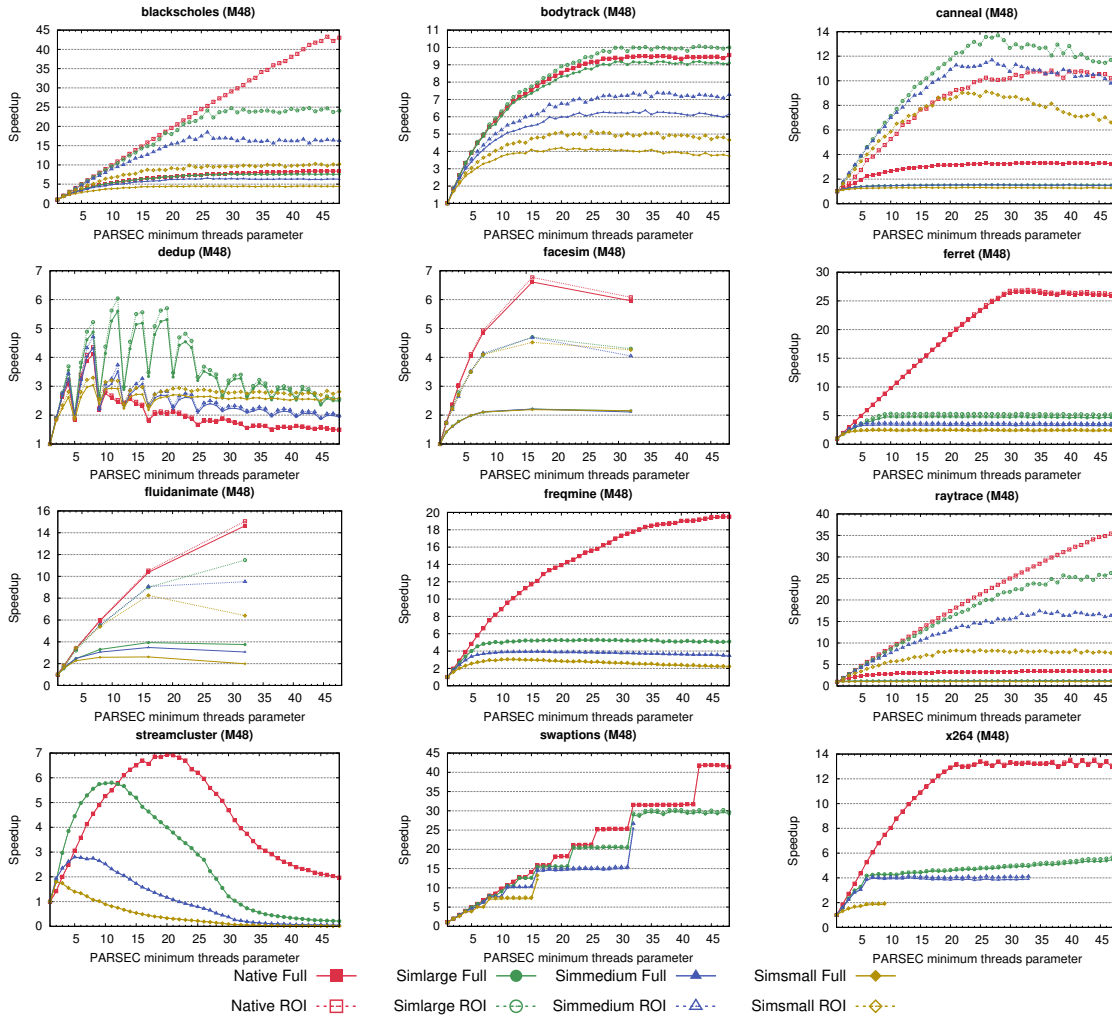


Figure 2.1: The speedup of several PARSEC benchmarks executed using different input set sizes for increasing thread counts, taken from [1] © 2016 IEEE

degradation with increasing thread counts. Together, both arguments question the need for global coherence in manycore systems, leading to architectures that confine coherence coverage to certain on-chip resources.

The Intel[®] Xeon Phi [47] is a 36 tile manycore system with 2 processing elements per tile. It offers various clustering modes that can be used to configure a group of tiles as part of a coherent NUMA domain. The cluster configuration process is a boot-time decision, and is limited to a selectable preset of configuration modes. Authors of [48] propose Coherence Domain Restriction (CDR), which restricts coherence to a cluster of tiles for MPSoCs. The CDR approach limits the number of sharers that can be part of a coherence domain, thereby limiting the necessary area overheads required for inter-tile coherence. CDR also explores limiting the number and location of the home-node to localize inter-tile communication traffic.

2.1.3 How is RBCC Different?

The RBCC concept is orthogonal to the discussed directory optimization schemes as it challenges the need for global coherence. Therefore, it can be classified under the “alternative to global coherence” category, . Compared to the Intel®Xeon Phi, RBCC supports run-time coherence region (re)configurations, as opposed to boot-time. This allows the coherence regions(s) to dynamically adapt to the application’s requirements. The work on CDR is quite similar to that of RBCC, i.e., both approaches allow restricting the coherence domain at run-time. However, CDR has been designed and evaluated for a manycore architecture with unified shared memory, which brings in the necessity of a home-node for directories. The RBCC concept is designed for a DSM-based tiled manycore system. The directories are therefore designed without the need for a home-node, alleviating the network-hops or look-up time that may be required to locate and access the home-node.

2.2 Eviction Policies

Generally, any design structure that experiences more demand than it can fulfil, requires a management policy to mitigate the mismatch. For example, in computing systems, cache structures are used to bridge the performance gap between processors and main memory. A cache is an invaluable hardware resource due to its quick memory access latency and limited capacity. Therefore, an eviction policy is used to manage the limited real-estate of a cache.

Primarily, eviction policies have been heavily mined in the context of cache memories as they greatly influence application performance. Existing cache eviction policies are either re-used or adapted to fit other hardware structures like sparse directories or Translation Lookaside Buffers (TLBs). This thesis focuses on eviction policies for cache memories and sparse directories. The terms eviction and replacement are used interchangeably throughout this thesis.

In 1966, László Bélády introduced the MIN algorithm [49] which provides the ideal solution for the cache replacement problem. However, Bélády's algorithm requires complete knowledge of all future memory access patterns in order to function correctly. As this is impossible to implement, Bélády's optimum solution is commonly used as a yardstick to measure the performance of other cache replacement policies. The performance of an eviction policy significantly depends on the memory access patterns exhibited by the workloads. Figure 2.2 [2] shows four commonly seen workload memory access patterns and their impact on different cache sizes. Cache-friendly workloads exhibit a high degree of temporal and spatial locality. Such memory access patterns benefit from an increase in cache capacity. Cache-fitting workloads, as the name states, are workloads whose data-sets can be completely accommodated into the cache. If the data-set sizes exceed the cache capacity, then the memory access patterns of such workloads will most likely cause cache-thrashing. The last example are streaming workloads which rarely exhibit any temporal or spatial locality properties. Such memory access patterns do not benefit from an increase in cache capacity. In order to cater to such diverse memory access patterns, cache eviction policies have been extensively explored, producing several replacement algorithms. Figure 2.3 illustrates a taxonomy of cache replacement policies.

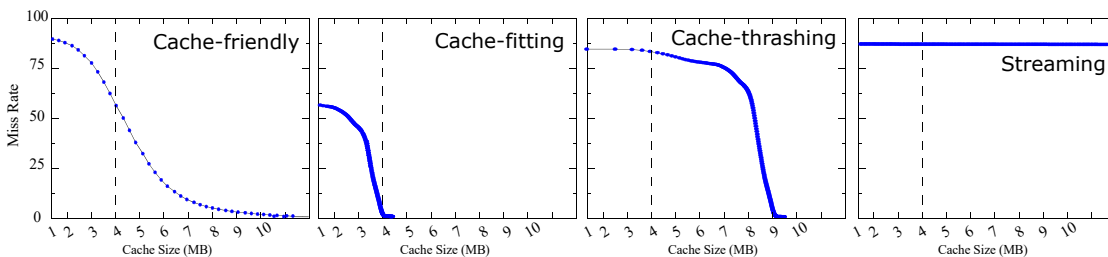


Figure 2.2: The diversity of different memory access patterns for increasing cache sizes using the LRU policy, taken from [2] © 2008 IEEE

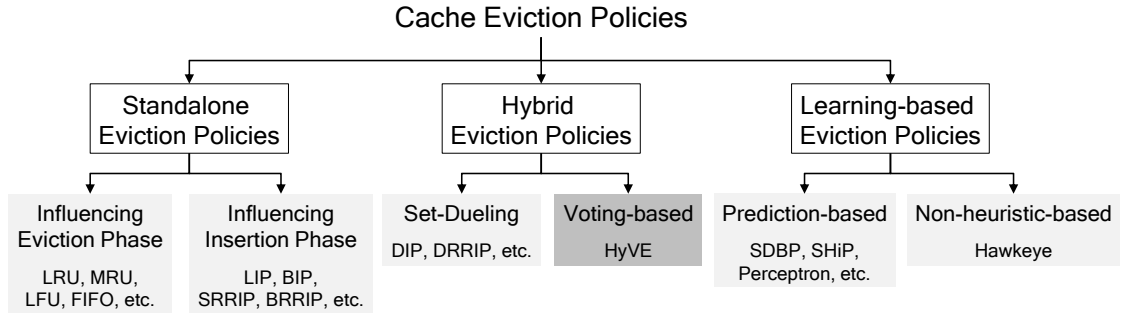


Figure 2.3: A taxonomy of cache eviction policies

2.2.1 Standalone Cache Eviction Policies

This category of cache eviction policies are designed to take eviction decisions based on a single eviction criterion. Commonly used eviction policies include LRU and LFU that utilize recency and frequency as the eviction criterion respectively. An eviction policy can be designed to influence a given cache block at different stages of its lifespan. Authors of [3] have classified this into four phases, which is also illustrated in Figure 2.4:

- *The Insertion Phase:* The eviction policy assigns/modifies the eviction metric of the cache block upon insertion into the cache memory.
- *The Promotion Phase:* The eviction policy modifies the eviction metric of the cache block upon a cache hit.
- *The Aging Phase:* The eviction policy modifies the eviction metric of the cache block relative to other cache blocks.
- *The Eviction Phase:* The eviction policy utilizes the eviction metric of the cache block to make an eviction decision.

Eviction policies like LRU, Most Recently Used (MRU), LFU, etc. do not influence the cache block in the insertion phase. They insert incoming memory blocks into the MRU position, protecting it from immediate evictions. While this might be favourable for cache-friendly memory access patterns, it could lead to cache pollution for others.

Authors of [4] introduce two eviction policies - LIP and Bimodal Insertion Policy (BIP), that additionally modify the eviction metric of a memory block upon insertion into

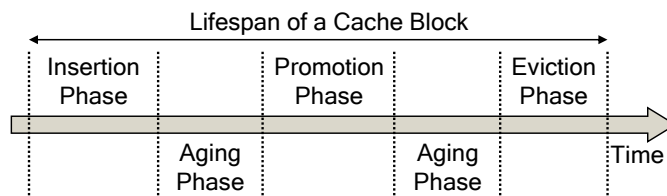


Figure 2.4: The lifespan of a cache block with different phases, adapted from [3]

the cache. LIP is similar to the LRU policy, but inserts incoming memory blocks into the LRU position instead. BIP builds upon this idea by inserting incoming memory blocks either into the LRU position or the MRU position based on a predefined probability factor. During the replacement phase, both policies evict cache blocks similar to the LRU policy. Both LIP and BIP offer resistance to cache thrashing, with the latter additionally capable of adapting to changes in the working-set.

Authors of [50] introduce two further eviction policies - Static Re-Reference Interval Prediction (SRRIP) and Bimodal Re-Reference Interval Prediction (BRRIP), that also influence memory blocks during the insertion phase. These eviction policies can be viewed as fine-granular versions of the Not Recently Used (NRU) policy. Both eviction policies use a N -bit Re-Reference Prediction Value (RRPV) to predict the re-reference interval of incoming memory blocks. SRRIP inserts incoming memory blocks with an RRPV of 2^{N-2} , indicating that it will be re-referenced in the distant-future. The RRPV is reset upon cache hits, indicating that the cache block will be re-referenced in the near-future. During the replacement phase, SRRIP evicts the cache block with an RRPV of 2^{N-1} . If not present, the RRPV of all cache blocks are incremented. BRRIP is an adaptive version of SRRIP, where incoming memory blocks are inserted with an RRPV of 2^{N-1} or 2^{N-2} based on a predefined probability factor. SRRIP offers scan-resistance (streaming workloads) while BRRIP is thrash-resistant.

The standalone eviction policies make eviction decisions based on a single eviction criterion. Therefore, they might be unable to optimize the cache performance for a wide range of application access patterns. In order to solve this, several policies are combined together to form hybrid eviction policies.

2.2.2 Hybrid Cache Eviction Policies

The goal of hybrid cache eviction policies is to combine the positive attributes of different standalone eviction policies, attempting to optimize for varying application access patterns. Authors of the Adaptive Replacement Cache (ARC) [51] blend LRU and LFU together, in order to utilize both recency and frequency metrics for the eviction process. ARC combines the two eviction policies together by maintaining two dedicated lists for the recency and frequency eviction metrics. Authors of [52] also combine LRU and LFU together using a different approach. They use a weighted methodology to factor-in the recency and frequency eviction metrics.

The Concept of Set-Dueling (SD)

A well-known and comprehensively-explored concept to combine two standalone eviction policies is Set-Dueling (SD) [4]. The primary idea of SD is based on the concept of Dynamic Set Sampling (DSS) [53] which shows that the overall behaviour of a cache memory can be determined by a subset of cache lines/blocks. Using this principle, SD classifies the cache memory into leader-sets and follower-sets, illustrated in Figure 2.5. The leader-sets are further divided equally among the two competing standalone eviction policies. A cache miss on any leader-set either increments or decrements the shared

Policy Selector (PSEL) counter as shown in Figure 2.5. The value of the PSEL counter indicates the cache memory’s preferred standalone eviction policy. Depending on this value, the follower-sets select the preferred standalone eviction policy for their eviction process. The value of PSEL counter varies dynamically throughout the course of an application, thereby guiding the follower-sets to choose the better performing standalone eviction policy.

Two popular hybrid cache eviction policies using SD are Dynamic Insertion Policy (DIP) and Dynamic Re-Reference Interval Prediction (DRRIP). DIP combines LRU and BIP together using the concept of SD. Therefore, DIP attempts to offer both recency-friendliness from the LRU policy and thrash-resistance from BIP. The DRRIP policy uses SRRIP and BRRIP as the dueling policies for the two leader-sets. Therefore, it offers both scan-resistance and thrash-resistance properties. The SD concept has also been used to combine recency and frequency metrics by mixing different segmented LRU versions [54, 55] together.

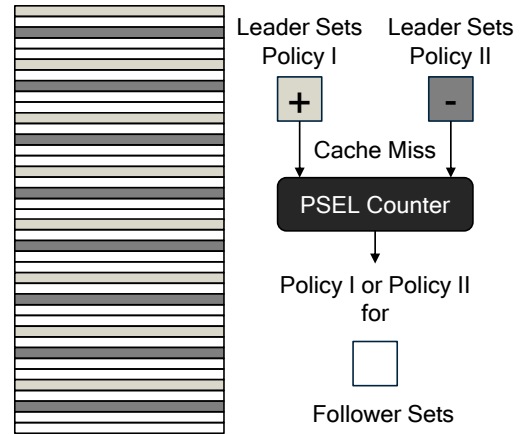


Figure 2.5: An example illustrating the concept of Set-Dueling (SD), adapted from [4]

2.2.3 Learning-based Cache Eviction Policies

Cache replacement policies have also been explored using learning-based techniques. Such cache eviction policies usually make use of a learning-subsystem to accurately predict the insertion values for incoming memory blocks.

The work in [56] uses a Sampling Dead Block Prediction (SDBP) to determine the usefulness of an incoming memory block. SDBP makes use of the program counter for its predictions. Similarly, the work in [57] uses a Signature-based Hit Predictor (SHiP) to determine the re-reference interval of a given cache block. Factors such as the memory region, program counter and the instruction sequence are used to predict the RRPV value for SRRIP. The works in [58, 59] alternatively use a perceptron-based learning technique to enhance the accuracy of predictors. This allows for better predictions as multiple features are used to determine the re-usability of a cache block. Authors of [60] introduce Hawkeye, which does not use heuristics to determine the re-reference interval of a cache block. Instead, it records memory accesses patterns and applies an algorithm similar to Bélády’s MIN solution to determine if an incoming memory block is cache-friendly or cache-averse.

In general, cache eviction policies have been heavily-mined by the research community, producing several innovative techniques. The aforementioned eviction techniques cover all types of cache eviction policies that are relevant in the scope of this thesis. For interested readers, who would want to delve deeper, the work in [3] is recommended,

as it provides a good summary on cache eviction policies including several alternative approaches.

2.2.4 Eviction Policies for Sparse Directories

More than two decades ago, directory-based coherence [26] was introduced by the Stanford DASH team [61, 62]. Next, these directory structures were subjected to various optimization techniques. The directory-width was optimized using coarse vector schemes and the directory-height was optimized by using sparse directory structures [9]. The introduction of sparse directories also brought-in the need for eviction decisions, due to potential conflicts and limited capacity. Eviction policies have been mostly explored in the context of cache memories. Therefore, the work in [9] uses existing replacement strategies such as LRU, Least Recently Accessed (LRA) and even a random scheme to evaluate their performance for sparse directories. Explorations concluded that the LRU policy is the best performer [9]. Further research to optimize sparse directory structures have produced several alternative/contemporary directory designs that minimize and/or even avoid the need for evictions.

Contemporary Directory Designs

The work in [63] uses a Directory Look-aside Table (DLT) to reduce sparse directory evictions. This additional DLT structure holds replacement candidates that are mapped using hash functions. If both the sparse directory and the DLT are full, the eviction candidate is selected using the LRU policy.

The *select directory* [42] design reduces the area consumption of sparse directories. Authors of [42] observe that a majority of memory blocks are temporarily private. Leveraging this principle, the *select directory* decouples the tag-array and data-array, thereby allowing for an overall smaller directory structure. The *select directory* uses the LRU policy for its eviction decisions. The *stash directory* [43] design minimizes evictions by suppressing *dirInvs* for private memory blocks. The *stash directory* also used the LRU policy for its eviction decisions. The *tiny directory* [64] design optimizes a sparse directory for area consumption. Upon reaching its maximum capacity, the *tiny directory* borrows bits from the LLC. The *tiny directory* uses a ratio of corrupted shared LLC reads coupled with an NRU policy for its eviction decisions.

The *cuckoo directory* [44] design aims to minimize both area consumption and the number of *dirInvs*. It employs a hash table and avoids evicting candidates entirely by re-inserting them to other non-conflicting positions. If a non-conflicting position is not found, the *cuckoo directory* uses a random replacement scheme. The *ZCache* [65] design interprets cache memories in a different way. It allows for higher associativity with smaller number of ways by using different hashing functions per way. The *ZCache* increases the number of viable replacement candidates across multiple cache ways. This allows it to find a suitable eviction victim which is decided by the LRU policy. The Scalable Coherence Directory (SCD) [46] borrows concepts from the *ZCache* design and

boasts better scalability and energy efficiency than the *cuckoo directory*. The SCD design incurs minimal *dirInvs*, but uses a random replacement policy when required.

2.2.5 How is HyVE Different?

The contribution of this thesis, HyVE, can be classified under the hybrid eviction policy category, similar to SD. While combining standalone eviction policies together using SD leads to effective and efficient solutions, it is limited to two eviction policies. By design, HyVE supports several standalone eviction policies to be incorporated together. This enables HyVE to potentially cover more optimization attributes. Compared to learning-based policies, HyVE does not use a learning subsystem, rather, it resolves to a consensus among its constituent eviction policies to decide on an eviction victim. HyVE can be seen as an alternative non-learning based approach for solving the cache replacement problem. Nevertheless, all three of SD, learning-based policies or HyVE are conceptually complementary approaches designed to overcome the cache replacement problem, each with certain advantages and disadvantages, as will be showcased in Section 4.4.6.

In the context of sparse directory designs, all aforementioned works either use the LRU policy or a random scheme to make eviction decisions. HyVE is a hybrid eviction policy that attempts to optimize for several sparse directory specific optimization criteria, as will be shown in Section 4.5. HyVE is evaluated for a regular sparse directory design that uses a bit-vector scheme to represent the exact sharer information, similar to [9]. Using a generic sparse directory design allows experimental analyses to solely focus on HyVE's characteristics, without involving the additional properties of contemporary directory designs. It is important to note that HyVE is not limited to generic sparse directory designs and can be applied to any directory design that requires a replacement decision.

3 Region-based Cache Coherence (RBCC)

3.1 The RBCC Concept

The fundamental idea of RBCC is to support the shared memory programming model, but with an alternative solution compared to global coherence. Global coherence not only suffers from scalability issues, but isn't even required, especially when a single application cannot efficiently utilize all available processing and memory resources of a manycore architecture. Table 3.1 summarizes the different coherence alternatives and their implications for large tile-based manycore architectures. The goal of RBCC is to enable scalable and flexible hardware-supported coherence for large manycore systems.

The RBCC concept is agnostic to architecture parameters such as, the type of memory, the number of memories, the number of processing units, the number of tiles and the type of interconnect. It is also independent of cache parameters like the number of levels in the cache hierarchy, the cache configuration and the coherence protocol. RBCC is applicable to any manycore system which requires hardware-supported inter-tile coherence. In order to design RBCC, evaluate its benefits and assess its challenges, a generic tile-based MPSoC platform is chosen.

3.1.1 Target Architecture

Figure 3.1 illustrates an MPSoC platform which serves as the target architecture to evaluate the RBCC concept. It is a heterogeneous tile-based manycore architecture consisting of two different types of memories - an SRAM-based distributed TLM as well as a DRAM-based main memory. The tiles are broadly classified as compute tiles and I/O tiles. Each computing tile consists of four LEON cores that are tightly coupled with private L1 caches, a shared L2 cache and a TLM. The I/O tile is basically a compute tile with additional modules such as off-chip main memory or an Ethernet interface.

The L1 caches hold data from the local TLM or main memory to speedup local memory accesses. A shared L2 cache is used to hold data either from remote TLMs or the main memory, to speedup remote memory accesses. The modules within a tile are connected using an Advanced High-performance Bus (AHB) interconnect which follows the

Table 3.1: Revisiting coherence alternatives for large tile-based manycore architectures

No Coherence	Global Coherence	RBCC
MPI-based Software Communication	Communication via Shared Memory	
Additional Programming Effort	Relatively Lower Programming Effort	
No Hardware Overheads	High Hardware Overheads	Low-to-Moderate Hardware Overheads

3 Region-based Cache Coherence (RBCC)

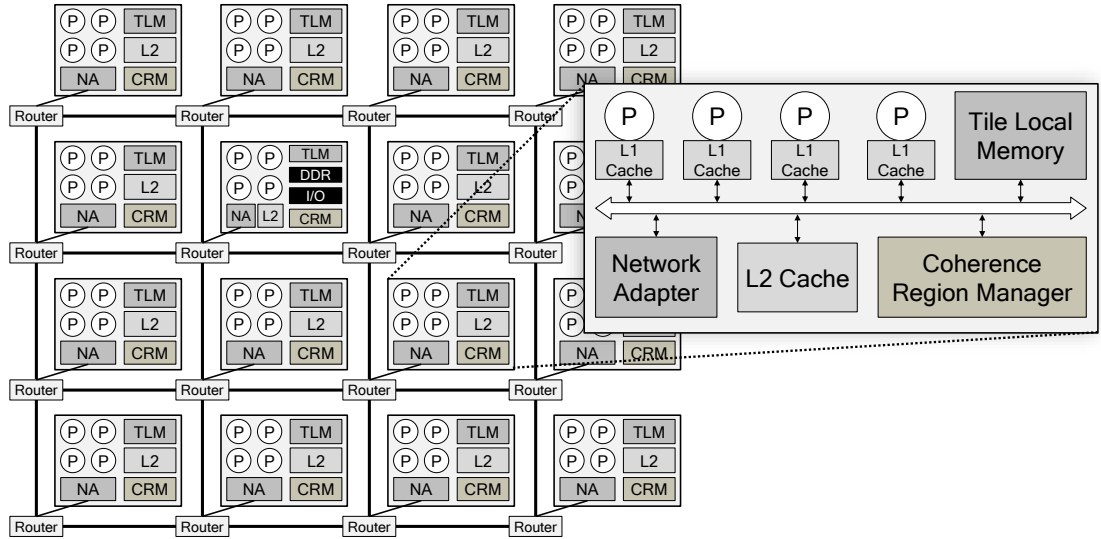


Figure 3.1: A heterogeneous DSM-based tiled manycore architecture used to evaluate RBCC. This architecture is also used in as part of the InvasIC project

Advanced Microcontroller Bus Architecture (AMBA) standard. A NoC-based interconnect is used to establish communication between different tiles. For this, a network adapter sub-module which is present within each tile acts as a gateway to send/receive information via the routers of the NoC.

Due to the presence of a shared bus medium, a snoop-based scheme is used to provide intra-tile coherence support. The L1 caches are configured with a write-through policy, and snoop on the AHB bus to invalidate themselves when necessary using the MI coherence protocol. The L2 cache is configured with a write-back policy and uses the MSI coherence protocol. Unlike the L1 caches, the L2 caches of different tiles do not share a common interconnect medium. Therefore, a directory-based scheme is required to keep them coherent. Here, RBCC is used as an alternative to global coherence, to provide a coherent view among the L2 caches, TLMs and the main memory.

3.2 RBCC Features

3.2.1 Scalability

RBCC provides inter-tile coherence support for a selectable cluster of tiles within a large manycore system. By confining coherence to within certain regions, it reduces the directory overheads that are required to maintain coherence. An example of a typical sparse directory structure is illustrated in Figure 3.2. The number of entries (height) of the directory is a design-time decision ¹, that can be represented as 2^I entries, where I is the number of index bits required to address the directory. Each entry of the sparse

¹It is a trade-off between the number of cache lines expected to be tracked and area consumption

directory holds the sharer information and auxiliary data for a given cache line. The width of each entry is computed as follows.

A *bit-vector* field is used to represent the number of sharers for a given cache line. The length of this field is given by $(N - 1)$, where N represents the number of tiles present in the manycore architecture. This field grows linearly for large manycore systems. A *tag* field T (bits), holding part of the memory address is stored to uniquely identify the cache line. The length of this field is given by $(AS - O - I)$, where AS is the

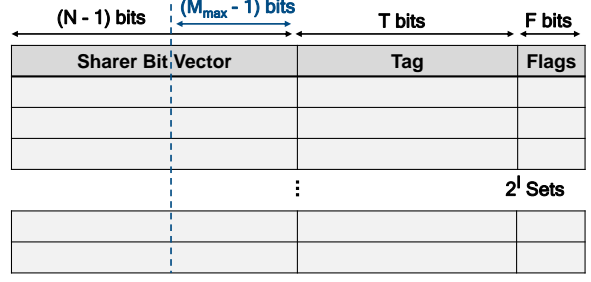


Figure 3.2: An example sparse directory structure

size of memory address space (32 bit, 64 bit, etc.) and O is the number of offset bits². Lastly, a *flag* field F (bits) is used to store additional information like the state or validity of the sharer information. The length of this field is design dependent.

If inter-tile coherence is offered globally for a manycore system with N tiles, the area consumed by the directory is given by:

$$directoryArea(GC) = [(N - 1) + T + F] * 2^I \quad (3.1)$$

By switching to inter-tile coherence support though the RBCC concept, the directory area consumption is given by:

$$directoryArea(RBCC) = [(M_{max} - 1) + T + F] * 2^I \quad (3.2)$$

where M_{max} is the maximum size of a coherence region. Therefore, the directory area reduced by using the RBCC concept compared to global coherence is given by:

$$directoryReduction = \left[1 - \frac{directoryArea(RBCC)}{directoryArea(GC)} \right] \quad (3.3)$$

$$directoryReduction(\%) = \left[1 - \frac{[(M_{max} - 1) + T + F]}{[(N - 1) + T + F]} \right] * 100$$

To better understand and visualize the directory reductions, a sparse directory with the following parameters is configured - 32Ki entries, a memory address space of 32 bits, offset of 5 bits, and a flag of 2 bits. The total number of tiles in the manycore system is varied from $N = 16$ to $N = 1024$. Depending on the total number of tiles, the maximum number of tiles within a single coherence region is varied from $M_{max} = 4$ to $M_{max} = N/2$. The directory reductions for this configuration is visualized in Figure 3.3. The magnitude of the directory reductions are mostly influenced by N and M_{max} . M_{max} is a configurable parameter that should be determined at design-time. It represents the

²The number of offset bits is determined by the size of a cache line

3 Region-based Cache Coherence (RBCC)

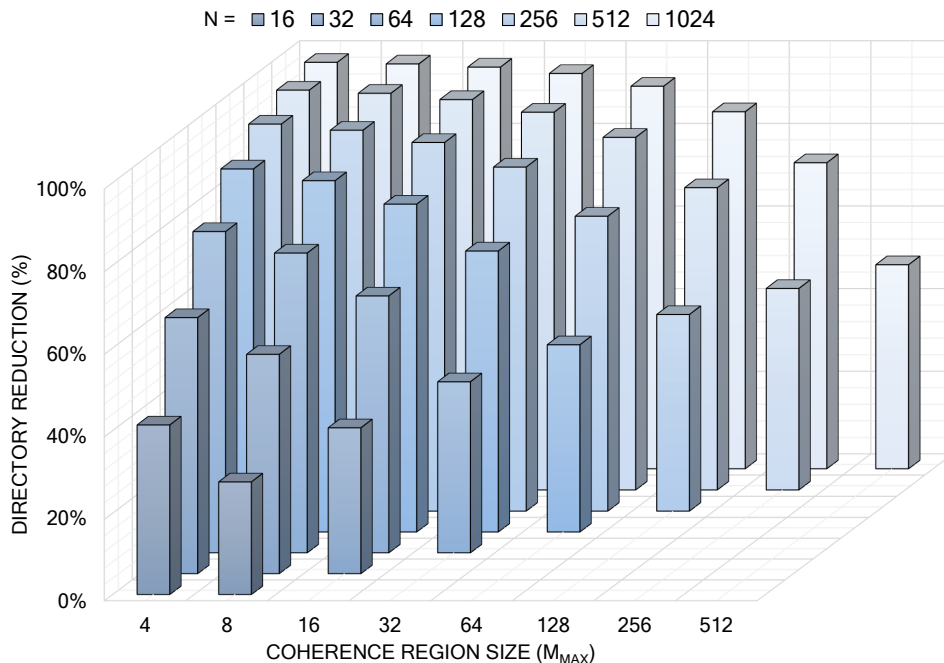


Figure 3.3: A visualization of the achievable directory area reductions when using RBCC compared to global coherence for different N and M_{max} combinations

maximum size of a single coherence region which can be decided by profiling the class of applications that are expected to run on the given manycore system. For example, a manycore system with $N = 16$ tiles and $M_{max} = 4$ tiles would reduce the directory overheads by 41.4% compared to global coherence. Further, the directory reductions significantly increase when $N \gg M_{max}$. This result is in line with the fundamental idea of RBCC, which is to offer inter-tile coherence to a subset of tiles within a large manycore system. Therefore, large manycore systems with moderately sized coherence regions will significantly benefit from using the RBCC concept compared to global coherence, making it scalable.

3.2.2 Flexibility

RBCC ensures scalability by limiting inter-tile coherence to a subset of tiles within a large manycore system. A configurable parameter M_{max} denotes the maximum number of tiles that can be part of a single coherence region. In order to ensure directory reductions, the size of M_{max} should be decided at design-time. For example, consider a manycore system consisting of $N = 16$ tiles and the maximum size of a coherence region $M_{max} = 4$ tiles. Now the question is “which 4 tiles of the available 16 tiles make up a coherence region”. Answering this question also at design-time would result in static coherence regions, similar to the Intel[®] Xeon Phi where a NUMA domain is configurable at boot-time [47]. By using such an approach, applications would need to

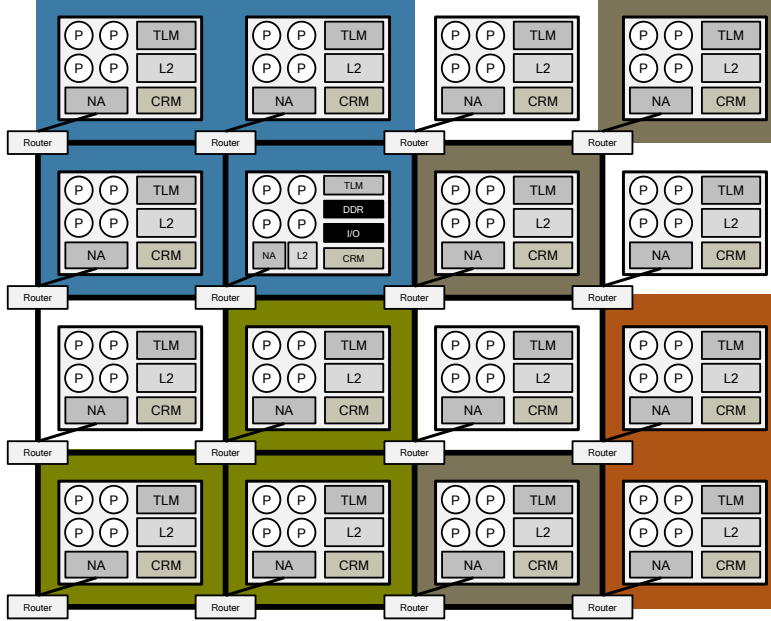


Figure 3.4: The InvasIC tile-based manycore architecture with several coherence regions

be restrictively mapped to only certain pre-defined coherence regions, which may result in under-utilization of the manycore system.

RBCC sidesteps this hurdle by enabling flexible coherence regions that can be configured at run-time. The idea is to allow applications to dynamically control coherence region parameters such as:

- The number of tiles within a coherence region ³,
- The spatial location and/or shape of the coherence region,
- The memory address range to be kept coherent within the coherence region.

With this, applications can dynamically configure and even re-configure coherence regions at run-time based on their ever-changing requirements. This feature perfectly aligns with the ideologies and requirements of the InvasIC project.

In order to maximize flexibility for applications, RBCC allows fine granular control over all coherence region parameters. For example, it is possible to configure single/-multiple coherence region(s) to be spatially disjoint or even overlapping with each other. Within a coherence region, the coherent memory range can be configured to only partially cover a custom memory address space. RBCC also allows uni-directional sharing between tiles within a coherence region, i.e., one tile guarantees a coherent view of its memory address space with the other, but not vice-versa. With such features, several applications with varying requirements can be simultaneously executed within their respective dynamically-created tailor-made coherence regions on large manycore system.

³The maximum number of tiles within a single coherence region is still M_{max}

3.2.3 Coherence-on-Demand

In a multi-threaded application spanning different tiles, usually not all data is shared by its threads/processes. Therefore, it would be sufficient to maintain a coherent view only on the memory range where shared application data reside. This would lower the amount of coherence actions thereby reducing the coherence traffic in the entire MPSoC. RBCC allows applications to dynamically configure coherence regions, even to cover a specific memory address range. However, the coherence region needs to be configured before the applications begins in order to maintain correctness. At this point, only the address ranges of statically allocated application data like initialized/uninitialized data are known. Application and OS code, processor stacks, etc. are usually private, and therefore do not need to be kept coherent. However, data to be allocated at run-time are still unknown, which forces the coherence region configuration to conservatively and unnecessarily cover the entire heap memory address space. This results in needless coherence actions that add to the network traffic and increases the communication latency.

As a solution, RBCC is designed to leverage hints from the OS such that it can provide inter-tile coherence to truly shared dynamically allocated data. The idea is to extend the *memory allocate* function of the OS to include the newly allocated data as part the coherent memory range. Similarly, when the *memory free* function is invoked by the OS, the coherent memory range is adapted to exclude the deleted data. This feature uses hardware-software co-design to provide tailored coherence for truly shared application data, enabling coherence-on-demand.

3.2.4 Auxiliary Features

The RBCC concept is architecture-agnostic, i.e., it can be applied to any generic tile-based manycore architecture. However, the target architecture used to conceptualize RBCC has also influenced its development, thereby adding some auxiliary features.

The architecture under discussion is a DSM-based tiled manycore system with two types of memories, placed at different locations. The TLM is an SRAM-based memory, distributed among every tile of the manycore system. The main memory is composed of DRAM-cells, located within a dedicated memory tile of the manycore system. The RBCC concept has been developed to seamlessly establish scalable and flexible coherence for both types of memories. As a consequence of housing distributed TLMs, the book-keeping directory structures have also been distributed accordingly. This avoids the need for home-node hops, thereby reducing the latency of coherence messages and lowering the network load.

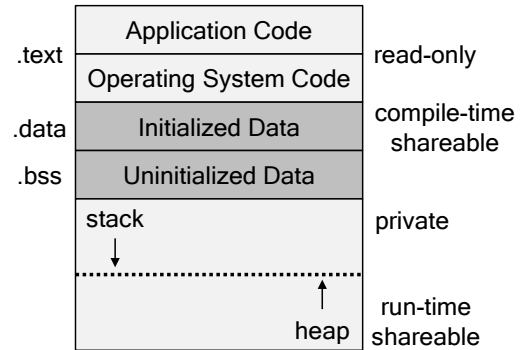


Figure 3.5: An example memory-map with code and data sections

Further, the design choices and implementation specific details when realizing RBCC have also necessitated several features, such as, providing a coherence barrier mechanism, resolving false-sharing problems, etc. As these features require design-specific knowledge, they are described from Section 3.4.4 onwards.

3.3 RBCC Design

3.3.1 The Coherence Region Manager (CRM)

The RBCC concept and its features are realized using a CRM. The CRM establishes a framework to create, maintain and tear-down coherence regions. It informs a given tile about which other tile(s) are permitted to have a coherent view over its share of memory. With this information, coherence is selectively enabled between a configurable subset of tiles within a large manycore system.

From a coherence perspective, the CRM is designed to perform all tasks of a generic directory-based coherence entity. Fundamental coherence actions of the CRM include tracking memory transactions like loads and stores, sending invalidation messages to clear stale cache data, and writing-back fresh data to the source memory (TLM or main memory). The CRM additionally contains custom logic which is used realize RBCC features such as dynamically (re)configuring coherence regions at run-time.

3.3.2 Architectural Design

The CRM is a hardware module distributed within every tile of the target manycore architecture, introduced in Section 3.1.1. Within the tile, there exist several design choices as to where the hardware CRM module could be instantiated. Various options were explored by a former colleague as part of a master thesis [66]. From this work, the design choices can be broadly classified into tightly-coupled or loosely-coupled.

A tightly-coupled design would be to integrate the CRM into an existing module of the target manycore architecture. For example, the CRM could be integrated as part of the L2 cache or even the network adapter, where it would receive information on memory transactions and carry out coherence actions. This design choice puts the CRM directly into the data-path, allowing explicit control over all memory transactions. However, the tight integration with a specific module would make the CRM design non-modular, i.e., it would be difficult to use the CRM as an Intellectual Property (IP)-core for any generic MPSoC platform. With a loosely-coupled design, the CRM would be instantiated on the AHB bus interconnect as a standalone module. It could use the AHB bus to listen to memory transactions and perform the required coherence actions. This design choice would make the CRM modular, but lack explicit control over memory transactions.

Most hardware-supported inter-tile coherence mechanisms usually use a tightly integrated design approach, which requires all memory transactions to first consult with the coherence manager before responding to the memory request. This thesis implements the CRM a loosely-coupled design, exploring a non-intrusive approach that works in parallel to memory transactions. Additionally, this design choice makes it relatively less-complex

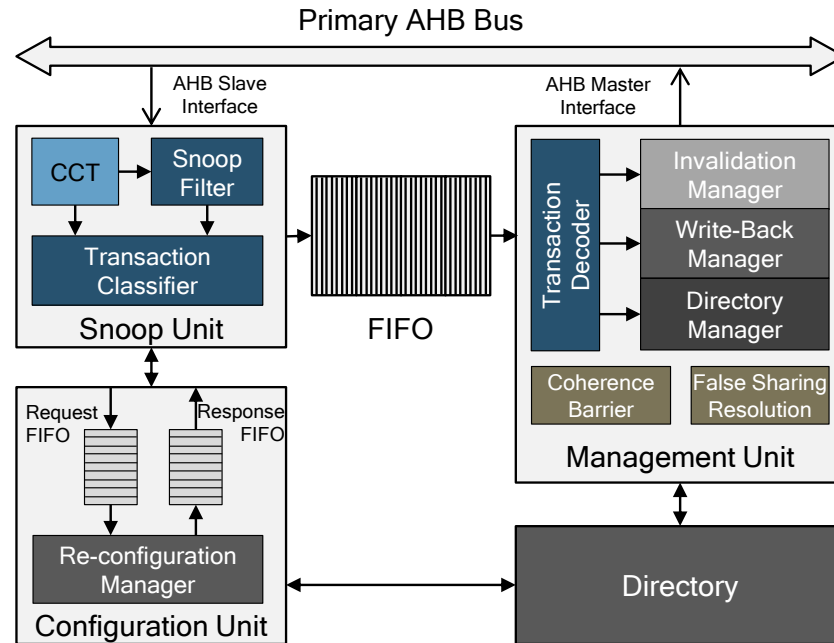


Figure 3.6: The internal block diagram of the CRM with its sub-modules

to deploy the CRM as an IP-core on a generic MPSoC platform. The subsequent challenges of this design choice such as controlling memory transactions during coherence actions and their solutions are described in Section 3.4.6. Lastly, the CRM is designed as a programmable hardware module. This allows users/applications to dynamically create and/or dissolve flexible coherence regions at run-time.

3.3.3 The CRM and its Sub-modules

Figure 3.6 depicts the CRM design as a block diagram. The CRM module can be viewed as several sub-modules, each designated with its respective tasks, working together to enable RBCC. The design of each CRM sub-module is described below.

Snoop Unit

The Snoop Unit is the front-end of the CRM, connected to the AHB bus interconnect. It has two main tasks - listening to all memory transactions on the AHB bus and maintaining information of all active coherence regions concerning the local tile. The Snoop Unit is equipped with an AHB slave interface that is used for two purposes. One, to passively listen to all memory transactions on the AHB bus interconnect. This may include transactions that are not directly addressed to the CRM. Two, to provide a memory-mapped address space that allows users/applications or even other hardware modules to directly interact with the CRM, for example, to configure coherence regions, CRM-CRM communication, check the status of the CRM, etc.

The Snoop Unit also contains a Coherence Configuration Table (CCT) to keep a record of all active coherence regions involving the local tile. Users/applications write into the CCTs of different tiles via the memory-mapped address space to configure a coherence region. Each entry in the CCT consists of five fields namely:

- *Region ID*: An identifier differentiating multiple coherence regions.
- *Start Address*: The start of the shared address range.
- *End Address*: The end of the shared address range.
- *Sharers*: A bit-vector representing the remote tiles that are allowed a coherent view over the shared address range.
- *Direction*: A bit-pair per Sharer representing either uni-directional or bi-directional sharing.

The role of each entry in configuring coherence regions is explained with an example in Section 3.4.

Management Unit

This sub-module serves as the back-end of the CRM. Its task is to react on transactions received from the Snoop Unit via the FIFO Interface. The transactions are sent to a transaction decoder, where they are interpreted and translated into coherence actions. In order to execute certain coherence actions, the Management Unit is equipped with an AHB master interface. The basic operations performed by the Management Unit to maintain inter-tile coherence are described in Section 3.4.

Configuration Unit

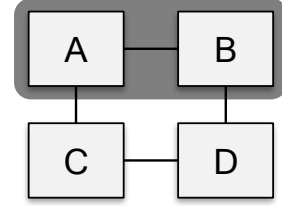
This sub-module is responsible for all coherence region configurations and re-configurations (to grow/shrink and/or migrate regions) at run-time. The Configuration Unit is equipped with internal interfaces to the Snoop Unit and to the Directory. Its primary task is to check if incoming coherence region configurations are new (first time) or an update to an existing active coherence region. For updates to existing coherence regions, i.e., a coherence region re-configuration, the Configuration Unit selectively clears the Directory entries. It also houses two FIFO modules, a request FIFO and a response FIFO to buffer multiple incoming coherence region configuration. Details describing the functionality of this sub-module is provided in Section 3.4.

FIFO Interface

The CRM module uses a FIFO Interface to connect the Snoop Unit and the Management Unit together. The FIFO Interface is not only used for communication, but also to buffer transactions. It also contains run-time configurable fill level indicators to avoid any FIFO overflows. The FIFO overflow handling procedure is described in Section 3.4

Table 3.2: An example of two CCT entries that establish a coherence region spanning two tiles

CCT Entry	Tile A	Tile B
Region ID	1	1
Start Address	80000000 ₁₆	80000000 ₁₆
End Address	807FFFFFFF ₁₆	807FFFFFFF ₁₆
Sharers	0010 ₂	0001 ₂
Direction	XX ₂ XX ₂ 11 ₂ XX ₂	XX ₂ XX ₂ XX ₂ 11 ₂



Directory

This sub-module is responsible for managing the book-keeping information that is required for inter-tile coherence. The design of the Directory follows the description of Section 3.2.1. Therefore, the storage requirements for the sharer bit-vector is reduced (RBCC concept), making it substantially smaller and scalable compared to global coherence directories. Currently, creating a region that exceeds the maximum number of tiles in a region activates an overflow flag which can be used to trigger an OS interrupt.

3.4 RBCC Functionality

3.4.1 Coherence Region Configuration

RBCC is activated by configuring the CCT present in the Snoop Unit of the CRM. To better understand the configuration process, consider an MPSoC with 4 tiles in total, wherein an application requests for a coherence region spanning two tiles, as depicted in Table 3.2. Also, assume that the application requests for bi-directional sharing of the complete TLM range, i.e., both tiles within the coherence region are allowed a coherent view over each other's entire TLM range. First, the application's request is handled by the OS which writes into the CCTs of the two tiles via the memory-mapped interface of the CRM. As shown in Table 3.2, both tiles (Tile A and Tile B) have the same Region ID of 1, as they are part of the same coherence region. The Start Address and End Address fields span the entire TLM address space of the respective tiles. The Sharers field of Tile A has a bit-vector indicating Tile B as a sharer and vice-versa. This allows both tiles to have a coherent view over each other's TLMs. The Direction field for both tiles has the bit-pair value of 11₂ (at the corresponding Sharer index) indicating bi-directional sharing. By modifying the Direction field (to 10₂ or 01₂), applications can enable fine granular coherence regions through uni-directional sharing.

With proper configuration of the CCTs, users can create and/or update an arbitrary number of coherence regions spanning different shapes and/or sizes with fine-granular sharing properties at run-time. The number of entries present in the CCT influences the number of coherence regions that can be active simultaneously. The maximum number of entries of the CCT should be decided at design-time, based on the size of the manycore system and the class of expected applications.

Re-configuration Overheads

The CRM allows coherence regions to be re-configured at run-time based on application requirements. This feature can be leveraged by resource-aware computing applications to dynamically add/remove processing and/or memory resources. It can also be used when a current application ends and a new application begins. The re-configuration process is similar to an initial configuration process, with an additional possibility that may involve resetting the sharer entries of the Directory. From the perspective of the CRM, re-configuration scenarios are classified into two categories. One, to modify the memory range, i.e., increasing/decreasing the coherent memory address range within an active coherence region. Two, to modify the number of sharers, i.e., adding/removing processing and memory resources to/from an active coherence region. If the Configuration Unit detects modifications to the start and/or end address fields, the Directory entries need to be reset, i.e., the sharer bit-vectors are set to zero. If the address range is reduced, the sharer bit-vector of the corresponding Directory entries are cleared. If the address range is expanded, no action is required as the entries are already clean (cleared when the address range is reduced).

For the example in Table 3.2, if Tile A's memory address range is reduced by half, (end address overwritten by $803FFFFFF_{16}$) the corresponding range of Directory entries are reset. If this address range is added back or re-assigned, no action is taken as the Directory entries are clean. Similarly, Directory entries are also reset upon modifications to the sharers field. However, compared to the memory range, the sharer bit-vector is only selectively cleared based on which tile was removed from the coherence region. For the example in Table 3.2, if Tile B is removed from the coherence region, the bit corresponding to Tile B in the Sharers field of Tile A's CCT is cleared (0000_2) for the entire shared memory range. If a tile is added to the coherence region, no action is required as the entries are clean. Updates to the Region ID or Direction fields do not require resetting the directory entries. Further details on the Directory resetting overheads are reported in Section 3.6.2.

Configuration Synchronization

The process of configuring/re-configuring a coherence region involves writing to several CCTs that are distributed among different tiles of the manycore system. Applications should start executing only after a requested coherence region has been successfully (re)configured. The time taken to (re)configure a coherence region depends on:

- The number of tiles requested by the application,
- The possible directory-resetting overheads in case of a re-configuration,
- The current NoC load.

These factors make the coherence region (re)configuration time non-deterministic. To synchronize this procedure, the coherence region (re)configuration follows a fork-join

3 Region-based Cache Coherence (RBCC)

model similar to multi-threaded applications, where each CRM responds with a coherence (re)configuration acknowledgment message upon successfully receiving the The tile executing the application’s main thread is defined as the coherence region initiator tile. First, the initiator tile writes into the CCT of the local CRM. This is followed by writes to all CCTs of the respective remote CRMs that are to be (re)configured as part of the coherence region. The CRM provides a hardware synchronization register present in the Configuration Unit, which can be used by the application’s main thread to keep track of all pending coherence (re)configuration acknowledgments. Upon successfully writes to all fields of the CCT, the CRM in every tile being (re)configured sends-out a coherence (re)configuration acknowledgment signal to the CRM of the initiator tile. The time taken by the CRMs to send-out the acknowledgment signal can vary depending on the possible Directory reset time and the NoC load. Once all coherence (re)configuration acknowledgment signals are received by the main thread, the application can safely start/resume its execution.

Referring to the example of Table 3.2, Tile A expects two response signals ⁴, one from itself and one from Tile B. During this waiting phase, the application’s main thread polls on the coherence synchronization register. Once Tile A receives all pending acknowledgment signals, the coherence synchronization register is cleared and the application can safely begin/resume its execution.

3.4.2 Coherence-on-Demand: RBCC-malloc()

The coherence region configuration and re-configuration procedures are designed to work at run-time. To leverage this functionality, a software wrapper around the regular *malloc()* function has been designed. Using this, applications can not only specify the amount of data to be allocated, but also if the data should be cache coherent ⁵. Referring to Figure 3.7, assume an application initially requests the OS for 4 tiles cache coherent tiles ❶. Assuming resources are available, the OS sets up a coherence region by

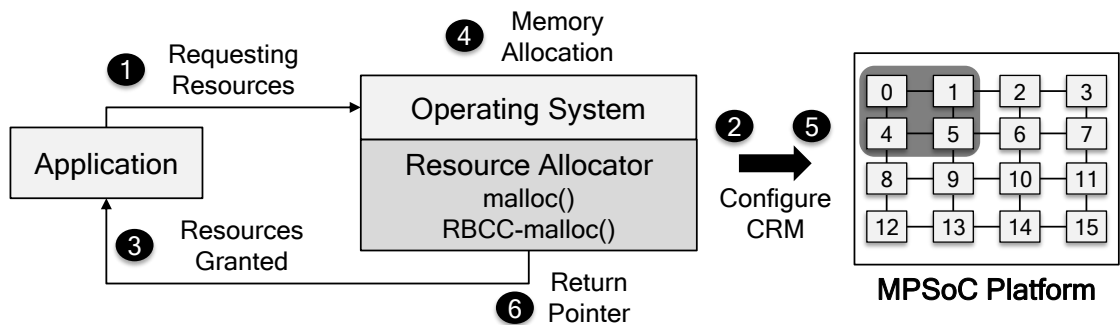


Figure 3.7: An example illustrating how `RBCC-malloc()` dynamically tailors coherence regions to only track actually shared applications’ working-sets at run-time

⁴Assuming the application’s main thread is running on Tile A

⁵This can also be abstracted by the OS, making it transparent to the application

writing to the CCTs of the corresponding tiles ②. The Start Address and End Address fields are already set to track compile-time known statically allocated shared data. Upon confirmation from the CRM, the OS grants the corresponding processing and memory resources to the application ③. During execution, all dynamic data allocation requests to the OS are handled either by a regular *malloc()* or RBCC-*malloc()* ④. If the data allocation request originates from within a coherence region, the OS can automatically choose RBCC-*malloc()* over *malloc()*. In this case, the OS first forwards the range of this “to-be-allocated” application data/working-set to the CRM ⑤, whose CCTs are adapted at run-time to additionally track this specific memory range. Upon a successful update, the OS returns a pointer to the newly allocated coherent working-set ⑥. Conversely, the *free()* is replaced by RBCC-*free()* to clean-up the CCTs followed by data de-allocation. The CRM configuration penalty/overheads are listed in Table 3.6. This hardware-software co-design mechanism is triggered by data allocation requests throughout application execution. RBCC-*malloc()* allows the CRM to transparently accommodate and track actually shared application working-set(s) at run-time.

3.4.3 Coherence Operations

With RBCC, coherence needs to be provided only to the subset of tiles and their corresponding shared address ranges that are configured as part of a coherence region. Therefore, the Snoop Unit uses information from the CCT to filter all memory transactions observed on the AHB bus. The filtered transactions are sent to a transaction classifier, where they are categorized. Then, the transactions are sent to the Management Unit of the CRM via the FIFO Interface, where they are translated into specific coherence actions. The different transactions and corresponding coherence actions are described below.

Directory Update

If the Snoop Unit detects a load transaction from a remote tile, the transaction classifier categorizes this as a directory update request and forwards it to the Management Unit via the FIFO Interface. The Management Unit decodes this request and triggers the directory manager, which updates the sharer bit-vector for the corresponding ⁶ cache block in the Directory. The Snoop Unit ignores load transactions that originate within the tile, as tile-local caches are kept coherent by a snoop-based scheme.

Invalidation Generation

If the Snoop Unit detects a store transaction either from within the tile or from a remote tile, the transaction classifier categorizes this as an invalidation generation request. In the Management Unit, the transaction decoder triggers the invalidation manager which accesses the Directory to read the sharer bit-vector for the corresponding cache block. Depending on the number and position of the set sharer bits, invalidation messages are

⁶The CRM obtains this information from the network adapter

3 Region-based Cache Coherence (RBCC)

sent out using the AHB master interface. These invalidation messages are addressed to the CRMs of remote tiles which share a copy of the data. The invalidation message also contains the memory address of the cache block which is to be invalidated in the remote caches.

Invalidation Execution

If the Snoop Unit detects an incoming invalidation message from a remote CRM, it triggers an invalidation execution request. This is decoded in the Management Unit and passed on to the invalidation manager, which executes the request in two steps. First, the corresponding memory address is invalidated in the L2 cache. This is done by writing to a specific control register of the L2 cache using the AHB master interface. Next, the corresponding memory address is invalidated in the L1 cache. This is done by performing a dummy store operation on the AHB bus which triggers the snoop-based coherence protocol. The dummy store operation is performed with the AHB IDLE.TRANS transfer mode, ensuring that the data is not modified⁷. After successfully invalidating the memory address in the L1 and L2 caches, the invalidation manager sends out an invalidation acknowledgment message to the source CRM which triggered the invalidation.

Write-back

If the Snoop Unit detects a store operation to a remote tile's TLM, the transaction classifier categorizes this as a write-back request. In the Management Unit, the write-back manager triggers a forced write-back of the corresponding memory address from the L2 cache. The explicit write-back is performed by writing to the control register of the L2 cache via the AHB master interface. Then, the L2 cache writes-back this cache block to the remote TLM via the NoC. After the cache block is successfully written-back at the remote TLM, the remote CRM sends a write-back acknowledgment message to the source CRM which triggered the write-back.

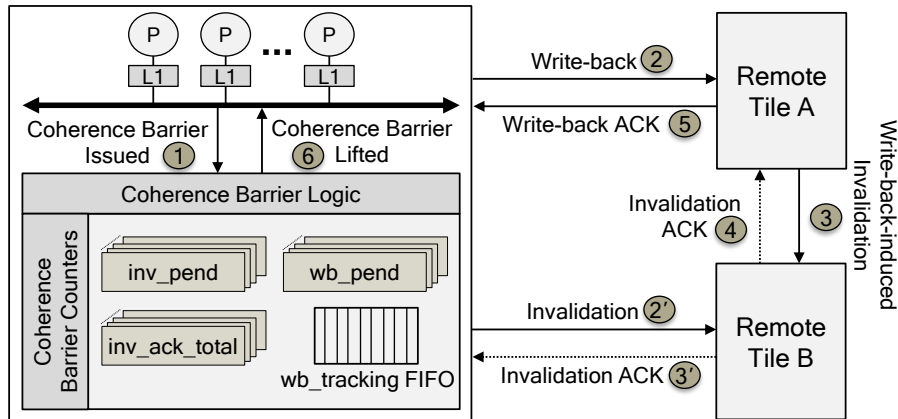
3.4.4 Coherence Barrier Mechanism

A coherence manager enforces a coherent view over all shared memory. However, this does not necessarily enforce “when” all coherence operations have been successfully executed, i.e., if all processing elements have the same view of the memory subsystem. This information is required by applications to synchronize and execute as intended. The CRM is equipped with a coherence barrier mechanism that can be used by applications for coherence-related synchronization. The coherence barrier is integrated as part of the Management Unit and uses a novel counter-based approach to track coherence messages. This approach allows the coherence barrier to work in parallel to ongoing transactions, thereby maintaining the non-intrusive property of the CRM design.

⁷The L1 caches have been modified to additionally snoop and react to idle transactions

Table 3.3: A summary different memory access operations and their coherence actions

Memory Operation	Potential Coherence Actions
Local Load	None
Remote Load	Update Directory
Local Store	Generate Invalidations
Remote Store	Write-back and Generate Invalidations

**Figure 3.8:** An example illustrating the internal block diagram and operation of the coherence barrier mechanism

The primary task of the coherence barrier is to inform applications if all coherence operations were successfully executed, i.e., sent-out and acknowledged. Table 3.3 summarizes the four basic memory operations, and the corresponding coherence actions taken by the CRM. Load operations do not modify memory and are acknowledged with data by default. Therefore, the CRM does not need to track them for the coherence barrier mechanism. Store operations modify memory and can trigger coherence actions like write-backs and invalidations. Therefore, the CRM keeps track of how many such messages were sent-out, and if all were successfully acknowledged.

Coherence Barrier Counters

The Management Unit keeps track of the ongoing/pending coherence messages using different counters.

Invalidation Tracker: This sub-module keeps track of pending invalidations to all tiles that are triggered by a local store operation. An *inv_pend* counter per tile is incremented and decremented when an invalidation is issued and acknowledged respectively (2') and (3') to/from Remote Tile B).

Write-back Tracker: This sub-module keeps track of pending write-back operations to all tiles. A *wb_pend* counter per tile is incremented and decremented when a write-back is issued and acknowledged respectively (2) and (5) to/from Remote Tile A).

3 Region-based Cache Coherence (RBCC)

Write-back-induced Invalidation Tracker: A write-back operation executed on the remote tile may trigger further invalidations. Therefore, a write-back acknowledgment should be sent, only when all invalidations triggered due to a write-back operation are successfully executed and acknowledged. In Figure 3.8, Tile A receives a write-back ② which further triggers invalidations ③ to Tile B. Only when Tile B sends back an invalidation acknowledgment ④, Tile A responds with a write-back acknowledgment ⑤. Tracking write-back-induced invalidations becomes a complicated process as there can be several ongoing write-back operations simultaneously. A look-up table approach noting down and comparing every invalidation’s parent write-back increases the area of the CRM and its delay characteristics. Instead, a novel counter-based approach is used to correctly acknowledge write-backs. An *inv_ack_total* counter that holds the total amount of invalidation acknowledgments per tile, is incremented upon receiving an invalidation acknowledgment. Upon triggering write-back-induced invalidations, the tile computes the future value of the *inv_ack_total* as follows:

$$inv_ack_total_{future} = inv_ack_total_{current} + inv_pend + 1 \quad (3.4)$$

This value, along with the source of the write-back is pushed into a write-back tracking FIFO. Upon receiving invalidation acknowledgments, the current *inv_ack_total* counter is compared to the pushed FIFO entry. If the FIFO value satisfies the current *inv_ack_total* counter ⁸, a write-back acknowledgment to the source tile is sent.

Coherence Barrier Logic

When an application triggers a coherence barrier on a given tile ①, a memory-mapped barrier register is set and the current values of the *inv_pend* and *wb_pend* counters are copied into shadow registers. The shadow registers are decremented upon every invalidation and/or write-back acknowledgment, and the barrier register is reset when they reach zero ⑥. Then, the application can safely resume as the barrier is lifted. The use of shadow registers allows other processing elements (most probably executing a different application) on the same tile to continue, while the coherence barrier logic operates in parallel ⁹.

3.4.5 False Sharing Resolution

False sharing is a well-known problem in manycore systems [67]. It occurs when multiple processing elements access independent variables, but are nevertheless considered as sharers because the variables themselves are located within the same cache line. False sharing can significantly degrade the performance of an application. Many solutions have been proposed to solve the false sharing problem. The work in [68] uses a compiler-based approach to avoid false sharing by transforming data. Studies in [69, 70] detect false sharing using compiler-based approaches, and the work in [71] resolves false sharing

⁸The coherence and acknowledgment messages should always be in-order

⁹Multiple barriers can be supported by increasing the number of barrier and shadow registers per tile

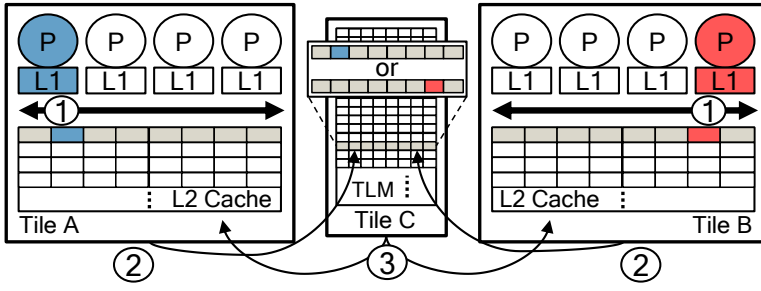


Figure 3.9: An example of the false sharing problem

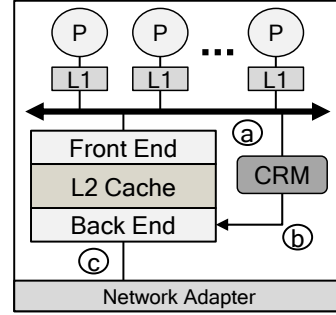


Figure 3.10: An example of false sharing resolution

at run-time using data migration. The work in [72] also detects and resolves the false sharing problem at run-time by using a writer-own protocol together with data migration. The study in [73] employs a complementary approach by making use of different cache line sizes. The idea is, with small cache line sizes, the effects of false sharing can be minimized. Therefore, non-shared data are assigned to large cache line sizes and shared data are assigned to small cache line sizes. The false sharing resolution strategy proposed by this thesis is different to the aforementioned works. First, the impact of false sharing on the manycore system is explained, followed by the proposed solution.

As explained in Section 3.3.2, the non-intrusive functionality of the CRM enables a modular design which is favourable for easy integration into other manycore systems. The challenge of this design choice is that it does not enforce flow control over load/store operations as they do not explicitly pass through the CRM. This, combined with the false sharing problem can not only deteriorate application performance, but also lead to data corruption. For example, consider Figure 3.9 where Tile A and Tile B are working on different words of the same cache line making them false sharers. A write operation ① on their respective words would result in remote write-backs ② of the entire cache line racing to Tile C's memory, where each of them further trigger invalidation messages ③ to each other. The final data in the TLM would be that of the write which lost the race, and the cache line in the respective L2 caches would be invalid. If the data path of the remote write operations were passing through the CRM, the second write would have received a negative acknowledgment, thereby forcing it to re-read the new data and write it again. As the CRM in Tile C only observes, and cannot control these write operations, data in Tile C's memory may be corrupted.

The solution to this data corruption problem, whilst maintaining a modular CRM design, involves three steps as illustrated by Figure 3.10:

- (a) Detect the modified word in the cache line,
- (b) Inform the L2 cache back-end as to which word was modified,
- (c) Write-back the modified word (not the cache line) to the remote memory.

3 Region-based Cache Coherence (RBCC)

For step (a), the AHB bus signals HSIZE and HBURST are used to decode the modified word of the cache line. The HSIZE signal represents the write data size ranging from a single Byte to a complete cache line. The HBURST signal indicates the type of bus transaction covering a single transfer to a 16 beat burst transfer. Monitoring both these signals, the CRM detects the modified word in the cache line. This information is propagated to the L2 cache back-end in step (b). Upon a write-back command from the L2, the L2 cache back-end uses this information and selectively writes-back only the modified word in the cache line through the network adapter in step (c). These steps happen in parallel to the CRM write-back operation, thereby adding no additional delay.

3.4.6 Auxiliary Functions

Apart from enabling coherence and providing coherence-related synchronization methodologies, the CRM is also equipped with some additional features that are described below.

FIFO Overflow Handling

The CRM module is designed to operate in parallel to load/store transactions, passively listening to memory transactions on the AHB bus. The Snoop Unit filters and classifies incoming transactions, before feeding them into the FIFO Interface. This process is performed in a streaming manner, without any back-pressure to the incoming transactions on the AHB bus. However, the Management Unit cannot guarantee to interpret and perform all the coherence actions in a streaming manner. This is due to the non-deterministic access times when using the AHB bus, for example, some coherence operations exhibit non-deterministic completion times depending on bus penalties, number of sharers, etc. These factors lead to a mismatch in production rate of the Snoop Unit (snooped bus operations) and consumption rate of the Management Unit (coherence actions) which could lead to loss of transactions. To alleviate this problem, the FIFO is designed with a predefined critical and safe level. If the FIFO reaches/exceeds the critical level, the CRM locks the AHB bus in order to stop new transactions from entering the FIFO. The lock is performed in a selective manner, i.e., only the CRM and network adapter bus masters are granted bus access during this locked-phase. Such a selective locking mechanism allows the CRM to empty its FIFO without causing deadlocks. However, locking the local AHB bus only stops transactions generated from within the tile from filling-up the FIFO. Remote tiles that are part of the coherence region could yet push transactions into the FIFO. Therefore, the CRM not only locks the local AHB bus, but also sends out a message to all remote tiles within the coherence region to selectively lock their respective local AHB buses. Once the FIFO reaches the safe level, the CRM unlocks the local bus, and sends out a message to all corresponding remote tiles, instructing them to unlock their respective AHB buses. This selective bus-locking mechanism is deadlock-free and allows the CRM's FIFOs to be safely emptied, ensuring lossless transactions.

Support for DMA Accesses

manycore systems are equipped with a Direct Memory Access (DMA) unit to accelerate inter-tile memory transfers. DMA memory transactions are carried out between dedicated hardware DMA modules on the source and destination tiles. As these memory transactions bypass the cache hierarchy, the CRM design is adapted accordingly. From the perspective of the CRM, DMA transfers are viewed as burst load (reading from the source tile) and store (writing to the destination tile) operations. The Snoop Unit filters ignores all burst load operations from the DMA module, as they are not cached and hence do not need to be traced by the Directory. For burst store operations by the DMA module, the Snoop Unit triggers the Management Unit to send out potential invalidations, as the data being written might have been cached previously.

3.5 Concept Evaluation - High-Level Simulation

Implementing the CRM directly as a hardware module requires detailed knowledge of the manycore system, its various sub-modules and their protocols. These low-lying system details are not entirely known or are under formulation, especially during the initial stages of a project. Deciding to go for a hardware implementation at this stage usually results in frequent modifications, thereby increasing the design time. Therefore, the RBCC concept is initially verified using a high-level simulation model. This approach allows the CRM to be validated quickly without the need for concrete system-level details.

3.5.1 Simulation Framework

An existing SystemC-based [74] cache simulator designed by a colleague at Lehrstuhl für Integrierte Systeme / Chair of Integrated Systems (LIS) is used as a starting point. The simulator was adapted and extended to incorporate all the necessary components such as the processing elements, the memory hierarchy and interconnect to mimic a DSM-based manycore system. All components communicate with each other using the TLM-2.0 standard protocol [75]. As an alternative to a typical SystemC top level module, the Synopsys Platform Architect Tool is used to connect all components together. The tool has a user-friendly Graphical User Interface (GUI) which eases the effort of connecting several components together.

The simulation framework is configured as a tile-based MPSoC with four processing elements per tile, similar to Figure 3.1. Each processing element has a 16 KiB private L1 cache (2-way, 256 cache lines per way, 8 words per cache line) using a write-through policy. The L1 caches use the MI protocol to keep coherent and are connected to a common TLM-2.0 compliant bus interconnect. Each tile is equipped with a tile-private 128 KiB L2 cache (4-way, 1024 cache lines per way, 8 words per cache line) using a write-back policy. The L2 cache uses the MSI cache coherence protocol. Each tile contains a 1 MiB SRAM-based TLM and the entire manycore system houses a DRAM-based global memory. Intra-tile coherence among the L1 caches is enabled by using a snooping-based

3 Region-based Cache Coherence (RBCC)

protocol. Inter-tile coherence between the L2 caches of different tiles is guaranteed by the CRM module present in every tile. For inter-tile communication, a NoC in conjunction with a network adapter is used. The NoC is configured as a 4x4 mesh using XY routing.

The focus of the RBCC concept is more towards the memory subsystem than the processing subsystem. Therefore, to reduce the overall simulation time, a trace-based simulation approach is used, where the internal operations of the processing elements like the pipeline stages, registers, etc. are abstracted. The processing elements in the simulation framework are designed to simply replay traces of a given benchmark that are obtained externally. The traces are then injected into the simulation framework which models the memory subsystem. The decision to use trace-based simulation trades-off exact modelling of the processor internal operations which are not required for high-level verification, in return for simulation speed.

3.5.1.1 Extracting Traces from the Gem5 Simulator

The Gem5 [76] simulator is used to generate execution traces of benchmarks, that are then fed to the trace-based processing elements of the SystemC simulator. The SystemC simulator concentrates on modelling the memory subsystem of the tile-based manycore architecture. Therefore, it is important that the extracted benchmark traces are independent of Gem5's memory subsystem modelling. For this reason, Gem5 is configured with an Atomic Central Processing Unit (CPU), which avoids capturing the timing characteristics of Gem5's memory architecture. Gem5 is compiled for the Alpha Instruction Set Architecture (ISA) ¹⁰ and is launched in Full System mode using a modified Linux kernel and a disk image containing the necessary benchmarks [77]. The benchmark's traces are recorded only in the Region of Interest (RoI). This omits any traces generated during Gem5's boot-up phase, the benchmark initialization and benchmark clean-up. During the RoI, the traces are further filtered to only include memory accesses, as this information is sufficient for the SystemC simulator. The benchmarks are executed with different Degrees of Parallelism (DoPs), and the resulting traces are formatted such that they can be used by the trace-based processing elements. Table 3.4 illustrates a sample set of memory access traces. Each trace consists of four fields:

- *Timestamp*: This field indicates the time at which the trace should be executed. The time is relative to each processing element and is represented in nano seconds.
- *Processor ID*: This field represents a particular processing element.
- *Load/Store*: This field indicates either a memory load (0) or store (1) operation.
- *Memory Address*: This field represents the memory address that is to be read-from or written-to.

It is important to note that the same trace file is fed to all processing elements of the simulator, and each processor executes its share of memory traces. For the example trace

¹⁰A Reduced Instruction Set Computer (RISC)-based ISA similar to the LEON

Table 3.4: An example trace file format

Timestamp	Processor ID	Read/Write	Memory Address
1000	3	0	80003000 ₁₆
1000	5	1	80005000 ₁₆
3000	3	1	80003030 ₁₆
5000	5	0	80005050 ₁₆

file of Table 3.4, processor 3 executes the first and third trace. When the simulation time reaches 1000 ns, processor 3 issues a memory load request to read data from address 80003000₁₆. Once the load request returns successfully, processor 3 waits an additional 3000 ns before issuing a store to address 80003030₁₆. Processor 5 executes the second and fourth traces in a similar manner.

3.5.1.2 Data Placement Strategies

The memory access traces obtained from Gem5 are unaware of the memory subsystem of the DSM-based manycore architecture which hosts two types of memories - a distributed TLM and a global DRAM memory. Therefore, the question “where should data reside?” arises. This question is answered with two algorithms that are described below.

First Touch (FT) Policy: This data placement strategy attempts to place a given memory block into the TLM of a tile where the memory block was first accessed. If the preferred TLM is full, the memory block is placed into the global memory. This algorithm has a linear complexity of $\mathcal{O}(n)$ where, n is the number of memory accesses.

Most Accessed (MA) Policy: In this data placement strategy, each memory block is placed into the TLM of a tile where the memory block was most accessed over the entire benchmark run. If the preferred TLM is full, the memory block is placed into the next preferred TLM. If all TLMs are fully occupied, the memory block is placed into the global memory. This algorithm exhibits a computational complexity of $\mathcal{O}(n) + \mathcal{O}(m)$, where n is the number of memory accesses and m is the number of unique memory blocks.

The data placement algorithms are executed on the memory access traces of the benchmarks obtained from the Gem5 simulator. The algorithm adapts the memory address fields in the trace file, to indicate the location of the memory block. The adapted trace file is fed to the trace-based processing elements to simulate the memory subsystem of the manycore architecture.

3.5.2 Experimental Setup

The purpose of high-level modelling is to quickly validate the concept of RBCC and to estimate its benefits. This is done by executing benchmarks on a manycore system configured with and without RBCC. For the case without RBCC, there is no inter-tile

3 Region-based Cache Coherence (RBCC)

coherence support, and all processing elements are configured within a single compute tile. The impact of data placement on benchmark performance is also evaluated.

Four workloads from the PARSEC Benchmark Suite - *blackscholes*, *swaptions*, *cannal* and *fluidanimate* are used for the experiments. All workloads are executed using the *simsml* input-set for three DoPs (4, 8, 16), that make up three different coherence region sizes (1, 2, 4) respectively. Every workload is executed on the manycore system with three different configurations:

- Inter-tile Coherence, placed with FT Policy: *RBCC::FT Configuration*,
- Inter-tile Coherence, placed with MA Policy: *RBCC::MA Configuration*,
- Intra-tile Coherence Only: *All-in-One (AiO) Configuration*.

3.5.3 Results and Analysis

The experiments are primarily evaluated for two criteria:

- Benefits of Inter-tile Coherence - AiO vs RBCC::MA,
- Impact of Data Placement - RBCC::FT vs RBCC::MA.

For easy comparisons, the execution time of each configuration is normalized to that of RBCC::FT with DoP(4).

blackscholes: This benchmark uses a small data-set which is not extensively shared by all processing elements. Therefore, it favours an increase in DoP and the number of processing resources. Figure 3.11 shows that with an increase in DoP, RBCC::MA outperforms AiO. This is because, in the AiO configuration, all processing elements share the common bus interconnect. With increasing DoPs, this creates a bottleneck for intra-tile communication, leading to performance degradation. Figure 3.11 also shows that data placement strategy has a negligible impact on execution time. This is due to the workload's small data-set, that allows all its data to reside within the TLMs.

cannal: This benchmark uses large data-sets and favours parallelism. Interestingly, the AiO configuration has a slightly better performance than RBCC::MA. Deeper investigations reveal that these differences are a result of additional remote accesses that are required for the RBCC::MA configuration. For the AiO configuration all data resides in the TLM. Regarding data placement, Figure 3.11 shows that there is a significant performance deterioration for RBCC::FT compared to RBCC::MA. For RBCC::FT, the workload's data-sets are sub-optimally placed in the TLMs, with several frequently-used memory blocks placed in global memory. This leads to excessive remote TLM and global memory accesses, underlining the importance of data placement.

swaptions & fluidanimate: Both these benchmarks exhibit similar characteristics with some minor differences. The *swaptions* workload use medium sized data-sets whereas the *fluidanimate* workload uses large data-sets. Figure 3.11 shows that both benchmarks tend to have a slightly better performance for the AiO configuration up to DoP(8). However, for DoP(16), both workloads experience the intra-tile communication bottlenecks

3.5 Concept Evaluation - High-Level Simulation

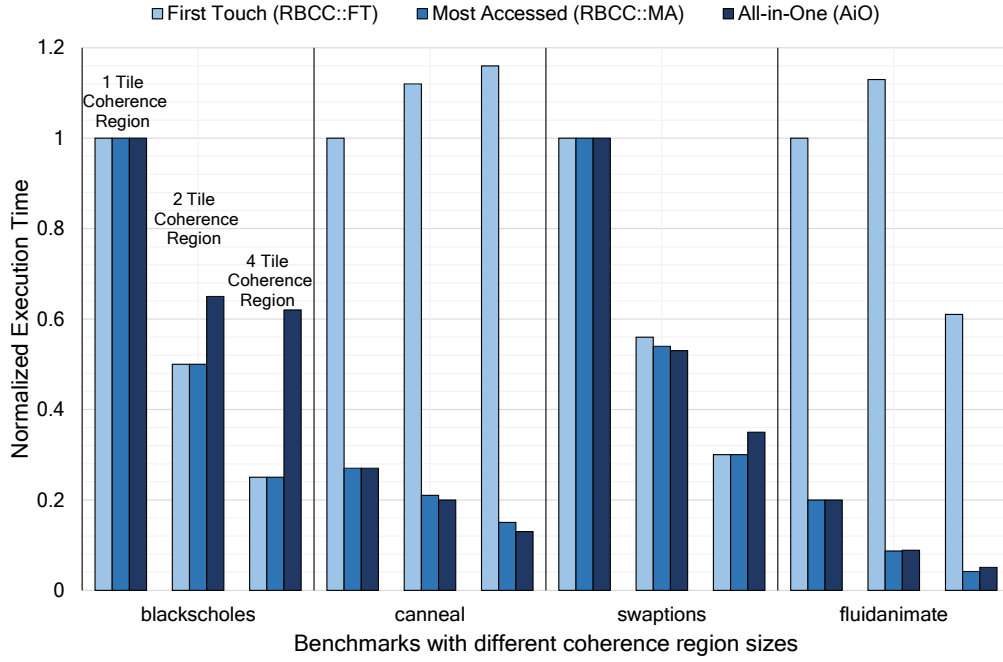


Figure 3.11: The normalized execution time of each benchmark with three DoPs (4, 8, 16) for the RBCC::FT, RBCC::MA and AiO system configuration

of a single tile, making the RBCC::MA configuration a better performer. The data placement strategies do not affect *swaptions* due to a relatively low data-set size. As *fluidanimate* uses larger data-sets, the chances of sub-optimal data placement is higher. Figure 3.11 shows that RBCC::FT’s performance significantly deteriorates for all DoPs compared to the RBCC::MA configuration.

The high-level simulation framework allowed for a quick implementation and evaluation of the RBCC concept. The simulation results provided three take-away points: One, it quickly verified RBCC-specific implementation details such as configuring the CCTs. Two, it showed that the benefits of inter-tile coherence depends on the applications’ characteristics and data-set size. Three, it highlighted the importance of data placement strategies and remote memory access penalties, especially when dealing with a distributed memory manycore architecture.

3.6 Hardware Implementation and Evaluation - FPGA Prototype

After validating the RBCC concept and evaluating the benefits of inter-tile coherence using high-level simulations, the next step was to implement the CRM design as a hardware module and integrate it as part of a DSM-based tiled manycore architecture.

3.6.1 Hardware Setup

For hardware evaluation, an FPGA prototyping platform is used to design and implement the CRM. It is integrated as part of a 4x4 tile-based MPSoC system as illustrated in Figure 3.1. The tile-based MPSoC design is spread across four interconnected Xilinx Virtex-7 (XC7V2000T) FPGA platforms. Each FPGA platform is loaded with a 2x2 tile design, leading to a 4x4 tile design consisting of 16 tiles and 64 processing cores in total. Each compute tile contains four LEON3 processing cores, each with their private 2 Way, 4 KiB/Way write-through L1 caches and a shared 4 Way, 16 KiB/Way write-back L2 cache. The L1 and L2 caches are Non-Inclusive Non-Exclusive (NINE) of each other. Each FPGA is equipped with a 32 MiB SRAM extension board, which is shared equally by four tiles on the FPGA. This grants each tile with an 8 MiB TLM. The MPSoC design also contains an I/O tile, which is similar to a compute tile with additional peripherals. One of the four FPGAs is equipped with a 1 GiB DRAM extension board and an Ethernet extension board which serve as main memory and an I/O peripheral respectively.

3.6.2 FPGA Resource Utilization and Timing

This subsection breaks-down and analyses the resource utilization of the CRM. All resource utilization data is obtained using the Vivado Design Suite. Table 3.5 reports the resource utilization of the CRM as a function of its sub-modules for a manycore system with $N = 16$ tiles and the maximum number of tiles within a single coherence region $M_{max} = 8$. The utilization of each CRM sub-module is further broken-down into FPGA logic resources like Look-Up Tables (LUTs), Registers (REGs), Multiplexers (MUXs) and FPGA memory resources like Block RAMs (BRAMs).

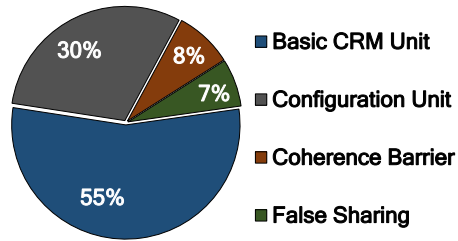
A major chunk of logic resources are consumed by the Snoop Unit of the CRM for its snooping, filtering and classification operations. As the Snoop Unit does not contain any storage element like a FIFO, its BRAM consumption is zero. The Management Unit of the CRM does not consume much logic resources. It mostly consists of an AHB master interface and various sub-modules which just trigger coherence actions. The Management Unit also hosts the coherence barrier and false sharing resolution mechanisms. The slight BRAM consumption of the Management Unit is due to the coherence barrier FIFOs. The Configuration Unit of the CRM also has a moderate resource utilization, mostly consisting of logic required for run-time re-configuration operations. The BRAM utilization of the Configuration Unit are due to the request and response FIFOs. The Directory and FIFO Interface sub-modules of the CRM are

Table 3.5: FPGA resource utilization of the CRM module in terms of LUTs, REGs, MUXs and BRAMs for a $N = 16$ tile manycore system with a coherence region size of $M_{max} = 8$

Modules	LUT	REG	MUX	BRAM
Snoop Unit	2547	2887	340	0
Management Unit	1531	1701	32	2
Configuration Unit	1156	2435	291	3
FIFO Module	140	40	0	7
Directory	112	14	0	13
Entire CRM Module	5486	7077	663	84
Relative to one Compute Tile	4.83%	12.62%	20.86%	12.62%
Relative to the Virtex-7 FPGA	0.44%	0.28%	0.07%	1.93%

the major consumers of FPGA memory resources. The Directory uses BRAMs to store its sharer bit-vectors that are needed for book-keeping purposes. For these results, the Directory is synthesized with a sparsity-8 compared to the 8 MiB TLM, and the FIFO Interface is dimensioned to buffer 4K transactions. As a reference, the area footprint of the CRM relative to a single compute tile as well as an empty Xilinx Virtex-7 FPGA are also provided.

Figure 3.12 breaks-down the logic utilization of the CRM by its features, like basic CRM operations, run-time re-configuration operations, coherence barrier mechanism and false sharing resolution. Obviously, the basic CRM operations consume the highest logic resources, followed by the Configuration Unit. The coherence barrier and false sharing resolution mechanisms only account for $< 10\%$ of the CRM.

**Figure 3.12:** Logic utilization break-down of the CRM by functionality

Sensitivity to Coherence Region Size

The maximum tiles within a single coherence region M_{max} is a design-time parameter which should be decided based on the size of the MPSoC and the class of applications that are intended to be executed. In this sub-section, the impact of varying M_{max} within the bounds of a 4x4 tile-based MPSoC design is investigated. All possible options of M_{max} are explored, ranging from the smallest $M_{max} = 2$ which is a maximum of two tiles within a coherence region, to the largest $M_{max} = 16$ which is global coherence. Figure 3.13 illustrates the FPGA logic resources consumed by one CRM hardware module for every increase in M_{max} . Note that numbers are normalized to the total available resources of a Xilinx Virtex-7 FPGA. The logic resources of the Snoop Unit, Configuration Unit and all FIFO sub-modules are independent of M_{max} ¹¹. The logic resources for Management

¹¹Slight variations are induced by Vivado's place and route algorithm

3 Region-based Cache Coherence (RBCC)

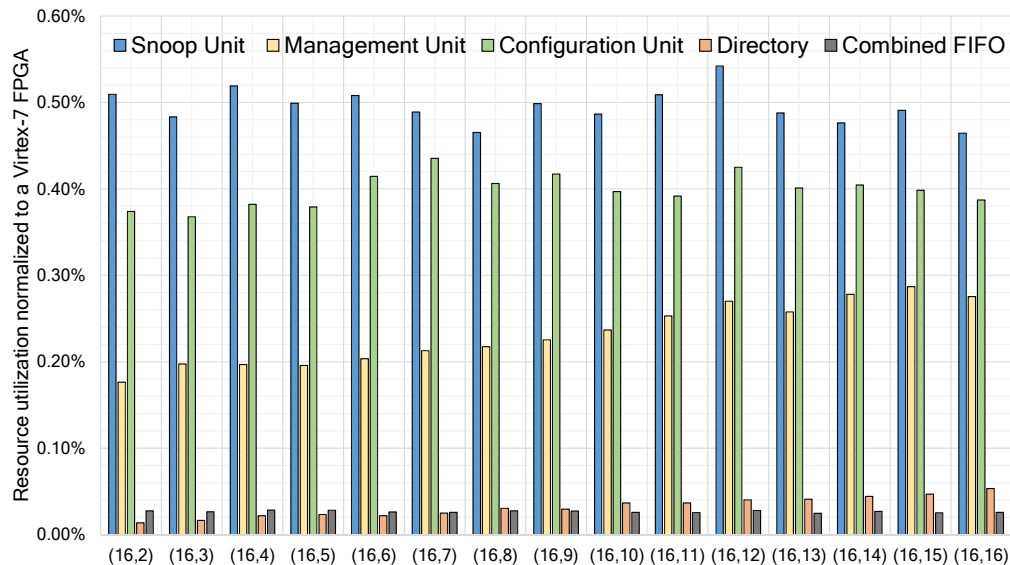


Figure 3.13: Logic utilization for a $N = 16$ tile manycore system with increasing coherence region sizes ($2 \leq M_{max} \leq 16$) normalized to a Virtex-7 FPGA

Unit and Directory sub-modules slightly increase when increasing M_{max} . This is due to the additional logic required to address a larger BRAM.

Figure 3.14 illustrates the FPGA memory resources consumed by the CRM, which are only due to the Directory and FIFO sub-modules. The smaller FIFOs required by the Configuration Unit and the coherence barrier mechanism in the Management Unit are independent of M_{max} . However, the BRAM utilization for the Directory sub-module increases linearly with an increase in M_{max} . For a $N = 16$ tile system with $M_{max} = 8$ tiles, the BRAM utilization can be reduced by 38% compared to global coherence. The savings of the synthesized Directory sub-module closely follow the theoretical directory saving results claimed in Section 3.2.1¹². This means that the BRAM savings will further increase for large manycore systems with moderately sized coherence regions.

CRM - Operation Numbers

Table 3.6 reports the number of clock cycles consumed by the Snoop and Management Units of the CRM to perform different coherence operations. The reported numbers are obtained using cycle-accurate ModelSim simulations performed in a

Table 3.6: Latency of different CRM operations

CRM Operations	Snoop Unit	Management Unit
CRM Configuration	15 clocks	18 clocks
Directory Update	3 clocks	3 clocks
Invalidation Generation	3 clocks	12 clocks
Invalidation Execution	3 clocks	20 clocks
Write-back	3 clocks	7 clocks
Coherence Barrier	3 clocks	3 clocks

¹²Slight deviations from theoretical numbers are due to the granularity of BRAM blocks on an FPGA

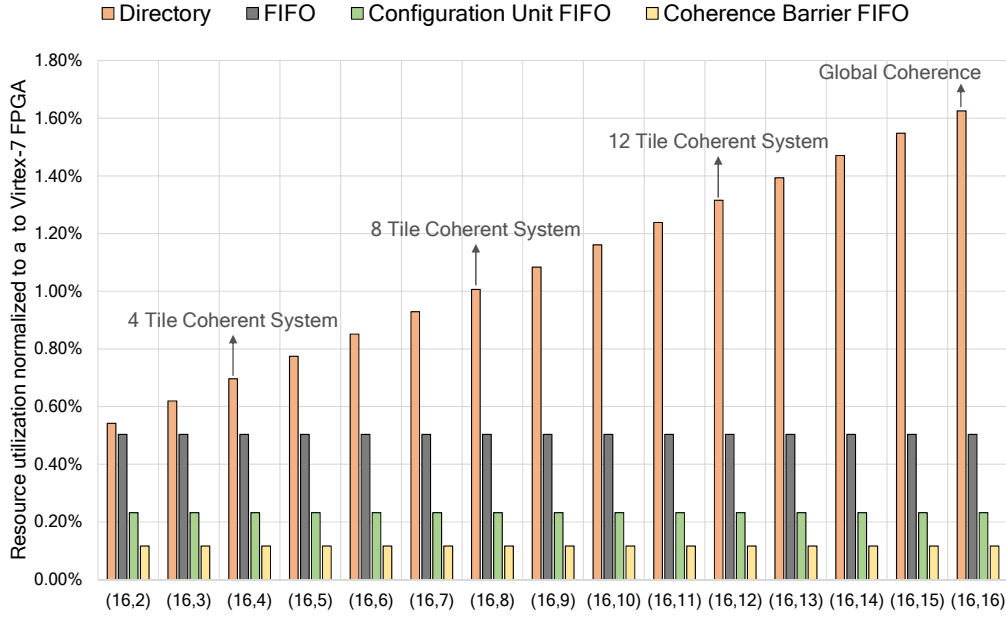


Figure 3.14: BRAM utilization for a $N = 16$ tile manycore system with increasing coherence region sizes ($2 \leq M_{max} \leq 16$) normalized to a Virtex-7 FPGA

traffic-free system. The clock cycles are measured from when the load/store operation is seen by the *Snoop Unit* on the AHB bus. The CRM is designed to keep these numbers as low as possible in order to minimize the overheads required for synchronization such as the coherence barrier mechanism. The false sharing resolution operation is performed in parallel to write-back operations, and therefore does not add additional timing overheads.

Run-time Coherence Region Re-configuration Overheads

The Configuration Unit of the CRM is responsible for the Directory resetting overheads due to a coherence region re-configuration at run-time. These overheads depend on specific changes to the Start/End address or Sharers fields. Figure 3.15 reports the re-configuration overheads for resetting the Directory of different sizes. The initial coherence region configuration is assumed to cover the entire TLM range of 8 MiB. Directory reset steps, ranging from 0.125 MiB to 7 MiB are chosen as the reset range for re-configuration. Note that these steps are

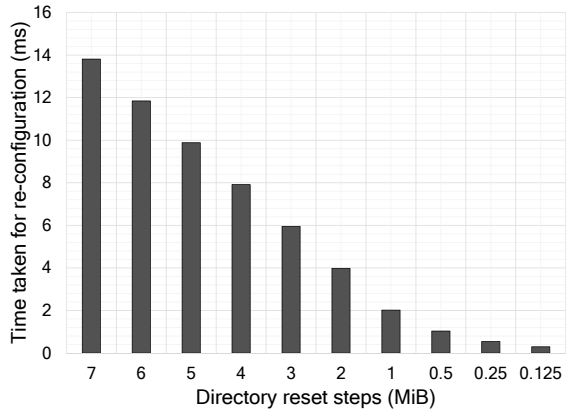


Figure 3.15: Directory re-configuration overheads for different reset steps

3 Region-based Cache Coherence (RBCC)

chosen only to get a feeling of the re-configuration overheads required for resetting the directory for different sizes, but are not specifically limited to the steps reported. The overheads increase linearly with the number of directory entries to be reset, i.e., larger memory range modifications would result in longer re-configuration overheads for the application. A practical example of the re-configuration overheads due to an application re-configuring its coherence region at run-time is presented in Section 3.6.4.2.

CRM - Timing Analysis

The Vivado timing analysis tool is used to obtain the maximum achievable frequency of the CRM module. For a standalone CRM design, the maximum delay reported is 9.398 ns (3.655 ns logic delay + 5.743 ns net delay) that allows the CRM to run at > 100 MHz on the FPGA prototype. However, the entire design is limited to an operating frequency of 50 MHz. This is mainly due to long logic paths in the L2 cache IP.

3.6.3 Experimental Setup

One of the major benefits of using the RBCC concept is the reduction of directory overheads that are achieved by confining hardware-supported coherence to a subset of tiles in manycore systems. The theoretical directory savings were already shown in Section 3.2.1 and reinforced on an FPGA prototype in Section 3.6.2. In the following subsections, the benefits of using the shared memory programming paradigm enabled using hardware-supported RBCC are compared to that of an MPI-based programming model. The benefits of different RBCC features such as coherence-on-demand and the overheads for run-time coherence region re-configurations are also evaluated.

Video Streaming Application

To evaluate the benefits of shared memory programming enabled using RBCC, a video streaming task is used. The task extracts features from an incoming video stream, and is used as part of a robotic vision application. The application is highly flexible in terms of its degree of parallelism, i.e., the number of processing elements can be increased/decreased at run-time, and on a per video frame granularity. The application also supports two parallel programming models - shared memory (RBCC) and MPI-based explicit software communication. These favourable properties make this application an ideal choice for evaluating the RBCC concept and its features.

The feature extraction task uses the Harris Corner Detection algorithm to extract feature points from an incoming video stream. From the hardware perspective, the application can be viewed as a recursive frame-by-frame operation illustrated in Figure 3.16. A host PC sends a video stream frame-by-frame via an Ethernet interface to the FPGA prototype. The video stream is received by the FPGA prototype on the I/O tile, where the input image is transferred via a DMA operation to the compute tile to be processed by the main application thread. The host PC also sends configuration infor-

mation such as: the number of processing elements and/or tiles to be used ¹³, the programming model to be used, and other image processing specific parameters. All these configuration parameters can be controlled from the host PC on a per-frame granularity.

Based on the configured parameters received from the host PC, the application's main thread parallelizes the feature extraction task by distributing the image to all participating processing elements. If the MPI-based mode (*mp*) is enabled, the different processing elements receive their share of the input image via software messages that are accelerated by a DMA engine. By design, this does not require any cache coherence as no data passes through the cache hierarchy. If shared memory mode (RBCC) is enabled, each processing element directly reads its share of the input image via remote load operations, assuming a coherent shared memory view. Coherence is enforced by the CRM by tracking and invalidating the stale image data in the corresponding remote L2 caches. After processing its share of the image data, each participating processing element transmits feature points back to the application's main thread. These feature points, along with other statistics (L2 cache hit rate, image distribution time, image processing time and total execution time) are transmitted back to the host PC via the Ethernet interface to be visualized as image overlay in real-time.

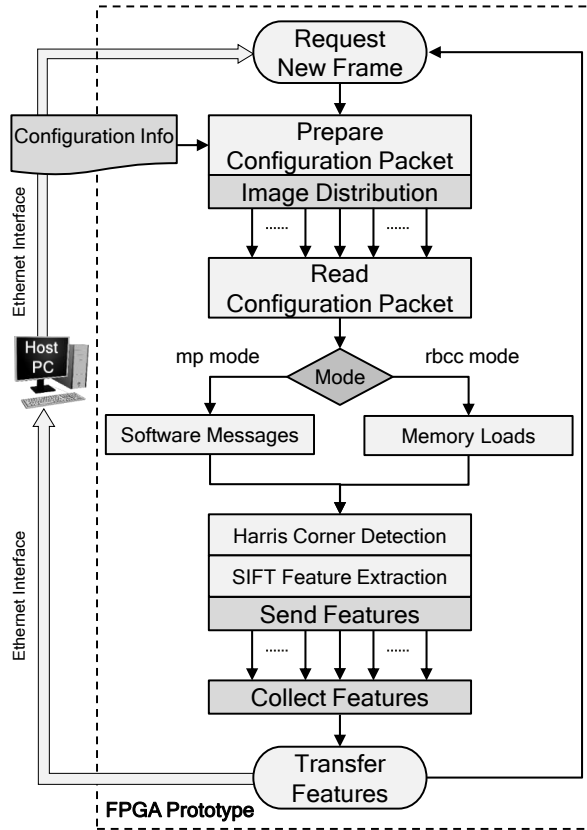


Figure 3.16: A flow diagram depicting the feature extraction task

3.6.4 Results and Analysis

The feature extraction task of the video streaming application is executed in both shared memory (*rbcc*) and MPI-based (*mp*) modes. Further, two coherence region configurations - *clustered* and *corner*, as depicted in Figures 3.17 and 3.18 respectively are used for the experiments. The two coherence region configurations have been chosen to demonstrate and evaluate RBCC's flexibility feature. Lastly, the impact of Background Traffic (BT) during application execution is investigated for both programming modes and the two coherence region configurations. The experiments with BT exposes the application

¹³A specific processing element can also be specified

3 Region-based Cache Coherence (RBCC)

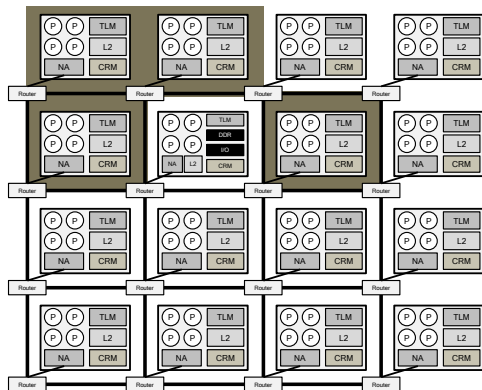


Figure 3.17: Clustered coherence region

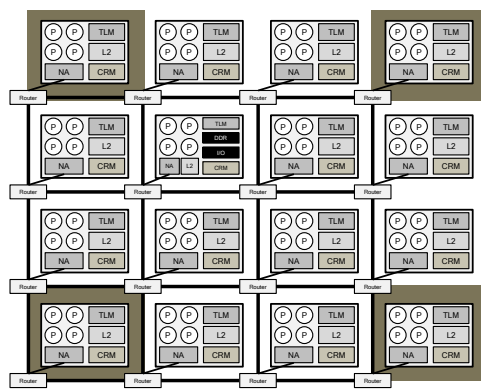


Figure 3.18: Corner coherence region

to a real-world scenario, where several applications are expected to be running simultaneously. BT is generated using a separate application that continuously sends synthetic DMA messages in parallel to the video streaming application on all tiles of the manycore system other than the configured coherence region tiles. The amount of BT load to be injected into the manycore system is controlled by varying the length of a DMA transfer. For the experiments, four DMA sizes are used: 50 kB, 100 kB, 250 kB and 500 kB. For *rbcc* mode runs, the coherence region is configured before the application begins. Additionally, the CRMs transparently adapts to the truly shared application working-sets at run-time using `RBCC-malloc()`. For *mp* mode runs, the CRM is not required, and is therefore disabled.

The application is fed with a video file from the host PC. The resolution of the video input determines the size of each frame to be transferred and processed on the FPGA prototype. The content of the video input determines the number of detected feature points, in-turn varying total application execution time. Considering the available memory on the FPGA prototype, video inputs with moderate resolution and duration are considered. Therefore, short clips (600–700 frames) of popular retro video games (inherently low-resolution) like *donkeykong*, *spaceinvaders*, *pacman* and *snake* are used. Each of the sample gameplay clips exhibit different video output characteristics (as will be shown in Section 3.6.4.1), providing a diverse input set to the feature extraction task.

Initial Observations

To get an impression of the application’s sensitivity with respect to the spatially different coherence region configurations and the BT load on the manycore system, some initial experiments were conducted and the observations are summarized below. In the absence of BT, the spatial locality of the coherence region configuration had virtually no influence on the application’s execution time for both programming modes, i.e. $T_{rbcc-clustered} \approx T_{rbcc-corners}$ and $T_{mp-clustered} \approx T_{mp-corners}$. This result is due to the fact that the manycore system contains little traffic, allowing all inter-tile communication to go on unimpeded.

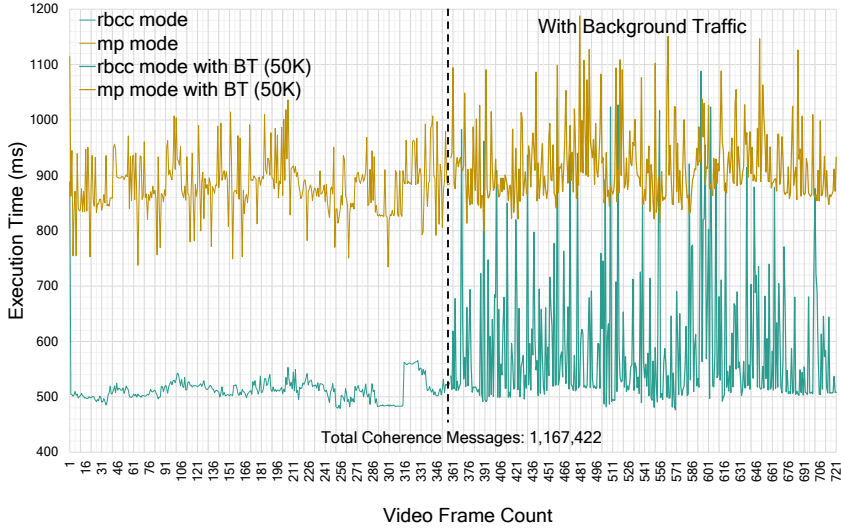


Figure 3.19: Per-frame execution time of the **donkeykong** video clip, with and without BT

With the injection of BT into the manycore system, the *clustered* coherence region configuration was still virtually unaffected¹⁴ for both modes. This is because, the tiles in the *clustered* configuration are tightly coupled, thereby limiting the impact of BT on inter-tile communication. However, the *corners* coherence region configuration showed sensitivity to BT. Therefore, all further experiments are conducted with the tiles setup in the *corners* coherence region configuration, both with and without the presence of BT, evaluated for the *rbcc* and *mp* programming modes.

3.6.4.1 RBCC mode versus MP mode

Figures 3.19 to 3.22 illustrate the per-frame execution time of the feature extraction task for both programming modes for all video clips. In order to better analyse the impact of BT, the second halves of these graphs show the per-frame execution time with the presence of BT (50 kB).

donkeykong: This clip has a steady background with moving objects, resulting in a relatively constant execution time for both programming modes. In the absence of BT, the execution time of the application using *rbcc* mode finishes 42% faster than that of the *mp* mode. In the presence of BT, there exist delay spikes in the execution time for both modes as the inter-tile communication is impeded. The application’s execution time using *rbcc* mode is still 35% faster than that of the *mp* mode.

spaceinvaders: This video clip shows a periodic removal of objects (aliens killed), which are then replaced by new ones (aliens re-spawned). Therefore, the execution time resembles a fading wave pattern. Without BT, the application’s execution time using *rbcc* mode is 31% faster than that of the *mp* mode. The injection of BT impacts both programming modes, but the *rbcc* mode finishes 29% faster than the *mp* mode.

¹⁴The execution time at-most increased by 5%

3 Region-based Cache Coherence (RBCC)

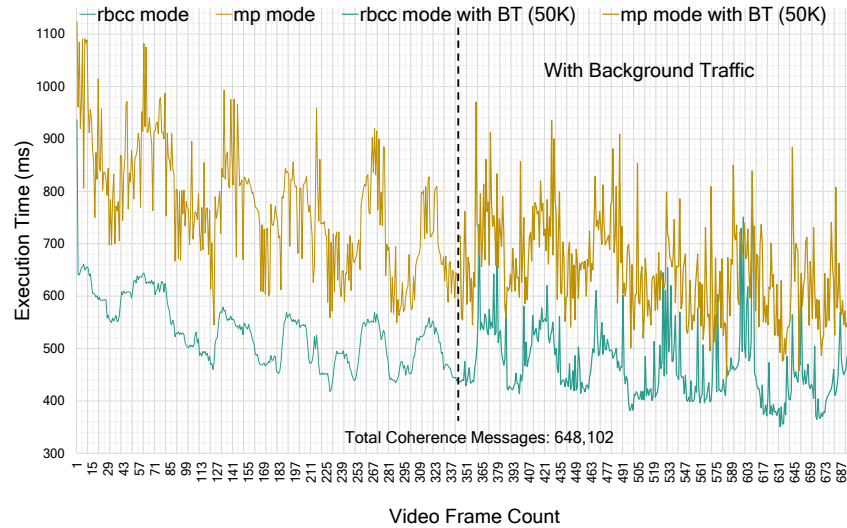


Figure 3.20: Per-frame execution time of the **spaceinvaders** video clip, with and without BT

pacman: In this popular video game clip, the number of extracted feature points are ever diminishing (eaten by pacman), resulting in a continuous decrease in the application’s execution time for both programming modes. Using the *rbcc* mode, the application finishes 37% and 31% faster than that of the *mp* mode with and without BT respectively.

snake: This video clip is sampled when the size of the snake is roughly constant. Therefore, the application’s execution time is relatively constant. In the absence of BT, the application’s execution time using the *rbcc* mode is 42% faster than that of the *mp*

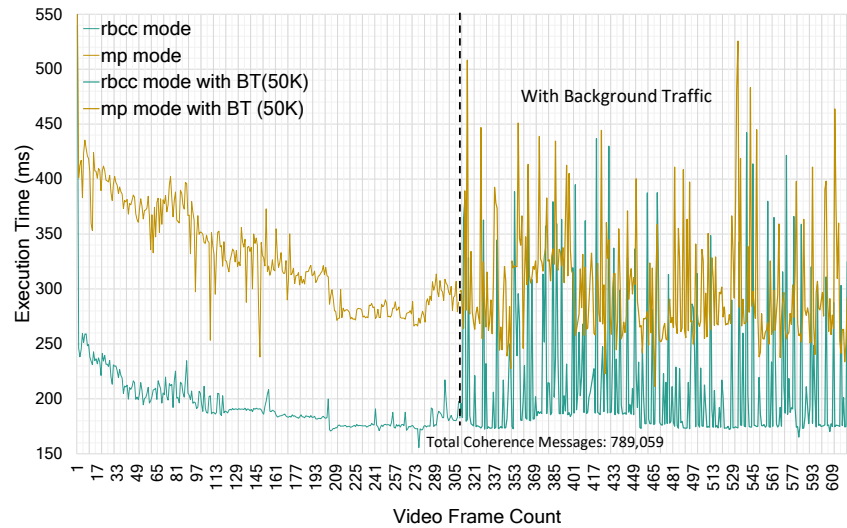


Figure 3.21: Per-frame execution time of the **pacman** video clip, with and without BT

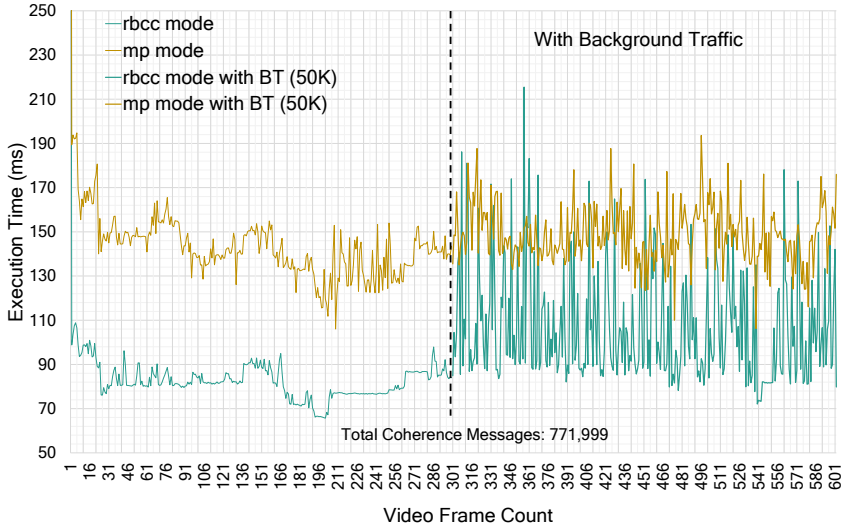


Figure 3.22: Per-frame execution time of the *snake* video clip, with and without BT

mode. With BT, the application’s execution time using the *rbcc* mode is 28% faster than of the *mp* mode.

Breaking-down the Execution Time

To better understand and interpret the differences of the two programming modes, the application’s execution time is broken-down. Referring to Figure 3.16, the major difference between the two programming modes are in the image distribution methodology. For the *mp* mode, the total execution time (T_{mp}) is divided into two parts namely Image Distribution Time (IDT) and Image Processing Time (IPT). IDT_{mp} represents the time taken by the application to distribute the image using explicit software messages which are DMA-assisted. IPT_{mp} represents the time taken by the application to process the image. It is important to note that all memory accesses during IPT_{mp} are local, as the data has already been copied by explicit software messages. For *rbcc* mode, the total execution time cannot be easily split into two distinct parts as done with the *mp* mode. The reason being, for the RBCC mode, the IDT and IPT phases overlap with each other, i.e. the overall execution time (T_{rbcc}) is a mixture of fetching remote data, followed by local processing which cannot be easily distinguished. Theoretically, IDT_{rbcc} represents the time taken by the application to simply convey the location of the shared image data to the respective remote tile. This consists of sending an address pointer of the shared image data to the remote tiles, whose time consumption is negligible. The IPT_{rbcc} can be viewed as the total execution time (T_{rbcc}) as it represents the time taken to process the image, which is a mixture of on-demand remote TLM accesses and local processing. The on-demand remote access penalties are reduced by the L2 cache which is kept coherent by the CRM.

3 Region-based Cache Coherence (RBCC)

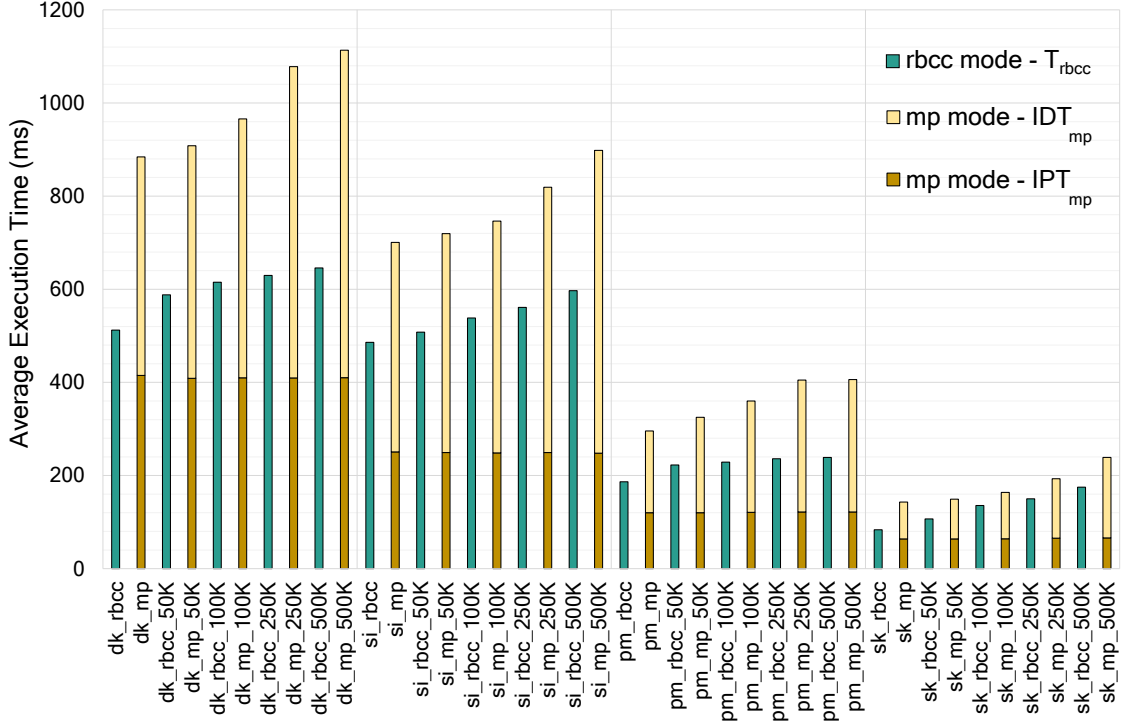


Figure 3.23: Average execution time of the *rbcc* mode and *mp* mode for all video clips with increasing BT

Therefore, for the execution time analysis, T_{mp} ($IDT_{mp} + IPT_{mp}$) is compared to T_{rbcc} . Additionally, a BT traffic sensitivity analysis is performed by increasing the BT load for all video clips. As illustrated in Figure 3.23, the IPT_{mp} is actually lower than T_{rbcc} for all video clips. This is because, all memory accesses in the *mp* mode result in local accesses, whereas in the *rbcc* mode they result in remote TLM accesses assisted by a local L2 cache. However, with the addition of IDT_{mp} , T_{mp} exceeds T_{rbcc} . This is due to the software overheads of explicit message passing required for image distribution to all remote tiles. IDT_{rbcc} is virtually negligible as each remote tile only receives a pointer to the shared image data and can therefore directly access the image during IPT_{rbcc} via remote load operations. For all *rbcc* mode runs, high L2 cache hit rates are recorded ($> 90\%$) for all video clips. This reduces the remote TLM access penalties and in-turn the overall processing time.

Regarding the sensitivity to BT, all inter-tile operations are affected due to the added network load, which is clearly seen in Figure 3.23. When using the *mp* mode, IDT_{mp} increases with an increase in BT for all video clips as it consists of remote DMA operations that are impeded by the added network traffic. However, IPT_{mp} remains constant with increase in BT, as all memory accesses during image processing result in local accesses. In *rbcc* mode, T_{rbcc} increases with BT as it consists of on-demand remote TLM accesses that are now slower due to the increased network load. But, in the *rbcc* mode, the application exhibits an overall lower execution time and saturates quicker than the *mp*

mode with increasing BT. This is visualized using the trend-lines of Figure 3.23. This is because, in the *rbcc* mode, inter-tile communication consists of quick load/store operations which have an overall lower latency in the presence of BT, compared to large DMA packets of the *mp* mode.

One could intuitively question whether the *mp* mode could mimic the *rbcc* mode by fetching data as required during processing. However, this would increase the message passing overhead penalty as each tile would send costly software messages in-between processing, which would further degrade the application’s performance, especially in the presence of BT. It is important to note that the *mp* mode uses a hardware DMA engine (burst accesses) to distribute the image whereas the *rbcc* mode uses regular load/store operations. For systems without a DMA engine, IDT_{mp} would also use regular load/store, further increasing the performance improvement of the *rbcc* mode.

3.6.4.2 Run-time Re-configuration Analysis

This section demonstrates and evaluates the benefits of RBCC’s flexibility feature by re-configuring coherence regions at run-time using the video streaming application. The malleable property of the video streaming application allows it to be expanded, shrunk or even relocated to different subset of hardware resources at run-time on a per-frame granularity. The CRM’s Configuration Unit sub-module is responsible for all run-time re-configurations. It makes sure that all requested coherence regions are created/destroyed, and that the corresponding Directory entries are cleared, before allowing the application to start/resume safely. Using two scenarios, this section demonstrates the benefits being able to dynamically re-configure the video streaming application at run-time using the CRM.

Scenario 1: Expanding the Coherence Region: The goal of this scenario is to demonstrate the benefits of dynamically expanding the coherence region to contain more hardware resources at run-time. Initially, the video streaming application is executed on a single compute tile, assuming that other applications have occupied the remaining tiles. When the other applications finish executing, the video streaming application has the opportunity to expand onto more hardware resources to increase its degree of parallelism, potentially reducing its overall execution time. It is assumed that new compute tiles will be available after processing every $1/3^{rd}$ of the total frames.

The results of this experiment are illustrated in Figures 3.24 and 3.25 using the *donkeykong* and *snake* video clips. It shows the per-frame execution time of the video streaming application as it expands to more compute tiles after processing a third of the total frames. For the *donkeykong* video input, on average, the execution time reduces by 38% when expanding from a single tile to two tiles, and by 21% when further expanding to four tiles. For the *snake* video input, on average, the execution time reduces by 47% when expanding to two tiles, and by 28% when further expanding to four tiles.

Expanding the coherence region only changes the sharers field. Therefore, all CRM Configuration Units that are part of the coherence region selectively reset their Directory entries and synchronize before the application can safely resume. For both video

3 Region-based Cache Coherence (RBCC)

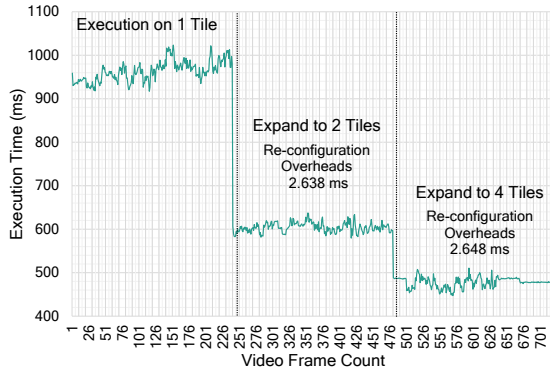


Figure 3.24: Expanding the coherence region using the **donkeykong** clip

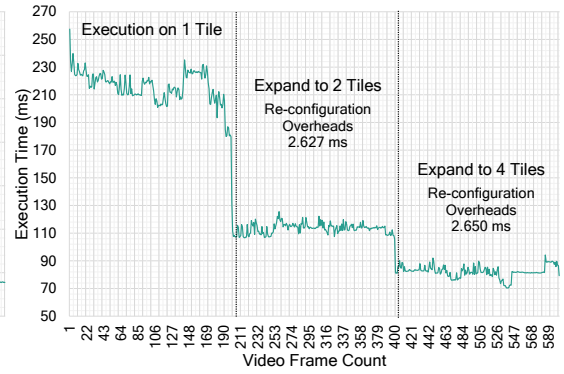


Figure 3.25: Expanding the coherence region using the **snake** clip

clips, the re-configuration time is ≈ 2.6 ms, which is negligible when compared to the application’s per-frame execution time.

Scenario 2: Relocating the Coherence Region: The goal of this scenario is to demonstrate the benefits of dynamically relocating the coherence region at run-time. Initially, the video streaming application is executed on four compute tiles that are in the *corners* coherence region configuration. Further, BT traffic (50 kB) is injected into the manycore system between all other tiles to impede inter-tile communication, thereby increasing the application’s execution time. This situation can be overcome by dynamically relocating to a *clustered* coherence region configuration, assuming that the necessary hardware resources are available. It is assumed that, after processing half of the total frames, the application decides to relocate to a *clustered* coherence region configuration. It is important to note that, relocation does not add more compute tiles, rather moves them closer to each other, which could potentially reduce the impact of BT on inter-tile communication.

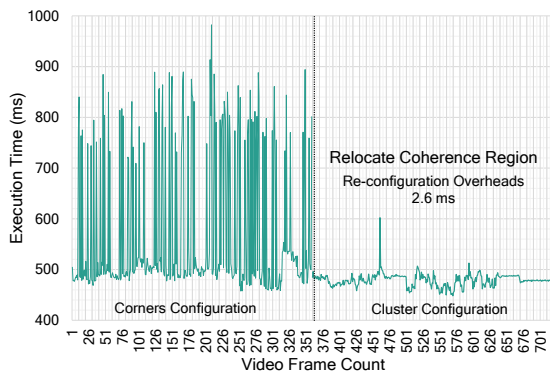


Figure 3.26: Relocating the coherence region using the **donkeykong** clip

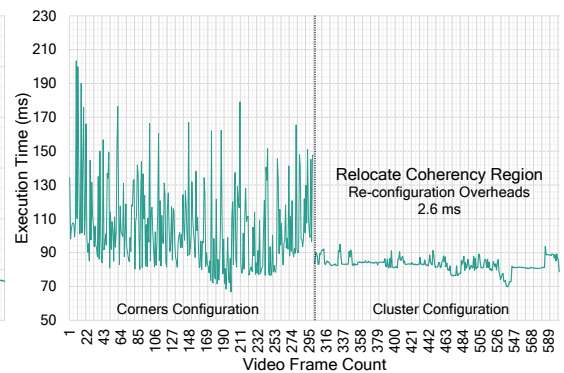


Figure 3.27: Relocating the coherence region using the **snake** clip

Figures 3.26 and 3.27 illustrate the per-frame execution time of the video streaming application when switching from a *corner* coherence region configuration to a *clustered* coherence region configuration, using the *donkeykong* and *snake* video clips. It can be clearly seen that the delay spikes induced by BT drastically reduce post relocation for both video clips. On average, this reduces the execution time by 17% and 21% for the *donkeykong* and *snake* video clips respectively. Again, the CRM's Configuration Unit overheads are negligible (≈ 2.6 ms) compared to the application's per-frame execution time.

Formal Analysis. For the explored video streaming application and the two scenarios, the re-configuration overheads do not have any negative impact on the application's performance. This is because, the re-configuration overheads are significantly smaller than the application's per-frame execution time. Therefore, triggering a re-configuration at any point in the application's execution lifespan will be beneficial. However, this may not be the case for all applications. Therefore, a formal analysis of when a re-configuration would be beneficial for a generic application, accounting for the CRM's re-configuration overheads is provided below.

The benefits of a re-configuration process depends on:

- Whether the application would benefit (reduction in execution time) from increased parallelism or relocation,
- The time when the re-configuration process is triggered,
- The time required for the re-configuration process.

Let T be the total execution time of an application and T_r be the total execution time of the same application with one re-configuration. T_r can be expressed as a combination of three parameters: the time before re-configuration (T_{br}), the time required by the CRM's Configuration Unit to reset the Directory (T_{reconf}) and the time after re-configuration (T_{ar}). The goal of a re-configuration should always be:

$$T_r < T, \text{ where } T_r = T_{br} + T_{reconf} + T_{ar} \quad (3.5)$$

Assuming that the re-configuration process reduces the application's execution time, the time after re-configuration can also be represented as:

$$T_{ar} = (T - T_{br}) \times \alpha, \text{ where } 0 < \alpha < 1 \quad (3.6)$$

By combining Equation (3.5) and Equation (3.6):

$$\begin{aligned} T_{br} + T_{reconf} + \{(T - T_{br}) \times \alpha\} &< T \\ T_{br}(1 - \alpha) + T_{reconf} + T\alpha &< T \end{aligned} \quad (3.7)$$

By rearranging and solving for the time elapsed before re-configuration:

$$T_{br} < T - \frac{T_{reconf}}{(1 - \alpha)} \quad (3.8)$$

An application benefits from a re-configuration only if Equation (3.8) holds true, where,

3 Region-based Cache Coherence (RBCC)

- T_{br} : The maximum time elapsed before re-configuration, beyond which a re-configuration will not be beneficial to the application.
- α : Represents whether an application benefits (reduction in execution time) from a bigger coherence region (increased parallelism) or a relocation of the coherence region (avoiding BT).
- T_{reconf} : Re-configuration time, which in this case consists of all CRM Configuration Units resetting their directory entries and synchronizing ¹⁵.

The task of monitoring these parameters, performing a feasibility check and triggering the re-configuration process is left to the OS. By profiling an application, the OS can determine whether triggering a re-configuration would be beneficial to the application's overall execution time. In case of an increase in hardware resources, α represents the expected speedup, which is calculated as the ratio of the application's execution time on more number of resources to that on lesser number of resources, as shown in Figures 3.24 and 3.25. In case of hardware resource relocation, α is calculated similarly but represents the speedup due to the mitigation of network traffic as shown in Figures 3.26 and 3.27. For the explored video streaming application, $\alpha < 1$, indicating that this application favours an increase in resources.

The re-configuration overheads T_{reconf} depend on changes to the start/end address or sharers fields which triggers a reset of the Directory entries. These overheads were already reported in Section 3.6.2. The video streaming application uses approximately 1.5 MiB of the TLM which corresponds to $T_{reconf} \approx 2.6$ ms as reported in Figure 3.15. A low T_{reconf} combined with a favourable α results in $T_r < T$ for both scenarios of the explored video streaming application. Similarly, for any given application, by tracking/monitoring these parameters (T_{br} , α , T_{reconf}), the OS can accelerate the application by triggering a coherence region re-configuration, such that Equation (3.8) is not violated.

3.6.4.3 RBCC-malloc() Analysis

RBCC-malloc() [18] tailors coherence support to actually shared application working-sets that are only known at run-time. This section shows how RBCC-malloc() reduces the intra-CRM communication traffic, thereby increasing the CRM's efficiency. The CRM's Snoop Unit uses the address range of the application's dynamically allocated data as an additional filter to discard irrelevant transaction messages. This reduces the load on the CRM's FIFO and Management Unit sub-modules. In order to quantify these savings, the amount of FIFO transactions with and without RBCC-malloc() are computed.

Figure 3.28 illustrates the number of FIFO transactions for all four video clips executed in the *corners* coherence region configuration with standalone RBCC (tracking the complete address range within a coherence region) and with RBCC-malloc() enabled (only track actually shared address ranges within the coherence region). With RBCC-malloc() enabled, the load on the FIFO is reduced by approximately 40% for the

¹⁵For some applications, this can additionally contain state transfers

3.6 Hardware Implementation and Evaluation - FPGA Prototype

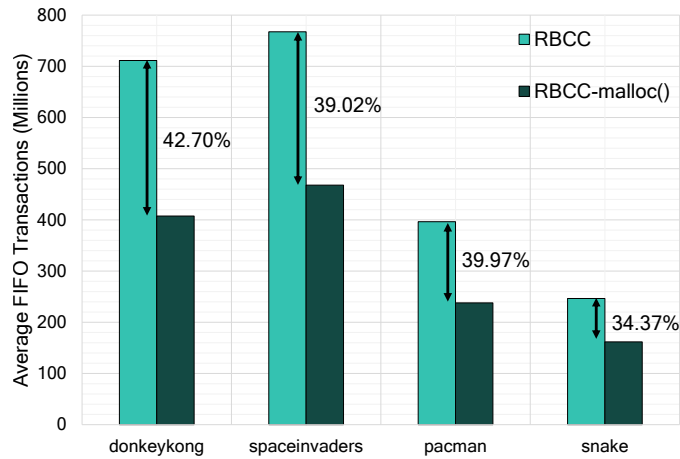


Figure 3.28: FIFO load reduction when using RBCC-malloc() for all video clips

video streaming application. This is because, RBCC-malloc() allows the Snoop Unit to filter-out unnecessary transactions that request to track private data structures such as tile local OS data, core instructions, core stacks, etc. Conservatively tracking these tile/core-private address ranges results in unnecessary traffic exchange between the FIFO and Management Unit, leading to redundant coherence actions.

3.7 Enabling Shared Memory Workloads

The previous sections evaluated RBCC and compared its performance with that of message passing techniques. This section describes how the concept of RBCC was applied to enable shared memory workloads on a DSM-based tiled manycore system. As described in Section 3.1.1, the InvasIC target architecture does not support inter-tile coherence. Therefore, all applications ¹⁶ use software DSM techniques like library-based (MPI) or language-based (X10) to support inter-tile communication and synchronization. While there are a large number of benchmarks that use MPI (NAS Parallel Benchmarks (NPB)) and X10 (IIT Madras Benchmark Suite (IMSuite)), the more commonly used shared memory workloads like the PARSEC and Stanford Parallel Applications for Shared Memory (SPLASH-2) benchmark suites are not supported on the InvasIC target platform.

Therefore the goal was to adapt the SPLASH-2 benchmarks to be executable on the DSM-based tiled manycore architecture. This required redesigning and/or adapting parts of the software stack in order to support the shared memory programming model. A “shared-memory software layer” was introduced by Tobias Langer, a colleague from the Department of Computer Science 4 (Distributed Systems and Operating Systems) at Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). This brought in several modifications, of which a few important ones are listed below:

- Mapping InvasIC-specific function calls to shared-memory (pthread) routines,
- Using the common DRAM memory for global shared variables,
- Adapting synchronization primitives.

3.7.1 Two Methodologies for Shared Memory Programming

These modifications opened up two methodologies to enable the shared memory programming paradigm - a software approach by using Virtual Shared Memory (VSM) [78] and a hardware approach using the RBCC concept. A brief description on the working principles of both these approaches is given below.

Working Principle - VSM

The software approach uses the concept of VSM to guarantee inter-tile coherency and consistency. As the name states, VSM grants applications a virtual shared memory view of the global memory. It uses a hardware Memory Management Unit (MMU) module in order to fetch data (at page granularity) from the DRAM and store it in the TLM to be used by the application. At every synchronization point, the VSM mechanism invokes a software routine that checks for possible changes between different TLMs and updates them accordingly. This ensures a coherent and consistent view of the memory for the application.

¹⁶The video streaming application is an exception as it supports both shared memory and MPI-based execution modes

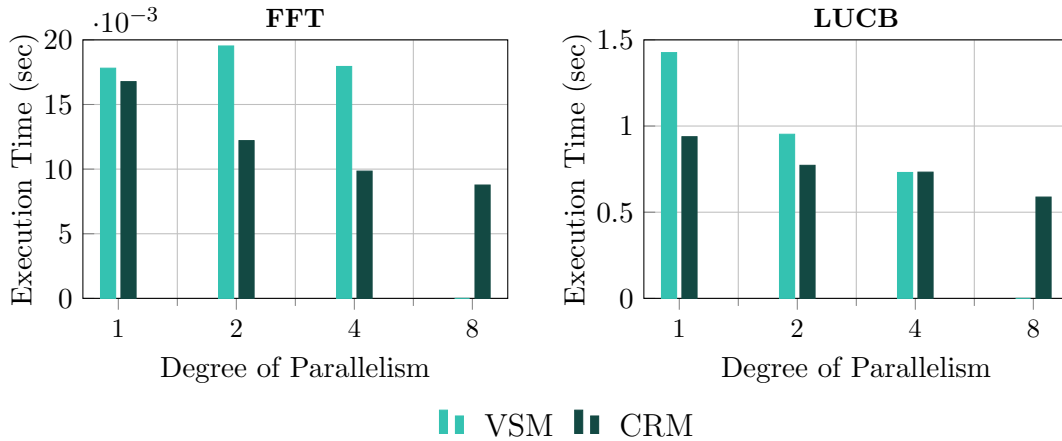


Figure 3.29: The execution time of two SPLASH-2 benchmarks using both VSM- and CRM-based approaches to enable shared memory programming, for different DoPs

Working Principle - RBCC

The hardware approach makes use of the CRM to guarantee inter-tile coherency and consistency. As application data resides in the DRAM, a coherence region is set up to include the memory tile in addition to the participating compute tiles. The application’s threads can then communicate directly via the coherent shared DRAM address space. At every synchronization point, the CRM’s coherence barrier mechanism is invoked. This makes sure that all coherence messages have been successfully executed, ensuring a consistent view of the memory subsystem.

3.7.2 Experimental Setup and Evaluation

The work on adapting the SPLASH-2 benchmarks to be executable on the InvasIC target architecture is currently ongoing. Therefore, only a few benchmark kernels have been used to demonstrate the two different approaches. The FPGA prototyping platform is set up similar to that of Section 3.6.1. The *fft* kernel is executed with the “-m12” problem size, and the *lucb* kernel is executed with its default configuration. In order to evaluate the cost of inter-tile communication, both benchmarks are executed with different Degrees of Parallelism (DoPs) (1, 2, 4 and 8), assuming only one core per tile. For example, a DoP of 4 spawns the benchmark of four tiles of the manycore system.

Figure 3.29 reports the execution time of the VSM and RBCC approaches, for different DoPs¹⁷. The execution time of both workloads reduce with increasing DoPs (except for *fft* with a DoP of 2 using VSM). For the *fft* kernel, the RBCC-based approach outperforms the VSM-based approach for all DoPs. For the *lucb* kernel, the same result holds true, except for DoP(4), where both approaches exhibit the same performance. The overall result indicates that the workloads generally execute faster (by up to 45%

¹⁷Due to ongoing implementation work, the VSM approach does not support the DoP of 8 yet

3 Region-based Cache Coherence (RBCC)

for *fft* and 34% for *lucb*) when using the RBCC-based approach compared to the VSM-based approach. This is expected since RBCC uses a dedicated hardware CRM module to handle inter-tile coherence, whereas, VSM orchestrates inter-tile communication and synchronization in software. Moreover, the VSM approach requires an MMU module which adds address translation overheads, that are not needed for the RBCC approach. However, referring to the working principles, the VSM approach uses the faster TLM, while the RBCC approach uses the global DRAM. Therefore, if application data can be placed within TLMs instead of the DRAM, coherence regions can be set up for the corresponding TLM address ranges. This would reduce memory access latencies, leading to further reduction in benchmark execution time for the RBCC approach.

These results in no-way rule out the applicability of the software-based approach. Most importantly, it can be used in computing platforms that do not support or cannot afford hardware-based coherence. Another use-case is to combine VSM with RBCC. This hybrid approach is described as part of future work in Chapter 5.

3.8 Additional Case-Study - RBCC and In-NoC Circuits (INCs)

The RBCC concept enables scalable and flexible inter-tile coherence for large tile-based manycore architectures. This requires sending and receiving coherence messages via the NoC interface of the manycore system. The latency of these coherence messages depend on factors like the NoC traffic and the hop-distance between tiles that are part of a coherence region. To reduce the latency of coherence messages, RBCC was combined with the concept of In-NoC Circuits (INCs) [5]. This research study was done in collaboration with colleagues from Institute for Information Processing Technologies (ITIV), Karlsruhe Institute of Technology (KIT), who work on innovative NoC concepts. This section provides a brief insight as to how the concept of RBCC was used with in combination INCs. An in-depth description of the INC concept, design, implementation and evaluation can be found in the research work [5].

3.8.1 Concept

An efficient NoC architecture plays an important role in optimizing inter-tile communication on manycore systems. NoCs generally use packet-switching for best-effort traffic or circuit-switching for low-latency communication. The idea of INCs [5] is built on hybrid NoC architectures that make use of both packet-switching and circuit-switching schemes. The packet-switched layer is used for regular inter-tile communication, and the circuit-switched layer is modified to be used as INCs for low-latency communication. The INCs have been designed with the following properties:

- They start and terminate between two routers of a NoC,
- They can skip/jump-over certain routers,
- They are created and/or destroyed at run-time by analysing ongoing traffic.

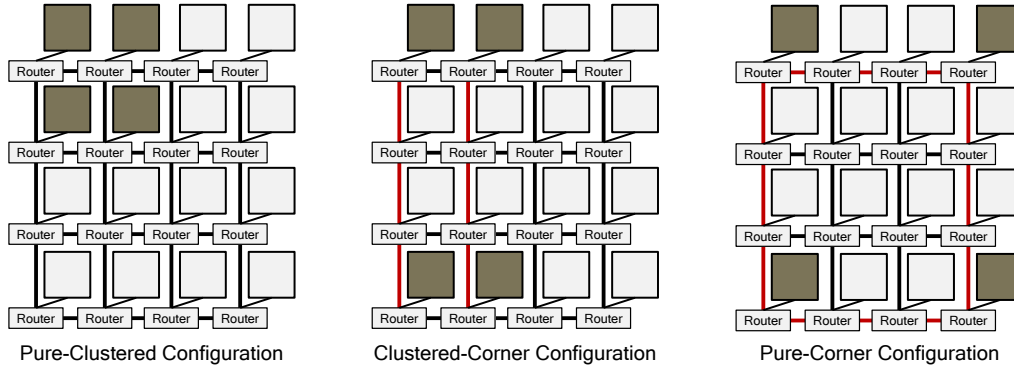


Figure 3.30: Different coherence region configurations on a 4x4 tile-based manycore system

The INCs exhibit properties such as dynamicity and flexibility that are similar to RBCC. This led to the idea of combining them together, i.e., dynamically establishing INCs between tiles of a coherence region. As a result, the coherence messages are accelerated by the INCs, without which they would use the regular packet-switched layer of the NoC.

3.8.2 Experimental Setup and Evaluation

The primary goal of using INCs with RBCC is to minimize the disadvantages of coherence regions that comprise of spatially distant tiles. In order to evaluate this, three different coherence region configurations have been explored on a 4x4 tile-based manycore system as illustrated in Figure 3.30. The INCs continuously monitor ongoing inter-tile coherence traffic and dynamically establish low-latency links between the tiles of a coherence region. The experiments were performed using two simulation platforms - the high-level cache simulator introduced in Section 3.5.1 and a cycle-accurate network simulator developed at the ITIV, KIT. The high-level cache simulator was modified to additionally provide network-related traces, which were in-turn fed into the cycle-accurate network simulator. A total of 8 benchmarks from the PARSEC and SPLASH-2 benchmark suites were chosen.

Figure 3.31 reports the average delay of inter-tile coherence messages with and without INCs for all benchmarks. The objective is for the *clustered-corner* and *pure-corner* configurations to achieve similar performance as the *pure-clustered* configuration. Therefore, the delay values have been normalized to that of the *pure-clustered* configuration. One evident result is that, with INCs enabled, the latency of inter-tile coherence messages is reduced for all benchmarks. For instance, enabling INC reduces the latency of inter-tile coherence messages by up to 40% (fft) and 45% (canneal) for the *clustered-corner* and *pure-corner* configurations respectively. This is a promising result, specifically for creating coherence regions comprising of spatially distant tiles. Furthermore, for the *clustered-corner* configuration, results show that with INC, the latency actually reduces by 3% – 12% compared to the tightly-coupled *pure-clustered* configuration. This additional reduction in latency is attributed to the fact that communication to/from the

3 Region-based Cache Coherence (RBCC)

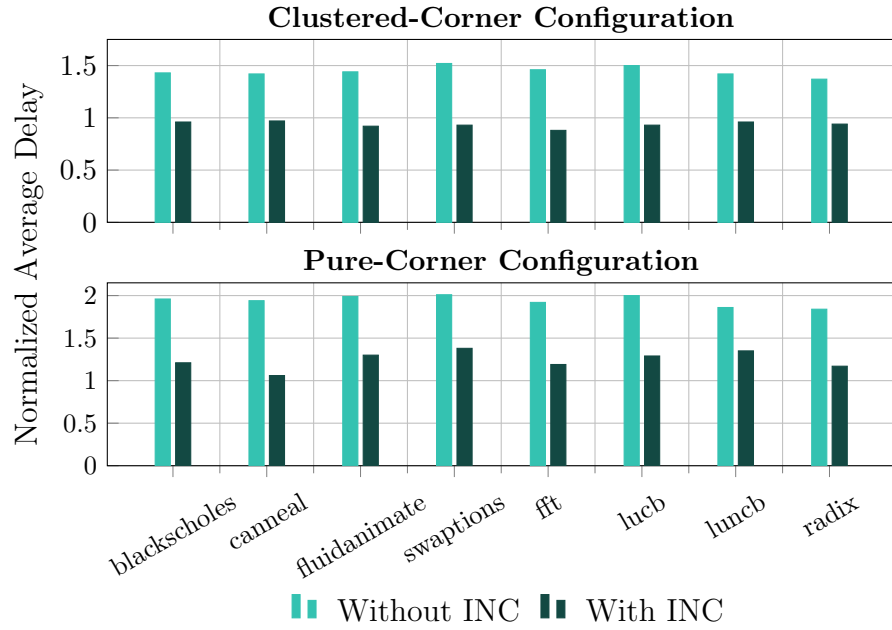


Figure 3.31: The normalized average delay of inter-tile coherence messages for the *clustering-corner* and *pure-corner* configurations with and without INCs for all benchmarks. Adapted from [5]

memory tile is also implicitly accelerated by the INCs. The take-away point is that RBCC combined with low-latency INCs allows for spatially distant tiles to be part of a coherence region, whilst maintaining reasonable inter-tile communication latencies.

4 Hybrid Voting-based Eviction Policy (HyVE)

4.1 The HyVE Concept

The primary function of any eviction policy is to select one victim for eviction from a given set of candidates (memory blocks). Standalone eviction policies make use of a pre-defined eviction criterion to evict the optimal candidate. For example, standalone eviction policies like LRU and LFU have been designed to optimize for eviction criteria such as recency and frequency respectively. Optimizing for a single eviction criterion might lead to sub-optimal eviction decisions, specifically for workloads that exhibit non-uniform memory access patterns. The ever increasing complexity of modern applications further accelerates the trend towards erratic memory access patterns.

This thesis proposes HyVE, a hybrid eviction policy that attempts to optimize for several eviction criteria. The primary idea of HyVE is to combine several standalone eviction policies together. To be incorporated within HyVE, the functionality of a standalone eviction policy requires slight modifications. Generally, standalone eviction policies select one eviction victim from a given set of candidates/memory blocks. This process inherently discards all information on other candidates, which could be useful. When used within HyVE, every standalone eviction policy is extended with ranking capabilities. This tweak transforms the binary victim-selection decision of the standalone eviction policies into a process where each policy can additionally express its opinion over all given candidates. As a final step, this opinion information is fed into a voting system, which decides on the actual eviction victim.

Figure 4.1 demonstrates the concept of HyVE using three constituent eviction policies. Instead of readily selecting an eviction victim, each constituent eviction policy within HyVE provides a rank to each eviction candidate based on their individual optimization attributes. For example, if *Policy I* was used as a standalone eviction policy, it would evict *Candidate C*. But within HyVE, *Policy I* just expresses its opinion over each candidate. In the example, *Policy I* expresses a strong preference to evict *Candidate C* followed by *A*, *B* and

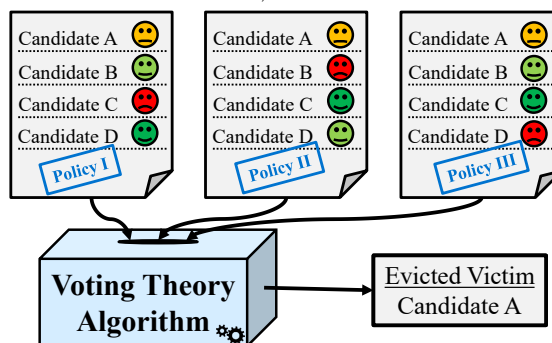


Figure 4.1: An abstract example demonstrating the basic concept of HyVE using three eviction policies casting their votes on four candidates

D. Similarly, all constituent eviction policies within HyVE rank the candidates based on their respective eviction criterion. The generated ranks are fed into a voting system which determines the actual eviction victim. In the example, *Candidate A* is selected as the eviction victim. Interestingly, none of the individual eviction policies independently expressed a strong preference to evict *Candidate A*. Nevertheless, *Candidate A* was evicted due to a consensus between all participating eviction policies within HyVE. This ability of HyVE to take unique eviction decisions than its constituent eviction policies makes it a new and unique eviction policy.

4.1.1 Voting Theory Background

HyVE requires a decision-making system to consolidate the opinions of its constituent eviction policies. This is done by borrowing concepts from the voting theory domain. Voting theory studies how opinions expressed by different decision makers can be consolidated together [79]. It is mainly applied in two contexts namely formal and informal. Voting in the formal context is commonly used for political elections or by juries. Informal voting is usually used by a group of people to

Table 4.1: Ranking Distribution

#Voters (V)	Preferences
8	$x > y > z$
6	$y > x > z$
3	$y > z > x$
6	$z > x > y$
4	$z > y > x$

decide which restaurant/bar to visit or to determine which movie to watch. The most common voting system in the informal context is plurality voting, which requires only little information from the voters. Plurality voting only collects information on which candidate is the best from each voter’s perspective, disregarding any additional information on alternate candidates. The candidate that receives the most votes is elected.

Consider an example ranking distribution listed in Table 4.1, where a total of $V = 27$ voters express their preferences on $C = 3$ candidates (x, y, z). If plurality voting is used, only the best candidate from the voters’ perspective is considered, disregarding any information on the other two candidates. For the example ranking distribution, 8 voters pick candidate x , 9 voters pick candidate y and 10 voters pick candidate z as their most preferred candidate. Therefore, candidate z would be elected. Existing standalone eviction policies can be viewed as using the plurality voting procedure to determine the eviction victim.

In 1770, Jean-Charles de Borda highlighted that plurality voting may lead to a choice that would lose, when compared to other alternatives, using pairwise comparisons by a majority of voters. For this to be possible, each voter would need to express their preference/opinion over all candidates, by providing ranks. Inspired by this idea, HyVE tweaks standalone eviction policies to additionally provide rankings for all eviction candidates. These rankings are fed into a voting system to determine the eviction victim. From HyVE’s perspective, its constituent eviction policies behave like voters and the elected candidate is the eviction victim. In this thesis, HyVE is evaluated using two voting methodologies namely the Borda Count and the (basic) Condorcet Method. Both voting methodologies make use of the additional preference/opinion information to determine the eviction victim.

Borda Count

Proposed by Jean-Charles de Borda, this methodology awards points to candidates based on their position in the individual rankings. Given C candidates, a candidate will receive n points when ranked first, $(n - 1)$ points when second, $(n - 2)$ points when third, and so forth. The candidate which accumulates the maximum points is elected. The points for each candidate $C_i \in C$ is calculated as:

$$B(C_i) = \sum_{i=1}^{|V|} \text{points}(C_i) \quad (4.1)$$

The voting outcome is computed as $\max_i B(C_i)$, where V is the set of voters. For the example ranking distribution,

$$\max(B(x) = 55, B(y) = 57, B(z) = 50) = B(y) \quad (4.2)$$

Therefore, candidate y is elected. From an implementation perspective, the candidates' individual points $B(x), B(y), B(z)$ are computed in parallel.

Condorcet Method

In this methodology, all pairwise comparisons are evaluated, and the candidate who wins the most pairings is the preferred choice by a majority of voters. If C is the number of candidates, then the number of pairwise comparisons is given by:

$$\frac{C \cdot (C - 1)}{2} \quad (4.3)$$

If V is the number of voters and n_{ij} are the number of voters who prefer candidate i over candidate j , then i is said to be globally preferred to j if:

$$n_{ij} > \frac{V}{2} \quad (4.4)$$

For the example ranking distribution,

$$n_{xy} = 14, n_{yz} = 17, n_{xz} = 14 \quad (4.5)$$

Since candidate x won twice, candidate x is elected.

The Condorcet Method can also be represented as a graph, as shown in Figure 4.2. It may happen that no candidate is elected due to the *Condorcet paradox* (cycle in the graph) which can occur when $C > 2$. There exist several methods to eliminate such conflicts, such as the Black method (falling back to the Borda Count), the Copeland method, etc. Interested readers are encouraged to refer to the following books [80, 79] for further information.

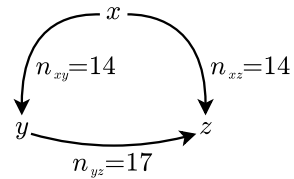


Figure 4.2: Condorcet Method

It is interesting to note that the three voting procedures elected different candidates for the same ranking distribution. This already indicates the importance of the voting methodology on eviction decisions.

4.2 HyVE: Features and Design

4.2.1 Rank Generation

The idea of HyVE is to combine several standalone eviction policies together to make an eviction decision. A standalone eviction policy needs to be modified to additionally provide ranks, in order to be incorporated as part of HyVE framework. The required ranking extensions are generated by simply leveraging existing logic of the standalone eviction policy. Most standalone eviction policies loop over the meta-data of all eviction candidates to determine the eviction victim based on a given eviction criterion. Upon finding an eviction victim, the knowledge gained over the remaining eviction candidates are discarded. With HyVE, this additional knowledge is retained and processed to generate the required ranks. For some standalone eviction policies (LRU, MRU, etc.), ranks can be generated directly from its meta-data (age), whereas for other standalone eviction policies (LFU, Re-Reference Interval Prediction (RRIP), etc.) the meta-data requires sorting before the ranks can be generated. Section 4.3.1 describes this in detail for the explored standalone eviction policies. The small yet effective ranking-extension modification allows HyVE to include virtually any standalone eviction policy.

4.2.2 Modular and Flexible Framework

The question “which, and how many standalone eviction policy should be incorporated within the HyVE framework?” can only be answered depending on the configured system architecture and class of applications. Therefore, HyVE is designed as a modular framework, capable of incorporating almost any standalone eviction policy. Furthermore, HyVE allows its constituent eviction policies to operate simultaneously in parallel. Each policy generates ranks independently which are fed into the voting system for evaluation. This modularity allows several standalone eviction policies to be accommodated within the HyVE framework.

Modern applications may contain varying phases, each exhibiting different memory access patterns. A fixed combination of standalone eviction policies making up HyVE might not be able to deal with such erratic memory access patterns. Therefore, HyVE is designed as a flexible framework, wherein, it can be configured as a subset, or even reduced to one of its constituent eviction policies at run-time. Similarly, the voting methodology is also made flexible, i.e., HyVE can be configured to switch to a different voting procedure at run-time. This flexibility allows HyVE to offer different optimization attributes, potentially covering a wider range of application characteristics.

Figure 4.3 illustrates the basic operation of standalone eviction policies and how they are incorporated as part of the HyVE framework. When used independently, every eviction policy selects an eviction candidate based on its respective evaluation metric. Within HyVE, every eviction policy is considered as a voter, whose task is to indicate their preference over the memory blocks (candidates), using an array of ranks. Finally, the rank arrays from multiple eviction policies are evaluated using a voting system to decide on an eviction victim. HyVE is made flexible by incorporating two additional sub-modules namely, an *eviction policy filter* and a *voting selector*. These sub-modules

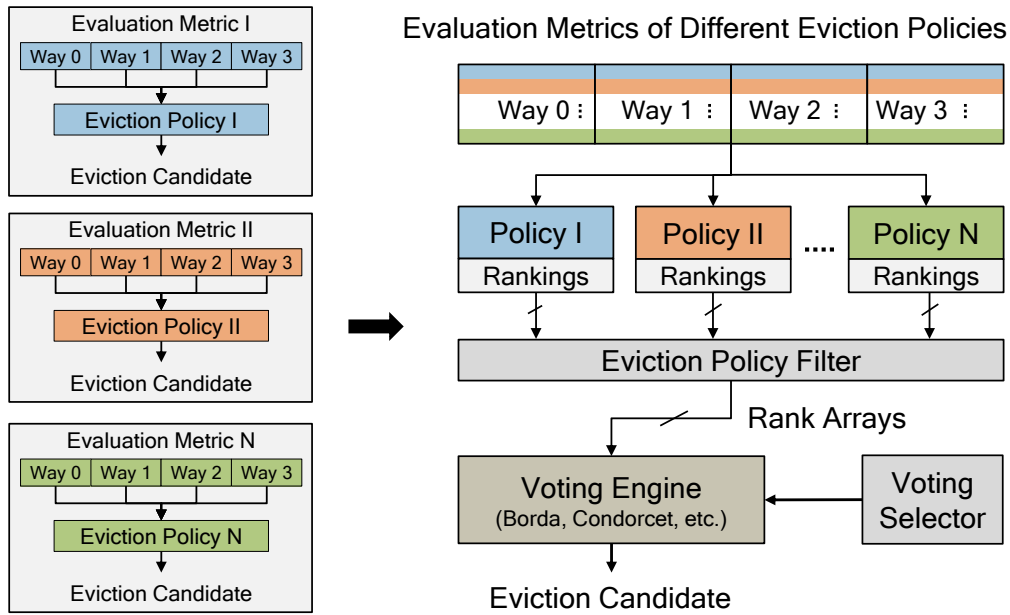


Figure 4.3: An example illustrating how standalone eviction policies are incorporated within the HyVE framework using a 4 way cache structure

are configurable, i.e., they enable HyVE to allow/block a subset of constituent eviction policies and/or switch the voting procedures at run-time.

4.2.3 Tie Handling

As HyVE adopts a voting system, ties can occur. When the Borda Count voting procedure is used, all ties are resolved by evicting one of the tied candidates randomly. When using the Condorcet Method voting procedure, ties are handled by falling-back to HyVE with Borda Count. Algorithm 1 describes a pseudo code of HyVE's eviction process using three example standalone eviction policies.

Algorithm 1 HyVE’s eviction process using three constituent eviction policies

```

1: if hit(addr) == true then                                ▷ Check if address is present
2:   Update Policy Attributes
3: else                                                       ▷ Miss Case
4:   if emptyWay then                                       ▷ Found Empty Way
5:     Fill Empty Way
6:     Update Policy Attributes
7:   else                                                       ▷ Eviction Decision Required
8:     ranks.Policy1 = Policy1(addr) } Computed in Parallel
9:     ranks.Policy2 = Policy2(addr) }
10:    ranks.Policy3 = Policy3(addr) }
11:    if HyVE(ranks) ≠ tie then
12:      return victim
13:    else                                                       ▷ Tied Ranks
14:      fallBack(HyVE(ranks))
15:      return victim
16:    end if
17:    Update Policy Attributes
18:  end if
19: end if

```

4.3 Case-Study 1: HyVE for Caches

As a first case-study, HyVE is evaluated for cache eviction policies with the focus on LLCs in a manycore system.

4.3.1 Ingredients for HyVE

Six popular standalone eviction policies are chosen to build and evaluate HyVE, for both the Borda Count and the Condorcet Method voting methodologies. It is important to note that any standalone eviction policy can be incorporated within the HyVE framework, and is not limited to the policies explored in this thesis.

Least Recently Used Policy: LRU is arguably the most common cache replacement policy. This policy stores the relative age information per cache block as meta-data, which is used as an eviction criterion. Upon a set conflict, LRU evicts the oldest cache block, making it a recency friendly eviction policy. Due to its popularity, several variants of the LRU policy have been explored that optimize for area [81, 82, 83] and performance [84, 85]. For the experiments in this thesis, a regular true LRU policy is used, but any LRU variant can be used as part of HyVE.

LRU Insertion Policy: LIP is an insertion-based policy introduced by authors of [4], whose eviction process is similar to that of the LRU policy. The difference is, LIP inserts new/incoming cache blocks into the LRU position, instead of the MRU position. This property makes LIP thrash resistant.

Bimodal Insertion Policy: BIP was also introduced by authors of [4]. BIP is an adaptive eviction policy which combines LRU and LIP. For the most part, BIP follows LIP, but switches to LRU based on a predefined bimodal throttle parameter (ϵ). By choosing an appropriate ϵ value, BIP can adapt to changes in the working-set, while retaining LIP’s thrash resistant property. For the experiments, $\epsilon = 1/32$, as recommended in [4].

Static Re-Reference Interval Prediction Policy: This policy views cache replacement as a RRIP [50] problem. It uses a N -bit RRPV to track the re-reference interval of each cache block. The RRPV value is set to $2^N - 2$ upon insertion and reset upon each re-reference. SRRIP evicts a cache block with a RRPV of $2^N - 1$. If not present, the RRPVs of all cache blocks are incremented till the maximum value is reached. The SRRIP policy is scan resistant, and can be used in two modes - Hit Priority (HP) or Frequency Priority (FP). For the experiments, SRRIP-HP is used as it is shown to outperform SRRIP-FP [50].

Least Frequently Used Policy: LFU evicts a cache block based on a frequency attribute. As meta-data, it stores the the number of times a cache block has been accessed. Upon set conflicts the least frequently used cache block is evicted, making it scan resistant.

First In First Out Policy: The FIFO policy evicts cache blocks in the same order in which they were added. Contrary to the previously mentioned policies, FIFO does not update its state when a cache block is re-accessed. For example, in an N -way cache it takes exactly N -misses to evict a newly inserted cache block. The FIFO eviction policy attempts to discard stale data by retaining each cache block for a maximum of N misses.

Ranking Extensions

In addition to the standalone variant, each of the six eviction policies have been designed as an alternative variant, extended with ranking capabilities to be used within the HyVE framework. For the LRU policy, the age meta-data (A) of each cache block in a N way cache lies in the range $0 \leq A \leq (N - 1)$. Furthermore, the age of each cache block is unique. This simplifies the rank generation process for LRU, as the age information of each cache block can be directly inferred as the rank. The same holds true for FIFO, as its functionality is similar to LRU, except that the meta-data is not updated on a cache-hit.

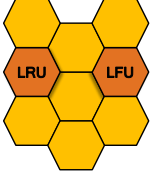
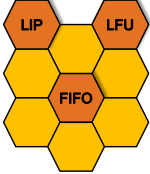
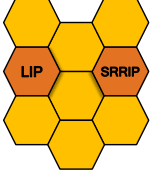
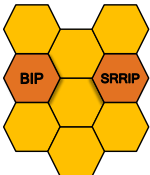
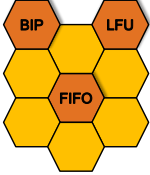
For the remaining standalone eviction policies, the meta-data may be in a format which cannot be directly inferred as ranks. For example, LFU’s meta-data holds the number of times a cache block has been accessed. These numbers do not necessarily adhere to an ordered and unique format like LRU. Therefore, the meta-data of different cache blocks are sorted, before converting them to ranks. For this purpose, a merge sort algorithm is implemented in hardware. The cost and complexity of this implementation is analysed in Section 4.4.8.

4.3.2 Exploring HyVE Flavours

With the described six standalone eviction policies, several HyVE variants/flavours can be produced by combining them in different ways. Determining the HyVE flavour with

4 Hybrid Voting-based Eviction Policy (HyVE)

Table 4.2: List of all explored HyVE flavours categorized into groups of two, three and four

	Flavors of 2	Flavours of 3		
	HyVE-c	LRU, LIP LRU, LFU LRU, SRRIP LRU, FIFO LIP, LFU	LRU, LIP, LFU LRU, LIP, SRRIP LRU, LIP, FIFO LRU, LFU, FIFO LRU, SRRIP, FIFO	
	HyVE-d	LIP, SRRIP LIP, FIFO BIP, LFU	LIP, LFU, FIFO LIP, SRRIP, FIFO BIP, LFU, FIFO	HyVE-a
	HyVE-e	BIP, SRRIP BIP, FIFO LFU, FIFO SRRIP, FIFO	BIP, LFU, FIFO BIP, SRRIP, FIFO	HyVE-b
		Flavours of 4		
		LRU, LIP, LFU, FIFO LRU, LIP, SRRIP, FIFO		

the best performance depends on factors such as the system architecture (cache size, memory hierarchy, etc.) and application characteristics. Therefore, an empirical approach is chosen, where all HyVE flavours are systematically evaluated and analysed. The total number of unique HyVE flavours that can be built using $N = 6$ standalone eviction policies is computed as:

$$\sum_{k=2}^N \binom{N}{k} = 15 + 20 + 15 + 6 + 1 = 57 \quad (4.6)$$

However, the goal of HyVE is not just to combine several standalone eviction policies together, but to do so wherein each policy exhibits orthogonal optimization attributes. Therefore, HyVE flavours containing eviction policies with similar optimization attributes are omitted from evaluation. This step minimizes bias in the voting process, which allows HyVE to consider the opinions of all its constituent eviction policies equally. So, from the 57 possible HyVE flavours, any combination involving the following eviction policies are omitted:

- (LRU, LIP, BIP), as BIP inherently combines LRU and LIP,
- (SRRIP, LFU), as both use frequency as the optimization metric (scan resistant).

These constraints reduce the total number of HyVE flavours to 23. Table 4.2 lists them categorized into groups with two, three and four HyVE flavours.

Table 4.3: Architecture configuration parameters for the sniper simulator

Parameters	Configuration Details
Processing Core	Quad-core operating at 2.66 GHz
Cache Line Size	64 B
Coherence	MESI Protocol
Private L1	32 KiB, 8 ways, 4 cycle data & 1 cycle tag access latency, LRU Policy
Private L2	256 KiB, 8 ways, 8 cycle data & 3 cycle tag access latency, LRU Policy
Shared LLC	{1,2,4,8} MiB, 16 ways, 30 cycle data & 10 cycle tag access latency, HyVE Policy
Main Memory	250 cycle memory access latency

4.4 Experimental Evaluation - HyVE for Caches

4.4.1 Simulation Framework

A manycore architecture simulator is required to evaluate the HyVE concept for LLCs. HyVE’s performance needs to be evaluated and analysed against existing standalone eviction policies for various benchmarks and cache parameters. A consequence of these requirements results in a substantial amount of simulation runs. Cycle-accurate simulators such as Gem5 [76], though accurate are highly time consuming. Simulating a single workload could consume several hours or even days, making it impractical.

The Sniper Multi-core Simulator

The sniper simulator [86] uses interval modelling to sacrifice some accuracy to reduce simulation time. Additionally, it supports multi-threaded execution, which further cuts-down the required simulation time. Therefore, this simulator is a viable option to explore a large number of simulation scenarios with HyVE. The sniper simulator supports three core models, each offering different levels of abstractions and simulation speeds:

- One-IPC (cache-only): Abstract simulation model with the least simulation time.
- Interval Model: Detailed model with average simulation time.
- Instruction-Window Centric Model: Most detailed simulation model consuming the highest simulation time.

Initially, the sniper simulator’s One-IPC model is used to explore different HyVE flavours. After the design-space has been narrowed-down to a few interesting HyVE flavours, the sniper simulator’s Instruction-Window Centric model is used to evaluate HyVE with other state-of-the-art eviction policies.

4.4.2 Experimental Setup

Target Architecture

The Intel Nehalem micro-architecture with four processing cores is used as a baseline system for evaluation. Figure 4.4 and Table 4.3 illustrate the target architecture and its

4 Hybrid Voting-based Eviction Policy (HyVE)

configuration details. The L1 and L2 caches are private to the processing cores and use LRU as the eviction policy. HyVE is deployed for the shared LLC.

For the experiments, 15 workloads from the PARSEC [87] and SPLASH-2 [88] benchmark suites are executed using *simlarge* input-sets. Each benchmark is evenly parallelized onto the four processing cores. By default, the Intel Nehalem micro-architecture uses an 8 MiB LLC. To compensate for using the *simlarge* input-set instead of the *native* input-set, the LLC size is reduced to 4 MiB for all experiments. A cache size sensitivity analysis is performed separately in Section 4.4.4 that evaluates HyVE’s impact on four different LLC sizes (1 MiB, 2 MiB, 4 MiB, 8 MiB). The HyVE flavours are initially evaluated using Borda Count as the voting methodology, followed by comparisons to the Condorcet Method. Lastly, HyVE is compared to other state-of-the-art eviction policies.

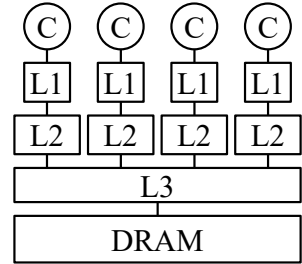


Figure 4.4: Target architecture

Initial Observations

All 23 HyVE flavours are simulated using 15 workloads from the PARSEC and SPLASH-2 benchmark suites. Reporting the results of all 23 flavours is impractical. Therefore, a detailed analysis of the following 5 HyVE flavours are presented in a concise manner:

- HyVE-a: LIP + LFU + FIFO
- HyVE-b: BIP + LFU + FIFO
- HyVE-c: LRU + LFU
- HyVE-d: LIP + SRRIP
- HyVE-e: BIP + SRRIP

The shortlisted variants consist of 2 HyVE flavours built with three standalone eviction policies and 3 HyVE flavours built with two standalone eviction policies. The selected 5 HyVE flavours represent almost all characteristics and properties of HyVE observed with other variants. Explorations with the remaining HyVE flavours are summarized towards the end of the analysis.

4.4.3 Analysing HyVE Flavours

This subsection evaluates HyVE’s impact on the LLC for the 4 MiB cache configuration using Borda Count as the voting methodology. Further, the performance of each HyVE flavour is analysed in detail using evaluation metrics like the eviction count, novelty of HyVE’s eviction decisions and the influence of HyVE’s constituent eviction policies on its eviction decisions. Figures 4.5 to 4.9 these evaluation metrics for HyVE-a to HyVE-e respectively. Each figure contains three types of plots:

- *Cache Misses*: This is a standard metric used to evaluate any cache eviction policy. The figures report the LLC misses for a given HyVE flavour. The LLC misses of its constituent eviction policies are also reported. The LLC misses are normalized with respect to the LRU policy ¹.
- *Eviction Count*: The figures report an eviction count plot for a selected benchmark. This metric helps to continuously visualize the performance of both HyVE and its constituent eviction policies throughout the course of the application. The solid, dashed and dotted lines represent the eviction count of the standalone policies. The outline of the coloured area represents the given HyVE flavour’s eviction count. This coloured area is further broken-down into two categories to analyse the nature of HyVE’s decisions:
 - *Known Eviction Victim*: This eviction decision would have been taken by at least one of HyVE’s constituent eviction policies.
 - *New Eviction Victim*: This eviction decision would have been taken by none of HyVE’s constituent eviction policies.
- *Normalized Opinion*: The figures also report an opinion plot for a selected benchmark. This metric visualizes the influence of HyVE’s constituent eviction policies on the eviction decisions. This is also a continuous plot measured throughout the course of the application. The opinion values are obtained by normalizing the ranks generated by the constituent eviction policies at every eviction decision, i.e., the opinion values always sum up to 1.

HyVE-a Analysis

Figure 4.5 shows the normalized LLC misses for HyVE-a compared to its composing standalone eviction policies (LIP, LFU and FIFO), for all benchmarks. With the exception of *canneal* and *streamcluster*, LIP and LFU greatly increase the cache misses compared to LRU. FIFO performs similar to LRU, improving it by up to 5% for *lu.cont* and *radix*.

Despite two of its constituents exhibiting bad performance, HyVE-a manages to match the performance of the better performing policy. It even outperforms its best constituent policy for the benchmarks *facesim*, *lu.cont*, *lu.ncont*, *radix*, *water.nsq* and *water.sp*. This shows HyVE’s ability to make better eviction decisions by resolving to a consensus among its constituent eviction policies.

Analysing the eviction plot for *facesim*, HyVE-a’s eviction count saturates compared to the standalone eviction policies. Also, a majority of HyVE-a’s decisions are unique. The opinion plot indicates that LFU had a strong opinion on most of HyVE-a’s eviction decisions, followed by FIFO and LIP. However, the eviction plot does not support LFU’s influence, as most of HyVE-a’s eviction decisions are unique. LFU has a high probability that its frequency counters are often equal, and can thereby generate equal ranks. These are falsely translated as strong opinions to evict any of the cache ways, resulting in LFU

¹The y-axis is restricted at 40%

4 Hybrid Voting-based Eviction Policy (HyVE)

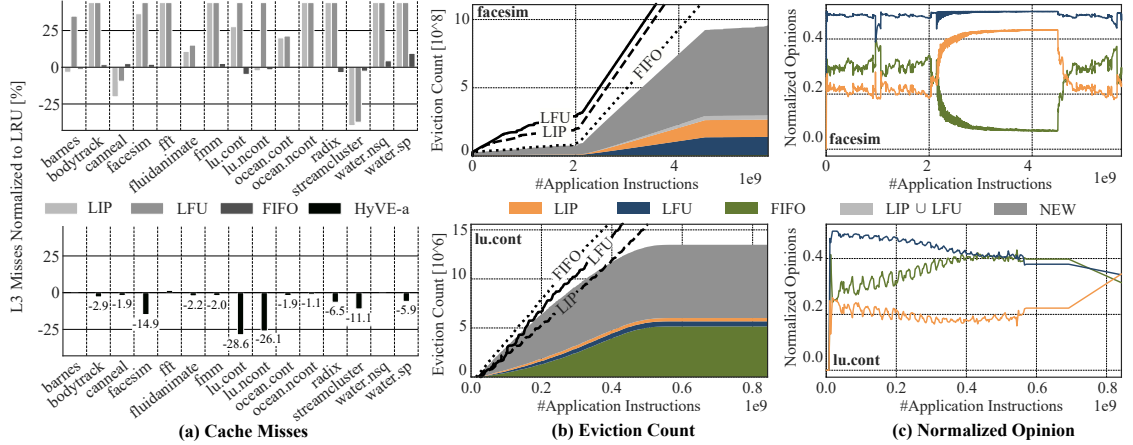


Figure 4.5: Normalized LLC misses for **HyVE-a** and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks

usually dominating the opinion plot. Accounting for this paradox, the major influencers can be interpreted as FIFO and LIP. The strong presence of new eviction decisions indicate that there was no clear winner, i.e., the eviction policies (voters) did not agree with each other in selecting an eviction victim. Therefore HyVE-a made new eviction decisions by resolving to a consensus, which resulted in a 15% reduction in cache misses compared to its best composing standalone eviction policy (FIFO).

For the *lu.cont* benchmark, HyVE-a improves the cache performance compared to its constituent eviction policies. The eviction plot shows that HyVE-a is closer to FIFO (best performer) at the beginning of the benchmark. Towards the end of the benchmark, HyVE-a is closer to LIP (poor performer), but saturates quickly. The opinion plot shows FIFO's growing influence during the initial phases, supporting the eviction plot. Towards the end of the benchmark, HyVE-a is influenced by LIP's opinion. It can be that favouring LIP's decision only towards the end of the benchmark improves HyVE-a's performance. By favouring different eviction policies at different benchmark phases, HyVE-a outperforms its best composing standalone eviction policies (FIFO) by 25%.

HyVE-b Analysis

Figure 4.6 shows the normalized LLC misses for HyVE-b, for all benchmarks. This flavour is similar to HyVE-a, with BIP replacing LIP. BIP improves the cache performance compared to LIP (Figure 4.5) for almost all benchmarks. HyVE-b mostly achieves similar performance to FIFO. It fails to match the benefits of BIP, especially for benchmarks *barnes*, *canneal* and *lu.ncont*. This can be accounted to the bad performance of LFU for the corresponding benchmarks. For the *water* benchmarks, it reduces the cache misses even though all three composing policies show poor performance.

The eviction plot for *water.nsq* shows HyVE-b's eviction count closer to that of FIFO, which is the better performer compared to BIP and LFU. The opinion plot indicates that approximately all three eviction policies equally influence HyVE's decisions. For

this benchmark, HyVE-b’s cache misses are closer to the best composing standalone eviction policy (FIFO), even improving it by $> 5\%$.

For *canneal*, HyVE-b’s performance is closer to FIFO, even though both BIP and LFU show improvements. The eviction plot for *canneal* shows HyVE-b’s eviction count following FIFO which is a poor performer for this benchmark. Neglecting the LFU paradox, the opinion plot shows that FIFO’s opinion is influencing HyVE-b’s decisions, which explains HyVE-b’s poor performance for this benchmark. This supports the statement that there is no “magic recipe” for HyVE, and its benefits vary with the application characteristics, as with any eviction policy.

Compared to HyVE-a, HyVE-b replaces LIP with BIP which is a better performer. However, HyVE-a shows better performance than HyVE-b (except for the *water* benchmarks). This shows that a poor performing standalone eviction policy could potentially have a positive impact when used along other eviction policies within HyVE.

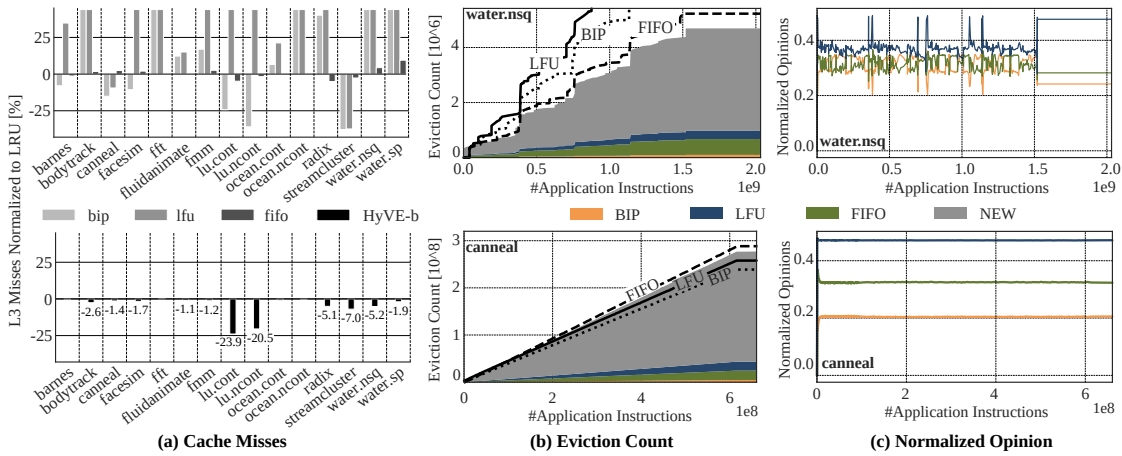


Figure 4.6: Normalized LLC misses for HyVE-b and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks

HyVE-c Analysis

Figure 4.7 shows the cache misses for HyVE-c (LRU and LFU). Except for *canneal* and *streamcluster*, LRU outperforms LFU for all benchmarks. HyVE-c mostly ignores the bad performance of LFU and matches the performance of the LRU policy, slightly improving it for some benchmarks.

For *bodytrack*, HyVE-c has good performance compared to its best composing standalone eviction policy (LRU). The eviction plot shows HyVE’s eviction count following the trend of LRU. The segmented area mostly contain unique evictions and LRU decisions, indicating HyVE-c’s preference. The opinion plot shows that both policies have approximately equal influence on the eviction decisions. Accounting for the LFU paradox, HyVE-c is expected to incline towards LRU. Comparing the cache misses, HyVE-c not only leans towards LRU, but improves it by 9% which can be explained by its new decisions.

4 Hybrid Voting-based Eviction Policy (HyVE)

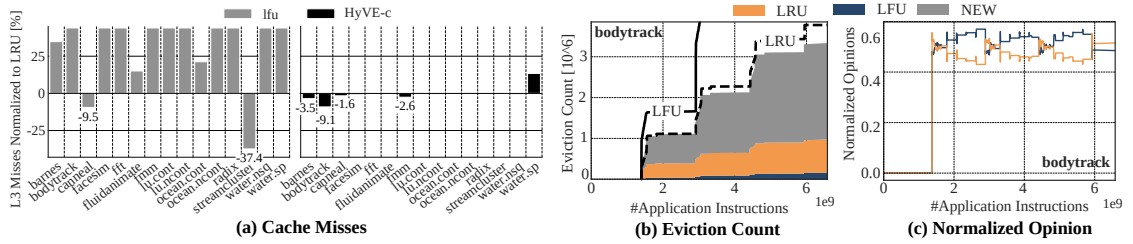


Figure 4.7: Normalized LLC misses for **HyVE-C** and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks

HyVE-d Analysis

Figure 4.8 shows the cache misses for HyVE-d which combines LIP and SRRIP. Overall, the SRRIP policy is a better performer compared to LIP, for all benchmarks. The trend of HyVE-d is similar to SRRIP, even improving upon it for the *barnes*, *cannear*, *facesim* and *lu.ncont* benchmarks. HyVE-d fails to achieve SRRIP’s performance for the benchmarks *fft* and *fmm*.

For the *lu.ncont* benchmark, both LIP and SRRIP exhibit good performance compared to the LRU policy. The eviction plot shows that HyVE-d’s eviction count is below that of both the standalone eviction policies, but with a similar trend. The opinion plot clearly show both policies having equal influence on the eviction decisions. For this benchmark, HyVE-d reduces the cache misses by an additional 13% compared to the best performing standalone SRRIP policy. This example shows HyVE combining two good performing eviction policies to further improve cache performance.

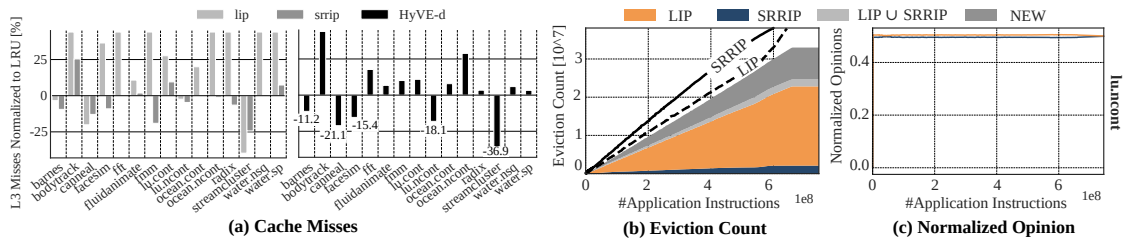


Figure 4.8: Normalized LLC misses for **HyVE-d** and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks

HyVE-e Analysis

Figure 4.9 shows the cache misses for HyVE-e, which is similar to HyVE-d, with BIP replacing LIP. Both BIP and SRRIP are strong policies, reducing the cache misses significantly compared to LRU. HyVE-e matches the performance of the best composing standalone eviction policy for most benchmarks, and even improves upon it for the *barnes* and *lu* benchmarks. There are also cases where HyVE-e follows the lesser performing policy (*bodytrack*).

4.4 Experimental Evaluation - HyVE for Caches

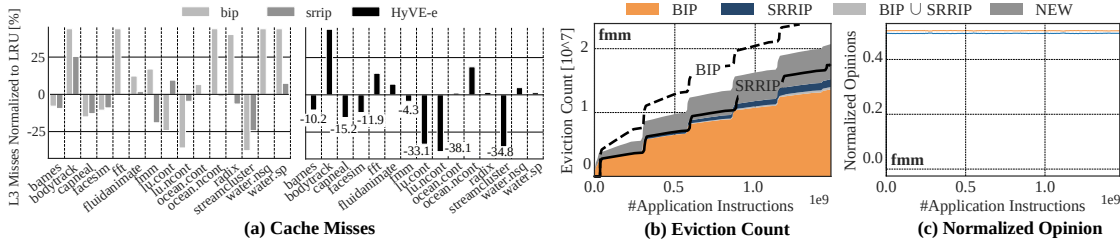


Figure 4.9: Normalized LLC misses for **HyVE-e** and its constituent eviction policies for all benchmarks, an eviction count and opinion analysis plot for selected benchmarks

For the *fmm* benchmark, SRRIP has a better performance than BIP. The eviction plot also shows a similar trend for HyVE-e, as its eviction count lies between SRRIP and BIP. The opinion plot shows that both policies have equal influence, but reassessing the eviction plot, BIP agrees with a large number of HyVE-e’s eviction decisions. This explains HyVE-e’s neutral performance for this benchmark, as it reduces the cache misses by 4% compared to LRU, which is between SRRIP and BIP.

Summary of other HyVE flavours

The LLC performance of the remaining 18 HyVE flavours and the standalone eviction policies are attached in Appendix A. On average, for all 15 benchmarks, 10 flavours improve the LLC performance compared to LRU, 6 flavours match the LLC performance of LRU, and 2 flavours degrade the LLC performance compared to LRU. For the explored benchmarks and HyVE flavours, HyVE-a exhibits the best LLC performance on average, covering a wide range of application characteristics.

4.4.4 Cache Size Sensitivity Analysis

The size of a cache is an important factor that influences the effectiveness of any eviction policy. If a cache is over-dimensioned, i.e., all application data comfortably fit into the cache, the smartness of an eviction policy has minimal impact on the application’s performance. The same holds true if a cache is under-dimensioned, as severe cache-thrashing makes any eviction policy ineffective. Therefore, the HyVE flavours are evaluated for different cache sizes. Figure 4.10 shows the normalized LLC misses for each HyVE flavour for the following cache sizes: 1 MiB, 2 MiB, 4 MiB and 8 MiB. For almost all benchmarks and cache sizes, HyVE-a and HyVE-b reduce the cache misses compared to the LRU policy. With increasing cache size, this performance is either constant or even improves, depending on the benchmark. The variant HyVE-c sometimes improves the cache performance, but mostly shows no changes with LLC size variations. Depending on the benchmark, HyVE-d and HyVE-e exhibit both good and bad cache performance compared to the LRU policy. Overall, the cache size sensitivity analysis shows that HyVE generally has a positive impact for different cache sizes. This shows that HyVE is applicable to different LLC configurations.

4 Hybrid Voting-based Eviction Policy (HyVE)

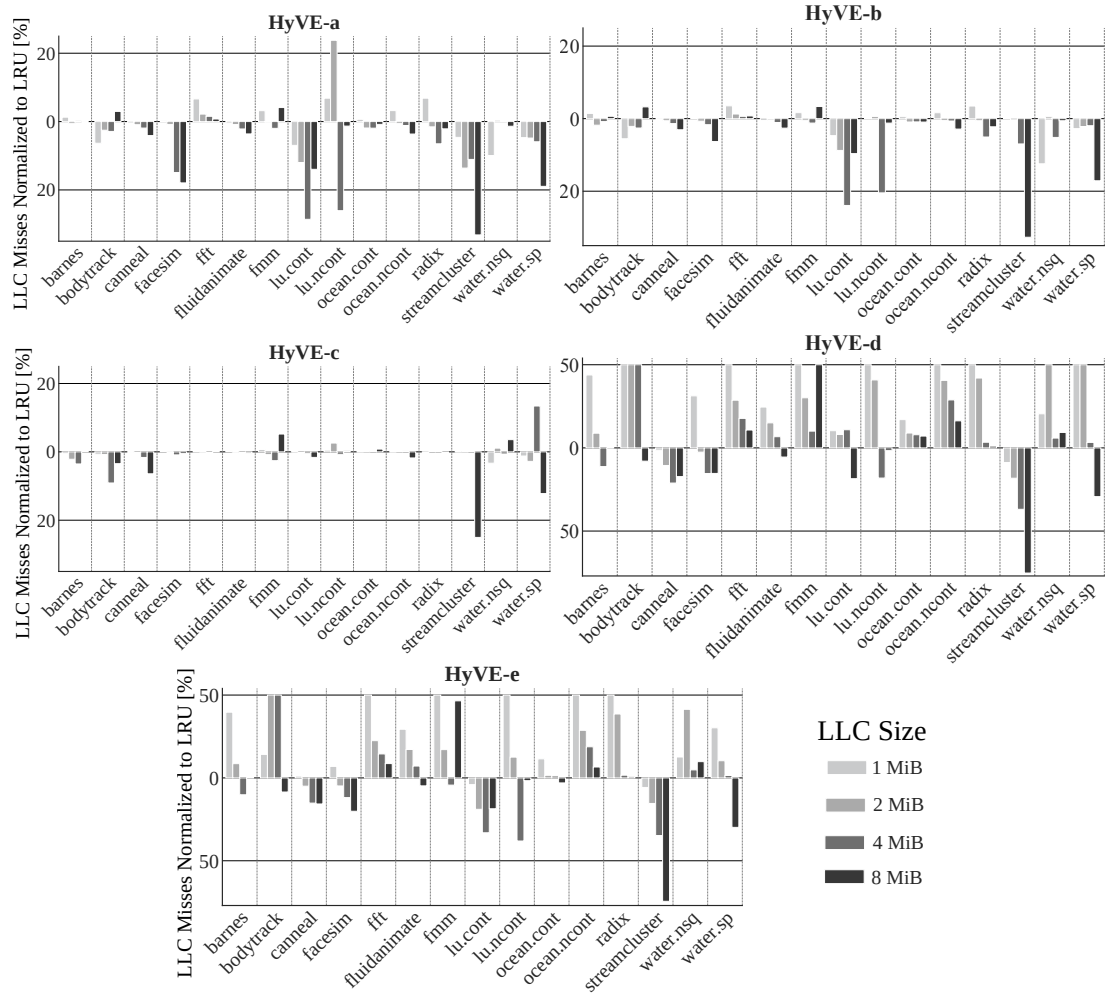


Figure 4.10: Normalized LLC misses with different cache sizes for all HyVE flavours

4.4.5 Voting Methodology Analysis - Borda Count vs Condorcet Method

HyVE’s voting algorithm plays an important role in determining the eviction victim [22]. Therefore, HyVE is re-evaluated with a second voting procedure for all benchmarks. For all 5 HyVE flavours, the voting methodology is switched from Borda Count to the Condorcet Method. Figure 4.11 reports the LLC performance of all flavours for all benchmarks. The LLC misses of each HyVE flavour are normalized to that of their Borda Count counterparts. For almost all benchmarks, HyVE with Borda Count outperforms HyVE with the Condorcet Method. The exceptions are *barnes*, *canneal*, *lu.cont* and *streamcluster*. This is an interesting result, especially from the voting theory perspective which considers the Condorcet Method “fairer” than the Borda Count. Although in theory, the Condorcet Method satisfies the Condorcet criterion, the results for cache evictions are against it. This can be explained due to the small number of voters which could easily bias HyVE with the Condorcet Method. This effect in particular can be

4.4 Experimental Evaluation - HyVE for Caches

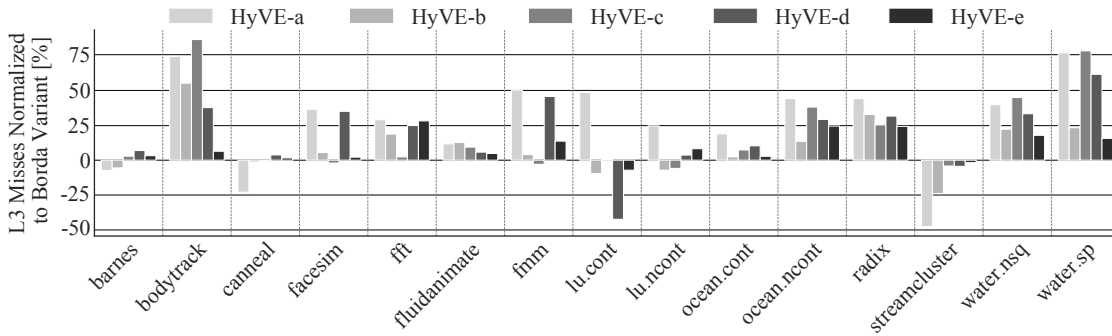


Figure 4.11: Normalized LLC misses of all HyVE flavours using the Condorcet Method

observed when three standalone eviction policies are involved, where two perform poorly. With the Condorcet Method’s pairwise comparisons, the two poorly performing voters have a greater influence on the decision process by taking advantage of two-against-one situation. The positives and negatives of using HyVE with the Condorcet Method is analysed in detail for two HyVE flavours and selected benchmarks.

HyVE-a-Condorcet. Figure 4.12 reports the eviction count plot of HyVE-a for four representative benchmarks, both with Borda Count and the Condorcet Method voting procedures. The plots show that HyVE-a with the Condorcet Method mostly follows the joint decision of LIP and LFU, whereas HyVE-a with the Borda Count selects new eviction candidates more often. This two-against-one situation is advantageous for the *canneal* and *streamcluster* benchmarks as LIP and LFU have a better performance than FIFO. However, the LLC performance suffers for *bodytrack* and *water.sp*, as FIFO outperforms LIP and LFU.

HyVE-d-Condorcet. Figure 4.12 also reports the eviction count plot of HyVE-d for four example benchmarks, both for Borda Count and the Condorcet Method. It is important to note that HyVE-d contains two voters. HyVE-d with the Condorcet Method only selects new eviction victims for the *lu.cont* benchmark, whereas for the others, it follows the trend of LIP. This can be due to the decisions of LIP being ranked relatively higher for SRRIP, therefore often winning the pairwise comparisons. In the case of *lu.cont*, new eviction victims are chosen most of the time allowing HyVE-d with the Condorcet Method to perform better than its constituent eviction policies.

The analysis of these benchmarks show that HyVE with the Condorcet Method can yet be used to obtain positive results, under certain conditions:

- If the rankings allow selecting new eviction victims,
- If the best constituent eviction policy exhibits good performance and is not hindered by the two-against-one situation.

4 Hybrid Voting-based Eviction Policy (HyVE)

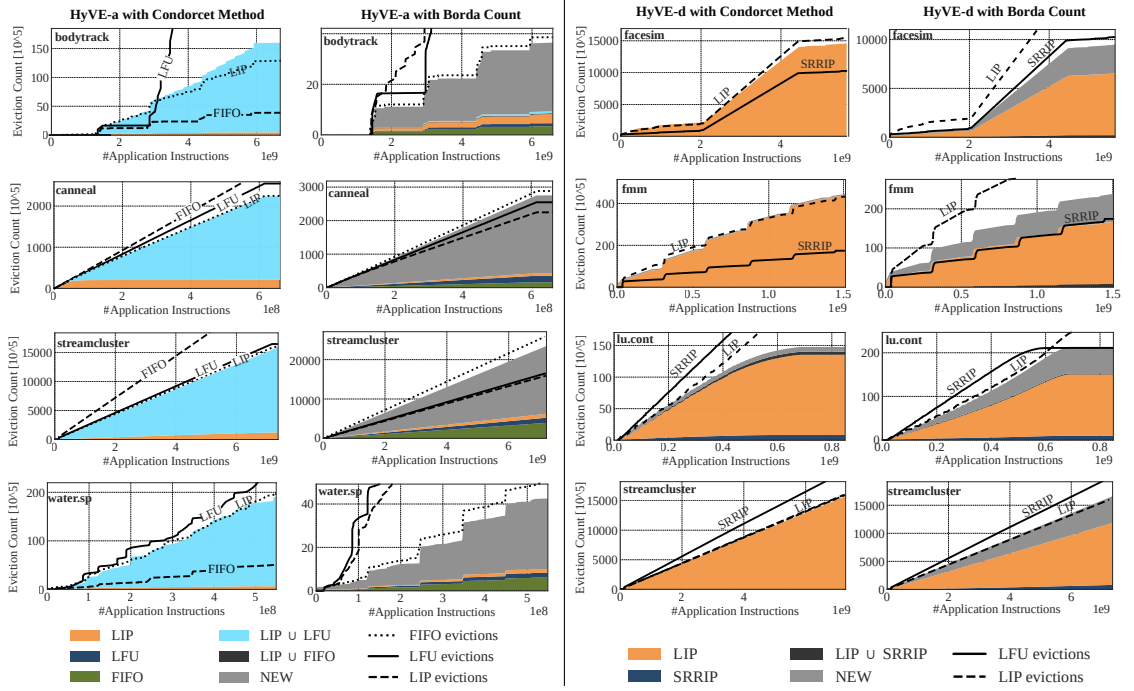


Figure 4.12: Eviction count plots for HyVE-a and HyVE-d using the Condorcet Method and Borda Count voting methodologies, for four representative benchmarks each

These observations and analyses underline the fact that the outcome of a voting system is only as good as its voters. When using the Condorcet Method, the properties of the standalone eviction policies must be carefully analysed, especially regarding the target applications. The Borda Count on the other hand seems to be more forgiving with its point-based system, compensating for majority-against-minority situations. From voting theory’s perspective, a more democratic system like the Condorcet Method is favoured for electoral systems. However, for the cache eviction task, the same voting system is counterproductive due to the small number of voters. To conclude, for the evaluated benchmarks and architecture configuration, the Condorcet Method does not yield much benefits over Borda Count.

4.4.6 Comparison to State-of-the-art Policies

HyVE is compared to two state-of-the-art cache replacement policies, one which uses the concept of SD, and another which uses a learning subsystem.

Set-Dueling (SD)

SD is the closest alternative to HyVE. It combines two standalone eviction policies using the concept of DSS [53]. HyVE’s performance for LLCs is compared against the DRRIP policy [50]. DRRIP can be viewed as a combination of SRRIP and BRRIP

4.4 Experimental Evaluation - HyVE for Caches

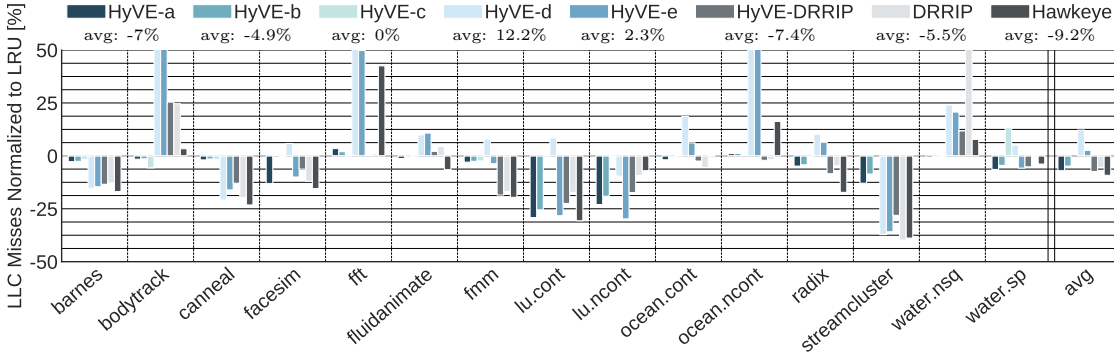


Figure 4.13: The normalized LLC misses of all benchmarks using the HyVE flavours, DRRIP and Hawkeye

using SD. For the experiments, SD has been implemented using a 10-bit PSEL counter and 32 leader sets per eviction policy [4]. For the BRRIP implementation, the bimodal throttle parameter $\epsilon = 1/32$ [50]. For a fair comparison to DRRIP, a new HyVE flavour namely HyVE-DRRIP is introduced. HyVE-DRRIP combines the SRRIP and BRRIP standalone eviction policies together using Borda Count as the voting methodology.

Hawkeye

Hawkeye [60] is a learning-based eviction policy that re-constructs Bélády’s optimal algorithm and predicts if an incoming memory block is cache-friendly or cache-averse. For the experiments, the Hawkeye implementation has been ported from the cache replacement championship simulator [89] to the sniper simulator. Hawkeye requires the program counter to perform predictions. Therefore, for these experiments, the sniper simulator’s mode is switched from One-Instructions-per-Cycle (IPC) (cache-only) to the Instruction-Window Centric model.

To compensate for the sniper simulator’s run-to-run variations, the simulations were repeated seven times. Therefore, the results presented in this section are an average of these seven simulation runs. Figure 4.13 compares the normalized LLC misses for the HyVE flavours, DRRIP and Hawkeye for all benchmarks ². Slight discrepancies between results of Figure 4.13 and Figures 4.5 to 4.9 are due to the switch in the sniper simulator’s core model. For the benchmarks *bodytrack*, *lu.ncont*, *water.nsq* and *water.sp*, there is at least one HyVE flavour exhibiting better LLC performance compared to DRRIP and/or Hawkeye. For the benchmarks *barnes*, *canneal*, *facesim*, *fluidanimate*, *ocean.cont*, *radix* and *streamcluster*, at least one of DRRIP and/or Hawkeye outperform HyVE. For the benchmarks *fft*, *fmm*, *lu.cont* and *ocean.ncont* both HyVE and DRRIP/Hawkeye exhibit the same LLC performance. On average, Hawkeye reduces the LLC misses by 9.2%, DRRIP by 5.5% and HyVE by 7.4%, compared to the LRU policy.

²Note that the y-axis is restricted to 50%

4 Hybrid Voting-based Eviction Policy (HyVE)

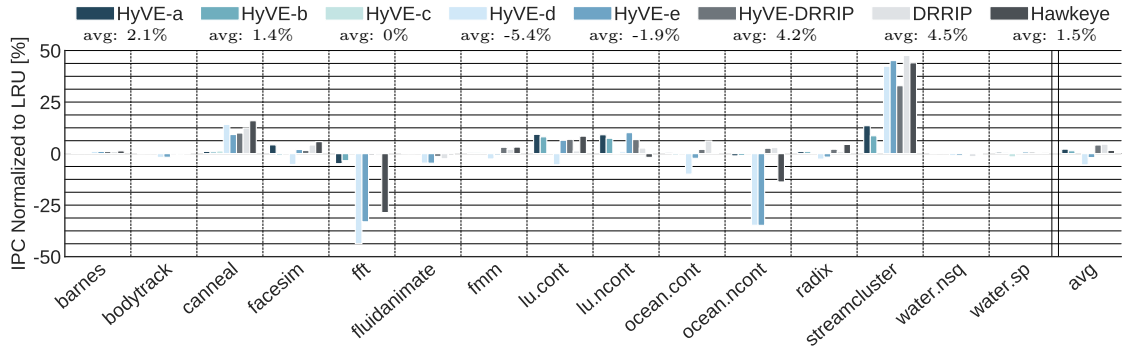


Figure 4.14: The normalized IPC of all benchmarks using the HyVE flavours, DRRIP and Hawkeye

Figure 4.14 reports the normalized IPC for the HyVE flavours, DRRIP and Hawkeye for all benchmarks². On average, DRRIP exhibits the best IPC, improving upon the LRU policy by 4.5%, followed by HyVE at 4.1% and Hawkeye at 1.5%. However, there are benchmarks where each replacement policy has a unique advantage over the other. HyVE introduces a new approach to tackle the cache replacement problem, borrowing ideas from the voting theory domain. Therefore, the concepts of SD, learning-based or HyVE can be viewed as unique and complementary to each other in solving the cache replacement problem, each with its advantages and drawbacks.

4.4.7 Take-away Points

Overall, the results show that HyVE improves the LLC performance for several applications compared to its constituent eviction policies. This can be credited to its design which combines several optimization criteria, and its ability to select new and unique eviction candidates. However, the results also underline that a single HyVE flavour does not, and even cannot guarantee the best performance for all applications. This statement is true even for existing standalone and hybrid eviction policies. Furthermore, deciding which, and how many standalone eviction policies to incorporate within the HyVE framework depends on various factors such as:

- The manycore architecture, memory hierarchy and cache configuration,
- The class of applications to be executed on the manycore architecture,
- The acceptable hardware footprint.

The HyVE design includes an eviction policy filter and a voting selector module. These modules can be configured/adapted at run-time depending on the factors mentioned above, to further enhance HyVE's performance. This is a research challenge in itself, and is discussed as part of the future work in Chapter 5.

4.4.8 Hardware Implementation

HyVE’s performance benefits as a cache eviction policy were evaluated using a simulation platform. However, if its hardware implementation costs are too high, it may outweigh the aforementioned performance benefits. Therefore, the HyVE framework has been realized on a Virtex-7 FPGA prototyping platform. HyVE is implemented with Borda Count as the voting methodology, as it showed better performance than the Condorcet method. HyVE is integrated as part of a light-weight cache controller sub-module, which is responsible for LLC eviction decisions. An example 4-way cache is used to analyse the overheads of the HyVE framework.

FPGA Resource Utilization

Table 4.4 reports the FPGA resources in terms of LUTs and REGs for the six standalone eviction policies and five flavours of HyVE. Additionally, each eviction policy’s area footprint relative to the cache controller sub-module is reported in parentheses. To further investigate the contributors to HyVE’s resource utilization, it is broken down as a function of its constituent eviction policies, their ranking extensions and the voting sub-module. Figure 4.15 illustrates this for the five HyVE flavours. The break-down shows that HyVE’s constituent eviction policies are the major contributors to its area footprint. HyVE’s ranking extensions do not add significant overheads, mainly due to two reasons:

- The ranking extensions leverage existing logic from the standalone eviction policies,
- An efficient implementation of the hardware *mergeSort()* algorithm.

The voting sub-module mostly contributes constant overheads for a given number of voters. Its area footprint is not dependent on the complexity of the standalone eviction policies.

Computational Complexity and Timing

Table 4.4 also reports the logic delay for the six standalone eviction policies and five HyVE flavours. HyVE’s logic delay numbers are very close to that of the standalone eviction policies. This is due to a combination of the HyVE concept and its efficient realization. HyVE’s operation can be broadly classified into two stages namely a rank generation phase and a rank evaluation phase. In the rank generation phase, the constituent eviction policies provide a rank for each cache way. For certain standalone eviction policies, the meta-data needs to be sorted first before a rank can be assigned. For this purpose, a hardware *mergeSort()* algorithm is used. The *mergeSort()* is implemented in an area efficient manner and scales with a complexity of $\mathcal{O}(\#ways \cdot \log\{\#ways\})$. Post sorting, each cache way is assigned a rank. It is important to note that all constituent eviction policies within the HyVE framework operate in parallel during the rank generation phase. In the rank evaluation phase, the generated ranks are fed into

4 Hybrid Voting-based Eviction Policy (HyVE)

Table 4.4: FPGA resource utilization (LUT, REG) and logic delay for all standalone eviction policies and HyVE flavours using Borda Count

Eviction Policies	LUT	REG	Logic Delay
LRU	22 (3%)	34 (7%)	1.169 ns
LIP	26 (4%)	34 (7%)	1.267 ns
BIP	29 (4%)	40 (8%)	1.199 ns
LFU	43 (6%)	50 (10%)	1.108 ns
FIFO	22 (3%)	34 (7%)	1.110 ns
SRRIP	41 (6%)	50 (10%)	1.110 ns
HyVE-a	158 (19%)	150 (24%)	1.130 ns
HyVE-b	162 (20%)	156 (24%)	1.147 ns
HyVE-c	117 (15%)	106 (18%)	1.227 ns
HyVE-d	133 (17%)	106 (18%)	1.192 ns
HyVE-e	136 (17%)	112 (19%)	1.147 ns

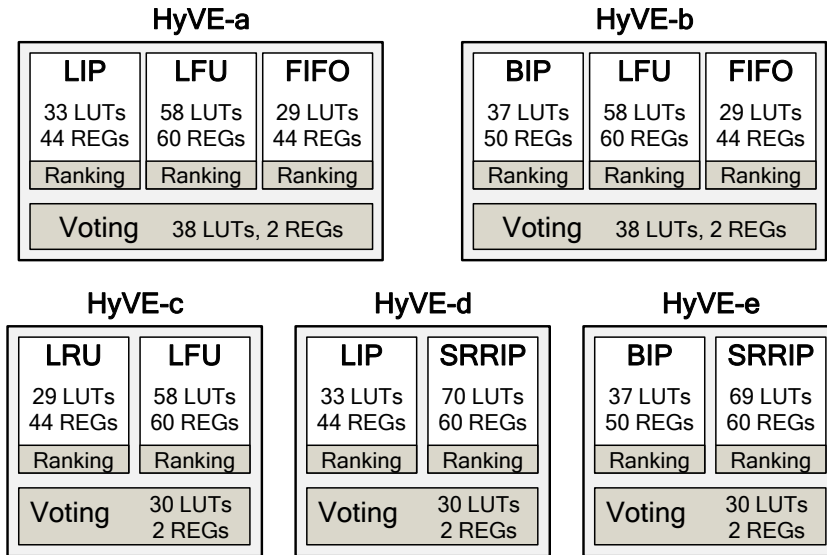


Figure 4.15: Breaking-down the FPGA resource utilization for each HyVE flavour

the voting sub-module, which selects the eviction victim. In order to maximize throughput, the rank generation and evaluation phases are pipelined. The efficient realization of the HyVE concept allows designers to build HyVE with almost any standalone eviction policy without much concern on the area and timing overheads.

4.5 Case-Study 2: HyVE for Sparse Directories

The HyVE concept is evaluated for a second use-case in the context of sparse directory eviction decisions. Commonly, sparse directories are used to support inter-tile coherence for DSM-based tiled manycore systems. From a structural perspective, a cache memory and a sparse directory are very similar. The basic functionality, which is to store and retain important data for faster re-accesses applies for both hardware structures. However, there exist several differences, which are summarized in Table 4.5. For cache memories, the goal of eviction policies is to closely follow the Bélády optimum solution [49]. This unattainable eviction algorithm provides the ideal solution for the cache eviction problem by assuming knowledge of all future memory accesses. However, for sparse directory structures, closeness to the Bélády optimum solution may not necessarily be the decisive optimization factor.

From a storage perspective, a cache holds application data that are fetched from the main memory. A sparse directory holds the sharer information for these application data. The sharer information (represented as bit-vectors) are generated internally upon the initial remote read request of an application data. This property decouples application data access statistics like recency, frequency, etc. from the sparse directory, rendering them pseudo-accurate. For example, in a cache, the LRU policy records the age meta-data for each cache way. When a particular cache block is accessed, the age of the respective cache way is updated to make it the youngest cache entry. Using the same example for a sparse directory, the LRU policy may not produce the expected results. A remote read request would only update the age meta-data in the sparse directory on a cache miss, or when it receives an invalidation message. Conveying the exact age meta-data on every cache access would drastically increase the system traffic. Therefore, even though the memory block may have been accessed several times in the remote data cache, the data usage pattern remains unknown to the sparse directory. This alters the intended functionality of the LRU policy, as the age meta-data in the sparse directory is not accurately represented.

The content within a cache memory and a sparse directory also play a major role at the time of an eviction. For a cache, the modified application data are written back to the main memory. The cost of this eviction process depends on various factors such as the distance to the main memory, the network traffic, etc. For a sparse directory,

Table 4.5: Fundamental differences between a data cache and a sparse directory in the context of eviction decisions

Properties	Data Cache	Sparse Directory
Storage Information	Application Data	Sharer Information
Source of Information	Main Memory	Internally Generated
Data Access Statistics	Accurate	Pseudo-accurate
Eviction Action	Write-back/No Action	Send Invalidations
Cost of Eviction	Fixed/Variable	Variable

invalidation messages are sent out when an entry is evicted. The number of invalidations sent out depend on the content of the sharer information. Therefore, the cost of this eviction process depends on factors such as the number of active sharers, their aggregated invalidation distance (network hop count), the network traffic, etc. These differences between a cache memory and sparse directory serve as motivation for alternate eviction strategies that optimize for different eviction criteria.

4.5.1 Architecture-aware Eviction Policies

Sparse directories are used in tiled manycore systems which exhibit NUMA latencies. The NUMA latencies greatly affect the cost of sparse directory evictions. An eviction policy which considers and specifically optimizes for such architectural differences could improve the performance of a sparse directory. This thesis introduces two such architecturally-aware eviction policies that optimize for eviction criteria other than the temporal attribute.

Least Number of Sharers (LNS) Policy

The Least Number of Sharers (LNS) policy is designed on the basis that a sparse directory entry with many sharers is more likely to be re-used. Therefore, during the eviction process, it evicts the entry with the lowest number of sharers. This minimizes the number of invalidations sent out after an eviction, thereby reducing the cost of evictions for the sparse directory.

For the eviction process, the LNS policy does not need any additional meta-data. It directly uses the sharer information (bit-vector) stored within the sparse directory to count the number of sharers. This reduces the area footprint of the LNS policy. Figure 4.16 illustrates an example LNS eviction operation. Using the sharer information, LNS computes the total number of sharers for each candidate within the sparse directory. The candidate with the lowest number of sharers is evicted.

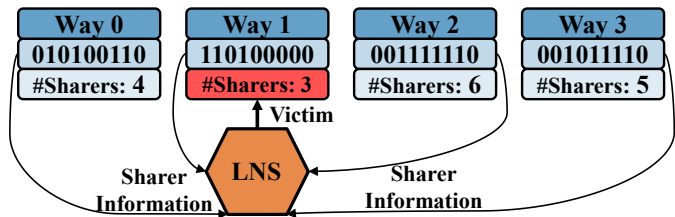


Figure 4.16: An example demonstrating the LNS eviction policy

Shortest Distance First (SDF) Policy

The Shortest Distance First (SDF) policy is designed to reduce coherence-related network traffic on a tiled manycore system. During the eviction process, the sparse directory entry which would induce the shortest number of aggregated network hops for invalidations is evicted.

For the eviction process, the SDF policy populates a small cost table depending on the network topology at run-time. The table contains the number of hops required to

reach every other tile, and is different for each tile in the manycore system. The size of this cost table grows with the size of the manycore architecture.

Figure 4.17 illustrates an example SDF eviction operation. Using the sharer information in conjunction with the cost table, the SDF policy

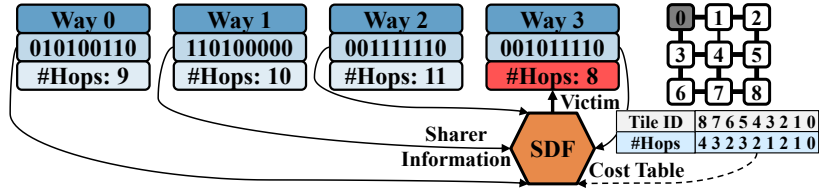


Figure 4.17: An example demonstrating the SDF eviction policy

computes the cumulative hop count for each eviction candidate of the sparse directory. The candidate with the least aggregated hop distance is evicted. For the example in Figure 4.17, the cost table is populated assuming an XY routing algorithm.

4.5.2 Building HyVE for Sparse Directories

To ensure a fair voting system (similar to HyVE for LLCs), it is desirable that HyVE's constituent eviction policies optimize for non-overlapping eviction criteria. For sparse directory eviction decisions, HyVE is built using three standalone eviction policies - the commonly used LRU and the two architectural-aware LNS and SDF eviction policies.

The LRU policy is similar to that described in Section 4.3.1 and evicts old and potentially irrelevant data. Figure 4.18 illustrates an example LRU eviction operation. The efficiency of the LRU policy is determined by its falsely-predicted eviction victims, i.e., the lower number of evicted victims returning to the sparse directory, the better.

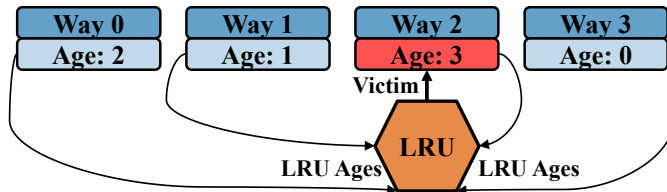


Figure 4.18: An example demonstrating the LRU policy

The goal of the LRU policy is to minimize this *eviction recurrence*. The architecture-aware eviction policies optimize for different criteria. The LNS policy attempts to minimize the total number of *dirInvs*. The SDF policy aims to reduce the network load generated by these *dirInvs*. Each of the three standalone eviction policies optimize for different eviction criteria that are summarized in Table 4.6. The three standalone eviction policies are extended with ranking capabilities and incorporated into the HyVE framework. An example HyVE victim selection decision is illustrated in Figure 4.19.

Table 4.6: Optimization attributes of the constituent eviction policies used for HyVE

Eviction Policy	Optimizing Attribute	Meta-data Storage
LRU	Eviction Recurrence	Age Entry per Memory Block
LNS	Number of <i>dirInvs</i>	None
SDF	Network Load of <i>dirInvs</i>	Cost Table per Tile

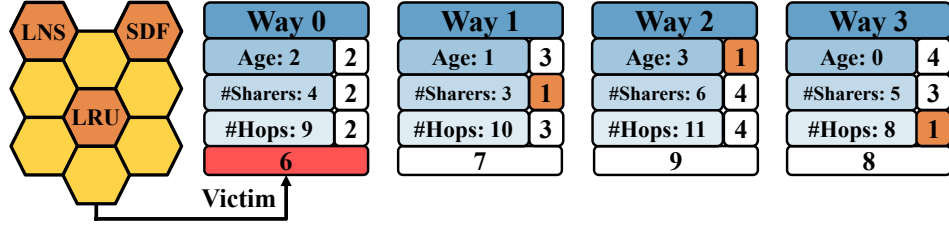


Figure 4.19: An example demonstrating HyVE with LRU, LNS and SDF as the constituent eviction policies

4.6 Experimental Evaluation - HyVE for Sparse Directories

4.6.1 Target Architecture

Sparse directory structures are used to support inter-tile coherence for tile-based many-core architectures. Therefore, HyVE for sparse directories is evaluated using a DSM-based tiled manycore architecture, similar to that used to evaluate the RBCC concept. For these experiments, HyVE has been integrated into the Directory sub-module of the CRM which holds the sharer information.

4.6.2 Experimental Setup

HyVE for sparse directories is initially evaluated using a simulation platform. The high-level SystemC simulation framework introduced in Section 3.5.1 which was used to evaluate RBCC serves as the base platform. HyVE is integrated into the sparse directory controller of the simulator, which is responsible for replacement decisions. A configurable eviction policy parameter is used to switch between either of the standalone eviction policies or HyVE for different simulation runs.

HyVE is evaluated using five workloads that exhibit diverse sharing and communication patterns, taken from the PARSEC and SPLASH-2 benchmark suites - *cannal*, *swaptions*, *fluidanimate*, *fft* and *lucb*. The *simsml* input set is used for all benchmarks. This generates sufficient memory access traces for evaluation, whilst guaranteeing reasonable simulation times. Each benchmark is executed with a parallelism of 16-threads, that spawn four tiles of the manycore architecture.

The focus of these experiments are to evaluate the impact of sparse directory evictions on the applications' execution time. However, the size of the caches also influence the application's execution time as showcased by the HyVE for caches experiments. With under/over-dimensional caches, the application's execution time would be dominated by cache-related properties that outweigh the impact of sparse directories. After extensive explorations, the cache sizes are fixed to lie between the range of under-dimensional and over-dimensional, known as the "knees" of the benchmarks. The L1 cache is set to 256 sets, 4 ways and the L2 cache is set to 16 Ki sets, 4 ways. The cache line size for both caches is 32 B. It is important to note that larger input sets would shift the knees of the benchmarks towards larger cache and sparse directory configurations.

Lastly, the global DRAM memory introduces variables like its access latency, memory bottleneck, etc. which also influence the application’s execution time. As the focus is on sparse directories, the DRAM memory is disabled for these experiments. Instead, the size of the TLM is increased to 16 MiB. This ensures that all application data fits into the distributed TLMs of the manycore system. With these cache and memory parameters, a sweep of different sparse directory configurations ranging from 128 sets, 8 ways to 8 Ki sets, 4 ways are explored. HyVE is evaluated with Borda Count first, followed by the Condorcet Method.

4.6.3 Results and Analysis

Figures 4.20 to 4.24 report the execution time of all benchmarks with standalone eviction policies and HyVE for various sparse directory configurations. The Figures also contain the execution time for three different Set-Dueling (SD) pairs, which is analysed in the coming subsections. All execution times are normalized to that of an ideal non-sparse directory which does not suffer from set-conflicts. Further, each benchmark is analysed in detail using additional evaluation metrics such as:

- *Eviction Count*: This metric reports the total number of sparse directory evictions which is used to evaluate the efficiency of a given eviction policy.
- *Directory-induced Invalidations (dirInvs)*: This metric represents the total number of invalidation messages that were triggered due to sparse directory evictions. The LNS policy attempts to minimize this evaluation metric at each eviction decision.
- *Time for Directory-induced Invalidations (dirInvs)*: This metric represents the total time taken to send out invalidation messages that were triggered due to sparse directory evictions. The SDF policy attempts to minimize this evaluation metric at each eviction decision.
- *Eviction Recurrence*: This metric represents the total number of times a sparse directory entry was re-accessed after being evicted. The LRU policy attempts to minimize this evaluation metric upon each eviction decision.

These supporting evaluation metrics are also reported in Figures 4.20 to 4.24, and are normalized to that of the LRU policy.

Individual Benchmark Analysis

In general, the execution time of all benchmarks increases as the size of the sparse directory decreases. This is an expected result caused by the additional directory-induced invalidations.

canneal: For this benchmark, the LNS and SDF policies perform better than the LRU policy for large sparse directory configurations. HyVE closely tracks both LNS and SDF with a maximum performance degradation of 7.4%. With smaller sparse directory configurations, the LRU policy exhibits better performance than LNS and SDF. As

4 Hybrid Voting-based Eviction Policy (HyVE)

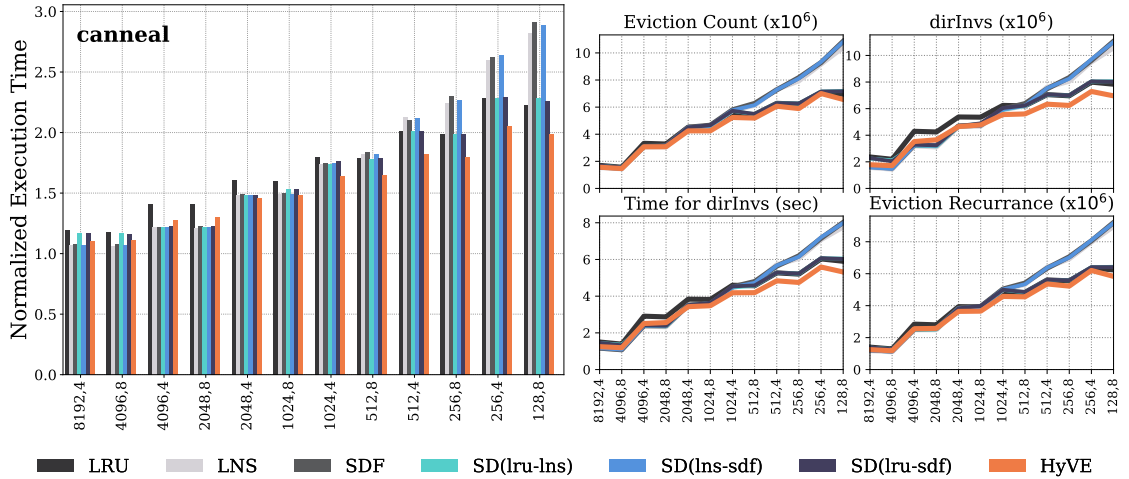


Figure 4.20: The execution time and additional evaluation metrics for the **canneal** benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)

for HyVE, it not only follows the LRU policy, but improves upon its results by further reducing the execution time by up to 11%. Compared to the LNS and SDF polices, HyVE reduces the execution time even up to 30%. This result underlines HyVE’s ability to combine multiple opinions to achieve better performance, even when the LNS and SDF policies have a negative impact when deployed as standalone eviction policies. These results are also supported by the additional evaluation metrics. For example, HyVE has a lower eviction count, generates lesser number of *dirInvs* than LNS, consumes lesser

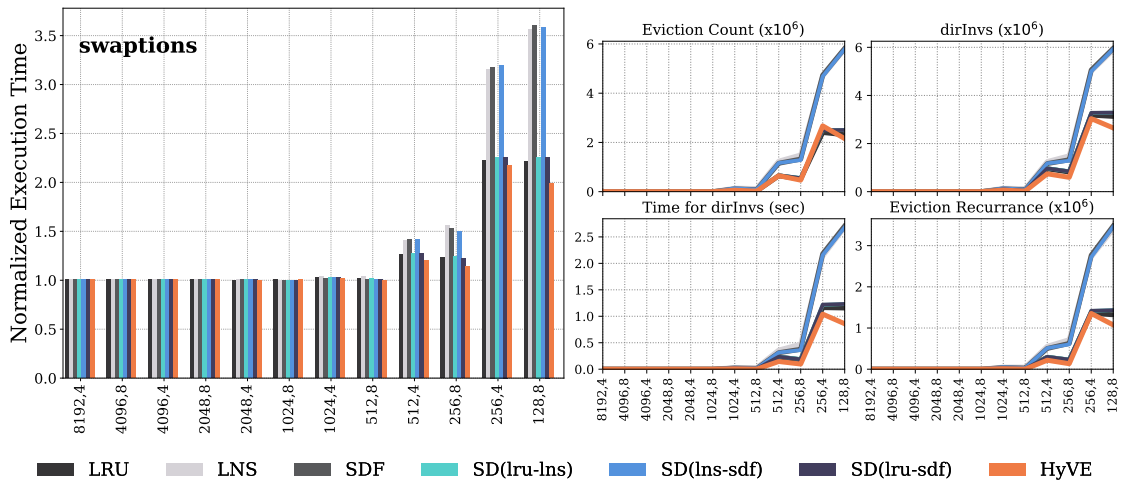


Figure 4.21: The execution time and additional evaluation metrics for the **swaptions** benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)

4.6 Experimental Evaluation - HyVE for Sparse Directories

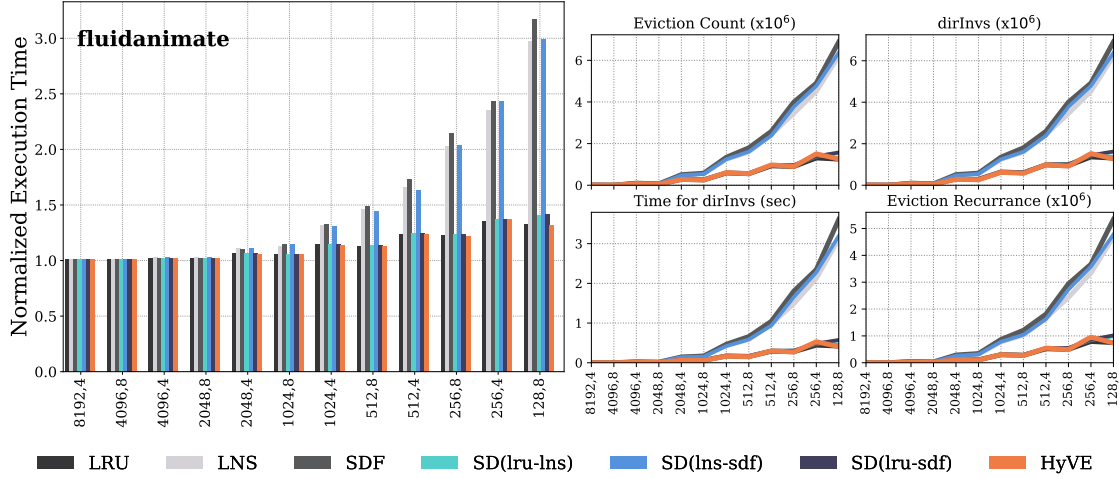


Figure 4.22: The execution time and additional evaluation metrics for the **fluidanimate** benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)

time for the *dirInvs* than SDF, and exhibits a lower recurrence count than LRU. For this benchmark, HyVE optimizes the eviction criteria of the standalone eviction policies better than themselves, making it a unique eviction policy.

swaptions: This benchmark does not induce evictions for large sparse directory configurations. Therefore there exist little differences between the eviction policies. As the size of the sparse directory decreases, the LRU policy starts performing better than LNS and SDF. HyVE tracks the performance of LRU and even improves upon it by up to

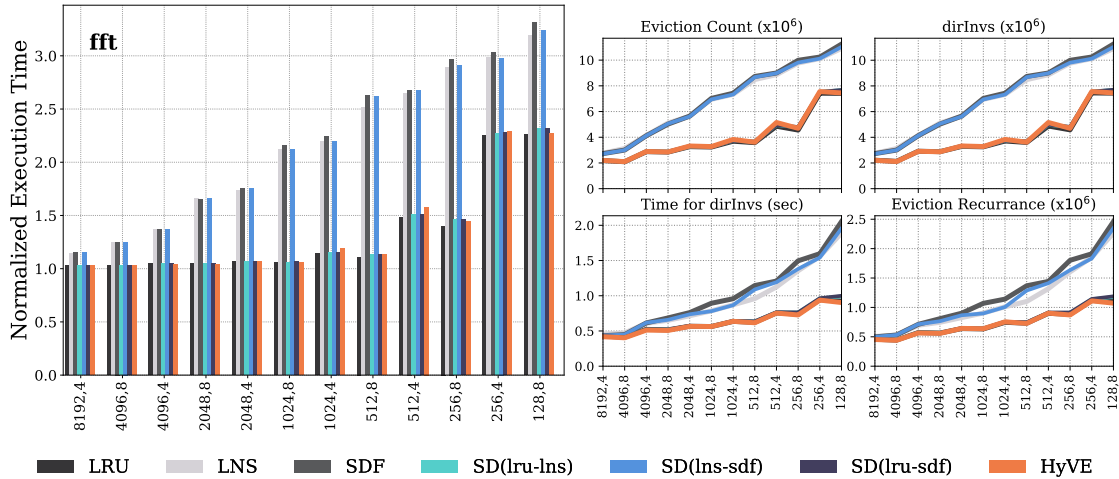


Figure 4.23: The execution time and additional evaluation metrics for the **fft** benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)

4 Hybrid Voting-based Eviction Policy (HyVE)

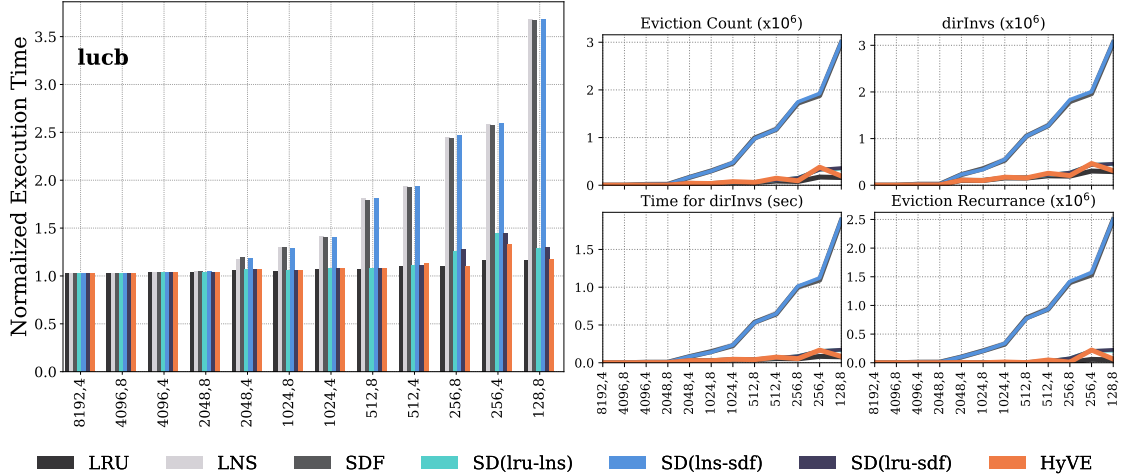


Figure 4.24: The execution time and additional evaluation metrics for the **lucb** benchmark using the standalone eviction policies and HyVE with Borda Count for different sparse directory configurations (sets,ways)

10%. This result again highlights HyVE’s ability to combine several opinions to obtain positive results. The additional evaluation metrics showcase similar trends to that of the execution time, reinforcing the results.

fluidanimate: For this benchmark, the LNS and SDF policies exhibit poor performance compared to the LRU policy. The additional evaluation metrics also support these results. This deems the LNS and SDF eviction policies as unfit for this particular benchmark. As HyVE uses LNS and SDF, its performance is also affected. The results show that HyVE takes the positives of LRU by minimizing the eviction recurrence, but is held back LNS and SDF in terms of the total number and time taken for *dirInvs*. As a result, HyVE mostly exhibits the same performance as the LRU policy for all sparse directory configurations. This result highlights the importance of HyVE’s ingredients.

fft and lucb: For these benchmarks, results show that the LNS and SDF eviction policies are clearly not good candidates. The execution time and additional evaluation metrics are similar to that of the *fluidanimate* benchmark. The major difference is that the number of bad eviction decisions taken by LNS and SDF are much worse compared to *fluidanimate*. Yet, HyVE manages to match the performance of the LRU policy. These results again highlight the importance of consciously selecting the constituent eviction policies to build HyVE.

Victim Distribution Analysis

HyVE’s decisions are further investigated by analysing its victim selection decisions. This metric attempts to quantize which of HyVE’s constituent eviction policies contributed to each of its eviction decisions, similar to the opinion plot of Section 4.4.3. Since there are three voters, every HyVE eviction decision can be categorized as follows:

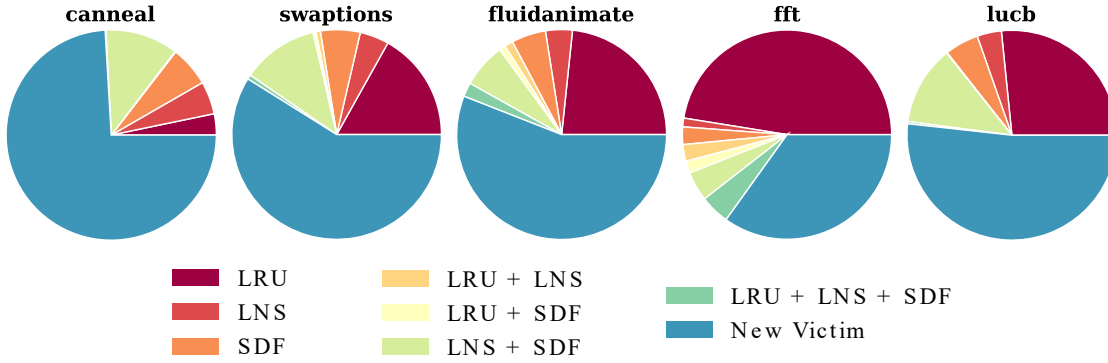


Figure 4.25: Victim distribution analysis of all benchmarks for the sparse directory configuration - 128 sets, 8 ways using Borda Count as the voting procedure

- *Majority Disagree:* A scenario where two voters strongly oppose each other, making the third voter's opinion decide the eviction victim. Depending on the opposing voters, the eviction victim is the same as either the LRU, LNS or SDF constituent eviction policy.
- *Majority Agree:* A scenario where two voters agree on an eviction victim, thereby rendering the third voter's opinion useless. Given that there are voters, the eviction victim is the same as either LRU+LNS, LRU+SDF or LNS+SDF.
- *Unanimously Agree:* A scenario where all voters have the same opinion on the eviction candidates. In this case, the eviction victim is the same as all three constituent eviction policies, i.e., LRU+LNS+SDF.
- *All Disagree/New Victim:* A scenario where all voters disagree, i.e., each of the three constituent eviction policies select a different eviction victim. In such a case, HyVE settles the dispute by resolving to a consensus, thereby selecting a new eviction victim.

To maintain conciseness, the victim distribution analysis results are reported for one of the sparse directory configurations - 128 sets, 8 ways in Figure 4.25. For the *canneal* benchmark, the *New Victim* category dominates the victim selection distribution. This is because, all three voters disagreed among each other at most eviction decisions. Therefore, HyVE took-over the decision-making process by resolving to a consensus among its voters. This idea of a consensus among the constituent eviction policies with different points-of-view can explain why HyVE exhibits better performance than their standalone counterparts.

The *New Victim* category also dominates the victim selection distribution for the *swaptions*, *fluidanimate* and *lucb* benchmarks. However, the remainder of the pie chart reports a large chunk categorized as *Majority Disagree* following the LRU policy. This implies, when not resolving to a consensus among its voters, the eviction decisions for these benchmarks fall under the *Majority Disagree* category, decided by the LRU policy. An exception can be made for the *swaptions* benchmark, which reports an equal chunk

4 Hybrid Voting-based Eviction Policy (HyVE)

of the *Majority Agree* category following LNS+SDF as well. This implies that *swaptions* occasionally followed the eviction decisions of LNS+SDF. This justifies the results for these benchmarks where HyVE mostly matched the performance of the LRU policy, with *swaptions* improving upon it.

For the *fft* benchmark, the *Majority Disagree* category dominates the victim selection distribution. Therefore HyVE’s eviction decisions mostly follow the LRU policy, which is justified by the execution time results for this benchmark.

Condorcet Method Analysis

The benchmarks are also executed with Condorcet Method as the voting methodology. These results and analyses are attached in Appendix B. The take-away point is, HyVE with Borda Count outperforms HyVE with the Condorcet Method. Again, this is an interesting result as the voting theory community consider the Condorcet Method a “fairer” voting methodology than Borda Count. However, when applied for data management in cache-based memories, the idea of consensual decisions provided aggregating different opinions seems to generally provides better outcomes.

Comparison to Set-Dueling (SD)

SD [4] is a state-of-the-art technique used to tackle the eviction problem. The primary idea of SD involves two standalone eviction policies “dueling” against each other, in a cache structure grouped into leader and follower sets. The concept of SD only allows combining two standalone eviction policies. Therefore the three standalone eviction policies used for HyVE are split into three unique pairs - SD(LRU-LNS), SD(LNS-SDF) and SD(LRU-SDF). The leader sets are spread-out uniformly within the sparse directory and the ratio of total sets to leader sets is equal to 16, as recommended in [4].

Figures 4.20 to 4.24 also report the execution time of HyVE with the Borda Count and the three SD pairs for all benchmarks. Results show that, SD usually follows the better performing eviction policy. Internal simulation counters report that the follower sets selected the better performing eviction policy about more than 80% of the time, with the exception of SD(LNS-SDF) where the number was more than 50%. As a consequence, the performance of SD deteriorates if it combines two poorly performing eviction policies. This is clearly seen for SD(LNS-SDF). Conceptually, HyVE differs from SD as it can incorporate more than two eviction policies. This is useful when designing an eviction policy that can optimize for several eviction criteria. This feature coupled with the flexible framework allows HyVE to potentially cover a wide range of application characteristics.

4.6.4 Highlighting HyVE’s Properties

Until now, HyVE was evaluated using standard benchmarks that use real-world memory access patterns. However, these benchmarks may not expose all theoretical properties of the standalone eviction policies or HyVE. Therefore, characteristic micro-benchmarks

were designed to highlight the strengths and weaknesses of the eviction policies, specifically HyVE. The goal of this section is to showcase HyVE’s advertised abilities that may not be observed using standard benchmarks.

Hardware Implementation

The proposed micro-benchmarks were designed to run without a dedicated OS (bare-metal C code). This provided a good opportunity to both verify HyVE for sparse directories on hardware, and to accelerate the experiments. HyVE for sparse directories (LRU, LNS, SDF) has been synthesized on an FPGA prototyping platform, similar to HyVE for LLCs. HyVE is integrated into the sparse directory controller module of a 4x4 tile-based manycore system, similar to that described in Section 3.6. As the simulation experiments showed that Borda Count outperforms the Condorcet Method, HyVE for sparse directories is designed with Borda Count as the voting methodology.

FPGA Resource Utilization. Table 4.7 reports the resource utilization of the sparse directory controller module for a 128 set, 4 way configuration. The number of LUTs and REGs for all three standalone eviction policies and HyVE are reported. As a reference, the eviction policy’s area footprint relative to the sparse directory controller is reported in parentheses. HyVE’s area footprint is greater than the standalone eviction policies, but is still comparable to them. Figure 4.26 breaks down HyVE’s LUT utilization as a function of its constituent eviction policies, the ranking extensions and the voting overheads.

Computational Complexity and Timing. In order to determine the eviction victim, the standalone eviction policies use a $\min()$ function which scales linearly with the complexity of $\mathcal{O}(\#ways)$. HyVE’s rank generation process uses a hardware $\text{mergeSort}()$ algorithm which scales with the complexity of $\mathcal{O}(\#ways \cdot \log\{\#ways\})$. Table 4.7 reports the logic delay numbers. Of the three standalone eviction policies, the SDF policy is the slowest in term of logic delay. Therefore it is pipelined to reduce the maximum delay path. Within the HyVE framework, SDF’s additional logic stage is spread over to the rank evaluation phase. This maintains HyVE’s two stage execution.

Table 4.7: FPGA resource utilization (LUT, REG) and logic delay for all standalone eviction policies and HyVE flavours using Borda Count

Eviction Policy	LUT	REG	Logic Delay
LRU	22 (3.2%)	34 (7.1%)	1.169 ns
LNS	38 (5.2%)	2 (0.4%)	1.130 ns
SDF	155 (18%)	31 (5.6%)	1.052 ns*
HyVE	366 (34.2%)	70 (11%)	1.117 ns*

*Pipelined Implementation: Additional latency of 1 clock cycle

LRU	LNS	SDF
29 LUTs 44 REGs	61 LUTs 12 REGs	238 LUTs 12 REGs
Ranking	Ranking	Ranking
Voting		38 LUTs, 2 REGs

Figure 4.26: HyVE break-down

4.6.4.1 Experimental Setup

One of the challenges when trying to highlight HyVE’s properties was to create a test environment where the memory access patterns of the micro-benchmarks could be carefully controlled. This is a key factor that allows designing specific test scenarios that expose the properties of standalone eviction policies as well as HyVE. Therefore, a test environment was designed with the following parameters:

- *Home Tile*: The tile whose TLM contains all data-sets that will be required by the processing elements of remote tiles during benchmark execution.
- *Team*: A group of tiles that work on the same data-set.
- *Data-set*: A continuous memory address range that is exclusively accessed by a single team during benchmark execution.
- *Frequency*: The rate at which a team requests a data-set entry from the home tile.

Figure 4.27 illustrates the team setup on a 4x4 tiled manycore architecture with three differently sized teams placed at varying aggregate distances from the home tile. Tile 0 is designated as the *Home Tile* to hold all data-sets. This decision allows the micro-benchmarks to use the maximum available hop-distance on the MPSoC. Team-1 consists of a single tile (15), Team-2 consists of two tiles (8, 9) and Team-3 consists of three tiles (1, 4, 5). This specific team groupings are designed intentionally to produce unique characteristics. Team-1 has the maximum aggregate distance from the home tile, but lowest number of sharers. This makes Team-1’s entries most favoured by the SDF policy and least favoured by the LNS policy. Inversely, Team-3 has the minimum aggregate distance, making it disfavoured by the SDF policy. However, it has the highest number of sharers, which makes it favoured by the LNS policy. Team-2’s aggregate distance and number of sharers lie in-between that of Team-1 and Team-3. This make it neutral for both LNS and SDF policies. The LRU policy is not an architecture-aware eviction policy. This makes it independent of architecture parameters like the teams’ size and location. Figure 4.27 summarizes these team specific properties.

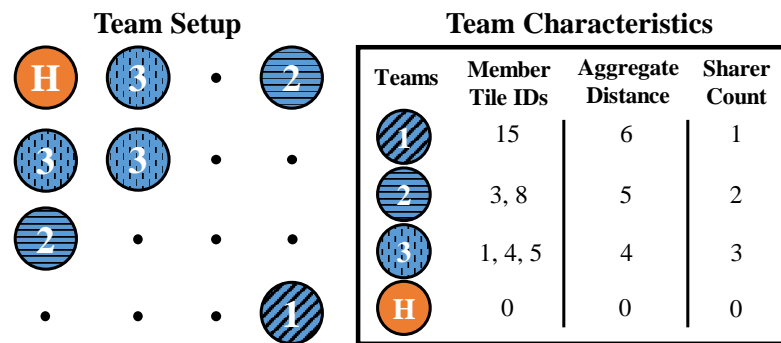


Figure 4.27: The three team configurations and their respective characteristics on a 4x4 tiled manycore architecture

Table 4.8: The test scenarios with combinations of different data-set sizes and their properties

Test Scenario (T3-T2-T1)	Test Property
S-S-L	Favoured by LNS, Disfavoured by SDF
L-S-S	Favoured by SDF, Disfavoured by LNS
L-S-L	Neutral
XL-S-XL	Favoured by LRU
XL-M-XL	Disfavoured by LRU

S = 128 memory blocks; M = 256 memory blocks; L = 384 memory blocks; XL = 512 memory blocks

Every team is allocated a unique data-set in the TLM of the home tile. The memory addresses of these data-sets are chosen such that they induce set-conflicts in the home tile’s sparse directory, thereby triggering evictions. To maintain fair competition between the different teams, the following conditions are enforced:

- The size of a team’s data-set \geq sparse directory set count,
- The total accesses per tile across all teams is the same, i.e., smaller data-sets correlate to higher memory access frequencies and larger data-sets correlate to lower memory access frequencies.

For example, if the size of Team-1’s data-set = (2×Team-2’s data-set) = (3×Team-3’s data-set). Each entry in Team-1’s data-set is accessed exactly once, while entries of Team-2 and Team-3 are accessed twice and thrice respectively. Leveraging this principle, five test scenarios are created using different data-set sizes for each team, which in-turn varies their respective memory access frequencies. Each test scenario is designed with different combinations of team data-set sizes (T3-T2-T1), reported in Table 4.8.

Each test scenario highlights the favouritism exhibited by a given standalone eviction policy, as it only optimizes for one eviction criterion. Table 4.8 also reports each test scenario’s favoured and disfavoured eviction policies. For example, LNS favours a Team-3 entry over other teams. Therefore, reducing Team-3’s data-set size increases its memory access frequency. This would increase the performance of the LNS policy. The same holds true for Team-1 and the SDF policy. The LRU policy favours entries based on their access recency which is correlated to the data-set access frequency. Therefore, LRU does not explicitly favour any team, but rather the most-recently refreshed data-set entries. For better understanding, the pseudo-code of the test scenario is provided in Algorithm 2.

4.6.4.2 Results and Analysis

Each test scenario is executed for all standalone eviction policies and HyVE. Figures 4.28 to 4.31 report the total number of evictions, total number of *dirInvs*, total hop count for *dirInvs*, total execution cycles (with and without BT), the average *dirInvs* per eviction, and the average hop count per *dirInv*.

Algorithm 2 Execution flow of the test scenarios

```

1: Set Team.Dataset.Size for all Teams
2: Set Team.Address.Start for all Teams
3: TotalAccesses =  $\max(\text{Teams.Dataset.Size})$ 
4: while Access < TotalAccesses do
5:   for each Team in Teams do
6:     Team.firstMember.write(Team.Address)
7:     if Team.members > 1 then
8:       for each member in Team do
9:         member.read(Team.Address)
10:      end for
11:    end if
12:    if Team.Address = Team.Dataset.End then
13:      Team.Address = Team.Address.Start
14:    end if
15:  end for
16:  Access = Access + 1
17: end while

```

S-S-L and L-S-S. For both these test scenarios, the access frequency of either the largest team (T3) or the farthest team (T1) is decreased. The effect of these two extreme criteria is reflected in the either performance improvement or performance degradation when using LNS or SDF respectively. For the S-S-L test scenario, Team-3 (highest sharer count) has a higher access frequency than Team-1 (smallest sharer count), which results in LNS making better eviction decisions. This can be observed in the total eviction count and total *dirInv* count plots where LNS is better than SDF. Conversely, for the L-S-S test scenario, Team-1 (farthest and smallest sharer count) has a higher access frequency than

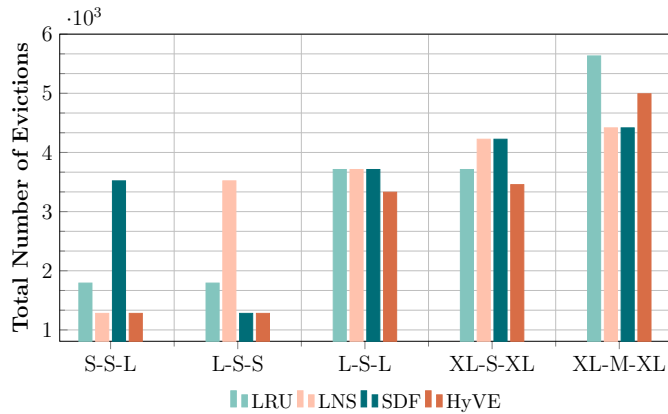


Figure 4.28: The total eviction count for multiple test scenarios using the micro-benchmarks

4.6 Experimental Evaluation - HyVE for Sparse Directories

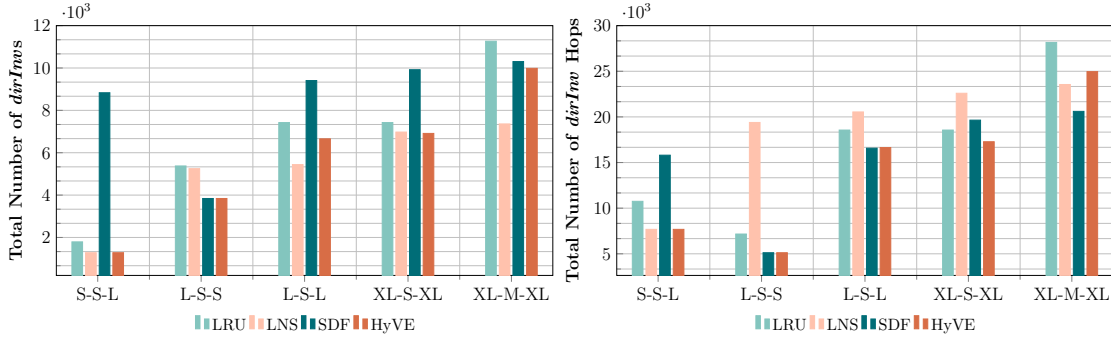


Figure 4.29: The total number of *dirInvs* and the total number of *dirInv* hops for multiple test scenarios using the micro-benchmarks

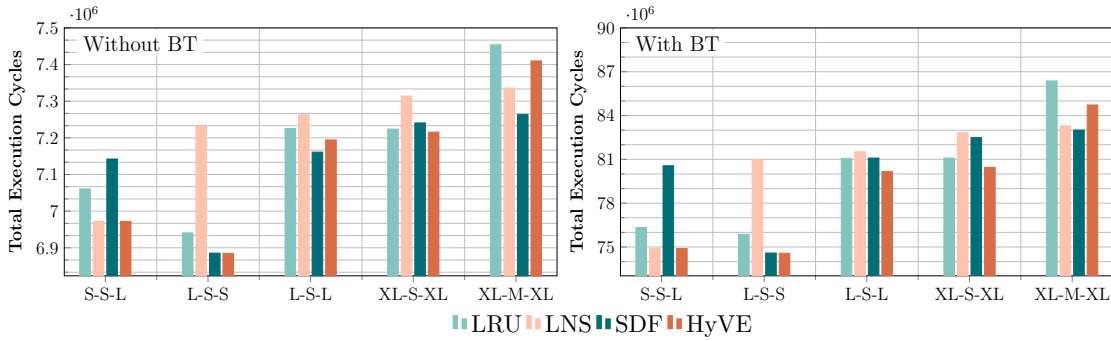


Figure 4.30: The total execution cycles for multiple test scenarios using the micro-benchmarks, with and without the presence of BT

Team-3, which leads to SDF making the best decisions. For both these test scenarios, HyVE closely follows the best decision maker, achieving the best results of either LNS or SDF.

L-S-L. This is a neutral test scenario where the attributes of LNS and SDF are considered equally. Although Team-2 has the highest access frequency, its other attributes (sharer count and distance) are set in-between that of Team-1 and Team-3. As a result, both LNS and SDF exhibit equal performance in terms of the total eviction count. However, by observing the total number of *dirInvs* plot, it can be seen that LNS is indeed sending-out the least number *dirInvs*, correctly optimizing for its eviction criteria. Similarly, it can be seen that SDF is indeed attempting to minimize the network load, by sending-out *dirInvs* that traverse the least number of NoC hops. For this test scenario, HyVE exhibits lower eviction counts than all its constituent eviction policies. Further, in the presence of BT, HyVE is the best performing policy for this test scenario. This highlights HyVE’s ability to incorporate multiple opinions and converge to new eviction decisions, making it a unique eviction policy.

4 Hybrid Voting-based Eviction Policy (HyVE)

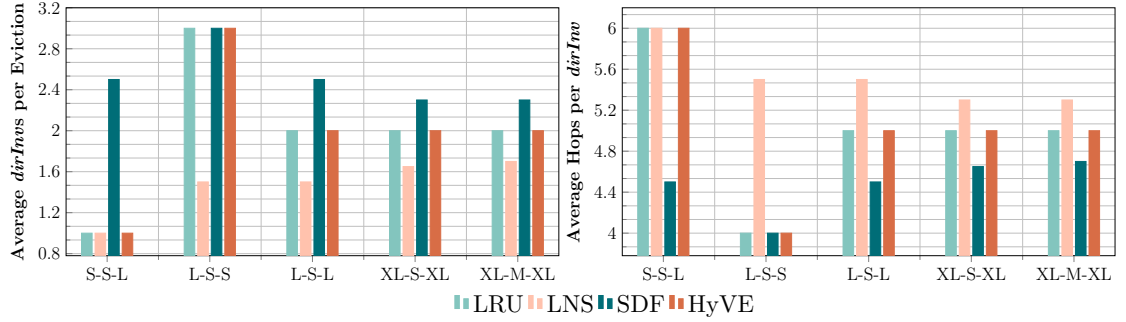


Figure 4.31: The average *dirInvs* per eviction, and the average hops per *dirInv* for multiple test scenarios using the micro-benchmarks

XL-S-XL and XL-M-XL. To devise a test for eviction recurrence, Team-1 and Team-3 are fixed with polarizing distances and sharer counts. This ensures the lowest access frequency compared to Team-2. In the XL-S-XL test scenario, LRU outperforms LNS and SDF due to Team-2’s small data-set size. The data-set entries are accessed frequently, refreshing their age and protecting them from evictions. Scaling-up Team-2’s data-set size in the XL-M-XL test scenario results in loss of age-information. Although the M-sized data-set has a relatively higher access frequency, it is not high enough to refresh the data entry ages at the right time in the sparse directory. For these test scenarios, HyVE has the best performance when the age-information of the high-frequency data-set (XL-S-XL) is updated early enough, even outperforming LRU. However, it suffers from LRU’s poor decisions in the XL-M-XL test scenario. Nevertheless, HyVE still exhibits a slightly better performance than LRU, owing to SDF and LNS making good eviction decisions in the absence of the age-information. These experiments highlight how different standalone eviction policies favour different memory access patterns. The results also show that HyVE can potentially neutralize negative biases and converge to democratic decisions.

5 Conclusion & Outlook

This thesis presented and evaluated two concepts, both optimizing the memory subsystem of modern computing systems. Both contributions are hardware-based solutions that perform architectural and micro-architectural modifications to enhance the performance of large manycore systems.

5.1 Conclusion

Region-based Cache Coherence (RBCC)

The need for hardware-based inter-tile coherence schemes on large tile-based manycore systems have always been debated, as global coherence schemes consume significant hardware area overheads, leading to scalability issues. As a result, alternative programming models or software-based coherence schemes have been proposed for such manycore systems. Moving away from the shared memory programming paradigm increases the programming-effort, and software-based coherence schemes generally do not have the same performance levels of their hardware counterparts. Therefore, this thesis proposed an alternative hardware-based inter-tile coherence methodology, that addresses the scalability challenges of global coherence schemes. The RBCC concept was motivated by the fact that a single application rarely uses all available processing and memory resources of a manycore system. Therefore, RBCC proposed an environment where inter-tile coherence could be established “as-required”, depending on the needs of the application. RBCC confined coherence support to within a subset of tiles (coherence region) of a manycore system. This significantly limited the book-keeping overheads required for inter-tile coherence. For example, RBCC reduced the directory overheads by 73% for a 64-tile manycore system with a maximum coherence region size of 8-tiles. The coherence regions were also designed to be flexible, i.e., they could be created, destroyed, re-sized and even re-located at run-time. RBCC also supported fine-granular sharing capabilities, i.e., inter-tile coherence could be tailored to truly shared application data at run-time. With these features, RBCC managed to create a *coherence-on-demand* environment for users/applications. The RBCC concept was evaluated both on simulation as-well-as an FPGA prototyping platform. Hardware-based inter-tile coherence enabled using RBCC was evaluated against message-passing based scheme using a video streaming application. Results showed that RBCC reduced the execution time by up to 42% compared to the message-passing based scheme. Further, the video streaming application was used to demonstrate the flexibility features of RBCC. The coherence region was expanded and re-located at run-time for two different scenarios, both of which improved the application’s execution time, with negligible re-configuration overheads. Lastly, the concept

of RBCC was used to enable shared memory workloads (SPLASH-2 benchmarks) on the DSM-based tiled manycore system. The performance of RBCC was compared to an alternative software-based VSM approach. Initial results showed that RBCC reduces the execution time by up to 45% compared to the VSM approach.

Hybrid Voting-based Eviction Policy (HyVE)

Cache memories are invaluable resources as they significantly reduce memory access latencies. This makes cache data management an important optimization aspect as it heavily influences the performance of a cache. For relatively simple applications, standalone eviction policy like LRU, LFU, LIP, etc. can maintain high cache hit rates. But as the computational demands and capabilities of modern applications increase, cache memories are subjected to non-uniform memory access patterns that cannot be optimized by a single cache eviction algorithm. To solve this problem, this thesis introduced HyVE, a modular framework that combined several standalone eviction policies together and evaluated their opinions using voting theory. HyVE was designed to consider several optimization criteria simultaneously, thereby making better eviction decisions than its standalone counterparts. The concept of HyVE was applied and evaluated as part of two case-studies - LLCs and sparse directories. For LLCs, different flavours of HyVE were explored empirically. Results showed that HyVE reduced the LLC misses by up to 25% compared to its standalone counterparts. HyVE was also compared to state-of-the-art hybrid (DRRIP) and learning-based (Hawkeye) eviction policies. On average, HyVE showed better performance than DRRIP (1.9%), but worse performance compared to Hawkeye (-1.8%). For sparse directory evaluations, HyVE was constructed with architecturally-aware eviction policies. Results showed that HyVE reduced the application execution time by up to 11% compared to the LRU policy. For both case-studies, there were applications where a given standalone eviction policy, a HyVE flavour or one of the explored state-of-the-art eviction policies which exhibited a unique advantage over the others. This re-emphasizes the statement that a single eviction policy does not provide the best performance for all applications.

5.2 Outlook

It is often stated that research work should open more doors than they close. The contributions of this thesis - RBCC and HyVE should not be viewed as final solutions, as there is always room for improvement.

Region-based Cache Coherence (RBCC)

Hybrid Hardware-Software Coherence Support

Towards the end of this thesis report, Section 3.7 described how shared memory programming could be used on a DSM-based tiled manycore architecture with two approaches - RBCC and VSM. These approaches could be combined together in two different ways.

The first idea is to make use of VSM’s paging functionality to bring application data into the TLMs, which allows for quick memory accesses. Then, coherence regions can be set up for the corresponding TLM address ranges, guaranteeing coherency and consistency of application data. This hybrid approach eliminates costly software routines of the VSM approach at the expense of additional hardware resources of RBCC.

The second idea is to make use of the VSM mechanism if/when an application requests for more processing and/or memory resources than the maximum coherence region size. Instead of declining this request, the application could yet be executed across multiple coherence regions, where RBCC guarantees inter-tile coherence within the coherence regions, and the VSM mechanism guarantees coherence between the coherence regions.

Exploring Hybrid Programming Models

Similar to the second hybrid RBCC-VSM idea, RBCC could also be combined with language-based software coherence schemes like X10. Currently, on the InvasIC target architecture, one X10-place is defined as one tile. Therefore all inter-tile communication is performed according to the Partitioned Global Address Space (PGAS) programming model. By re-defining the notion of one X10-place to several tiles ¹, RBCC can be used to maintain inter-tile coherence within an X10-place, and the PGAS programming model can be used to communicate between different X10-places. Of course, re-defining the size of one X10-place requires modifications to both the X10 run-time system and the hardware design, similar to the RBCC and VSM approaches to shared memory programming.

Accelerating Coherence Invalidations

Section 3.8 described how inter-tile coherence messages were accelerated using the concept of In-NoC Circuit (INC). Recently, colleagues at ITIV, KIT proposed another NoC optimization concept known as *block-based multicast routing*. The idea is for the NoC to efficiently handle similar messages that are intended for different destinations. The CRM’s remote invalidation messages contain the same information which needs to be sent-out to multiple destination tiles, making them prime candidates to exploit this NoC feature. Normally, the CRM sends-out invalidation messages sequentially. But with the block-based multicast feature, the CRM could bundle several invalidations into a single message ². This approach could not only reduce the time taken by the CRM to send-out several invalidations, but also reduce intra-tile bus traffic.

Hybrid Voting-based Eviction Policy (HyVE)

One of the “lessons learnt” from the HyVE explorations was that a HyVE flavour cannot provide the best performance for all possible memory access patterns. To tackle this problem, HyVE is equipped with an eviction policy filter and a voting selector module.

¹Ideally, to the maximum size of a single coherence region

²One additional message is required for a set of invalidations that indicates the destination tiles

5 Conclusion & Outlook

This makes HyVE a highly flexible framework that is capable of changing to different HyVE flavours and/or even the voting methodology at run-time.

Enabling Fine Granular Control

The eviction policy filter module could be further extended to make non-binary decisions. Instead of entirely adding/removing a standalone eviction policy, the generated ranks could be scaled-up or scaled-down. This would enable a weighted-HyVE framework, where the opinions of each constituent eviction policy can be managed on a fine-granular level.

Feedback Mechanisms

With ample fine-granular and run-time available control knobs, the challenge is to efficiently tune them to improve application performance. Decisions like “which HyVE flavour to transform into?” or “when to transform into a different HyVE flavour?” require additional information.

One approach, inspired by the concept of Set-Dueling (SD), is to extract this information from the cache memories. SD uses a PSEL counter that indicates and selects the better performing eviction policy, based on its cache hits. HyVE could be equipped with a similar feedback mechanism, where the PSEL counter could “steer” HyVE’s decisions towards the better performing policy. This option was explored as part of a master thesis [90], leveraging the weighted-HyVE extensions. This showed promising results for certain workloads, motivating for further explorations in this direction.

Alternatively, the tuning information can also be obtained from the application. One approach could be to profile the expected class of applications and pre-determine the best HyVE flavour. This could be further enhanced if applications can provide hints at run-time, indicating the type of memory access patterns in its upcoming phase. This information can be used to tune HyVE accordingly. If no application-specific information is available, machine learning techniques could be used to predict the type of memory access patterns. However, the challenge is to obtain a suitable and reliable training input-set.

Lastly, when exploring add-on’s for HyVE, the expected hardware implementation overheads and complexity should also be considered. It is crucial to maintain a light-weight HyVE design that does not add to the latency of cache memory accesses. Exploring and overcoming these challenges would lead to a self-adapting HyVE framework that could further enhance memory subsystem performance.

Bibliography

- [1] G. Southern and J. Renau. Analysis of parsec workload scalability. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 133–142, 2016. doi:10.1109/ISPASS.2016.7482081.
- [2] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. Adaptive insertion policies for managing shared caches. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 208–219, 2008.
- [3] A. Jain and C. Lin. Cache replacement policies. *Synthesis Lectures on Computer Architecture*, 14(1):1–87, 2019. URL: <https://doi.org/10.2200/S00922ED1V01Y201905CAC047>, arXiv:<https://doi.org/10.2200/S00922ED1V01Y201905CAC047>, doi:10.2200/S00922ED1V01Y201905CAC047.
- [4] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA '07*, pages 381–391, New York, NY, USA, 2007. ACM. URL: <http://doi.acm.org/10.1145/1250662.1250709>, doi:10.1145/1250662.1250709.
- [5] L. Masing, A. Srivatsa, F. Kreß, N. Anantharajaiah, A. Herkersdorf, and J. Becker. In-noc circuits for low-latency cache coherence in distributed shared-memory architectures. In *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoc)*, pages 138–145, 2018. doi:10.1109/MCSoc2018.2018.00033.
- [6] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974. doi:10.1109/JSSC.1974.1050511.
- [7] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. doi:10.1109/N-SSC.2006.4785860.
- [8] T. N. Theis and H. . P. Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, 2017. doi:10.1109/MCSE.2017.29.
- [9] A. Gupta, W. dietrich Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *In International Conference on Parallel Processing*, pages 312–321, 1990.

BIBLIOGRAPHY

- [10] J. Teich et al. Invasive computing: An overview. In *Multiprocessor System-on-Chip*, 2011.
- [11] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166, 2011. doi:10.1109/PACT.2011.21.
- [12] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE Micro*, 31(1):42–55, 2011. doi:10.1109/MM.2011.8.
- [13] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Paillet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 108–109, 2010. doi:10.1109/ISSCC.2010.5434077.
- [14] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012. URL: <https://doi.org/10.1145/2209249.2209269>, doi:10.1145/2209249.2209269.
- [15] H. Shan, J. Singh, L. Oliker, and R. Biswas. Message passing vs. shared address space on a cluster of smps. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 8 pp.–, 2001. doi:10.1109/IPDPS.2001.925009.
- [16] A. Bilas, C. Liao, and J. Singh. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pages 282–293, 1999. doi:10.1109/ISCA.1999.765958.
- [17] A. Srivatsa, S. Rheindt, T. Wild, and A. Herkersdorf. Region based cache coherence for tiled mpsoCs. In *2017 30th IEEE International System-on-Chip Conference (SOCC)*, pages 286–291, 2017.
- [18] A. Srivatsa, S. Rheindt, D. Gabriel, T. Wild, and A. Herkersdorf. Cod: Coherence-on-demand – runtime adaptable working set coherence for dsm-based manycore architectures. In D. N. Pnevmatikatos, M. Pelcat, and M. Jung, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 18–33, Cham, 2019. Springer International Publishing.
- [19] A. Srivatsa, M. Mansour, S. Rheindt, D. Gabriel, T. Wild, and A. Herkersdorf. Dynaco: Dynamic coherence management for tiled manycore architectures. *International Journal of Parallel Programming*, Jan 2021. URL: <https://doi.org/10.1007/s10766-020-00688-6>, doi:10.1007/s10766-020-00688-6.

- [20] A. Srivatsa, S. Rheindt, O. Lenke, L. Nolte, T. Wild, and A. Herkersdorf. *Tackling the MPSoC Data Locality Challenge*, pages 85–117. 04 2021. doi:10.1002/9781119818298.ch5.
- [21] A. Srivatsa, S. Nagel, N. Fafous, N. Anh Vu Doan, T. Wild, and A. Herkersdorf. Hyve: A hybrid voting-based eviction policy for caches. In *2020 IEEE Nordic Circuits and Systems Conference (NorCAS)*, pages 1–7, 2020. doi:10.1109/NorCAS51424.2020.9265136.
- [22] N. A. V. Doan, A. Srivatsa, N. Fafous, S. Nagel, T. Wild, and A. Herkersdorf. On-chip democracy: A study on the use of voting systems for computer cache memory management. In *2020 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 984–988, 2020. doi:10.1109/IEEM45057.2020.9309925.
- [23] A. Srivatsa, N. Fafous, N. Anh Vu Doan, S. Nagel, T. Wild, and A. Herkersdorf. Exploring a hybrid voting-based eviction policy for caches and sparse directories on manycore architectures. In *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, 2021.
- [24] A. Srivatsa, N. Fafous, N. A. V. Doan, T. Wild, and A. Herkersdorf. Method for evicting data from memory. In *Publication Number: WO/2021/175725, International Application Number: PCT/EP2021/054888*, Sep 2021.
- [25] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture, ISCA '89*, page 2–15, New York, NY, USA, 1989. Association for Computing Machinery. URL: <https://doi.org/10.1145/74925.74927>, doi:10.1145/74925.74927.
- [26] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. *SIGARCH Comput. Archit. News*, 18(2SI):148–159, May 1990. URL: <https://doi.org/10.1145/325096.325132>, doi:10.1145/325096.325132.
- [27] M. P. Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.
- [28] R. F. van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel’s single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1):73–83, February 2011. URL: <https://doi.org/10.1145/1945023.1945033>, doi:10.1145/1945023.1945033.
- [29] B. D. de Dinechin. Kalray mppa®: Massively parallel processor array: Revisiting dsp acceleration with the kalray mppa manycore processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–27, 2015. doi:10.1109/HOTCHIPS.2015.7477332.

BIBLIOGRAPHY

- [30] E. Rustad. Whitepaper: A qualitative discussion of numaconnect vs infiniband and other high-speed networks. June 2019. URL: <https://www.numascale.com/wp-content/uploads/2019/06/numascale-vs-infiniband.pdf>.
- [31] D. Eadline. Whitepaper: Redefining scalable openmp and mpi price-to-performance with numascale’s numaconnect. June 2019. URL: <https://www.numascale.com/wp-content/uploads/2019/06/WhitePaper-Redefining-Scalable-Price.pdf>.
- [32] S. Nürnberger, G. Drescher, R. Rotta, J. Nolte, and W. Schröder-Preikschat. Shared memory in the many-core age. In *Euro-Par 2014: Parallel Processing Workshops*, pages 351–362, Cham, 2014. Springer International Publishing.
- [33] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, SOSPP ’89*, page 211–223, New York, NY, USA, 1989. Association for Computing Machinery. URL: <https://doi.org/10.1145/74850.74871>, doi:10.1145/74850.74871.
- [34] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, PPOPP ’90*, page 168–176, New York, NY, USA, 1990. Association for Computing Machinery. URL: <https://doi.org/10.1145/99163.99182>, doi:10.1145/99163.99182.
- [35] J. Cordsen, T. Garnatz, M. Sander, A. Gerischer, M. Gubitoso, U. Haack, and W. Schroder-Preikchat. Vote for peace: implementation and performance of a parallel operating system. *IEEE Concurrency*, 5(2):16–27, 1997. doi:10.1109/4434.588280.
- [36] K. Li. Ivy: A shared virtual memory system for parallel computing. In *ICPP*, 1988.
- [37] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA ’05*, page 519–538, New York, NY, USA, 2005. Association for Computing Machinery. URL: <https://doi.org/10.1145/1094811.1094852>, doi:10.1145/1094811.1094852.
- [38] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, September 2007.
- [39] Tile processor architecture overview for the tilepro series. *Tilera Corporation*, March 2011. URL: https://www.inf.pucrs.br/~moraes/prototip/artigos/tilera_architecture.pdf.

- [40] R. E. Kessler. The cavium 32 core octeon ii 68xx. In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–33, Aug 2011. doi:10.1109/HOTCHIPS.2011.7477487.
- [41] J. Hennessy, M. Heinrich, and A. Gupta. Cache-coherent distributed shared memory: perspectives on its development and future challenges. *Proceedings of the IEEE*, 87(3):418–429, March 1999. doi:10.1109/5.747863.
- [42] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang. Selectdirectory: A selective directory for cache coherence in many-core architectures. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 175–180, March 2015.
- [43] S. Demetriades and S. Cho. Stash directory: A scalable directory for many-core coherence. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 177–188, Feb 2014. doi:10.1109/HPCA.2014.6835928.
- [44] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 169–180, Feb 2011. doi:10.1109/HPCA.2011.5749726.
- [45] D. Chaiken, J. Kubiawicz, and A. Agarwal. Limitless directories: A scalable cache coherence scheme. *ASPLOS IV*, pages 224–234, New York, USA, 1991. ACM. doi:10.1145/106972.106995.
- [46] D. Sanchez and C. Kozyrakis. Scd: A scalable coherence directory with flexible sharer set encoding. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, Feb 2012. doi:10.1109/HPCA.2012.6168950.
- [47] A. Sodani et al. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, Mar 2016. doi:10.1109/MM.2016.25.
- [48] Y. Fu, T. M. Nguyen, and D. Wentzlaff. Coherence domain restriction on large scale systems. In *48th Intl. Symp. on Microarchitecture, MICRO-48*, pages 686–698, New York, USA, 2015. ACM. doi:10.1145/2830772.2830832.
- [49] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. doi:10.1147/sj.52.0078.
- [50] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA '10*, pages 60–71, New York, NY, USA, 2010. ACM. URL: <http://doi.acm.org/10.1145/1815961.1815971>, doi:10.1145/1815961.1815971.
- [51] N. Megiddo and D. S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, April 2004. doi:10.1109/MC.2004.1297303.

BIBLIOGRAPHY

- [52] K. M. AnandKumar, A. S. D. Ganesh, and M. S. Christy. A hybrid cache replacement policy for heterogeneous multi-cores. In *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 594–599, Sep. 2014. doi:10.1109/ICACCI.2014.6968209.
- [53] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 167–178, 2006.
- [54] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.
- [55] H. Gao and C. Wilkerson. A dueling segmented lru replacement algorithm with adaptive bypassing. *JWAC 2010 - 1st JILP Workshop on Computer Architecture Competitions: cache replacement Championship*, 06 2010.
- [56] S. M. Khan, Y. Tian, and D. A. Jiménez. Sampling dead block prediction for last-level caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186, 2010.
- [57] C. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer. Ship: Signature-based hit predictor for high performance caching. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.
- [58] E. Teran, Z. Wang, and D. A. Jiménez. Perceptron learning for reuse prediction. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.
- [59] D. A. Jiménez and E. Teran. Multiperspective reuse prediction. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 436–448, 2017.
- [60] A. Jain and C. Lin. Back to the future: Leveraging belady’s algorithm for improved cache replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA '16*, page 78–89. IEEE Press, 2016. URL: <https://doi.org/10.1109/ISCA.2016.17>, doi:10.1109/ISCA.2016.17.
- [61] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of scalable shared-memory multiprocessors: the dash approach. In *Digest of Papers Comcon Spring '90. Thirty-Fifth IEEE Computer Society International Conference on Intellectual Leverage*, pages 62–67, 1990. doi:10.1109/CMPCON.1990.63654.
- [62] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992. doi:10.1109/2.121510.

- [63] X. Shi, F. Su, and J. Peir. Directory lookaside table: Enabling scalable, low-conflict, many-core cache coherence directory. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 111–118, Dec 2014. doi:10.1109/PADSW.2014.7097798.
- [64] S. Shukla and M. Chaudhuri. Tiny directory: Efficient shared memory in many-core systems with ultra-low-overhead coherence tracking. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 205–216, 2017. doi:10.1109/HPCA.2017.24.
- [65] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '13*, pages 187–198, Washington, DC, USA, 2010. IEEE Computer Society. URL: <https://doi.org/10.1109/MICRO.2010.20>, doi:10.1109/MICRO.2010.20.
- [66] A. Kalbande. Hardware support for configurable cache coherence in tiled many-core architectures. Master’s thesis, Technical University of Munich, Germany, 2015.
- [67] J. Torrellas, H. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994. doi:10.1109/12.286299.
- [68] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '95*, page 179–188, New York, NY, USA, 1995. Association for Computing Machinery. URL: <https://doi.org/10.1145/209936.209955>, doi:10.1145/209936.209955.
- [69] T. Liu, C. Tian, Z. Hu, and E. D. Berger. Predator: Predictive false sharing detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '14*, page 3–14, New York, NY, USA, 2014. Association for Computing Machinery. URL: <https://doi.org/10.1145/2555243.2555244>, doi:10.1145/2555243.2555244.
- [70] T. Liu and X. Liu. Cheetah: Detecting false sharing efficiently and effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO '16*, page 1–11, New York, NY, USA, 2016. Association for Computing Machinery. URL: <https://doi.org/10.1145/2854038.2854039>, doi:10.1145/2854038.2854039.
- [71] T. Liu and E. D. Berger. Sheriff: Precise detection and automatic mitigation of false sharing. *SIGPLAN Not.*, 46(10):3–18, October 2011. URL: <https://doi.org/10.1145/2076021.2048070>, doi:10.1145/2076021.2048070.

BIBLIOGRAPHY

- [72] V. W. Freeh and G. R. Andrews. Dynamically controlling false sharing in distributed shared memory. In *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*, pages 403–411, Aug 1996. doi:10.1109/HPDC.1996.546211.
- [73] M. Waliullah and P. Stenstrom. Classification and elimination of conflicts in hardware transactional memory systems. In *2011 23rd International Symposium on Computer Architecture and High Performance Computing*, pages 96–103, Oct 2011. doi:10.1109/SBAC-PAD.2011.18.
- [74] P. R. Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM. URL: <http://doi.acm.org/10.1145/500001.500018>, doi:10.1145/500001.500018.
- [75] J. Aynsley. *OSCI TLM-2.0 language reference manual*. Open SystemC Initiative, ja32 edition, jul. 2009.
- [76] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. URL: <http://doi.acm.org/10.1145/2024716.2024718>, doi:10.1145/2024716.2024718.
- [77] M. Gebhart, J. Hestness, E. Fatehi, P. Gratz, and S. W. Keckler. Running parsec 2.1 on m5. Technical report, The University of Texas at Austin, Department of Computer Science, October 2009.
- [78] T. Langer, J. Rabenstein, T. Hönig, and W. Schröder-Preikschat. No coherence? no problem! virtual shared memory for mpsocs. In *2021 SC Workshops Supplementary Proceedings (SCWS)*, 2021.
- [79] H. Nurmi. *Voting Theory*, pages 101–123. Springer Netherlands, Dordrecht, 2010. URL: https://doi.org/10.1007/978-90-481-9045-4_7, doi:10.1007/978-90-481-9045-4_7.
- [80] W. D. Wallis. *The Mathematics of Elections and Voting*. Springer International Publishing, 2014.
- [81] K. So and R. N. Rechtschaffen. Cache operations by mru change. *IEEE Transactions on Computers*, 37(6):700–709, June 1988. doi:10.1109/12.2208.
- [82] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42Nd Annual Southeast Regional Conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM. URL: <http://doi.acm.org/10.1145/986537.986601>, doi:10.1145/986537.986601.

- [83] H. Ghasemzadeh, S. Sepideh Mazrouee, and M. R. Kakoe. Modified pseudo lru replacement algorithm. In *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS '06*, pages 368–376, Washington, DC, USA, 2006. IEEE Computer Society. URL: <http://dx.doi.org/10.1109/ECBS.2006.52>, doi:10.1109/ECBS.2006.52.
- [84] T. Johnson and D. Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94*, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. URL: <http://dl.acm.org/citation.cfm?id=645920.672996>.
- [85] E. J. O’Neil, P. E. O’Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, SIGMOD '93*, pages 297–306, New York, NY, USA, 1993. ACM. URL: <http://doi.acm.org/10.1145/170035.170081>, doi:10.1145/170035.170081.
- [86] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014. doi:10.1145/2629677.
- [87] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.
- [88] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture, ISCA '95*, pages 24–36, New York, NY, USA, 1995. ACM. URL: <http://doi.acm.org/10.1145/223982.223990>, doi:10.1145/223982.223990.
- [89] T. C. Simulator. URL: <https://github.com/ChampSim/ChampSim>.
- [90] S. Nagel. Exploring a hybrid voting-based eviction policy for last level caches. Master’s thesis, Technical University of Munich, June 2020.

A Results of all Explored HyVE Flavours

The evaluations of HyVE for LLCs explored various HyVE flavours consisting of two, three and four standalone eviction policies as listed in Table 4.2. Section 4.4.3 analysed five HyVE flavours in detail. The LLC performances of remaining 18 HyVE flavours are reported in Figure A.2. To make comparisons easy, the LLC performances of the standalone eviction policies as well as the already analysed 5 HyVE flavours are reported in Figures A.1 and A.3 respectively. Note that all LLC misses are normalized to that of the LRU policy.

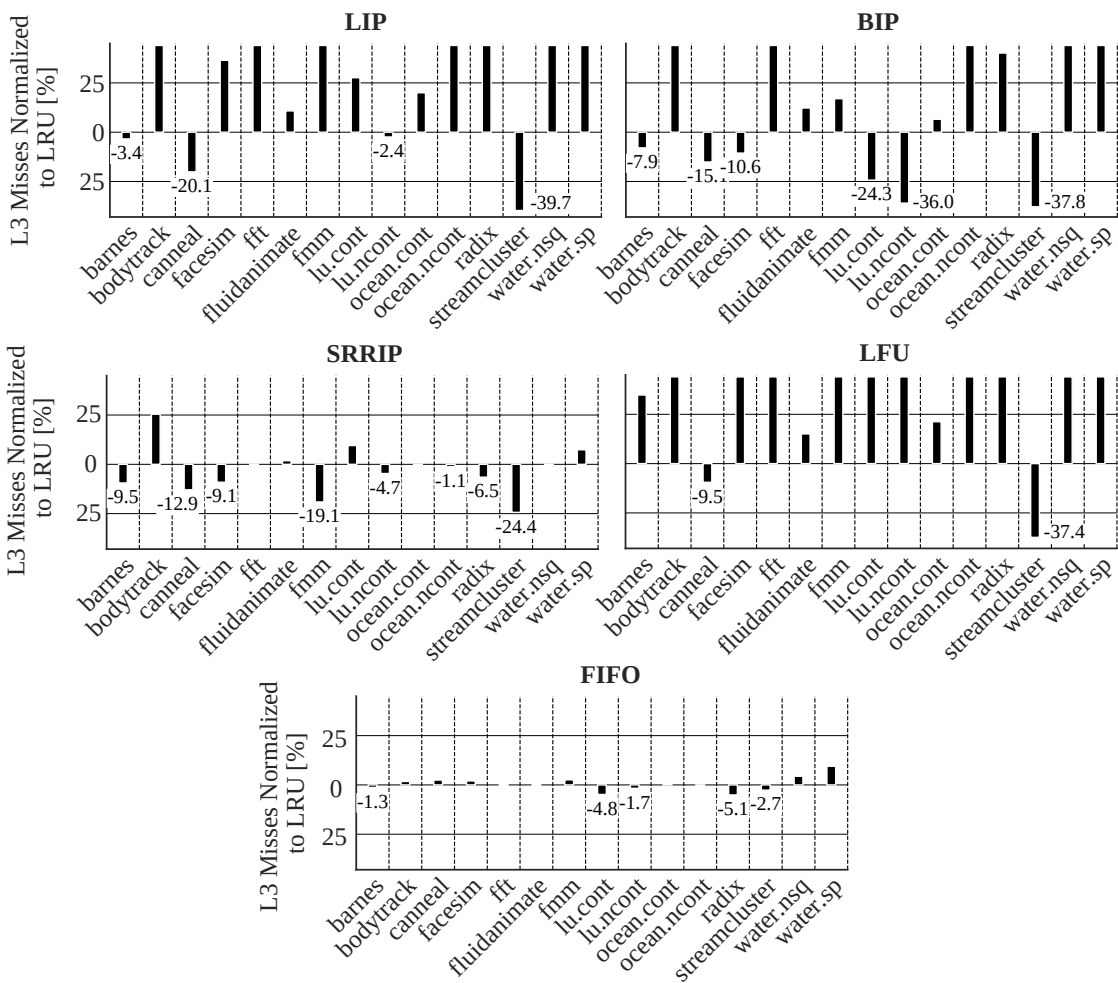


Figure A.1: Normalized LLC misses of the standalone eviction policies for all benchmarks

A Results of all Explored HyVE Flavours

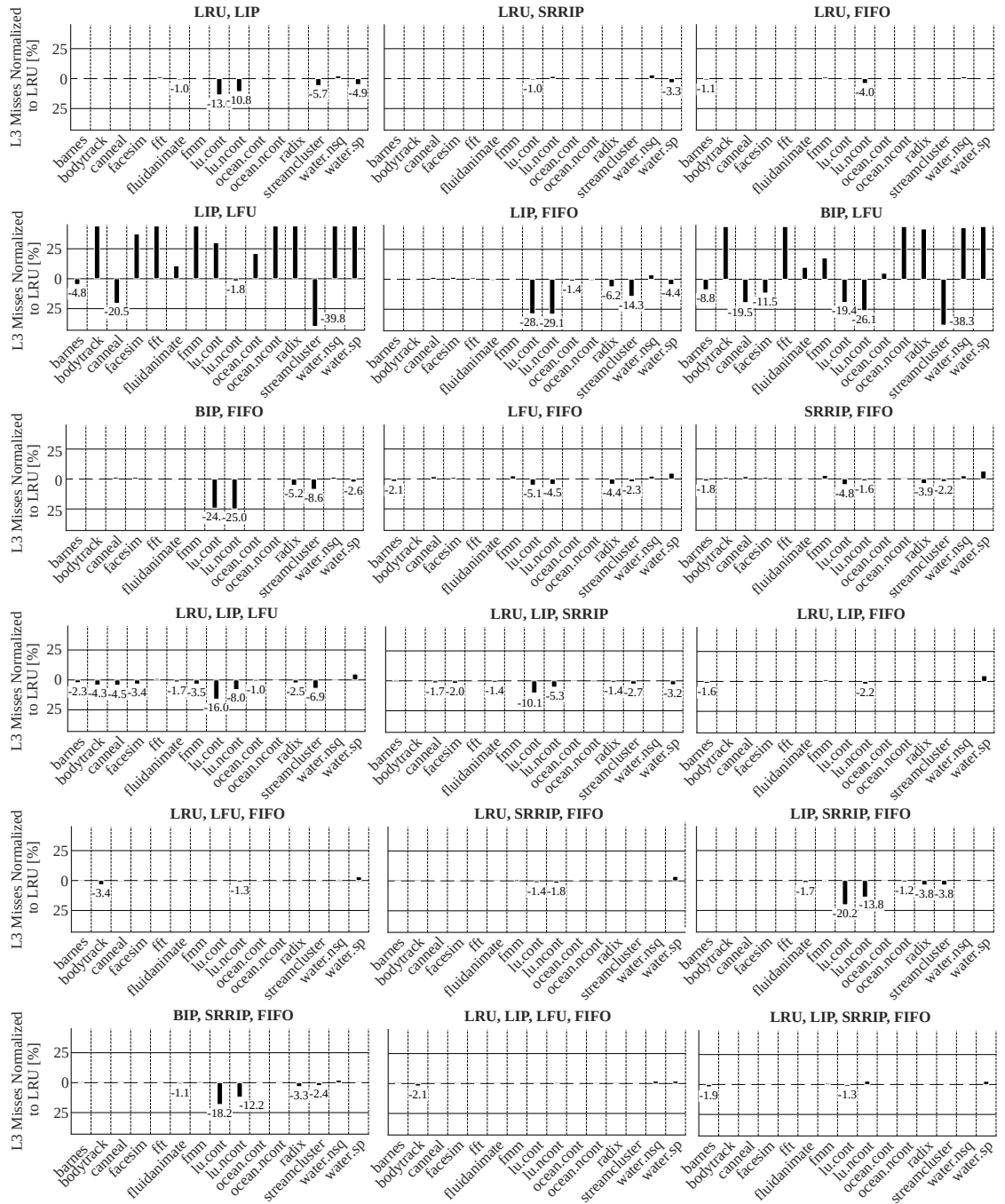


Figure A.2: Normalized LLC misses of the 18 remaining HyVE flavours for all benchmarks

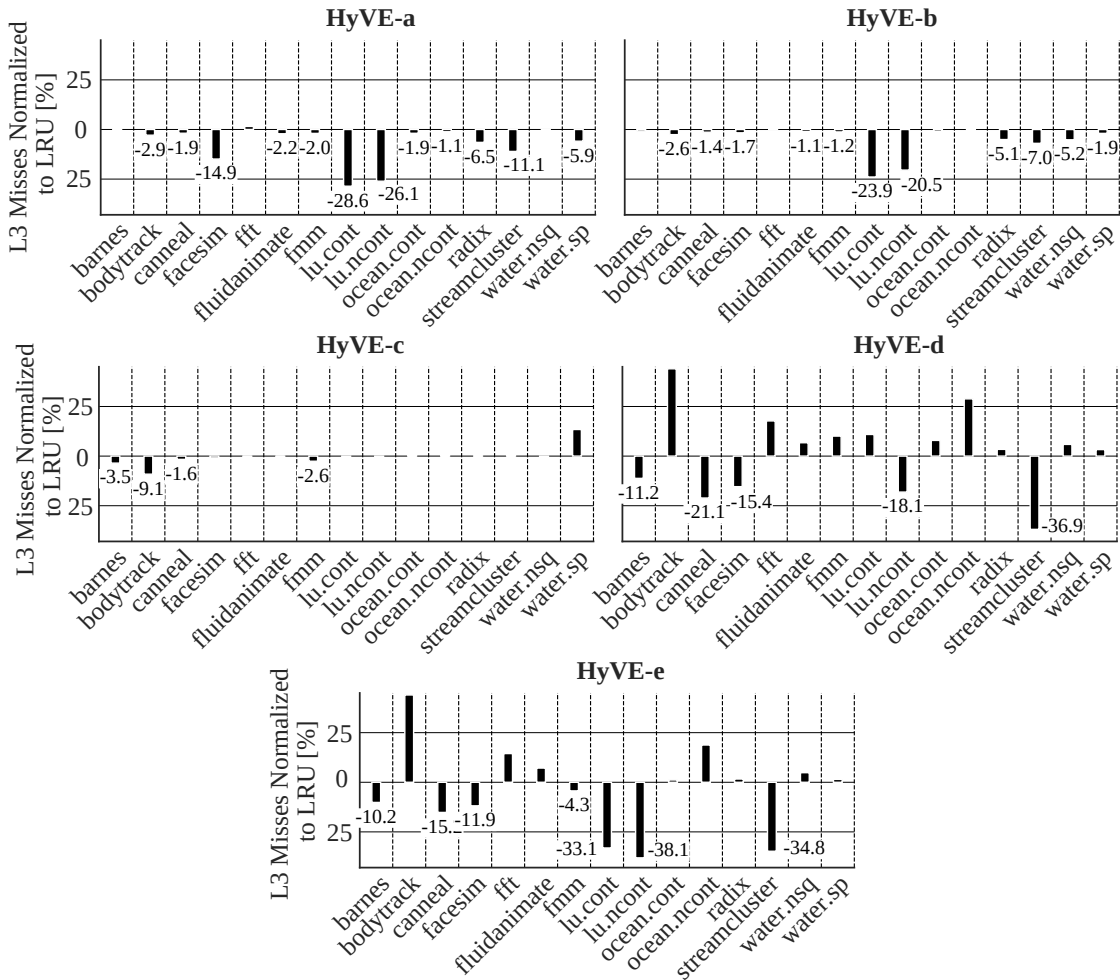


Figure A.3: Normalized LLC misses of the already analysed 5 HyVE flavours for all benchmarks

B Analysing HyVE with the Condorcet Method for Sparse Directories

Figure B.1 reports the execution time for all benchmarks using the Condorcet Method as the voting methodology. In general, HyVE with the Condorcet Method exhibits poorer performance than HyVE with Borda Count, similar to the results with HyVE for LLCs. This can be specifically observed for the results of *fft* and *lucb*.

Figure B.2 reports the victim selection distribution analysis for HyVE with the Condorcet Method with the sparse directory configuration of 128 sets, 8 ways. Interestingly, the victim selection distribution for HyVE with Condorcet contains significant eviction decisions categorized as *Majority Agree* following LNS+SDF. This implies that HyVE's decisions were mostly biased by LNS and SDF which exhibit poor performance. Further, both LNS and SDF use the same sharer information (bit-vector) to determine the eviction victim. Though they optimize for different eviction criteria, due to this commonality, their decisions tend to be similar. This can be observed for the *lucb* benchmark, where HyVE with the Condorcet Method almost always follows the eviction decisions of LNS+SDF. This results in a two-against-one situation, omitting the influence of the LRU policy altogether. The same effect is observed for *swaptions*, where HyVE with the Condorcet Method has a worse execution time than HyVE with Borda Count. This influences HyVE's behaviour, leading it to closely track the performance of LNS and SDF.

From these results, one could conclude that the Borda Count is a superior voting procedure than the Condorcet Method. However, from the voting theory perspective, the Condorcet Method is the fairer voting methodology. Perhaps for eviction decisions, a fairer voting procedure may not be the best choice as the Borda Count compensates for some bias due its point-based ranking system. Further, from the experiments with HyVE for LLCs, the Condorcet Method is known exhibit bias when incorporated into a voting system with a small number of voters. As a take-away point, the statement 'HyVE with Borda Count seems to outperform HyVE with the Condorcet Method for the cache eviction problem' holds true.

B Analysing HyVE with the Condorcet Method for Sparse Directories

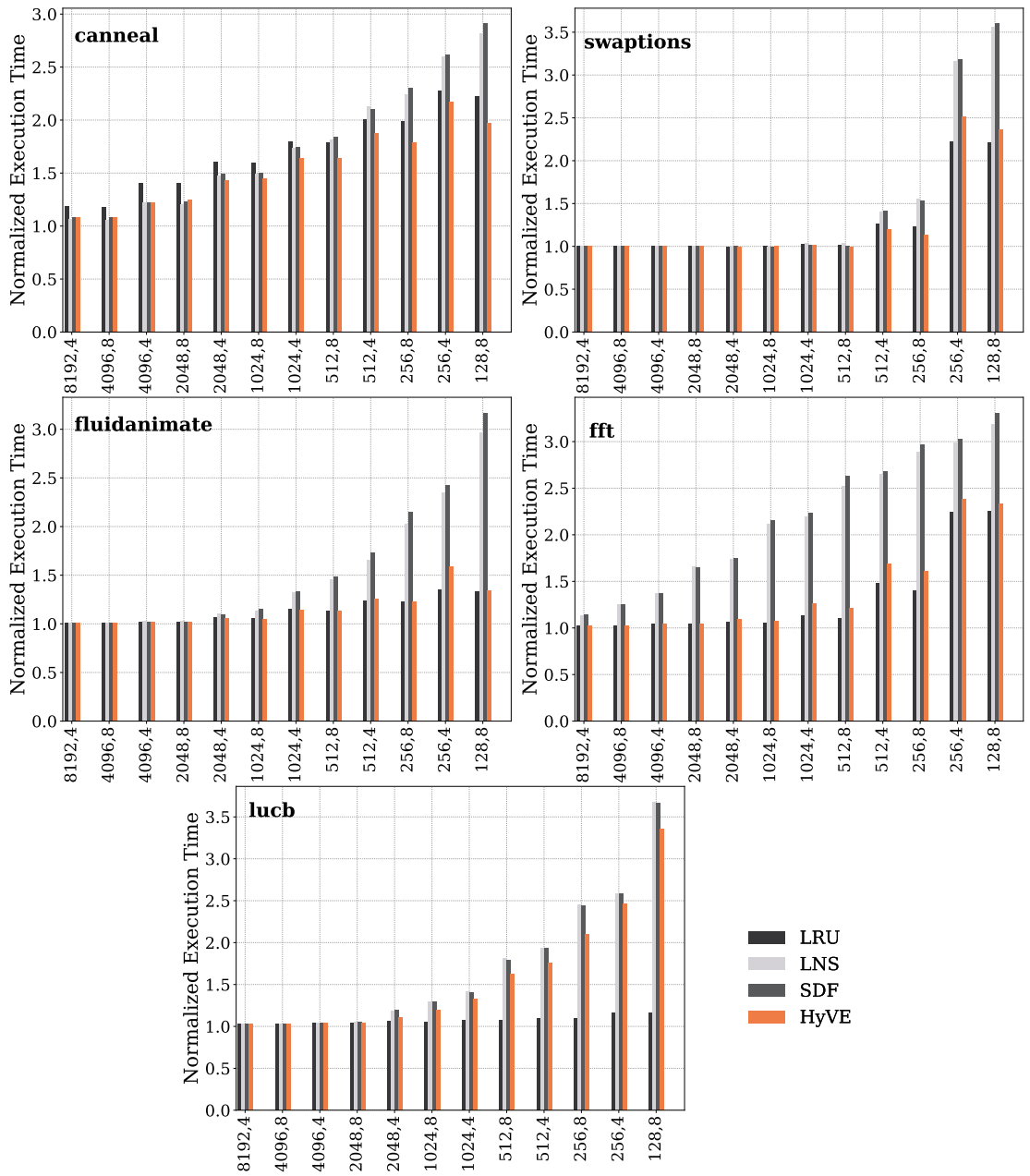


Figure B.1: The execution time of all benchmarks using the standalone eviction policies and HyVE with the Condorcet Method for different sparse directory configurations

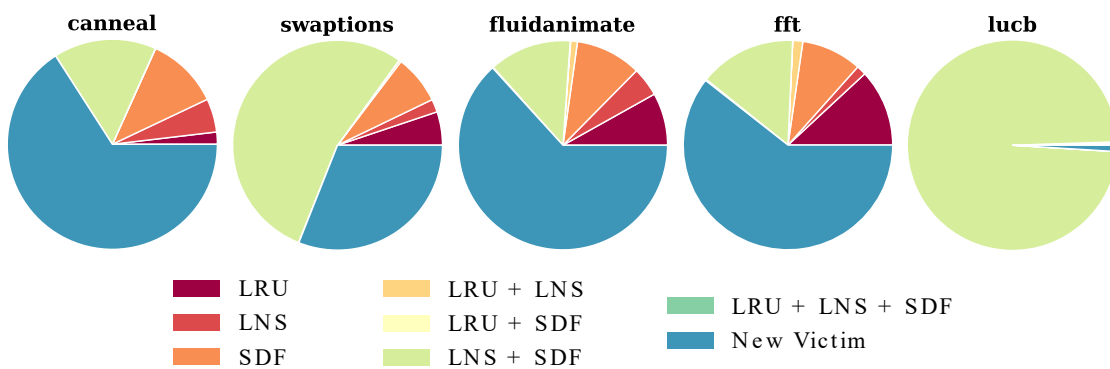


Figure B.2: Victim distribution analysis of all benchmarks for the sparse directory configuration - 128 sets, 8 ways using the Condorcet Method as the voting procedure