

Copyright
by
Mochamad Asri
2020

The Dissertation Committee for Mochamad Asri
certifies that this is the approved version of the following dissertation:

**Hardware-Assisted Data Movement Optimizations for
Heterogeneous System Architectures**

Committee:

Andreas Gerstlauer, Supervisor

Calvin Lin

George Biros

Jonathan Beard

Lizy K. John

Mohit Tiwari

**Hardware-Assisted Data Movement Optimizations for
Heterogeneous System Architectures**

by

Mochamad Asri

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2020

Dedicated to my grandmother, Mamih.

She passed away as I was writing this dissertation.

Mamih, I wish I could have said one last goodbye to you.

Thank you for taking a great care of me since I was a kid.

Thank you for loving me, and for all the wonderful memories.

Rest in peace now, I love you.

To my loving father, this PhD is as much yours as mine.

I wish you could be here and attend my defense.

But I am sure you will be watching from up there.

Thank you for teaching me the joy of learning.

Thank you for being my greatest hero. I love you.

-Anden-

Acknowledgments

First and foremost, I would like to express a special appreciation and thanks to my advisor, Prof. Andreas Gerstlauer. This work would not have been possible without his sharp guidance, endless patience and unwavering support. As much as I like to critically self-question myself, Andreas always challenges my ideas and solutions in an intellectually demanding and stimulating way. Living up to such exacting standards is never easy and can get very overwhelming and exhausting at times. But from him, I learned that good enough is never great enough, great enough is never best enough, and best enough is just a start of the quest for perfection.

I would like to also express my sincere gratitude to my dissertation committee: Prof. George Biros, Prof. Lizy John, Prof. Mohit Tiwari, Prof. Calvin Lin, and Dr. Jonathan Beard. Working with George has certainly broaden my view on algorithm designs and their interactions with the hardware. Having regular meetings with Lizy in my early PhD years has helped me build a strong foundation on benchmark characterization and performance profiling. Mohit was the faculty that convinced me to join UT and I am glad I did. From Calvin, I developed a great interest and passion in memory/cache systems. His wisdom in this field along with his unparalleled positivity and open-mindedness certainly encourage me to always look for and enjoy working

on highly challenging problems. And finally, crossing paths with Jonathan is one of the key highlights in my PhD journey. He was the first person who introduced me to near-memory topic that later becomes a part of this dissertation. Jonathan is one of the smartest, most driven, and most compassionate mentors I have ever met. I continue to draw inspiration from him, not just to do better research, but even beyond that, to be a better human being.

My graduate school experience would have been different without this wonderful group of educators: Prof. Yale Patt and Prof. Nina Telang. I would like to thank them for giving me amazing opportunities to be a TA of their EE 306 class. Through Prof. Patt, I experienced unforgettable moments of being a TA in an American school for the first time. Something I would never have thought I could do as an international student. Such a new experience continued four years later, as I was appointed by Prof. Telang as the head TA of her class. Not only did I enjoy taking more responsibilities as the head TA, but I also learned from Prof. Telang how respect, compassion, and kindness can go a long way in teaching. I am very grateful for the opportunities that both of them had given to me.

I would also like to express my sincere appreciation to a fantastic group of research mentors I met along the way: Prof. Mattan Erez, Dr. Ardavan Pedram, Dr. Akanksha Jain. Mattan was the first faculty that introduced me to research in computer architecture. I am thankful of the feedback he presented to me that helped sharpen my research skills during my early PhD years. From Ardavan, I got my feet wet on accelerator research that later

became a building block of my research. Although it can be psychologically challenging working with him at times, I am grateful for all the useful knowledge and wisdom he had shared with me during my early years. And finally, this appreciation would not be complete without mentioning Dr. Akanksha Jain. Seeing how Akanksha was always excited when talking about caching and prefetching during the class, certainly helped build my interest and passion in caching/memory systems during my early PhD years. I am also grateful that later in my final years I got to work on a project with her which later became a part of this dissertation. It has been fun and a pleasure at the same time to work with her, a smart researcher and a wonderful human being.

My PhD journey would not have been as much fun without my fellow members of the SLAM research group: Kishore Punniyamurthy, Kamyar Mirzazad, Alex Cathis, Erika Susana Alcorta, Dimitris Gourounas, Zhuoran Zhao, Sabine Francis, Xinnian Zheng, Wooseok Lee, Dongwook Lee, Seogoo Lee. I spent my good and bad days with them, experienced my ups and downs with them, and shared my brightest and darkest moments with them. Nothing compares to having them as my family here, and I am grateful for our bonds and friendships. I will miss the daily lab chats, campus gossips, our regular lunch and boba time. I wish you guys all the best for the rest of your PhD career.

I also thank other graduate students for all the wonderful graduate school memories: Jiajun Wang, Reena Panda, Kyushick Lee, Esha Choukse, Jungrae Kim, Dong Wan Kim, Sangkug Lym, Yongkee Kwon, Anyesha Ghosh,

Majid Jalili, Siavash Zangeneh, Ben Lin, Tian Tian, Stephen Pruett, Prateek Sahu, Willy Vasquez, Qinzhe Wu, Marko Vasic, Jay Patel, Otitoaleke Akinola, Suyogya Karki, Mahsid Alamdar, Bagus Hanindito, Ahmad Zulfikar, Reynaldi Fifariz, Joseph Amick. Special shout-out to Alex Hsu who has been my intensive partner in work and crime for the last two years. Alex and I started a cache replacement project when we took Calvin's class, and later we developed the project further and eventually became a chapter of this dissertation. I am lucky to have crossed paths with him, a great colleague and an amazing friend.

I am also grateful for the good friends I made and for their support during my PhD time. I would like to thank Fron, Mbak Butet, Mas Djoko, Mas Widodo, Mas Candra, William Teh, Randy, Qoni, Ibu Tami, Ibu Sandra, Ibu Gina, Ibu Evie, Hari, Chris, Kaisa, Schrouilde, Alessia, Alessio, Aarne, Susan, Marie, Diego, Sudesh, Bruno, Hannah, Halley, Krishnendra, Urmish, Gururaj, Patrick, Jiecao, Mikako, Brittany, Paloma, Roman, Ivona, Ralph, Lora, Tara, Rachel, Mandy, Marissa, Raul, Clint, Anna. The dinners, road trips, hikes, potlucks, dance lessons, runs, volley-ball games, karaoke sessions, and hangouts with them made sure I had a life outside of my PhD.

I would like to also express my appreciation to all the staff of the Consulate General of Republic of Indonesia (KJRI) in Houston for their help during my PhD study. Special thanks to the Consul General, Ibu Nana Yuliana, for your constant support and encouragement, Ibu Romanna for your help with passport extension, Ibu Enah and Bapak Bambang for the wonderful hospitality at the Konjen. While I am 10,000 miles away from my home in Indonesia

during my study, the friendly gatherings, dinners, sport/music events, and certainly amazing Indonesian food at KJRI made sure I can always feel home here in Texas.

This dissertation would not be complete without giving credit to Annie Szotkowski. Her immense support, unequaled love and compassion have kept me going through the difficult years of graduate school. Thank you for always being on my side and my biggest supporter. Thank you for all the food surprises and watermelons that powered me through every paper deadline. Thank you for all the scooter rides, bike rides, fun trips, karaoke, two-step dancing, jamming sessions, porch fest, movies, blood donations, sushi, ramen, pool burger and buffet palace nights, all of which kept me sane during these final years. Thank you for understanding me and being my bedrock during the most difficult times. Thank you for always believing in me even when I almost gave up. Thank you for being you.

Finally, I would like to express my deepest gratitude for my amazing family: My mother (Mamah) and my sister (Enong). Thank you Mamah for your relentless support and night prayers. It was not until I heard she could only stand up for less than five minutes nowadays that I realized she is physically aging while I am gone. And this journey must have made it even harder for her than it was for me. Thank you Mamah, for all the unconditional love and incredible patience. And thank you Enong for taking care of Mamah while I am here. Thank you for all the birthday and cake surprises for Mamah. I wish I did not have to live so far away from you for the past 15 years, so

that I could spend more time together with you. You have sacrificed a lot for me. From now on, let me return your kindness. But first, let's travel together with Mamah and the kids once the COVID-19 pandemic ends!!

Hardware-Assisted Data Movement Optimizations for Heterogeneous System Architectures

Publication No. _____

Mochamad Asri, Ph.D.
The University of Texas at Austin, 2020

Supervisor: Andreas Gerstlauer

Heterogeneous systems have emerged as state-of-the-art computing solutions. Such systems consist of acceleration units that provide massive compute capabilities within limited power budgets. Compute and/or memory intensive regions of applications are often offloaded to these accelerators and as such they naturally exchange data with the host core. This complex scenario poses a key challenge: how do we optimize data movement between the host core and accelerators from a holistic system-level perspective?

My research focuses on addressing the above question. Data movement optimizations can be explored in two flavors: 1) Maximizing locality and keeping the data close to its compute, and 2) moving the actual computation itself close to the data. Such optimizations fundamentally depend on applications as well as their interaction with underlying architectures. Exploring associated tradeoffs first and foremost requires an accurate modelling infrastructure.

To that end, I first propose a systematic simulator calibration methodology to provide a faithful baseline for accurately modeling targeted system architectures. Results show that an unrepresentative baseline can cause misleading conclusions in heterogeneous architecture studies. Using this calibrated simulator, I then study accelerator integration and respective data movement tradeoffs under consideration of different on- and off-chip coupling scenarios. From this study, I observe that applications can benefit from integrating accelerator closer to the chip, providing up to 20% better performance with 17% less total energy consumption over an off-chip integration. However, significant software modifications are required to fully unlock such benefits. Furthermore, traditional software overheads for accelerator invocation and offload can further limit acceleration benefits.

To address such challenges, I propose three hardware-assisted approaches that enable transparent optimization of data movement in heterogeneous architectures with little to no software or programmer overhead.

To perform automatic, software-transparent fine-grain data staging and synchronization between on-chip integrated components, I first introduce a novel Cache-Managed, Fine-Grain Accelerator Staging and Pipelining in On-Chip Heterogeneous Architectures (CASPHAr). CASPHAr tracks and synchronizes producer and consumer accesses at cache line granularity. As soon as some fraction of shared data is produced and becomes ready in the LLC, the data will be delivered for processing in the waiting consumer, reducing data spills due to unnecessarily long lifetimes of shared data in the cache. Results

show that CASPHAr can boost performance by up to 23% and achieve energy savings of up to 22% over baseline accelerations.

I further introduce Flock, a shared cache management scheme that targets improved system-wide performance benefits among heterogeneous components when running independent kernels/applications. Different from prior work, Flock employs holistic performance proxies that capture cache misses/hits in each level of the memory hierarchy when selecting a cache partitioning. This enables Flock to increase the system throughput by taking a global view of a core’s performance instead of focusing on the LLC misses/hits in isolation. Moreover, Flock applies a new shared cache replacement scheme that adapts to the varying access rates of different individual cores, preventing domination of a core with high access intensity. Results show that Flock improves performance by up to 12.5% over state-of-the-art solutions.

Finally, I present the Non-Uniform Compute Device (NUCD) system architecture for low-latency and generic accelerator offload to move computations closer to the data. Different from conventional offload mechanisms that rely primarily on device drivers and software queues, the NUCD system architecture extends a host core micro-architecture to enable a low-latency out-of-order task offload to heterogeneous devices. Results demonstrate that the NUCD system architecture can achieve an average performance improvement of 21%-128% over a conventional driver-based offload mechanism. This in turn enables whole new forms of fine-grain task offloading that would otherwise not see any performance benefits.

Table of Contents

Acknowledgments	v
Abstract	xi
List of Tables	xviii
List of Figures	xix
Chapter 1. Introduction	1
1.1 Application/Architecture Tradeoffs	1
1.2 Simulation Infrastructure	2
1.3 Data Movement Optimizations	3
1.4 Hardware Support	4
1.5 Thesis Statement	7
1.6 Contributions	7
Chapter 2. Related Work	12
2.1 System Simulations	12
2.2 Accelerator Integration/Data Movement Tradeoffs	15
2.3 Accelerator Staging and Pipelining	16
2.4 Cache Management for Co-Running Applications	18
2.5 Accelerator Offload	20
Chapter 3. Systematic Simulator Calibration Methodology	23
3.1 Calibration Methodology	24
3.1.1 Core Pipeline Calibration	26
3.1.1.1 Instruction Latencies	28
3.1.1.2 Pipeline Width	30
3.1.1.3 Instructions and Micro-ops	32

3.1.2	Cache System Calibration	35
3.2	Results	38
3.2.1	Baseline Refinement	39
3.2.2	Impact on Heterogeneous Design	43
3.3	Summary and Conclusions	44
Chapter 4.	Accelerator Integration/Data Movement Tradeoffs	45
4.1	FMM Application and its Execution Flow	46
4.2	Accelerator Coupling	47
4.2.1	External Off-chip Coupling at Shared DRAM Level	49
4.2.2	Internal On-chip Coupling at Shared Cache Level	51
4.3	HW/SW Co-Optimization	53
4.3.1	GEMM Acceleration	54
4.3.2	Cache-Aware Blocking and Memory Hiding	55
4.3.3	Prefetching, Pipelining and Overlapping	55
4.4	Results	59
4.4.1	DRAM Accesses	61
4.4.2	Execution Time	63
4.4.3	Energy	65
4.5	Summary and Conclusions	67
Chapter 5.	Cache-Managed Accelerator Staging and Pipelining	68
5.1	CASPHAr Overview	68
5.1.1	Base System Architecture	69
5.1.2	CASPHAr System Architecture	71
5.1.3	Software Interface	74
5.1.4	Discussions and Possible Variations	76
5.2	CASPHAr Design	77
5.2.1	Base Cache Architecture	77
5.2.2	Synchronization Flow	80
5.2.3	CASPHAr-Aware Cache Replacement	84
5.3	Evaluation	86
5.3.1	Performance and Energy Results	87

5.3.2	Why CASPHAr Works	91
5.4	Summary and Conclusions	97
Chapter 6.	Cache Management for Co-Running Applications	98
6.1	Background	98
6.1.1	Performance Proxy	99
6.1.2	Sharing Model	101
6.2	Flock Overview	103
6.3	Flock Design	104
6.3.1	Base Cache Management	105
6.3.2	Partition Allocator	106
6.3.3	Relative Aging	113
6.3.4	Hardware Overhead	115
6.4	Evaluation	116
6.4.1	Comparison With Prior Art	118
6.4.2	Why Flock Works	119
6.4.3	Epoch Size Sensitivity	126
6.5	Summary and Conclusions	127
Chapter 7.	Low-Latency Task-Level Accelerator Offload	129
7.1	Background	129
7.2	NUCD Overview	131
7.3	The NUCD Programming Model	132
7.3.1	Task Identification	133
7.3.2	NUCD Instruction	133
7.3.3	Offload Flow	135
7.3.4	Multi-Task Offload	136
7.4	The NUCD Micro-Architecture	138
7.4.1	Accelerator Invocation and Context Transfer	138
7.4.2	Accelerator Execution and Address Translation	141
7.4.3	Accelerator Retirement and Sequential Consistency	142
7.4.3.1	Scoreboarding	143
7.4.3.2	Dependency Tracking with LSQ	143

7.4.3.3	Sequential Retirement with ROB	144
7.4.3.4	Putting It All Together	145
7.5	Experiments and Results	147
7.5.1	Speedup	149
7.5.2	Offload Overhead	150
7.5.3	Scaling	152
7.6	Summary and Conclusions	153
Chapter 8.	Summary, Conclusions, and Future Research	155
8.1	Dissertation Summary	155
8.2	Future Work	156
	Bibliography	158
	Vita	180

List of Tables

3.1	System configuration.	25
3.2	GEMM micro-op breakdown comparison.	32
3.3	Micro-ops of GEMM on MARSSx86.	33
3.4	Cache miss-rates on each level.	37
3.5	L2 set breakdown when executing GEMM.	38
4.1	System configuration.	60
5.1	CASPHAr-aware replacement policy extensions.	85
5.2	System configuration.	86
6.1	Speedup due to better cache management depends on the number of LLC hits <i>and</i> the proportion of memory accesses that reach the LLC. (MAT represents total memory access time.) .	100
6.2	Flock hardware cost (4-core, 32-way 8MB LLC)	115
6.3	Simulated system configuration.	117
6.4	Applications used in our evaluation	117
7.1	Summary of simulation parameters.	148
7.2	Workload parameters.	148

List of Figures

3.1	Core Pipeline Architecture.	26
3.2	Microbenchmark for testing instruction latencies.	27
3.3	Flowchart of the improved commit logic, accounting for different micro-op implementation in the simulator.	34
3.4	Microbenchmark for testing L1-D accesses.	36
3.5	Normalized DGEMM runtime on original and calibrated simulators.	39
3.6	Runtime improvement of five benchmarks on each calibration step.	39
3.7	Normalized DGEMM runtime improvement as a function of prefetch degree.	40
3.8	Normalized runtime of different simulators on five benchmarks.	41
3.9	Normalized runtime of PARSEC benchmarks running on original and calibrated simulators, with respect to hardware platform.	42
3.10	Normalized runtime of SPEC benchmarks running on original and calibrated simulators, with respect to hardware platform.	42
3.11	Speedup of benchmarks using original and calibrated simulators.	43
4.1	Data flow graph (DFG) of U-list execution and data movement patterns.	46
4.2	Accelerator coupling at different levels of the memory hierarchy.	49
4.3	Data movement pattern when sharing LLC.	52
4.4	Comparison of CPU-only with CPU + LAP, total execution cycles and system energy ($q = 8$). While LAP offers significant GEMM acceleration, we observe that simply coupling the LAP on LLC does not make remarkable difference compared to the LAP on the DRAM. More effort is needed to unlock additional gains.	54
4.5	Runtimes and energy with blocking and no latency hiding ($q = 8$).	56
4.6	Software blocking, pipelining and overlapping.	59
4.7	DRAM accesses with different integration options across varying q	61

4.8	Execution cycles with different integration options across varying q	63
4.9	DRAM energy with different integration options across varying q	65
4.10	Total energy with different integration options across varying q	66
5.1	Baseline system architecture.	69
5.2	CASPHAr system architecture.	71
5.3	Execution flow on a CASPHAr system.	76
5.4	CASPHAr LLC architecture.	77
5.5	CASPHAr synchronization on a synchronization miss but cache hit.	80
5.6	CASPHAr synchronization on a cache miss and cache eviction.	81
5.7	DRAM accesses.	88
5.8	DRAM energy.	89
5.9	Execution cycles.	90
5.10	Sensitivity to consumer slowdown rate (<i>stencil</i>).	91
5.11	Fraction of staged lines that are evicted before they get consumed.	92
5.12	Histograms of produced but unconsumed shared cache line occupancies.	93
5.13	Histograms of produced shared cache line lifetimes until consumed.	95
5.14	Execution cycles and DRAM accesses of CASPHAr vs. manual optimizations (<i>FMM</i> and <i>spmv</i>).	96
6.1	The number of LLC accesses on different evaluated systems normalized to big core accesses	102
6.2	Top-level block diagram of Flock cache.	103
6.3	Hawkeye cache architecture.	105
6.4	Block diagram of our Partition Allocator.	107
6.5	Relative aging's block diagram.	114
6.6	Performance improvement of evaluated replacement policies normalized to LRU on homogeneous and heterogeneous systems	119
6.7	Speedup normalized to LRU for two cores sorted by Flock. Flock outperforms ASM on homogeneous CPU-CPU systems, heterogeneous CPU-CPU systems, and heterogeneous CPU-GPU systems.	119

6.8	Decomposing Flock’s benefits.	120
6.9	Comparison of Real CPI and Flock’s estimated CPI for xalancbm. Flock’s estimated CPI closely tracks real CPI.	121
6.10	Flock’s partitioning differs from ASM because it takes a holistic view of performance and allocates more cache space to core 1. Flock achieves higher performance than ASM due to better partitioning.	122
6.11	Average partition allocations of each core and their actual cache occupancy when running 470.lbm and 401.bzip2 mix with three different sharing models. Flock’s sharing model achieves the best performance.	123
6.12	Cache Occupancy of Flock, Flock-IMP and Flock-EXP when running 470.lbm and 401.bzip2 mix on a 2-core homogeneous system. Flock strikes the right balance between explicit partitioning and implicit replacement.	124
6.13	Cumulative cache occupancy and speedup on a CPU-GPU running 401.bzip2 and histogram. Flock achieves better performance than TAP and Hawkeye because it dedicates more cache to the CPU.	125
6.14	Cache Occupancy and IPC of each core over time when running bzip and histogram mix using Flock on a heterogeneous CPU-GPU system. Compared to TAP, Flock intelligently allocates more cache to the CPU. This results in better performance for the CPU and no performance difference for the GPU.	126
6.15	Flock’s sensitivity to epoch sizes.	127
7.1	System-wide speedup with and without offload overhead.	130
7.2	Envisioned system architecture.	131
7.3	Task identification in the NUCD programming model.	133
7.4	Register format of the nucd instruction.	134
7.5	Offload flow from the host core to the accelerator and back.	135
7.6	NUCD system architecture.	137
7.7	Extended host core architecture for invocation and context transfer.	139
7.8	LSQ and ROB snapshot of the host core during acceleration. An instruction marked as red in the LSQ is blocked until NUCD acceleration completes to satisfy dependencies. Other instructions can execute subject to sequential ROB retirement.	145

7.9	Speedup under different offload mechanisms with varying accelerators.	149
7.10	Fraction of execution time spent on offload.	151
7.11	Performance scaling with number of accelerator cores on <i>meabo2</i> and <i>stride</i> applications.	152

Chapter 1

Introduction

The end of Moore’s law era means that single-socket, general-purpose hardware is quickly approaching its limits. A possible path to increasing single-socket performance is to consider a special-purpose hardware (hereby an “accelerator”). By sacrificing programmability, accelerators provide the best performance with respect to time-to-solution and energy consumption. Compute and/or memory intensive regions of applications are often offloaded to these accelerators and as such they naturally exchange data with the host core. This complex scenario introduces a new and unique research challenge: how do we optimize data movement between the host core and accelerators from a holistic system-level perspective? My research specifically aims to address this question.

1.1 Application/Architecture Tradeoffs

Data movement optimizations are closely tied to architectural integration of the accelerator with the host system. Where in the memory hierarchy do we place the accelerator in order to reduce data movement and communication overhead? Traditionally, accelerators have been integrated as off-chip

devices placed near DRAM. This requires all data to be exchanged through DRAM. Such off-chip data movement can negate performance benefits of acceleration and can account for up to 50% of dynamic access energy in HPC applications [37].

More recently, accelerators have been moved in or near main memory [13, 141], but this only affects how efficiently the accelerator itself can access memory. To minimize or completely avoid off-chip DRAM transfers, accelerators can instead be integrated on the same die and coupled by sharing the last-level cache (LLC) with CPUs [87, 137, 29]. However, such a design also has its own shortcomings. If data exceeds LLC capacity, expensive spills to DRAM will still take place—depending on application locality patterns.

Overall, deciding on whether on-chip or off-chip placement is better depends strongly on application and architecture interactions. Understanding associated tradeoffs and optimization opportunities requires comprehensive design space exploration studies. Different system architectures result in different data movement scenarios and tradeoffs, which often are non-obvious and require quantitative analyses.

1.2 Simulation Infrastructure

Such design space explorations and optimizations first and foremost require an accurate modeling infrastructure. Cycle-accurate simulators are widely used in architecture research as modeling infrastructures and a certain degree of performance inaccuracy is expected in such simulators [61]. Some

of the mismatches are the result of modeling the system at a lower level of detail compared to RTL or gate-level simulation, thereby trading accuracy for speed. This can cause a significant discrepancy between the model and the targeted system. Such mismatches may be tolerable in micro-architectural studies where the emphasis is often on the relative but not absolute effect of architectural modifications on performance. Nevertheless, an inaccurate baseline model can still lead to wrong conclusions, e.g. when relative contributions of different micro-architecture components are not properly accounted for. Moreover, an accurate baseline is essential when modeling heterogeneous systems. An inaccurate baseline CPU model can lead to severe over- or under-estimation of the speedups and energy savings achieved by various optimizations. It is therefore crucial that simulators used in both system- or micro-architecture studies be calibrated to match real-world platforms.

1.3 Data Movement Optimizations

Generally speaking, data movement optimizations can be achieved in two ways. First, maximizing locality to keep data close to its compute. Data is initially brought in from its source, and kept resident in local memories. The goal is to let the data reuse/exchange happen at the local memory as much as possible. This approach particularly benefits applications with good locality. On the other hand, we can also move the actual computation itself close to where the data is. This is, in particular, suited for applications that exhibit poor locality. Parts of computations are delegated to devices near the data.

And ideally, the cost of delegating such applications should remain low.

Existing prior arts have investigated several techniques to provide solutions in this space. First, to maximize data locality with compute in case of interdependent kernels, software-centric data tiling and blocking optimizations have been widely applied. They essentially break larger operations or data into smaller pieces, so that data can fit into the shared space. With such software optimizations, streaming applications can benefit a lot from closer on-chip integration of accelerators. By contrast, if not managed carefully, spills can easily negate on-chip integration benefits. This approach, however, has a major caveat that the software has to be rewritten and re-optimized for every different and new hardware to achieve the best system design.

Prior work also has proposed a wide variety of mechanisms to move the actual compute closer to data. In such approaches, accelerators are placed in or near memory and computations are offloaded to such far devices. This is mainly done using device drivers acting as an interface between software and the hardware. While such approaches excel in portability and productivity aspects, the resulting overhead for device initiation and actuation can not be neglected. As I will demonstrate in later chapters, a wide range of applications suffer from this approach, which can severely limit the net acceleration benefits.

1.4 Hardware Support

In summary, existing approaches come with large software costs and overheads. In this dissertation, I instead ask the question whether can we

achieve equivalent or better benefits through hardware support with little to no software or programmer involvement. I consider such possible hardware supports for three different contexts: 1) Maximizing data locality between dependent kernels; 2) Maximizing data locality between independent kernels/applications running concurrently; and finally, 3) minimizing offload costs when moving computations closer to data.

First, in case of maximizing data locality between dependent kernels, a key question is whether the shared memory space (typically the LLC) can automatically manage locality and sharing with relatively minimum software involvement? Dependent kernels exhibit producer-consumer data exchange patterns. A producing kernel computes and writes a set of data, and later, the consuming kernel reads the data to perform its own task. Traditional approaches enforce the consumer to not start reading the data until the producer finishes writing the complete staged data. However, if the produced data size exceeds the cache capacity, some data will get spilled to DRAM before it can get consumed. If the cache can instead manage production and consumption status of data such that a consumer is allowed to access data as soon as it is produced, such spills can potentially be avoided and locality between dependent kernels can be maximally exploited.

In the context of maximizing locality of independent applications running concurrently, a key challenge is to intelligently manage the shared cache capacity such that it yields the maximum system-wide throughput. Different applications running on different types of cores can have varying degree of

performance sensitivity as a function of allocated cache capacity. Such variations between cores get exacerbated in heterogeneous systems where different cores can have vastly different performance sensitivity. In particular, latency-tolerant accelerators with high access intensity will enjoy an increased amount of cache hits when given more space, but such increased hits do not necessarily translate into performance benefits. Allocating more cache space to such cores may thus be counterproductive to overall system performance. Existing cache management solutions—a vast majority of which were designed for homogeneous CPU systems—do not address the complex needs of such heterogeneous systems. This requires novel approaches for shared cache management transparently integrated into the hardware.

Finally, in the context of moving the computation closer to the data, a key challenge is how to minimize the costs of offloading, initiation, context transfer and synchronization. To handle such low-level operations and procedures, traditional approaches rely on device drivers to provide a clean abstraction for the software to interface with the accelerator device. However, as I will show later, associated software and driver costs can quickly outweigh any potential acceleration benefits. This is particularly the case for accelerators placed nearer to memory, which rely on task-level parallelism and multi-threading to hide latency. Being able to launch a large number of potentially fine-grain tasks with little programming and offloading overheads is critical for such near-memory accelerators. Enabling hardware support to mitigate such offload overheads can allow a wide range of accelerated applications

to enjoy greater benefits from such near-memory accelerations.

1.5 Thesis Statement

My thesis demonstrates that data movement optimizations on heterogeneous systems can be assisted by hardware mechanisms that allow less intrusive software changes while achieving equivalent or higher benefits as careful manual optimizations. This, first and foremost, requires a well-calibrated simulation infrastructure to faithfully study application- and architecture-dependent data movement tradeoffs. Using such infrastructures and leveraging insights gained from tradeoffs studies, I show that hardware-assisted data movement optimizations can be achieved through cache-managed fine-grain accelerator staging and pipelining, effective shared cache management for co-running applications, and low-latency task-level accelerator offload.

1.6 Contributions

The specific contributions of this dissertation are as follows:

1. **Systematic simulator calibration methodology** [21]:

I propose a simulator calibration methodology for accurate accelerator integration studies. I show a systematic, step-by-step approach to calibrate a widely-used cycle-accurate x86 simulator, starting from validation of core pipeline operation up to memory system behavior. I also evaluate the potential impact such a calibrated simulator can have on

accelerator-rich architecture and design space exploration studies. Our calibrated simulator shows on average less than 10% error across a wide range of benchmarks. Using this calibrated baseline simulator, I further quantify the impact of such calibration on an accelerator-rich architecture case study. Our results show that an accelerator estimated to obtain a speedup of 10.3x using unrefined simulators can in reality only achieve a 5.3x speedup when an accurate CPU model is employed.

2. Accelerator integration/data movement tradeoff study:

Using the above calibrated simulator, I then study accelerator integration and its data movement tradeoffs under consideration of different architecture coupling cases. I also show how algorithms and architectures can be co-designed to unlock further system-wide performance improvement under different coupling scenarios. Our results show that naive on-chip integration of a hardware accelerator alone reduces DRAM accesses by up to 26% compared to an off-chip integration. With careful algorithm and placement optimizations, DRAM accesses can be reduced further by up to 2x for a total of 3x savings over an off-chip integration. Moreover, when more aggressive algorithm/architecture optimizations of software pipelining and overlapping with on-chip accelerator coupling are applied, I am able to achieve up to 37% additional performance gains and up to 35% less energy consumption compared to a baseline on-chip accelerator integration. When similar optimizations are applied to an off-chip coupling, on-chip integration provides up to 20% better performance with

17% less total energy consumption.

3. **Cache-managed accelerator staging and pipelining:**

I further present an approach for Cache-Managed, Fine-Grain Accelerator Staging and Pipelining in On-Chip Heterogeneous Architectures (CASP_HAr). CASP_HAr supports transparent and efficient data sharing between CPUs and accelerators. Instead of relying on software to coordinate preparation of input data, synchronization with accelerators and processing of results at sub-block level, CASP_HAr tracks, synchronizes and coordinates production and consumption of data at cache-line granularity directly in the LLC. Using existing cache interfaces, a shared cache line is marked as ready for any dependent consumer to access as soon as it is produced. In doing so, CASP_HAr reduces data spills due to unnecessarily long lifetimes of shared data in the cache caused by accelerator staging at too coarse granularity. Moreover, fine-grain synchronization in hardware inherently results in overlapping and pipelining of producer and consumer executions. Our evaluation results show that CASP_HAr can achieve results similar to or better than manual system-level data movement optimizations in software, improving performance by up to 23% and reducing energy consumption by 22% compared to baseline acceleration

4. **Cache management for co-running applications:**

I then introduce Flock¹, an effective cache management for on-chip heterogeneous systems that targets an improved aggregate system-wide performance benefits when co-running independent applications. Different from prior work, Flock observes cache misses/hits in each level of the memory hierarchy as a performance proxy instead of only LLC misses/hits. Flock then selects a cache partitioning that maximizes system speedup, and uses that information to assign an insertion priority to each cache line. Finally, Flock protects the initially selected partitioning by adaptively adjusting the eviction policy depending on the cores' access intensity, preventing domination of a core with high intensity. Our evaluation results show that Flock improves performance by up to 12.5% over state-of-the-art solutions.

5. **Low-latency task-level accelerator offload** [20]:

Finally, to support an efficient mechanism for moving computation closer to data, I introduce the non-uniform compute device (NUCD) system architecture (SA). The NUCD SA enables generic, hardware-assisted and driver-less low-latency compute offload to task-level accelerators. It provides a canonical programming model that adds a single instruction to indicate a region of interest to offload. This instruction provides fast identification of critical information to setup an accelerator context, such as code region size, memory footprint/granularity, and output/input reg-

¹In the natural world, many organisms that travel in groups exhibit emergent behavior known as *flocking*, in which simple local rules lead to elegant, ordered aggregate behavior.

isters. The NUCD microarchitecture then handles all offload, initiation, synchronization and forwarding of an accelerator context without driver involvement. NUCD tightly integrates with a general-purpose core’s out-of-order mechanisms to maintain standard sequential programming semantics while enabling transparent parallel execution between the host core and accelerators. Results show an average system-wide performance benefit of 21%-128% when compared to a more traditional driver-based offload mechanism and up to $2.6\times$ speedup for applications utilizing fine-grain tasks that achieve no performance gains under a traditional driver model.

Chapter 2

Related Work

In this chapter, related works for each specific contribution in this thesis are discussed. We first describe related work in existing system simulators and present prior art in heterogeneous system research utilizing such simulators. We then discuss related work on accelerator integration/data movement tradeoffs and its co-optimizations. We finally discuss different architecturally-orchestrated approaches on optimizing data-movement, where prior art on data staging and pipelining, and shared cache management and accelerator offload are further discussed.

2.1 System Simulations

One of the earliest simulators in the architecture community is SimpleScalar [30]. Since then, many simulators have emerged and are able to model complete computer systems including both user and operating system (OS) code, such as Zsim [116], Sniper [32], gem5 [25] and MARSSx86 [104].

Zsim [116] and Sniper [32] use binary instrumentation or interval simulation to provide fast and scalable system validation. They achieve high simulation speeds by trading off accuracy and raising the level of abstraction

in modeling of architectural features, especially for hardware and OS interactions. However, as our results will show, baseline accuracy and architectural detail of a simulator are essential for modeling of heterogeneous systems, where overheads of interactions between accelerators, the OS and the host CPU can have a large influence on overall results. Higher-level simulators such as Zsim and Sniper are unable to support this level of modeling granularity.

Among cycle-accurate full-system simulators, gem5 [25] is a discrete-event simulator supporting x86, ARM, PowerPC, SPARC, and MIPS targets for both in-order and out-of order CPUs. MARSSx86 [104] was designed to provide fast simulation of x86 systems by integrating a x86 timing model with QEMU. In this study, we target typical accelerated high-performance computing (HPC) applications that rely heavily on vectorized instructions and floating-point performance. In this domain, the potential for acceleration is best exploited. gem5 has only marginal support for vectorized instructions. A number of Streaming SIMD Extensions (SSE) are not completely implemented. This results in many of our targeted HPC benchmarks failing to run on gem5 due to unimplemented instructions. By contrast, MARSSx86 has a more complete and accurate x86 ISA implementation, including SSE and x87 floating-point support. As such, we use MARSSx86 as the baseline simulator in this work.

MARSSx86 has been employed by several heterogeneous system studies as their baseline model to show the potential of their proposed accelerator and its improvements. Esmailzadeh *et al.* [50] proposed a Neural Processing

Unit (NPU) tightly coupled to the processor pipeline to accelerate small code regions. Zhu *et al.* [143] proposed a software-assisted hardware accelerator for the critical style-resolution kernel within a Web browser engine. Yang *et al.* [137] presented an approach to utilize the CPU and GPU resources that are integrated on the same die sharing the on-chip L3 cache. While all of the aforementioned heterogeneous system studies show significant speedup benefits, none of them evaluated that their baseline simulator models accurately represent the targeted machine’s performance.

Several studies have evaluated the performance mismatch between cycle-accurate full-system simulators and real machines. Butko *et al.* [31] have investigated the runtime mismatch between a cycle-accurate gem5 model and a real ARM. Gutierrez *et al.* [61] further leveraged the insights of this work to account for the accuracy of micro-architectural characteristics and how they affect the overall runtime accuracy. While there have been several such studies on identifying sources of errors on a particular cycle-accurate simulator, to the best of our knowledge, there exists no prior work that specifically reports and discusses x86 simulator and MARSSx86 accuracy. Furthermore, there are no published studies that consider the calibration and impact of the baseline simulator accuracy in the context of heterogeneous systems design. Different from previous work, this work proposes a detailed and systematic x86 simulator calibration methodology that is applied to obtain a highly accurate MARSSx86 model. Moreover, we quantify the implication of such calibrations on a heterogeneous system design and design space exploration case study.

2.2 Accelerator Integration/Data Movement Tradeoffs

Accelerators are often designed as standalone components that communicate with the rest of the system through some interface [75, 82]. While a large amount of ASIC-based accelerator research has focused on the performance and energy improvements [129, 35, 63, 105], system-level integration and implications have received less consideration.

Initially, such ASICs were envisioned as external devices located in a separate chip different from the host system. As such, any data transfer has to be exchanged through DRAM. To address the overhead of off-chip memory accesses, researchers have proposed methods for hiding communication latency using overlapped computations, e.g. via special direct memory access (DMA) [142, 118, 73] and DMA pipelining mechanisms [105]. More recently, processing-in-memory (PIM) architectures have gained renewed attention [91]. With advancements in 3D technology, integration of general-purpose or specialized compute units into the logic layer of 3D stacks has been investigated to exploit massive memory bandwidth at low energy consumption. While such approaches in general help move accelerator computation closer to memory, not every application pattern can benefit. In particular, most PIM work assume limited to no data sharing between the PIM accelerators and the host [13, 141]. By contrast, in domains where data is frequently shared and exchanged, the potential benefit of PIM acceleration is limited and unclear.

Alternatively, we can place the accelerator on the chip. To optimize on-chip cache-level integration, caching and cooperative prefetching policies have

been proposed [87, 137]. Other approaches allow accelerator local memories to become part of the shared cache substrate, thereby extending LLC capacity when accelerators are not in use [41]. Within this context, PIM-like approaches closely couple accelerators with on-chip caches [11], but such architectures are again targeted at offloading operations that manipulate data already residing in caches.

In all cases, existing work is limited to hardware- and architecture-managed integration issues. Existing work does not evaluate and compare actual system-wide benefits and tradeoffs of different on-/off-chip accelerator couplings. In this work, we study and demonstrate cross-layer acceleration tradeoffs from a system-level and hardware/software perspective, specifically in the context of HPC applications. Furthermore, not only do we study hardware-centric optimizations, but also we present both software as well as hardware-assisted optimizations to fully exploit performance and energy opportunities of acceleration.

2.3 Accelerator Staging and Pipelining

Several software mechanisms have been proposed to optimize data staging and pipelining that maximize data locality in shared cache level. Compiler or library support has been proposed to exploit parallelism and regular communication patterns in stream programs by formulating a set of cache-aware optimizations that automatically improve data locality [117, 60, 52, 85, 92]. More recently, several works have investigated compiler flows to provide

an end-to-end approach that takes a high-level specification specifically of a machine learning applications written in existing frameworks such as TensorFlow or PyTorch, and generates low-level optimized code including data reuse/exchange optimization for a diverse set of hardware back-ends, including mapping to different sets of accelerators during run-time [28, 36]. All such approaches unlock system-level gains and data movement reductions at the expense of increased software/compiler burden. As we will describe further in the later chapters, for applications that exhibit regular streaming exchange patterns with relatively large data footprint, software/compiler support can equally orchestrate data movement to maximize data locality and unlock pipelining/overlapping. However, with a small data footprints and potentially repetitive invocations on a coarse-grain accelerator, software/compiler optimizations are limited to only allowing accelerator execution once the CPU finishes preparing a minimum amount of data. Thereby, limiting otherwise available data-flow parallelism at finer granularity.

By contrast, several hardware mechanisms have been investigated to achieve fine-grain data staging optimizations between interdependent kernels that can avoid caveats of software-based approaches. The work in [66] has analyzed contention challenges of live data shared by heterogeneous cores that can lead to cache spills, but no specific hardware techniques have been proposed or studied to tackle such challenges. More recently, several approaches have been proposed for fine-grain data staging and synchronization. The work in [46] has employed full/empty bits in DRAM managed by a shared memory

controller to overlap host-to-device memory copies with computation. They assume a discrete memory space between CPU and accelerator and thus are limited to the context of explicit data copy/transfer between CPU and accelerator private memories. The scheme in [119] instead extends a DMA controller with local full/empty information to synchronize data transfers at cache line granularity. However, they require modifications on the accelerator side to comply with their design, and staging support is limited to one direction (i.e., CPU producer and accelerator consumer) in a DMA context with specific transfer patterns. In both schemes, data is shared through off-chip memories. Data exchanges still have to happen at the DRAM level and management of limited on-chip storage capacities is not addressed. Expensive spills and refetches from/to DRAM still occur with such schemes since data can only be exchanged at the DRAM level. This work addresses such a problem, by providing hardware support for data exchange, coordination and orchestration at shared cache level in on-chip heterogeneous architectures.

2.4 Cache Management for Co-Running Applications

Managing the shared cache when co-running independent applications essentially requires partitioning the cache among different applications. There exist two key components when partitioning the cache. First, the partitioning decision that decides how much space to allocate per core. And second, how to enforce such a partitioning decision in the shared cache.

For the former, existing solutions use LLC hit sensitivity in general

as performance proxy [108, 78, 77]. They all are based on Mattson’s stack distance algorithm [94], which is able to provide miss rate predictions of multiple LLC configurations in parallel. A core having more potential hits if given more space will be given more partitions. As we will describe later, estimating performance by considering LLC misses in isolation alone can have fundamental drawbacks. Conceptually, the speedup provided by a cache not only depends on the the number of extra potential LLC hits, but also on the proportion of time spent outside the LLC.

To enforce partitioning decision, there exist two main approaches. Explicit cache partitioning schemes have been proposed to provide utility-based quality-of-service among independent applications on homogeneous CPUs that are sharing the cache [108, 125, 97, 126, 90]. Unfortunately, these solutions have limitations. Explicit partitioning solutions allocate static amount of space for cores and this rigidity can lead to low cache efficiency in case of rapidly changing workload behavior.

By contrast, implicit partitioning schemes enforce partition amount through the replacement policy by assigning different priority to incoming lines during cache line insertion. SDBP [81] and SHiP [132] use a PC-based predictor to remember past evictions and to predict whether incoming lines should be inserted with a high or low priority. TAP [87] is an adaptive insertion-based policy for CPU-GPU multi-core processors. Li *et al.* propose a finer-grained cache management policy to cope with heterogeneous CPU-GPU accesses, leveraging behavior variations among various LLC sets [89]. Hawkeye [69]

learns from Belady’s optimal solution in the past to predict future caching decisions. Glider [121] improves upon Hawkeye by using a machine learning inspired predictor to better predict the optimal solution. However, as we will describe later, these approaches can be too adaptive by nature. Dominant cores can overtake cache space and skew the initially selected partition amount since there is no boundary that divides the partition.

2.5 Accelerator Offload

A number of prior works have shown the performance benefits and energy savings of moving the computation closer to data by offloading kernels to accelerators [14, 59, 141]. While the performance gains demonstrated by these works are promising, they place less emphasis on the efficiency of integrating and interfacing such accelerators with the host core.

In practice, several software and/or hardware extensions are required to integrate an accelerator into a system architecture. Hardware optimizations to integrate accelerators typically begin with a mechanism to communicate “jobs” between a host/master and the endpoint device. This is often in the form of a ring buffer or other form of synchronized communication (e.g., VIRTIO [8]).

Several software approaches exist in literature that attempt to optimize various aspects of the CPU-accelerator runtime/system software interface. Gdev [122] allows GPUs to access main system memory directly and has been shown to be beneficial for low-latency applications. PTask [39] provides an OS abstraction for GPU computing resource and data transfer manage-

ment. It presents a dataflow programming model that exposes information to enable the OS kernel to better assist with performance isolation and data movement coordination. Finally, Pagoda [139] presents a runtime system that dynamically manages GPU resources to improve utilization for narrow kernels with limited parallelism where the actual time spent executing (vs. offload overhead) is relatively low. While the above approaches have demonstrated improvements in data and resource management, they do not address optimization of the actual offload process.

Several hardware mechanisms have been proposed to mitigate accelerator offload costs. Lustig et.al. [46] present fine-grained host CPU-accelerator synchronization. Using full-empty bits to track data transfer completion, the accelerator can process data as soon as it is ready without waiting for full offload completion. While this can benefit by amortizing data transfers during offload and kernel computation, the offload process itself is still performed by the runtime software. Other mechanisms such as AMD’s extended task queue mechanism [84] and Arm’s Revere System Architecture [19] aim to reduce the overheads of interacting with device work queues. However, the actual offload overhead itself is not fundamentally reduced. Other approaches enhance the ISA to support fine-grain accelerator offload through the host core [14, 99, 127]. However, they target instruction-level offloading and thus are not able to support light-weight tasks that consist of instruction streams larger and more complex than just a simple arithmetic operation.

In summary, existing works fall into either traditional driver-centric

offload, hardware-based offloading targeted at instruction-level granularity, or hardware-software mechanisms to accelerate work queues. Driver-based approaches tend to suffer from the non-trivial latency caused by the runtime software and system overhead that can limit the benefit of offload for lightweight tasks. Existing microarchitecture-driven mechanisms are limited to instruction-level offload, and are typically unable to handle lightweight tasks. Lastly, mechanisms that focus solely on acceleration of the work queue tend to reduce the overhead of initiating an accelerator, but do very little for the core-side invocation and targeting of that accelerator.

Chapter 3

Systematic Simulator Calibration Methodology

In this chapter¹, we demonstrate a simulator calibration methodology and evaluate the potential impact such a calibrated simulator can have on accelerator-rich architecture studies [21]. Using microbenchmarking techniques, we show a systematic, step-by-step approach to calibrate a widely-used cycle-accurate x86 simulator, starting from validation of core pipeline operation up to memory system behavior. We target high-performance computing (HPC) applications built around core dense linear algebra operations as a domain in which accelerators are extensively used. We use MARSSx86 [104] as the baseline simulator for this work. We chose MARSSx86 since it is the full system simulator with the broadest x86 support. Specifically, MARSSx86 supports the widest range of vectorized ISA extensions among all cycle-accurate simulators. This is essential in providing a realistic baseline for HPC applications that leverage such vector extensions for improved floating-point perfor-

¹Contents of this chapter previously appeared in:
[21] M. Asri, A. Pedram, L. K. John, and A. Gerstlauer. Simulator Calibration for Accelerator-Rich Architecture Studies, SAMOS, 2016.
I proposed, designed, and implemented the main ideas and collaborated with the co-authors on writing the paper.

mance

3.1 Calibration Methodology

In the following, we present our methodology to calibrate a x86 CPU model for HPC applications. We use a bottom-up methodology that starts from the simplest components of a system and gradually treats and evaluates those as pieces within a larger system. This includes gradually developing both the simulated hardware layers as well as the software benchmark stacks on top. The calibration process is partitioned into two main steps: It starts from the core pipeline evaluating its accuracy with the simplest piece of microbenchmark code. Afterwards, we build on top of the calibrated pipeline model to further observe memory system behaviour. As we will describe in the following, we use microbenchmarks of increasing complexity to calibrate the pipeline and the cache model step by step. Using microbenchmarks that go hand in hand with the hardware layer being calibrated helps us to efficiently unravel and tune the unknown detailed micro-architectural parameters of the targeted machine.

While our general step-by-step methodology should be applicable to any x86 simulator or target machine, in this work we specifically demonstrate our calibration methodology as applied to MARSSx86 targeting an Intel Core i7-920 CPU for HPC applications.

We use General Matrix Matrix Multiplication (GEMM) as the basis for our microbenchmarking stack. GEMM is a key kernel at the core of many

Parameter	i7-920	MARSSx86 Model
Fetch Width	4	4
Dispatch Width	4	4
Issue Width	5	5
Commit Width	4	4
Writeback Width	4	4
Cache Block Size	64B	64B
L1 I-cache Size	128kB	128kB
L1 I-cache Associativity	8-way	8-way
L1 D-cache Size	128kB	128kB
L1 D-cache Associativity	8-way	8-way
L2 Cache Size	256kB	256kB
L2 Associativity	8-way	8-way
L3 Cache Size	8MB	8MB
L3 Associativity	16-way	16-way

Table 3.1: System configuration.

HPC applications. It is nicely layered, reflects the peak performance, uses a wide variety of compute-intensive instructions, including vectorized instructions, can exploit locality at all levels of memory hierarchy, and has optimized implementations available [128].

Table 3.1 details the MARSSx86 configurations compared to our target Intel processor. We used publicly available information to match the target machine configuration, while simultaneously conducting validation checks on the given configuration through microbenchmarking and calibration. We used the PAPI [98] performance counter tool to get the total number of cycles on the i7-920 as compared to MARSSx86’s statistics from its simulated CPU model.

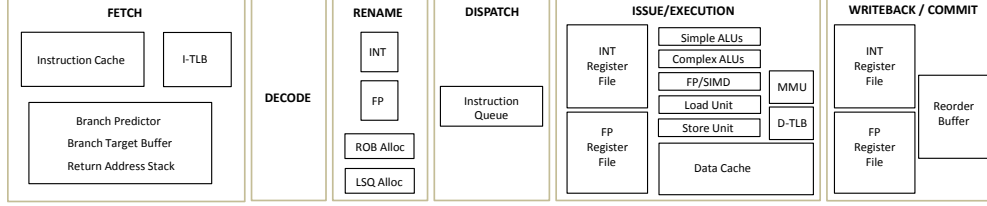


Figure 3.1: Core Pipeline Architecture.

3.1.1 Core Pipeline Calibration

A typical out-of-order (OoO) pipeline architecture model is depicted in Figure 3.1. The front-end of the pipeline (from the FETCH until the DISPATCH stage) works in-order until the RENAME stage. The RENAME stage renames (i.e. maps) the architectural source and destination register IDs into physical Register File IDs and insert the instructions into the Issue Queue (also referred to as Reservation Station).

Instructions wait in the Issue Queue until they are selected for execution. When an instruction is selected, it reads its dependent instructions via the wake-up logic, reads its source operands from the physical Register File, and is executed in Functional Unit(s). After execution, an instruction’s result is broadcasted via a bypass network such that any dependent instruction can use it immediately. The instruction retires once it reaches the head of the Reorder Buffer (ROB), and updates the corresponding Register Alias Table.

In order to calibrate pipeline performance, we first use a variant of the inner-most micro-kernel of GEMM that performs a *rank-1 update* [128]. The microbenchmark is designed to evaluate the peak performance of the

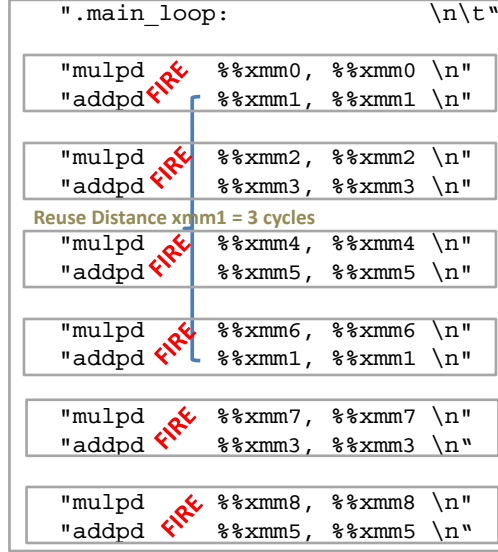


Figure 3.2: Microbenchmark for testing instruction latencies.

pipeline. It is modified to exclusively perform register-to-register computations, i.e. without any load/store operations from L1 to registers. This allows us to investigate the execution behavior of the OoO pipeline without any potential interference of the load/store unit and cache system. Using such an instruction stream, we compare the peak performance produced by the pipeline model and the targeted i7 machine. Any discrepancies are a sign of potential issues in the pipeline model.

Figure 3.2 illustrates our microbenchmark that repeatedly executes SSE multiply and add (*mulpd* and *addpd*) instructions in a loop, similar to a subset of rank-1 update operations in GEMM. SSE instructions are intensively used in our targeted HPC applications. Each *addpd* and *mulpd* operates on *xmm* registers that each hold 128-bit values and can operate on two double-

precision values in a cycle. Furthermore, the i7-920 can perform four double precision (DP) FLOPs/cycle [2]. Consequently, the microbenchmark is expected to run at near-peak performance of four DP-FLOPs/cycle (FPC).

3.1.1.1 Instruction Latencies

At the core level, instruction latencies are one of the first parameters to calibrate against the targeted machine. Any mismatch will directly impact the accuracy of the processor model.

One of the biggest challenges in calibrating instruction latencies is observability. In the simulator, it is trivial to measure the cycles an instruction takes to complete as the behaviour of the model is fully visible and tractable. However, in a real CPU, there is no easy way to measure how many cycles an instruction requires to finish. Hence, a systematic approach to make instruction latencies observable is required.

We explain our approach using Figure 3.2 as a motivating example. We define the reuse distance of destination registers as the number of distinct references between two references to the same location. Therefore, the reuse distances of *addpd* and *mulpd* instructions are 3 and 6 cycles. Consequently, if the corresponding latencies for *addpd* and *mulpd* are less than 3 or 6, respectively, the microbenchmark will execute in nearly 4 FPC (i.e, peak performance in the target machine). Otherwise, *addpd* and *mulpd* would have to stall and wait for the dependency to resolve, resulting in a degradation in performance. We measured the microbenchmark shown in Figure 3.2 to

execute nearly in peak performance on the i7-920, achieving a FPC of 3.94. We then varied the reuse distances in the microbenchmark until performance degrades to further determine instruction latencies. Using microbenchmarks with varying reuse distances, we measured *addpd* and *mulpd* latencies to be 3 and 5 cycles, respectively.

By contrast, the MARSSx86 model, with a similar configuration as the i7-920, could only achieve an FPC of 2.4. Per-cycle event and log traces reveal that the MARSSx86 pipeline is not able to commit at a rate of 4 FPC because an *addpd* instruction was waiting on the previous *addpd* instructions to retire. We further observed that the instruction latencies of *addpd* and *mulpd* were hardcoded to 6 cycles in MARSSx86. We adjusted latencies to their targeted values of 3 and 5 cycles, respectively. We utilized a similar approach to calibrate the instruction latencies of other SSE instructions. We first determined potential latency candidates, based on the technical documentations provided in [53]. We then crafted similar microbenchmarks with varying reuse distances for these instructions to measure latencies on the target machine and adjust the MARSSx86 model accordingly.

As described in the above example, the key points in calibrating instruction latencies are as follows: First, coming up with a baseline microbenchmark that would run in peak performance in the pipeline. Second, using this baseline, calibrating most frequent arithmetic instructions latencies by including those instructions in the microbenchmark. Finally, one can determine the actual instruction latencies in the targeted machine by modifying the reuse

distances in the microbenchmark until the performance deviates from the peak. By employing this generic methodology, one can easily come up with microbenchmark variants in order to determine instruction latencies for any given x86 instruction set.

3.1.1.2 Pipeline Width

Once instruction latencies are calibrated, we move to a higher level in the pipeline hierarchy. In a pipelined system, every stage should ideally have a uniform production and consumption rate. Any non-uniform rate makes the system operate at less than peak performance. For example, in the pipeline architecture depicted in Figure 3.1, if the processor FETCHes up to four instructions, but can only DECODE two instructions at a time, the system throughput will be limited to two. In this scenario, the DECODE stage is the performance bottleneck.

The pipeline width varies from processors to processor between 2-wide and 8-wide. Hence, validation of the system width is an integral part of the pipeline calibration. In order to determine the system width, we use the following approach: In general, pipeline performance increases as a function of system width, assuming a sufficiently independent instruction stream and no cache miss penalties. Using such an instruction stream, one can vary the system width gradually and observe how performance improves. Note that the system width we refer to here is not merely the issue width. Performance bottlenecks can equally reside in other stages. As such, by width we refer

here to the parameters of the entire system, including front-end and back-end. Hence, changing the width to four means that all fetch, rename, dispatch, issue, commit, and writeback widths are set to four accordingly. With this, the pipeline performance should increase as the system width grows.

We run the entire GEMM benchmark to increase test program complexity and evaluate the pipeline width across a mix of instructions each progressing differently through different stages of the pipeline. The GEMM benchmark employs a combination of a broad range of compute-intensive and vectorized instructions. We make minor modifications of the benchmark such that every load and store related instruction is omitted, thus avoiding interference of the cache system. The real i7 achieves a performance of 3.94 FPC for this benchmark. By contrast, we observed that the latency-calibrated MARSSx86 could only reach an FPC of 2.45. In addition, increasing the pipeline width of MARSSx86 did not bring expected performance improvements. In our special case, the problem was due to MARSSx86 internally hardcoding the issue width to four. After removing the hardcoded assignment and making issue width configurable based on the provided config file, this problem was fixed. We then observed that with a pipeline width of seven or eight, the MARSSx86 core model produced 90% and 102% of i7-920 peak performance, respectively. Therefore, a pipeline width of seven or eight could be assumed to give a reasonable configuration assuming there is no cache miss penalty. However, this is not truly the case. For example, for variants of the original microbenchmark with large enough reuse distance, a width of seven would artificially inflate

	i7-920	MARSSx86	Gap
# of Instructions	229 Million	225 Million	2%
# Micro-Ops	229.2 Million	465 Million	202%

Table 3.2: GEMM micro-op breakdown comparison.

performance beyond the peak FPC of 4 that the real machine is capable of. In the following, we explain the root issue and the key insights that support it. The main reason for this mismatch is because in MARSSx86, the system width refers to a granularity of micro-operations (micro-ops) while a real x86 machine defines widths at instruction granularity. Hence, if we apply a similar width configuration as in the i7, MARSSx86 will achieve a lower number of instructions per cycle (IPC) in cases where it has to execute more micro-ops than instructions compared to the real x86 machine.

3.1.1.3 Instructions and Micro-ops

Table 3.2 illustrates the breakdown of instructions and micro-ops of GEMM executing on the i7-920 and MARSSx86. As expected, the number of dynamically executed instructions matches closely within the range of possible measurement errors. By contrast, an x86 instruction is internally broken down into simple RISC-like micro-operations, where the gap between micro-ops executed in the i7-920 and MARSSx86 is 202%. We further examine the detailed micro-op breakdown. Table 3.3 presents the breakdown on MARSSx86, where 96% of the micro-ops belong to SSE instructions. Moreover, 94% of the SSE micro-ops belong to multi-op instructions, which are typically broken down into two smaller micro-ops. From this observation, and from data in Ta-

SSE		NON-SSE	
96%		4%	
Multi-Op	Single-Op	Multi-Op	Single-Op
94%	6%	87%	13%

Table 3.3: Micro-ops of GEMM on MARSSx86.

ble 3.2, we infer that the i7-920 has more single-op SSE instructions than the MARSSx86 core model.

In order to account for this discrepancy, we considered the following options:

1. Altering the internal micro-op implementation in the MARSSx86 core model, which arguably is the ideal option. However, there is no publicly available documentation on how each x86 instruction is broken down into micro-operations.
2. Fixing the processor pipeline width statically to seven or eight to achieve a reasonable performance match for the larger GEMM kernel. However, opting for this approach would make the system configuration of the core model artificially unaligned to the system configuration of the real machine. If different applications have different single- to multi-op instruction ratios, the core model would not correctly capture the behavior.
3. Keeping the base system configuration aligned with the real target machine parameters, but *dynamically* adjusting the *execution width* to account for multi-op instructions. In this way, the adjustment in the core model will be orthogonal to the system configuration.

micro-op is ready to retire, we check whether it belongs to the SSE class. If not, we normally commit and increment the commit counter. However, if it is of SSE type, we first check whether it is a multi-op SSE instruction. Every micro-op in the MARSSx86 model will have *uop.som* and *uop.eom* flags, indicating the first and last operation in a sequence of micro-ops for one instruction. If an instruction is a single-op instruction, both flags will be set to 1. We alter the commit logic flow as follows: If *uop.som* is 1, i.e. it is either the first micro-op of a multi-op instruction or a single-op instruction, we commit and also increment the commit counter. In all other cases, we commit but do not increment the counter. In this way, a multi-op SSE instruction is treated as single micro-operation at commit time. With this, we are able to closely match performance of the real machine for the larger GEMM kernel while keeping the basic pipeline width set at its correct value of four.

3.1.2 Cache System Calibration

In general, there are multiple levels of cache hierarchy in modern processors. A unit validation at each level is required before calibrating the complete cache system.

Our targeted machine has three levels of caches. We examine L1 cache access behaviour first. We run the GEMM benchmark now including load and store instructions. We collected statistics of the number of L1 accesses both from MARSSx86 and the targeted machine. The resulting L1-D statistics show discrepancies. In particular, the number of accesses from MARSSx86 is

	Target i7	MARSSX86
#of L1-D Accesses	12,583,748	25,165,871
<pre> .main_loop: \n\t" "movaps (%rax), %%xmm0 \n" "movaps (%rax), %%xmm1 \n" "movaps (%rax), %%xmm2 \n" "movaps (%rax), %%xmm3 \n" "movaps (%rax), %%xmm4 \n" "movaps (%rax), %%xmm5 \n" "movaps (%rax), %%xmm6 \n" "movaps (%rax), %%xmm7 \n" "movaps (%rax), %%xmm8 \n" "movaps (%rax), %%xmm9 \n" </pre>		

Figure 3.4: Microbenchmark for testing L1-D accesses.

74 million, while the targeted machine issued only 46 million L1-D accesses.

The previous experiment gave us a key insight that every SSE instruction is broken down into two micro-ops in MARSSx86. In our previous microbenchmark, we used *addpd* and *mulpd* as a motivating example. In reality, the GEMM benchmark uses a wide variety of SSE instructions, including load-store type of instructions. For such instructions, one load is executed as two load micro-ops. As such, accessing the L1-D cache with a SSE load instruction would logically generates two accesses to the cache.

To confirm this, we crafted a simple microbenchmark similar to previous ones but consisting of SSE load instructions that move data from the cache to *xmm* registers. Figure 3.4 shows the microbenchmark and the corresponding

	MARSSx86			i7-920		
	L1-D	L2	L3	L1-D	L2	L3
Total Accesses	43.3M	4.7M	4M	46.2M	4.8M	207K
Miss Rate	11%	85%	3%	10%	4.5%	11%

Table 3.4: Cache miss-rates on each level.

L1-D accesses when executed in MARSSx86 model and the target machine. As shown in the figure, the number of L1-D accesses in MARSSx86 are twice as high as in the target machine. This confirms that accessing the L1-D cache with a SSE load instruction creates two different accesses in MARSSx86, whereas it issues only one access in the target machine. We investigated other variants of SSE load and store instructions and found similar mismatches in the number of L1-D accesses.

While this does not affect simulated performance, we made the following changes in the MARSSx86 model to address reported statistics: Whenever encountering a load/store instruction in the pipeline, we first identify whether it is SSE, and then adjust the runtime statistics counter. Whenever there is a cache accesses from an SSE instruction, the cache access will be treated only once. Hence, the statistic collections are aligned with the target machine.

We use the calibrated L1 cache model to further investigate other levels of the cache system. Table 3.4 shows the statistics of cache miss-rates at each level in MARSSx86. Miss-rates are measured when executing GEMM. We observe that the L2 miss rate in MARSSx86 is significantly higher than in the real machine, causing a significant performance mismatch.

To find the cause for this issue, we collected the number of competing

Set	# of way	# of Competing Blocks	Miss	Hit
# 0	8	19	18,387	2,291
# 1	8	46	15,637	882
...	...	...	...	...
# 56	8	42	15,754	921

Table 3.5: L2 set breakdown when executing GEMM.

blocks, defined as blocks that are of the same set, and their corresponding miss and hit numbers. Table 3.5 shows a snippet of the set breakdown. It is evident that competing blocks cause a significant miss rate per set. By profiling the miss address patterns, we find that the majority are from neighbouring addresses. Whenever a miss requesting block N occurs, then it is likely that the following misses are for blocks $N+1$, $N+2$, $N+3$, $N+4$, etc. The i7-920 tackles this problem by the support of its hardware prefetcher. Thereby, the capacity misses in the relatively small L2 cache can be avoided.

We subsequently implemented a stream prefetcher in MARSSx86 to model the hardware prefetcher in the real i7. Once a miss occurs, our model will aggressively prefetch 16 neighbouring cache blocks. As our results will show, using this stream prefetcher, we succeeded in improving the performance of the L2 cache to closely match the real system.

3.2 Results

In this section, we discuss calibration results. We modified MARSS to account for the issues we covered in the previous section. We further evaluated the refined model to quantify the impact of this calibration on heterogeneous

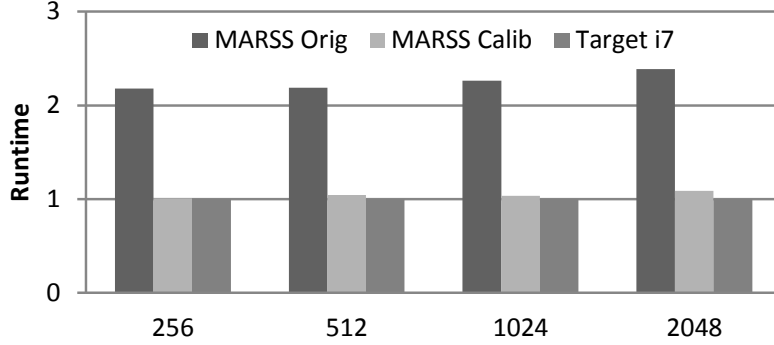


Figure 3.5: Normalized DGEMM runtime on original and calibrated simulators.

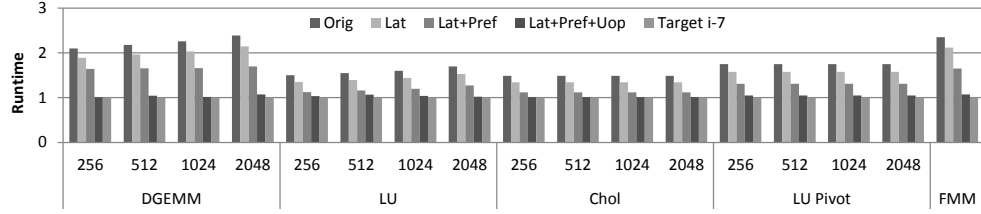


Figure 3.6: Runtime improvement of five benchmarks on each calibration step.

system design.

We use four applications that are representative in high-performance computing: GEMM, FMM, LU and Cholesky factorizations. We also evaluated the calibrated model on general-purpose SPEC2006 [65] and PARSEC [24] benchmarks in order to further validate its accuracy. In all cases, overhead of calibrations was observed to be negligible at less than 2% reduction in simulation speed.

3.2.1 Baseline Refinement

Figure 3.5 shows the runtime of GEMM for different input matrix sizes on the original MARSS, the calibrated MARSS, and the target i7. The original

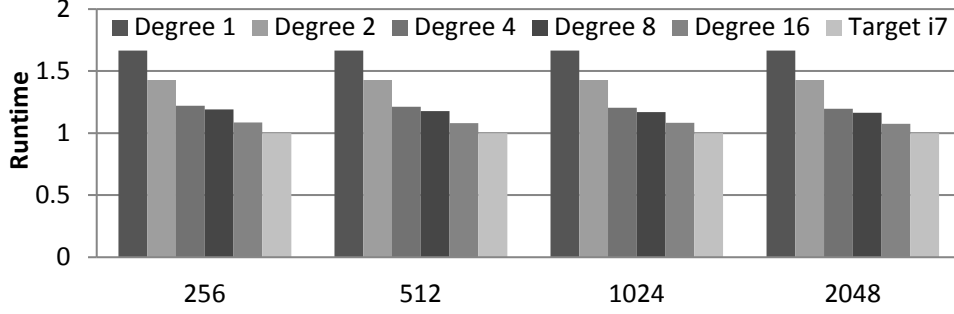


Figure 3.7: Normalized DGEMM runtime improvement as a function of prefetch degree.

MARSS model predicts that the execution takes twice as long as in reality. By contrast, the calibrated MARSS model captures the behavior of the target i7 accurately with less than 5% error.

Figure 3.6 illustrates the accuracy improvements when running the five benchmarks with varying problem sizes for progressive calibration steps. *Orig* refers to the original MARSS, *Lat* to the latency-calibrated MARSS, *Lat+Pref* to the *Lat* version implemented with a simple stream prefetcher, and *Lat+Pref+Uop* to *Lat+Pref* with modified commit logic. A MARSS model with calibrated instruction latencies improves accuracy roughly by 10%. A stream prefetcher further boosts accuracy up to 40% overall. Finally, the fully calibrated MARSS model captures the behaviour of the target machine very accurately. It achieves near-identical runtime with the target i7, with less than 7% error across all HPC applications.

We also observed the impact of the prefetch degree on the simulated performance. Figure 3.7 shows the normalized performance of the MARSS

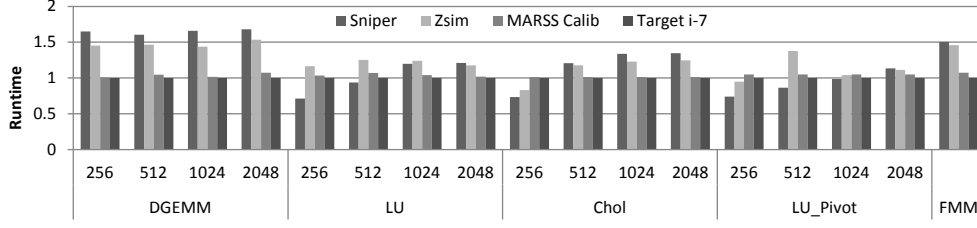


Figure 3.8: Normalized runtime of different simulators on five benchmarks.

model employing different prefetch degrees. Increasing the prefetch degree generally increases the core performance and simulator accuracy. In particular, a prefetch degree of 16 on the MARSS model closely matches the performance of the targeted i7. This justifies our selection of realizing a stream prefetcher with a degree of 16.

Figure 3.8 shows the accuracy of our calibrated MARSS as compared to Zsim and Sniper. As mentioned before, gem5 is not able to run these benchmarks in their optimized forms due to its limited x86 instruction support. Among the supported simulators, the calibrated MARSS provides the best accuracy compared to the target machine. Zsim and Sniper in general are 20-50% less accurate due to their abstracted timing model. However, as a consequence, they possess a significantly faster simulator speed than MARSS.

To further evaluate the accuracy of our calibrated simulator, we compared performance when running SPEC and PARSEC benchmarks (Figures 3.9 and 3.10). We ran all C/C++ benchmarks for SPEC2006. For PARSEC, we excluded facesim, vips, and x264 benchmarks as they failed to successfully execute in MARSS. In this work, we targeted calibration for HPC applications.

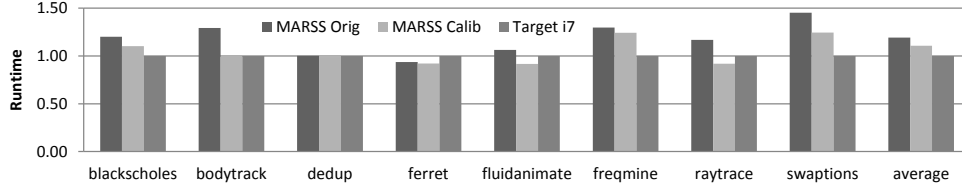


Figure 3.9: Normalized runtime of PARSEC benchmarks running on original and calibrated simulators, with respect to hardware platform.

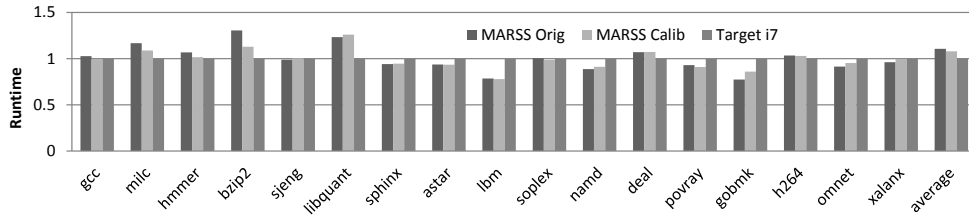


Figure 3.10: Normalized runtime of SPEC benchmarks running on original and calibrated simulators, with respect to hardware platform.

As results show, our calibration improves accuracy even for many general-purpose benchmarks while not significantly affecting results in cases where the original MARSS already matches well. An uncalibrated MARSS executes PARSEC and SPEC benchmarks with an average error rate of 19% and 9%, respectively. By contrast, the calibrated MARSS improves PARSEC and SPEC accuracy to 11% and 7% on average. Nevertheless, truly targeting other application domains would require applying our calibration methodology using representative microbenchmarks that would likely improve the accuracy even higher.

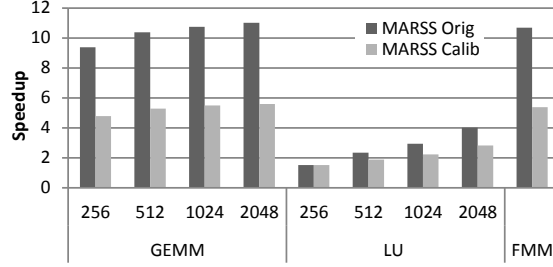


Figure 3.11: Speedup of benchmarks using original and calibrated simulators.

3.2.2 Impact on Heterogeneous Design

To quantify the impact of simulator calibration on heterogeneous system design, we integrated a publicly available cycle-accurate simulation model of a Linear Algebra Processor (LAP) [105] as an example accelerator into the system. We modified applications to offload the GEMM kernel operation to the LAP with the help of a device driver. The CPU thereby copies all the necessary data to the main memory before invoking the accelerator. The LAP then performs the GEMM and interrupts the host CPU when it is finished.

Figure 3.11 shows the overall system speedup when running GEMM, LU, FMM applications with all GEMM kernels offloaded to the LAP. An uncalibrated MARSS achieves average speedups of 10.3x when offloading a complete GEMM. However, when the calibrated core model is employed, the speedup is reduced to 5.3x on average.

For LU factorization, the application consists of three kernels, one of which is GEMM. When the problem size is relatively small, the speedup difference between the original and the calibrated model is small. However, as the

problem size grows beyond 512, the time spent in GEMM rises significantly. In particular, for the 2048 problem size, the original MARSS estimates the speedup to be 4x. However, when the calibrated baseline model is integrated, the estimated speedup is reduced to 2.8x. If the accelerator was designed to have only a smaller gain by itself, due to the cost of integration overheads, an uncalibrated model could potentially project a system-wide speedup even though none or a negative one exists in reality.

We observe similar behavior in FMM. Similar to LU, FMM consists of several kernels with one of them being GEMM. An unrefined MARSS core model reports a 10.3x speedup. However, using the calibrated model, estimated speedup is reduced to 5.3x.

3.3 Summary and Conclusions

We presented a systematic methodology for calibration of cycle-accurate x86 simulators and the resulting impact on heterogeneous system design. Results show it has on average less than 10% error including SPEC and PARSEC benchmarks. We further showed how an unrepresentative baseline CPU model can significantly over/under-estimate speedup when analyzing and evaluating heterogeneous system designs. This can result in misleading design decisions. We conclude that an accurate simulator baseline is particularly paramount when modeling heterogeneous systems.

Chapter 4

Accelerator Integration/Data Movement Tradeoffs

In this chapter, using the calibrated simulator, we study accelerator integration and data movement tradeoffs from a system-level perspective. We consider different architecture coupling scenarios in the memory hierarchy, chiefly off-chip and on-chip placement. We present a comprehensive algorithm/architecture co-design case study and its corresponding data movement tradeoffs for a complex HPC application consisting of multiple interdependent kernels executing on a host CPU assisted by a custom hardware accelerator. Specifically, we study these tradeoffs for an application that consists of multiple compute-intensive accelerated kernel calls that are combined with complex data-structure accesses and memory reshuffling. Such workload patterns are common in many HPC applications but as an exemplary application we study the Fast Multipole Method (FMM) [47] widely used in computational physics and machine learning [48, 26, 92, 85].

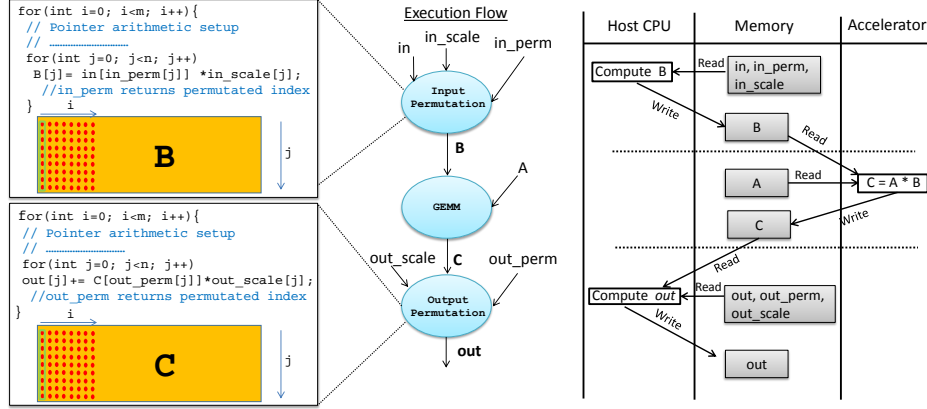


Figure 4.1: Data flow graph (DFG) of U-list execution and data movement patterns.

4.1 FMM Application and its Execution Flow

We use the Fast Multipole Method library PVFMM [47, 92] a well-optimized library that has demonstrated up 60% efficiency on x86 architectures. We consider the near-interaction phase of volume FMM (a particular variant of FMM methods), referred as the “*U-list*” operation. In well-optimized FMM codes the U-list accounts for 50% of execution time.

Figure 4.1 shows the three main parts in the U-list computation, namely the input permutation, the main GEMM Kernel, and the output permutation. The preprocessing or the input permutation step applies a permutation and scaling operator K to the input array, labeled “*in*” in Figure 4.1, and assembles the result into the matrix B . This step mainly consists of light-weight computation along with data shuffling and pointer chasing. The main GEMM computation kernel then uses matrix B together with matrix A (which resides

in memory) as inputs to a GEMM that produces matrix C as the result. The width of matrices B and C is equal to the number of times the matrix block A and this can be hundreds or even thousands of columns wide. Since the GEMM requires a significant amount of compute resources, it is the main candidate for accelerator offloading. Once the GEMM is complete, the matrix C is sent to the host CPU which performs the post-processing computation to assemble the final result in the array, labeled “*out*”. This again involves a combination of light-weight computation with data reshuffling and irregular pointer chasing.

This workflow represents typical program patterns observed in HPC applications consisting of a mix of irregular data transformation and compute-intensive stages. Throughout this chapter, we use the U-list as a representation of application patterns requiring significant data exchanges between irregular code executing on host CPUs and compute-intensive kernels offloaded to accelerators.

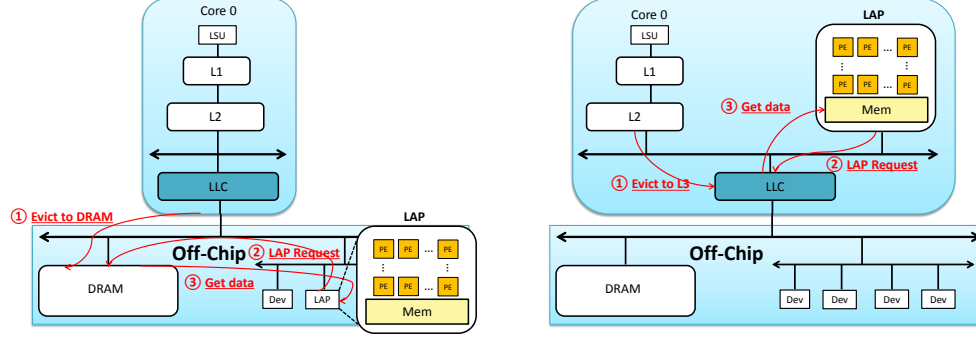
4.2 Accelerator Coupling

We discuss different accelerator coupling options from a system architecture perspective. Based on their granularity of coupling, hardware accelerators can be fundamentally classified as *tightly coupled* or *loosely coupled* [33]. Tightly coupled accelerators are attached to the main processor pipeline as one or more specialized functional units (FUs) located inside or very close to the main processor core. They typically realize fine-grain, short-latency com-

putations, such as floating-point, vector or SIMD operations. As such, they generally share key resources with the processor core (e.g., register file, MMU and L1 data cache) while in some cases possessing their own additional internal storage. Similar to co-processors, tightly coupled accelerators require special instruction set architecture (ISA) extensions to manage their operation.

By contrast, loosely coupled accelerators are located outside of the main processor core and connected via cache, memory or I/O buses, or over on- or off-chip interconnect. Loosely coupled accelerators are often also called *coarse-grained*, due to the fact that they typically operate on coarser invocations of complete tasks, kernels or even an entire application. They typically operate asynchronously, where triggering of their operation and synchronization of completion and results is achieved through memory-mapped I/O, polling or interrupt mechanisms. Proper communication and synchronization with such accelerators is accomplished through device drivers for handling of interrupts, cache flushes and/or memory copies. GPUs and network/cryptography processors [82, 17] are examples that belong to this class.

In this work, we focus on integration tradeoffs of loosely coupled hardware accelerators. We use the LAP [105] as an example for coarse-grain custom hardware acceleration of GEMM operations in the FMM U-list. We study different LAP coupling scenarios and their implications and system tradeoffs. Specifically, we investigate two different cases (Figure 4.2): (1) the accelerator is externally coupled as an off-chip device sharing DRAM with the host CPU, and (2) the accelerator is located on the same chip as the host CPU



(a) Off-chip coupling at shared DRAM. (b) On-chip coupling at shared LLC.

Figure 4.2: Accelerator coupling at different levels of the memory hierarchy.

sharing the last level cache (LLC). In this study, we assume a modern, high-end x86 CPU with a typical aggressive wide-issue and out-of-order processor micro-architecture.

4.2.1 External Off-chip Coupling at Shared DRAM Level

In this scenario (Figure 4.2a), the LAP is connected to the external, off-chip system bus. The LAP accesses all its necessary input and output data in the DRAM through the system bus and a DMA controller. To avoid coherency protocol overhead, we assume that LAP and CPU are non-coherent. Instead, as shown in Figure 4.2a, before the LAP attempts to read inputs from the memory, if there exists any dirty data in the host CPU caches, the software has to explicitly manage coherency and tell the CPU to evict relevant data from its caches to DRAM. For example, this can be done via the “*clflush*” instruction in the x86 ISA. Once data is synchronized and the LAP has been triggered, it

will operate directly on data in DRAM. The LAP will issue memory requests to DRAM, subject to DRAM latency. The LAP will prefetch data and overlap computation and communication through its internal scratchpad memory. At the end of one invocation, the LAP will have stored all output data back to DRAM, from where it can be accessed by the CPU. Before accessing the LAP results, the CPU driver will ensure that all previously cached copies are invalidated.

In a PIM setting, the accelerator would be coupled and integrated directly in or near DRAM. The main benefit is increased bandwidth for DRAM accesses by the accelerator. However, GEMM on the LAP is compute-bound and not bandwidth-limited, i.e., we would not expect to see any performance benefits from such in-memory integration. As such, we exclude this as a separate case in our studies. From an energy perspective, our simulation models currently do not explicitly account for off-chip interconnect energy. This underestimates energy costs for DRAM accesses. At the same time, in a PIM case, off-chip interconnect overhead would only be incurred for DRAM accesses by the CPU, which would in turn again be closer to our reported results. In literature, dynamic energy costs of 20pJ/bit are reported for off-chip DRAM interfaces in current 40nm technology [79] on top of around 60pJ/bit for actual DRAM accesses including all dynamic read/write and activation energy as modeled in our setup. In addition, both interconnect and DRAM incur similar proportions of static versus dynamic power [72]. Our reported DRAM energy results can thus be scaled accordingly to account for additional interconnect

costs in either PIM or non-PIM scenarios.

4.2.2 Internal On-chip Coupling at Shared Cache Level

To avoid communication via external DRAM accesses, we couple the LAP at the LLC by connecting it to the internal bus network between L2 and LLC—in the same way as other cores are interconnected in multicore systems. As shown in Figure 4.2b, the LAP shares the LLC with other cores. It will access all data through the LLC. Since the LAP itself explicitly manages all memory and internal scratchpad accesses, we want to avoid the overhead of it having to participate in the L2 coherency protocol. Instead, the LAP expects necessary data to be in LLC or DRAM when it starts computing. However, since data might be cached in the CPU’s L1 or L2, the CPU now has to explicitly evict data to the LLC. In our setup, caches are non-inclusive and write-back. The *clflush* instruction by default evicts the targeted block all the way to DRAM. If the block exists in the LLC, it will be invalidated and further evicted. In our scenario, we wish to manage the eviction and keep or store the block in the LLC by not invalidating and potentially updating an already existing, or by allocating a new LLC cache line that potentially replaces and evicts others. There are two plausible options to support this: (1) adding a special instruction in the ISA that handles eviction only to the LLC, or (2) setting a configurable register in the LLC controller to identify if a block should be invalidated or kept/stored when software eviction happens. We opted for the latter implementation due to its simplicity and transparent

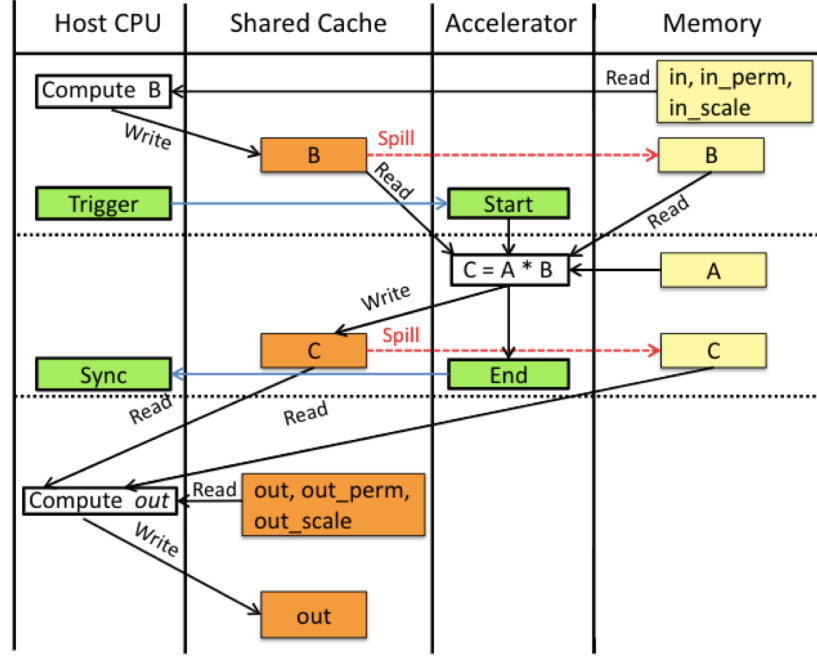


Figure 4.3: Data movement pattern when sharing LLC.

change from the software perspective.

Figure 4.3 summarizes the data movements for this case. The CPU evicts necessary data to the LLC (B). Afterwards, the LAP accesses the LLC to get its input data to operate on. Note that, depending on the size of B and the size of the shared cache, some fraction of B might get evicted and spilled to the main memory before it can get used by the accelerator. In this case, the accelerator has to refetch/reload some fraction of B from main memory. Similarly, the LAP writes results (C) back to the LLC, which, in case of capacity misses, can lead to previous results being evicted to DRAM.

Meanwhile, a closer integration with potentially lower distance and latency would be to couple the accelerator to the CPU at the level of L1 or L2 caches and buses. While we considered this option, it is less appropriate for loosely coupled coarse-grain accelerators. Such accelerators usually afford a greater area budget than tightly coupled ones. This allows them to operate with complex data paths and internal memory for acceleration of complete tasks or kernels. Coupling a greater area budget close to the main CPU would pose challenges in achieving timing closure and meeting tight clock frequency constraints set for latency-sensitive L1 and L2 caches. Moreover, buses and interconnect of private L1 and L2 caches are usually not designed to support multiple connected cores. Furthermore, when task-level accelerators are allowed to access and share a private L1 or L2, the capacity pressure of the relatively small caches increases. Hence, potential performance degradation resulting from cache pollution and cache trashing can negate acceleration benefits, especially when executing on CPU cores and accelerators concurrently. A number of recent works have proposed acceleration engines placed in between L1 and L2 [11]. However, for reasons outlined above, such approaches do not target task/kernel level invocations, but are restricted to acceleration at finer instruction-level granularity.

4.3 HW/SW Co-Optimization

We further discuss HW/SW co-optimization opportunities to exploit aforementioned coupling options and optimally take advantage of the under-

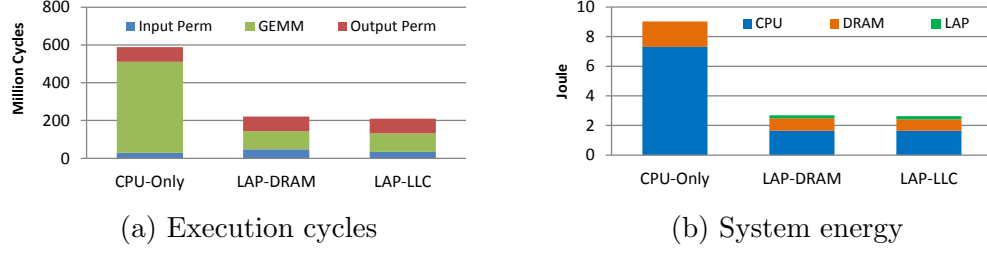


Figure 4.4: Comparison of CPU-only with CPU + LAP, total execution cycles and system energy ($q = 8$). While LAP offers significant GEMM acceleration, we observe that simply coupling the LAP on LLC does not make remarkable difference compared to the LAP on the DRAM. More effort is needed to unlock additional gains.

lying system architecture.

4.3.1 GEMM Acceleration

In a basic setup, we first evaluate acceleration benefits by simply offloading all GEMM calls in the U-list to the LAP hardware accelerator while input and output permutations remain on the CPU. Figure 4.4a and Figure 4.4b compare total system cycles and energy consumption of a pure software solution to GEMM hardware acceleration with a LAP coupled at either DRAM (LAP-DRAM) or LLC (LAP-LLC) levels. Basic hardware acceleration provides $1.4\times$ - $4\times$ speedup and $2\times$ - $5.6\times$ energy savings across varying q , predominantly due to $5\times$ - $6\times$ performance and energy gains from GEMM acceleration on the LAP alone, where FMM parameters (see Section 4.4) determine the proportion of time and energy spent on GEMMs in the U-list. At the same time, due to the extra data flushing and communication overhead, cycles spent in input permutations increase. However, results show that in

such a basic hardware/software setup, coupling the LAP at LLC level only yields minor performance and energy improvement compared to DRAM coupling. The main reason is that the total data exchanged by the CPU and the accelerator during one GEMM invocation can reach more than 3x the total capacity of the 8MB LLC. This requires a significant amount of shared data written to the LLC to be evicted and spilled to DRAM, from where the other component has to read it again. In other words, a large fraction of data is exchanged through the DRAM in either case, i.e., even when the LAP is coupled to the LLC.

4.3.2 Cache-Aware Blocking and Memory Hiding

In order to minimize data movement and unnecessary round-trip transfers to DRAM with LLC coupling, all data should ideally be exchanged between CPU and the accelerator exclusively through the LLC. As Figure 4.1 shows, in the U-list, CPU and LAP communicate by exchanging matrices B and C during GEMM calls. Since the LAP operates on both matrices simultaneously in a streaming fashion, ideally both matrix B and matrix C should remain resident in the LLC for the duration of such calls.

4.3.3 Prefetching, Pipelining and Overlapping

We modify the U-list such that the program operates on smaller, blocked versions of matrix B and C whose total size is equal or smaller than the LLC capacity. We applied such a blocking optimization and evaluated its impact

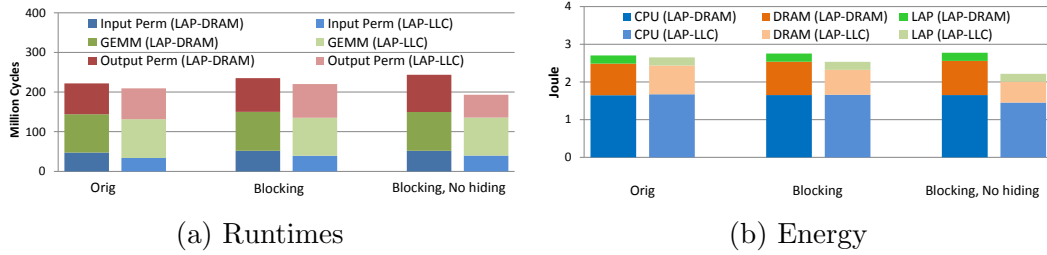


Figure 4.5: Runtimes and energy with blocking and no latency hiding ($q = 8$).

on energy and performance. As Figure 4.5a and Figure 4.5b show, however, contrary to expectations, total execution cycles and energy increase by just 7% and 2% in DRAM coupling— LLC coupling doesn’t help. The overall increase in cycles and energy is mainly due to less available instruction-level parallelism and additional kernel launch overhead when processing matrices in smaller chunks. At the same time, as will be detailed in Section 4.4, the total number of DRAM accesses is significantly reduced in the blocked case with LLC coupling. However, such a reduction does not lead to system-wide performance and energy benefits as would normally be expected. From our findings, this is chiefly due to two reasons: (1) effective hiding of DRAM performance overheads in existing hardware and software implementations; and (2) high baseline idle power of DRAMs, which limits relative gains as we will further quantify in Section 4.4.

From a performance perspective, both accelerators and HPC software executing on modern CPUs are already well optimized to effectively hide memory latencies behind computations or other parallel memory operations. DRAM accesses are hidden and any reductions in DRAM overhead do not

translate into overall performance benefits. The LAP, similar to other modern accelerators, is designed to overlap data (pre-)fetching with simultaneous computations. Therefore, any data fetching costs are well amortized by the computation, regardless of coupling. Our experiments show that the LAP’s communication cost of fetching data from the corresponding cache or memory is merely 10% of the total execution time, where costs differ by only 2% between different couplings.

On the software side, input and output permutations are memory bound. However, they are characterized by ample memory-level parallelism. As such, they are bandwidth but not latency sensitive, where memory bandwidth available to software is dictated by CPU limitations and largely independent of data coupling. To further study potential effects of latency differences on the software side, we modify the output permutation in the U-list to make it more latency sensitive and measure the impact on performance. Modern out-of-order processors are equipped with non-blocking caches and memory miss-level parallelism to allow issuing a new memory request even before the previous one completes. In such processors, latency sensitivity, i.e., the ability to hide memory latencies in software code is mainly a function of dependency chains between instructions in the CPU pipeline. In our pre-optimized U-list implementation, latencies for accessing the accelerator output C in the output permutation are hidden behind access latencies for other elements of C and matrix M . We modify the code by introducing artificial dependencies and thus removing parallelism. Results are included in Figure 4.5a. In the

latency-sensitive (but functionally incorrect) case, DRAM coupling runtimes are increased, but 25% performance improvements are seen in LLC vs. DRAM coupling. This indicates that latency sensitivity can be an important aspect in optimally exploiting accelerator coupling.

As seen by the 7% performance penalty in the previous section, blocking of matrices leads to reduced instruction- and memory-level parallelism, which is fully exposed for accesses to auxiliary matrices stored in DRAM. However, software prefetching can be applied to reduce this overhead. When GEMM operations are executing on the LAP, the CPU is generally idling until the LAP signals completion and the output permutation can resume by processing the C result. However, we can note that auxiliary $M.scale$ and $M.perm$ metadata needed by output permutations is independent of GEMM execution and can thus be prefetched ahead. We modify the code flow to accommodate this optimization, such that the CPU prefetches the required metadata for output permutation while the LAP is performing the GEMM.

Moreover, with blocking of matrices into tasks with smaller granularity, we can apply software pipelining to exploit additional overlapping and parallelism between accelerator and CPU. As shown in Figure 4.6, with input permutation, GEMM, and output permutation tasks denoted as $in(i)$, $gemm(i)$, and $out(i)$, respectively, where i represents the iteration phase of the blocking, while the LAP is operating on one iteration of the GEMM, the CPU simultaneously performs output and input permutations for the portions of B and C from the previous and next iterations, respectively. If the time to

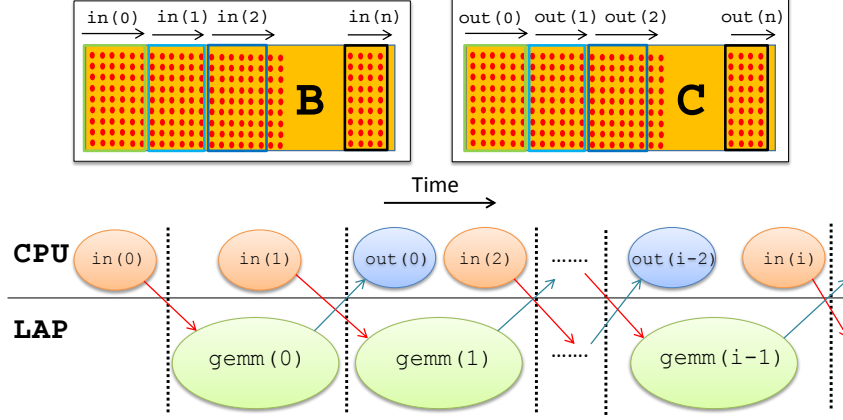


Figure 4.6: Software blocking, pipelining and overlapping.

execute $gemm(i)$ on the LAP is equal to the total time for $in(i+1)$ and $out(i-1)$, the pipeline will be fully overlapped and balanced. In our study, we employ a simple partitioning strategy based on workload characterization of $in(i+1)$, $gemm(i)$, and $out(i-1)$ to further balance blocked and overlapped executions.

4.4 Results

We present our evaluation methodology and comprehensive results across different acceleration options. As discussed before, we adopt FMM U-list as a representative application exhibiting heterogeneous data exchanges between host CPU and an accelerator.

The main FMM parameter in our experiments is q , the polynomial order used in each leaf node of the FMM tree. In all experiments we keep the total problem size N fixed, $N = 2.4e5$. Given q , we adjust the number of leaf nodes in the FMM tree so that N remains fixed. The number of leaf nodes controls

Parameter	Core Model
Fetch Width	4
Dispatch Width	4
Issue Width	5
Commit Width	4
Writeback Width	4
Cache Block Size	64B
L1-I\$ Size	32kB
L1-I\$ Associativity	8-way
L1-D\$ Size	32kB
L1-D\$ Associativity	8-way
L2 Size	256kB
L2 Associativity	8-way
LLC Size	8MB
LLC Associativity	16-way
DRAM	
DDR Type	DDR3-1600
Specification	Micron MT41J256M4

Table 4.1: System configuration.

the second dimension of matrices B in the GEMM (Figure 4.1). The size of B varies according to the total number of leaf nodes and the symmetry direction of the near-field interaction. B has height $q^3/6$ and width approximately $6N/(q^3/6)$ (the approximate factor 6 is because of symmetry and blocking). Hence for problem size $N=2.4e5$ and $q=10$, the size of B is approximately 286-by-5K and for $q=4$ it is approximately 35-by-41K. The order q determines one dimension of the GEMM (the dimension of the square matrix A is $q^3/6$, Figure 4.1). While the size of B (and C) remains constant, the smaller q is, the smaller A will be and the thinner the GEMM becomes (see Section 4.1). Simulations with small q better resemble low-order finite element and other stencil-based codes. Larger values of q are more typical in PVFMM.

We use the previously calibrated cycle-accurate MARSSx86 full-system

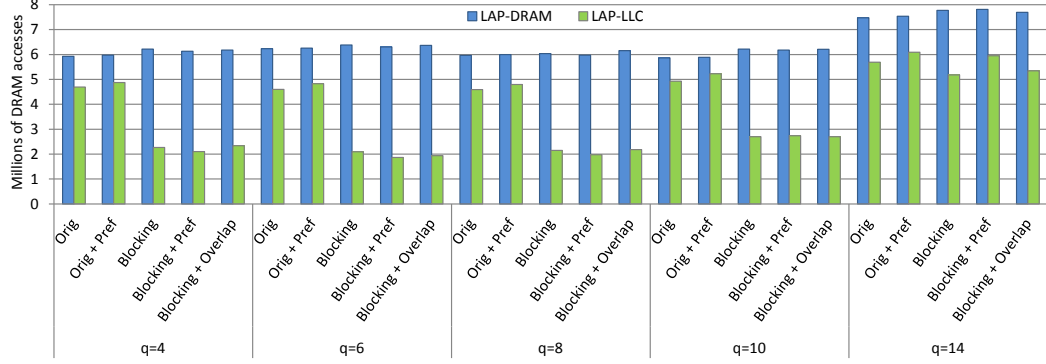


Figure 4.7: DRAM accesses with different integration options across varying q .

simulator for architecture evaluation. As an example of a hardware accelerator, we integrate a cycle-accurate model of the LAP. We feed statistics obtained from MARSSx86 into McPAT [88] to model the total CPU energy. Finally, we attach DRAMSim2 [114] to model an accurate DRAM delay and energy. Table 4.1 shows the detailed system configuration.

4.4.1 DRAM Accesses

We first evaluate impact of integration options on DRAM overhead, which is the main optimization goal affected by different accelerator couplings. Figure 4.7 shows the total number of DRAM accesses in the U-list under different HW/SW coupling options across varying q . We compare the original U-list with basic acceleration of GEMM calls (*Orig*) to a U-list with prefetching (*Orig + Pref*), with cache-aware blocking (*Blocking*), with blocking and prefetching (*Blocking + Pref*), and with blocking and software pipelining *Blocking +*

Overlap) for both DRAM and LLC coupled LAP accelerators. As expected, when the accelerator is coupled as an external device requiring data to be always exchanged via DRAM, none of the software optimizations significantly affect DRAM accesses. DRAM accesses are slightly increased with blocking due to reduced cache and scratchpad locality with smaller matrices.

By contrast, when the accelerator is coupled at the LLC, a cache-aware blocking approach reduces the number of DRAM accesses by 50%. The blocking case is optimized to ensure that all data exchanged between CPU and accelerator fits into the LLC, avoiding DRAM round trips for CPU-LAP communication. However, blocking benefits decrease with higher q . In the $q = 14$ case, due to algorithmic limitation, the smallest chunk that matrices B and C can be partitioned into already exceeds LLC capacity. Hence, significant capacity misses are unavoidable and result in additional DRAM accesses.

Prefetching does not generally affect DRAM access counts, but in the blocked case (*Blocking + Pref*), a minor decrease is observed. This is due to the Least-Recently Used (LRU) cache replacement policy. With prefetching, data already in the LLC is marked as used, which reduces the likelihood of it being evicted before being needed next as compared to other, less recently used lines. However, adding prefetching to a non-blocking approach instead increases the number of DRAM accesses (*Orig + Prefetch*). Due to the large number of capacity conflicts in the non-blocking case, prefetched data is likely be evicted again from the cache before it can even be used.

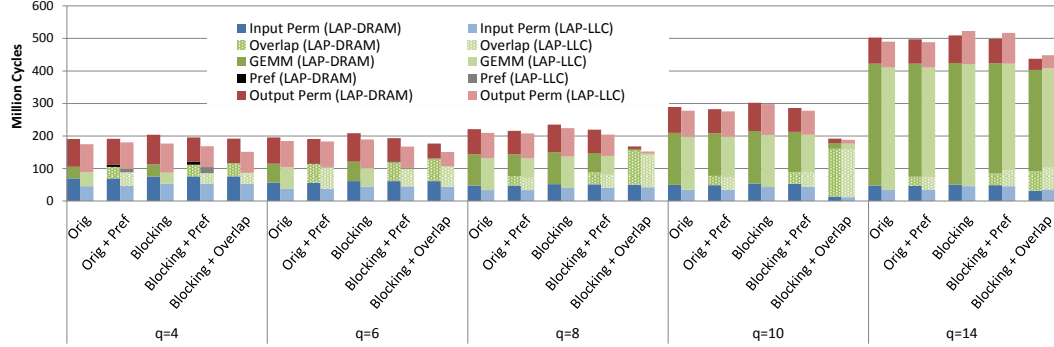


Figure 4.8: Execution cycles with different integration options across varying q .

4.4.2 Execution Time

Figure 4.8 shows execution cycles for a complete U-list execution under different optimization and coupling options across varying q . Total execution time is broken down into cycles spent on different parts, including *Overlap* periods in which the GEMM on the LAP runs in parallel with either prefetching or input and output permutations on the CPU. Note that *Pref* refers to extra cycles spent on prefetching when it can not be completely hidden behind a shorter GEMM.

Among HW/SW optimizations, due to additional overheads, performance is slightly decreased with blocking alone. Prefetching can in turn be applied to compensate such overheads. The best improvement is observed from pipelining and overlapping (*Blocking + Overlap*). At lower q , execution times are dominated by permutations and there are only small gains from overlapping them on the CPU with the GEMM on the LAP. As q grows, first input and then output permutations are hidden behind the increasingly dominating

GEMM (except for non-overlapped invocations needed to fill and flush the software pipeline). At $q = 10$, both permutations are maximally overlapped and balanced with the GEMM. This yields a 32% better performance compared to the non-optimized solution (*Orig*). However, these speedups almost exclusively stem from exploiting available parallelism between accelerator and CPU, i.e. they are not a function of data movement optimizations and, ultimately, do not vary significantly across different accelerator couplings.

Although the number of DRAM accesses in Figure 4.8 strongly depends on accelerator coupling and is significantly reduced with LLC blocking, the impact on performance is generally relatively minor, with a maximal gain of up to 20% comparing LLC to DRAM coupling. As discussed in Section 4.3.2, this is due to effective hiding of DRAM access delays in existing hardware and software. Closer observation show that input permutations consume more execution cycles in *LAP-DRAM* than *LAP-LLC* due to flushing overhead to the DRAM off-chip memory. In case of the GEMM, as it becomes smaller and thinner with smaller q , the computation to memory ratio and data locality in the LAP decrease, making it more memory bound. With ample parallelism, the LAP is not latency sensitive, but coupling it to the LLC with higher bandwidth than DRAM benefits performance resulting in 10% faster GEMM execution in LLC vs. DRAM coupling at $q = 4$ (higher DRAM bandwidth, e.g. through in-memory LAP placement could potentially alleviate such differences). However, at bigger q , GEMM executions differ by less than 2%. Finally, given their latency-insensitive nature, output permutations are largely unaffected by cou-

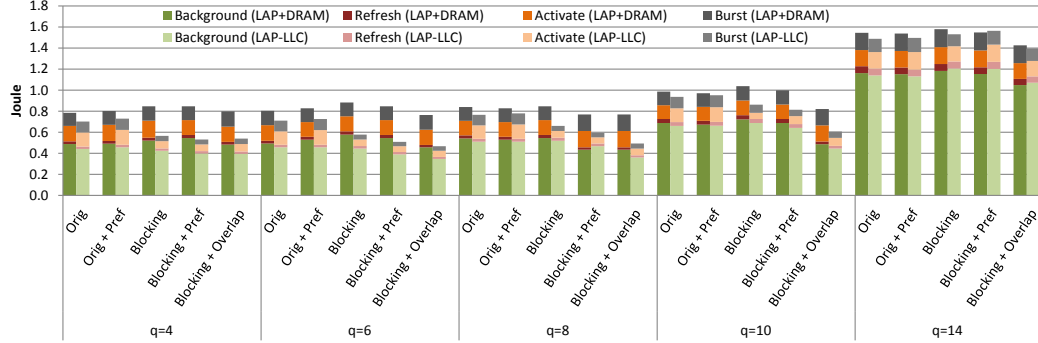


Figure 4.9: DRAM energy with different integration options across varying q .

pling. However, due to increased overhead and cache pressure, performance actually decreases with LLC coupling and blocking at larger q . Coupled with previously discussed blocking limitations, this makes LLC worse than DRAM coupling at $q = 14$.

4.4.3 Energy

We further evaluate potential energy benefits of different architecture and algorithm options. We first focus on DRAM energy savings as the primary aspect that is affected by data movement and accelerator placement/coupling optimizations. Figure 4.9 shows the DRAM energy breakdown across different coupling options and varying q . We break energy consumption down into static background energy due to leakage, energy spent on periodic refresh, and dynamic energy consumption dependent on activation and read/write activity. As can be observed, static background and periodic refresh energy is the most contributing factor. Such idle baseline power contributes from 60% up to 80% of total energy consumption. As such, while dynamic energy reduces by

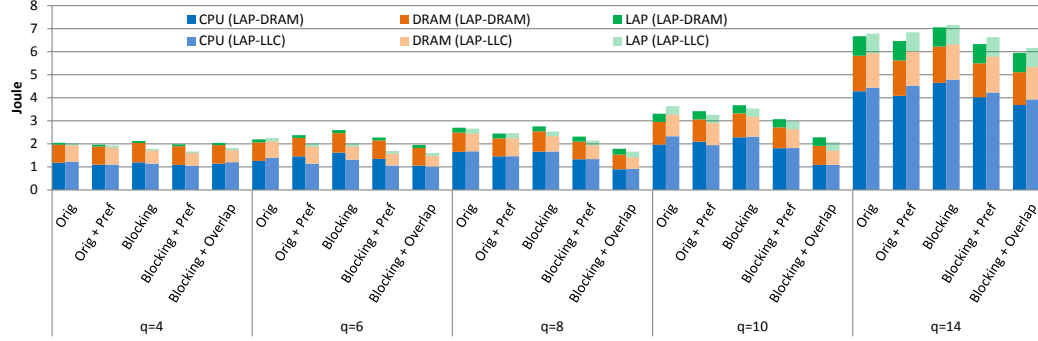


Figure 4.10: Total energy with different integration options across varying q .

50%-60% proportionally to the reduced number of DRAM accesses shown in Figure 4.7 with blocking and LLC coupling, total energy improvements are limited due to small share of dynamic energy.

At the full system level (Figure 4.10), this translates into even smaller savings. The graph shows the total system energy broken down by contributions from individual system components. As can be seen, the CPU consumes the largest portion of energy spent, followed by the DRAM and the LAP with a comparably tiny fraction. Due to additional base overheads and relatively constant energy contributions by the CPU and the LAP, savings in dynamic DRAM energy across different integration options and data movement optimizations have limited effect on total energy. Variations in energy consumption are predominantly due to differences in static idle and leakage power accumulated over different total execution times. This is the case both at the DRAM and full system levels, where the CPU is also the dominant contributor of static leakage power (while its dynamic energy varies little given the same workload). In other words, energy savings are primarily driven by

improvements in performance.

Through a combination of dynamic DRAM energy and execution time, i.e. static idle power savings, when coupling the LAP at LLC and applying aggressive software pipelining and overlapping optimizations, reductions in DRAM data movement between the LAP and CPU reduce the total system energy by up to 17% compared to coupling at the DRAM level. All in all, primarily driven by execution time savings, pipelining and overlapping yields up to 43% lower energy consumption compared to the original solution, but is, again, largely independent of the architecture coupling.

4.5 Summary and Conclusions

To summarize, we study system-level performance and energy saving benefits of hardware acceleration under different placement scenarios and software optimizations for a state-of-the-art Fast Multipole Method (FMM). We investigated accelerator coupling options at the architecture level and proposed several algorithm co-optimizations to maximize acceleration benefits. We showed that, in the context of a complex real-world application, simply placing an accelerator closer to the chip does not always lead to significant performance/energy wins. We demonstrated that an on-chip integration alone without supporting software optimizations provided limited gains. Only when more aggressive software/hardware co-optimizations are applied, additional gains can be unlocked.

Chapter 5

Cache-Managed Accelerator Staging and Pipelining

In the previous chapter, we showed that streaming applications such as FMM can benefit a lot from closer integration. However, significant software modification are required to achieve such benefits. In this chapter, we introduce CASPHAr, a LLC architecture for Cache-Managed, Fine-Grain Accelerator Staging and Pipelining in On-Chip Heterogeneous Architectures. CASPHAr performs automatic, software- and hardware-transparent fine-grain data staging and synchronization between CPUs and accelerators in hardware and achieves similar and/or better benefits than manual optimizations shown in the previous chapter. Using existing cache interfaces and stall mechanisms, CASPHAr provides drop-in synchronization and staging support for on-chip integrated architectures with only minimal software changes.

5.1 CASPHAr Overview

We first provide an overview of the key CASPHAr concepts. We first explain the basic system architecture on which CASPHAr is based. We then introduce the CASPHAr paradigms on top of this base architecture. Finally,

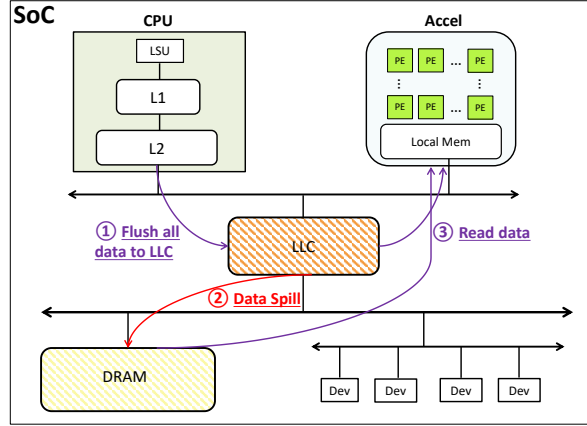


Figure 5.1: Baseline system architecture.

we demonstrate how CASPHAr interfaces with software.

5.1.1 Base System Architecture

Figure 5.1 shows the baseline system architecture we assume in this work. The accelerator shares the LLC with CPU cores. It is coupled at the interconnect network between the L2 cache and LLC in the same way as other cores are interconnected in multicore systems. CASPHAr’s main concepts are independent of the coherency management approach used. For the remainder of this work, we assume accelerators to be non-coherent, which is a system-architecture design commonly used in previous work [58, 40]. Since the accelerator itself explicitly manages all memory and internal scratchpad accesses, it is typically desirable to avoid the overhead of having it participate in the L2 coherency protocol. Previous work has shown that enforcing coherency between CPU and accelerators can have significant overhead, gener-

ating significant bus and cache coherency traffic that potentially limits acceleration benefits [27, 134]. Nevertheless, CASPHAr would equally function and co-exist in a coherence system, subject to potentially additional data lookup energy consumption generated by the coherence/snooping protocol.

In this work, we assume that the accelerator expects necessary data to be in the LLC or DRAM when it starts computing. Consequently, the programmer through the software ensures explicit flushes of any dependent data to the LLC before triggering the accelerator. This is a similar approach used in previous work [58, 67, 56, 135]. As can be seen from Figure 5.1, the CPU stages data to the accelerator by first flushing it to the LLC. The accelerator is synchronized to not start reading the data until the CPU completes flushing the complete staged data. If the data footprint is below the LLC capacity, all staged data will be resident in the LLC. However, if the data size exceeds LLC capacity, some data will get spilled to DRAM. Thus, when the accelerator subsequently reads the data, it may find a fraction of it in the LLC and some other part in DRAM due to the earlier spills.

Similarly, once the accelerator is finished computing, it writes the result back to the LLC for the CPU to access. If result size exceeds LLC capacity, as discussed previously, spills to DRAM will be incurred, and the CPU may experience LLC misses requiring some of the data to be fetched from memory when it starts accessing the results.

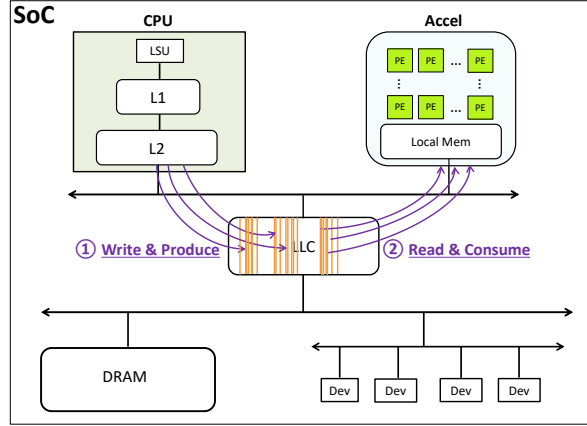


Figure 5.2: CASPHAr system architecture.

5.1.2 CASPHAr System Architecture

In contrast to the base architecture, CASPHAr frames the data staging and synchronization as a fine-grain in-cache producer and consumer transaction. CASPHAr allows a producer to stage data and synchronize with the consumer at cache line granularity. Therefore, instead of waiting for the full data footprint to be written to the LLC, CASPHAr enables CPUs and accelerators to exchange any dependent data as soon as it becomes ready.

Figure 5.2 illustrates the data movement flow in CASPHAr. The CPU stages data to the accelerator by storing individual items normally into a shared memory address (*Write*). Once the CPU has finished writing a location, i.e. after the last write to a shared memory address, the CPU marks the data line as ready in the LLC (*Produce*). In the process, the CPU needs to ensure that the produced data is available in the shared LLC. Consequently, the CPU needs a way to simultaneously flush data to the LLC and mark it as produced.

We discuss our approach to achieve this. The default x86 ISA has a software flush instruction (`clflush`) that evicts the targeted line all the way to DRAM, i.e. if the block exists in the LLC, it will be invalidated and further evicted. In our scenario, we wish to manage the eviction and keep or store the line in the LLC by not invalidating and potentially updating an already existing, or by allocating a new LLC cache line that potentially replaces and evicts others. At the same time, we need to provide a mechanism to indicate that a flushed line has been produced. There are two plausible options to support this: (1) adding a special instruction in the ISA that handles both eviction only to the LLC and flagging as produced, or (2) setting a configurable register in the LLC controller to identify a range of addresses that should not be invalidated but kept/stored and flagged when software eviction happens. We opted for the latter implementation due to its simplicity and transparent change from the software perspective. This approach will also not require any modifications of the accelerator design for cases when the accelerator is the producer.

CASPHAr then tracks the produced status of individual cache lines in the LLC. It tracks the status even if a cache line is spilled to and later re-fetched from main memory. The accelerator in turn accesses any shared memory location in a normal fashion (*Read*). If a shared location has not been produced yet, CASPHAr will indicate a standard miss condition and use regular miss handling mechanisms to block or delay the access until the data is ready. Finally, once the accelerator has finished accessing the shared location,

it resets the line back to the default status (*Consume*).

Communication of accelerator results back to the CPU follows a similar flow, but in the opposite direction, i.e. with the accelerator being the producer and the CPU consuming shared data. In this case, the CPU needs a mechanism to indicate that a shared line has been finally consumed, i.e. that the last access has occurred. Since this is equivalent to marking the line as candidate for eviction, we utilize the same evict-to-LLC instruction to indicate both production and consumption of data. Note that this will also evict data from L1 and L2 caches to the LLC on consumption. However, since consumption is equivalent to indicating that data will not be accessed further, it is actually beneficial to evict it from and thus free up space in the higher-level caches.

All combined, unlike the base architecture, a consumer in CASPHAr does not need to wait until the full data region has been staged. Instead, it can access any cache line as soon as it is produced. Hence, data spills occurring in the base architecture due to long lifetime of shared data in the cache caused by coarse-grain data staging and synchronization can be avoided. Moreover, CASPHAr takes advantage of such fine-grain synchronization by inherently allowing accelerator executions to overlap with data staging on the CPU in an out-of-order/dataflow style. As a result, CASPHAr allows for system-level pipelining of interdependent kernels to be seamlessly achieved through its microarchitecture.

Overall, CASPHAr acts like a synchronized shared buffer of arbitrary size between producers and consumers that is realized transparently between

LLC and DRAM. Note that CASPHAr supports irregular, random and non-conforming access patterns of producers and consumers. However, achievable benefits depend on the order and timing of shared memory accesses made by the involved components. CASPHAr will be able to avoid data spills as long as the maximum number of simultaneously alive (i.e. produced but not yet consumed) shared data items is smaller than the LLC size. It will remain beneficial as long as the peak shared buffer fill state is smaller than the complete shared data region size. In the worst-case, producers and consumers access items in opposite order, or consumers do not start processing data until producers are already finished. In such cases, CASPHAr will automatically fall back to a default base architecture execution.

5.1.3 Software Interface

To fully leverage the CASPHAr concept, software will require only minimal modifications. CASPHAr envisions physically contiguous memory regions dedicated to data exchanged between CPUs and accelerators. Any data structures that are shared between components are physically mapped to this region. Such an approach is widely used to simplify accelerator data exchange [64, 54]. The programmer can access the shared regions from conventional user-space virtual addresses, which are mapped to a physically contiguous memory region by the OS. CASPHAr in turn allows the user program or OS to configure the start and end address of the shared regions through a set of memory-mapped configurable register in the LLC controller. The CAS-

PHAr cache controller will then use this information to identify if a cacheline access belongs to a shared region and hence requires tracking.

Such an approach does not require any modifications to the basic application code. However, to make use of fine-grain synchronization and overlapping opportunities, the accelerator must be invoked to execute in parallel to the CPU. As such, triggering of the accelerator must be hoisted to before the start of CPU pre-processing/data preparation and synchronizing with its completion must be delayed until after post-processing of the results. In the process, the software will also set and reset the region registers in the LLC to enable and disable fine-grain tracking and synchronization of shared region accesses in the cache. Furthermore, the software must flag fine-grain production and consumption of data during pre-/post-processing.

Figure 5.3 summarizes the software modifications and resulting execution flow on a CASPHAr system. The host CPU triggers the accelerator, and the accelerator immediately begins its execution. The host CPU and the accelerator can then execute in parallel, with all producer-consumer transactions tracked and managed seamlessly by the LLC. Once the software has finished writing any data line required by the accelerator, it uses a *clflush* instruction to flush the data to and mark it as produced in the LLC. This requires inserting corresponding primitives into the application code. Similarly, once the CPU finishes reading any data produced by the accelerator, *clflush* instructions are inserted to mark data as consumed and invalidate it in its local L1 and L2 caches.

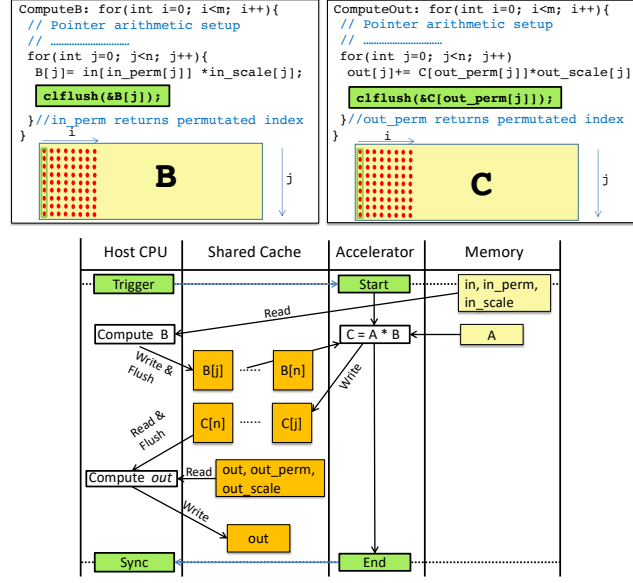


Figure 5.3: Execution flow on a CASPHAr system.

5.1.4 Discussions and Possible Variations

In CASPHAr, the software has to explicitly mark if a cacheline is produced/consumed using `clflush` instructions. This can be automated using compiler support as demonstrated in prior work [38, 138, 95]. However, for cases where producer and consumer data access patterns are irregular and/or data-dependent, where it is difficult for the compiler to statically determine the last write/read to/from a cacheline, an explicit synchronization hint from the programmer may still be required. Alternatively, marking a cacheline produced or consumed could be associated with an event such as the end of a kernel, instead of at single read/write granularity. Finally, for code where using producer-consumer patterns is particularly unsuitable, user can default to

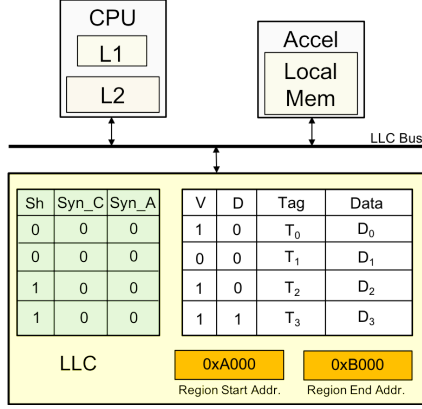


Figure 5.4: CASPHAr LLC architecture.

existing software-based synchronization mechanisms.

5.2 CASPHAr Design

In this section, we describe the CASPHAr cache architecture that implements our fine-grained data staging and synchronization. We start by describing the microarchitecture of CASPHAr and its fine-grain staging and synchronization flow. We then detail techniques to tackle challenges in performing such synchronization in the LLC, such as a tracking mechanism of staged data during residence as well as eviction, and CASPHAr-aware extensions to existing replacement policies.

5.2.1 Base Cache Architecture

CASPHAr extends a conventional cache architecture to support CPU/accelerator synchronization at cacheline granularity. In order to do so, additional meta-data bits are added to the cache structure for every cacheline (Figure 5.4).

The *Sh* bit indicates a shared cache line between the CPU and accelerator. As mentioned before, CASPHAr requires that any data structures shared between the CPU and the accelerator to be physically mapped to contiguous memory regions as similarly adopted in related works [64, 54]. CASPHAr distinguishes between shared regions for data transfers from CPU to accelerator and for transfers from the accelerator back to the CPU. CASPHAr is equipped with a set of memory-mapped configuration registers in the LLC controller to mark the start and end address of the two shared regions from user or OS space. The cache controller uses this information to identify if a cache line access belongs to a shared region and which component is producer and consumer. If a line is in the shared region data, its *Sh* bit is set to 1. CASPHAr employs two registers *Region Start Addr* and *Region End Addr* to track the start and end address of shared region as depicted in the Figure 5.4

Syn_C is a bit for synchronization when the CPU is the producer and the accelerator is the consumer. It is set to 1 when the CPU has produced the latest data ready to be consumed by the accelerator, i.e. when the CPU issues a *clflush* instruction. When the accelerator attempts to access the line, CASPHAr first checks if the *Syn_C* bit is set to 1. If so, it means the cache line is ready to be consumed and the accelerator will be allowed to access the cache line and proceed accordingly much like a typical cache hit. Otherwise, the cache line is still being processed by the CPU and is not ready. In this case, the accelerator will experience a synchronization miss, and will be notified by the cache controller once the cache line becomes ready for consumption,

i.e. when the *Syn_C* bit is set to 1. As with any other miss, the accelerator can issue other independent operations while waiting for that cache line to exploit memory-level parallelism. By contrast, *Syn_A* is the synchronization bit used when the accelerator produces data that later will be consumed by the CPU. *Syn_A* is set to 1 when the accelerator finishes producing data to be consumed by the CPU. The CPU will then similarly be synchronized and allowed to perform operations on that cache line as soon as the *Syn_A* bit is set to 1. *Syn_C* and *Syn_A* bits are reset whenever the accelerator or CPU, respectively, marks the data as consumed. On the CPU side, the *clflush* instruction thereby prioritizes resetting of *Syn_A* over setting of *Syn_C*.

Meta-data bits in the cache are initialized every time the configuration registers are set. On every region register (re-)configuration, the *Sh* bits are updated to reflect tracking status of every line in the cache. Furthermore, they are updated every time a line is brought into the cache. By contrast, the *Syn* bits of all lines in the cache are reset to zero on a configuration register update (and otherwise tracked as described above). Synchronization status of evicted lines is tracked by storing their meta-data in memory on eviction and re-initializing *Syn* bits when the line is brought back into the cache. We assume backup full/empty (F/E) bits to be associated in the main memory with each shared memory region, either as part of extended memory arrays or in a separate memory region, similar to previous work [15, 12, 46].

In the following, we show a CASPHAr system for a one producer and one consumer scenario. CASHPar can be extended to handle multiple

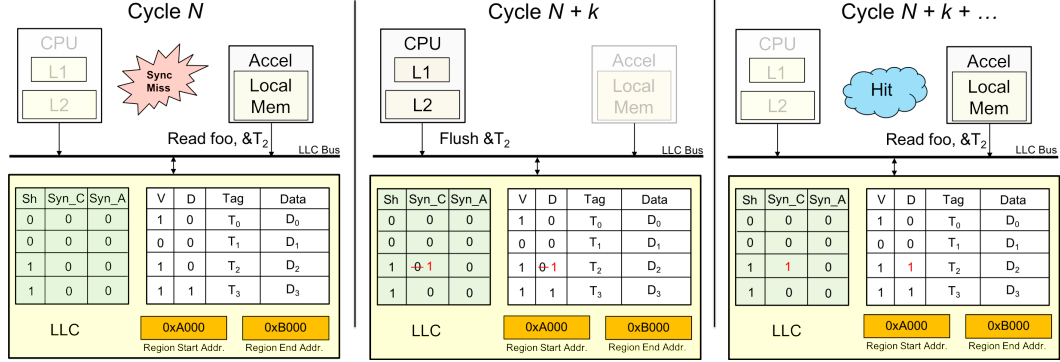


Figure 5.5: CASPHAr synchronization on a synchronization miss but cache hit.

producer-consumer scenarios by incorporating CPU/accelerator IDs into the synchronization bits, much like coherence systems with multiple CPUs.

5.2.2 Synchronization Flow

In the following, we explain the CASPHAr cache line synchronization process using a simple direct-mapped LLC organization as an example.

Synchronization miss: Figure 5.5 shows the synchronization flow in CASPHAr for a synchronization miss when the requested line is in the cache but not yet ready. In this example, the accelerator is a consumer and the CPU is the producer of memory address T_2 . When the accelerator attempts to read T_2 , it sees that the *Syn_C* bit is 0, indicating that the data is not ready and is still being processed by a consumer. A moment later, the CPU finishes processing the data and issues a software flush to memory address T_2 . It will set the *Syn_C* and *D* bits to 1, marking the cache line ready and produced.

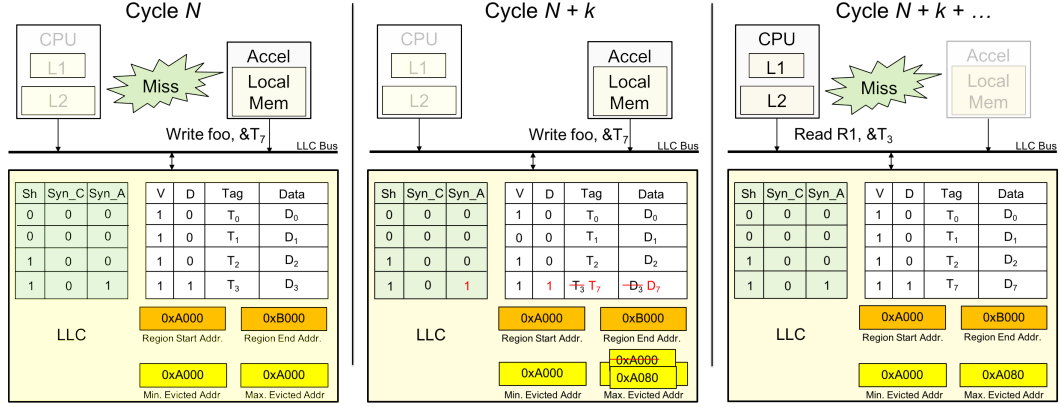


Figure 5.6: CASPHAR synchronization on a cache miss and cache eviction.

The cache controller then wakes the accelerator up by sending a signal to let it retry a read for memory address T_2 , in the same way that a cache miss wakes up a CPU instruction once the targeted cache line is ready. This time, the accelerator sees that the *Syn_C* bit is 1, i.e. that the line is produced and ready. The accelerator then accesses the cache line as a hit and the process resumes.

Producer miss: Depending on the data access patterns and capacity conflicts, some cache lines might get evicted from the cache before they are produced. Figure 5.6 summarizes the synchronization flow in a case of pure cache miss for a case where the accelerator is the producer and the CPU the consumer. The accelerator attempts to write to a memory address T_7 . However, cache line T_7 is not in the cache and consequently, the request is forwarded to the DRAM through the memory controller. Once the cache line T_7 is finally serviced, it is inserted into the cache and the synchronization process proceeds

as explained above.

Consumer miss and eviction range registers: In the scenario shown in Figure 5.6, line T_3 has to be evicted in order to make room for line T_7 . However, T_3 itself is unconsumed. As in a conventional cache, such a line will be evicted and written back to memory. On a subsequent consumer access to such an evicted line, a miss will normally be serviced and the line will be brought back into the cache. However, if an evicted line has not yet been produced, it does not need to be brought back into the cache on a consumer miss. The consumer will experience a synchronization miss and be stalled until the producer updates the line.

CASPHAr employs a set of eviction range registers to optimize such unnecessary refetching. Two 64B registers, *Min* and *Max*, record the range of the smallest and largest address of produced but evicted cachelines. For every eviction of a produced cacheline that belongs to the shared memory region, *Min* and *Max* are updated when the evicted address is smaller or larger than the current *Min* or *Max*, respectively. When a consumer accesses the cache, CASPHAr will first check if the cache has the targeted line. On a miss, CASPHAr will consult the eviction range registers to check if the targeted cacheline has actually been produced but evicted. There exist two possible scenarios: 1) The cacheline's address is outside the range of *Min* and *Max*. This guarantees that the cacheline has not been previously produced. The cacheline is not fetched from memory and the consumer is stalled until the producer updates the line (and as a result, brings it back into the cache). 2)

The targeted cacheline's address falls into the range of *Min* and *Max*. This in turn indicates two possibilities: the cacheline has been produced and evicted, or due to false positives in range tracking, the line has been evicted but not actually produced. In either case, the line will be brought from main memory to the LLC. CASPHAr will check the F/E information as a part of meta-data in main memory. If it finds the F/E bit is set, the consumer will subsequently be released. Otherwise, the consumer will be stalled and the synchronization will proceed normally.

In Figure 5.6, when the produced but unconsumed line T_3 is evicted, the *Max* eviction register is updated to match T_3 's cacheline address $0xE003$. As more evictions occur, the *Max* eviction register is updated to $0xE080$. When the CPU later accesses T_3 , it will experience a miss. CASPHAr sees the targeted cacheline's address $0xE003$ falling into the range defined by the *Min* and *Max* registers (i.e., the range of $0xE000$ - $0xE080$), and it assumes that the cache line can be accessed but has been evicted. Consequently, it will allow the access to proceed, issue the request to the memory controller and service the DRAM access. By contrast, if the targeted cacheline's address would be outside of the *Min* and *Max* range, it indicates that the evicted data requested by the consumer has not yet been produced. Thus, CASPHAr will not service the miss but instead continue and only wake up the corresponding consumer once the data is brought into the cache and marked by its producer.

As an alternative to eviction registers, evicted cachelines can be tracked by extending the LLC with an eviction bitvector, where each bit of the vec-

tor will mark whether a cache line that has been evicted has already been produced. This bitvector can be indexed relative to the start of the physical address of the shared region. However, this approach requires extra storage in the LLC and its overhead scales linearly with the size of the contiguous shared address space. By contrast, using eviction range registers requires only two 64B registers and, more importantly, the overhead remains constant independent of the shared memory region size.

5.2.3 CASPHAr-Aware Cache Replacement

The CASPHAr cache architecture enables fine-grain synchronization and data staging. However, if such capabilities are not managed carefully, they can skew the replacement policy’s decision in efficiently managing cache space. A replacement policy can assign higher residence priority for some cache lines that have been consumed recently even though their lifetime in the cache has likely already expired. Similarly, it can unintentionally evict some cache lines that are ready and staged but not yet consumed. We describe how CASPHAr addresses these challenges by helping existing replacement policies make better decisions in the presence of fine-grain data-staging and synchronization.

CASPHAr allows any existing replacement policy to be extended to leverage fine-grain synchronization information. More specifically, a replacement policy can use feedback from CASPHAr by prioritizing evictions for lines that have been consumed, i.e. whose lifetime has likely already ended. In doing so, CASPHAr can inherently and as quickly as possible make room for other

LRU Extensions	RRIP Extensions
Cache Hit: (i) move block to MRU	Cache Hit: (i) set RRPV of block to 0
Cache Miss: (i) look for consumed block (ii) if found, evict such a block (iii) if not found, evict LRU block (iv) move newly installed block to MRU	Cache Miss: (i) search for consumed block (ii) if found, go to step (vii) (iii) search for max RRPV value (iv) if found, go to step (vii) (v) increment all RRPVs (vi) go to step (iii) (vii) evict block, set RRPV value to maxRRPV-1

Table 5.1: CASPHAr-aware replacement policy extensions.

staged lines that are ready for consumption in the LLC. CASPHAr achieves such a goal in the following way. When the cache has to replace a line to make room for a new one, it will first look for replacement candidates that are already consumed. If there is no such cache lines, the conventional replacement policy is applied.

Table 5.1 shows examples of how CASPHAr can be integrated with existing replacement policies to make them aware of synchronization extensions. We show LRU and RRIP [71] as examples of how replacement policies can be enhanced and integrated with CASPHAr. In a traditional LRU replacement policy, a block with least-recently used value will get evicted/replaced. However, *CASPHAr-LRU-Ext* will instead look for a block that has been consumed. If such a block exists, CASPHAr will evict that block. If there is no such block, it will fall back to a traditional LRU policy. Similarly, *CASPHAr-RRIP-Ext* will search for a consumed block first in eviction. If such a block does not exist, it will fall back to a traditional RRIP mechanism that employs a *Re-Reference Prediction Value (RRPV)* to make eviction decisions.

Parameter	Core Model
Issue Width	5
Cache Block Size	64B
L1-I\$ Size	32kB, 8-way
L1-D\$ Size	32kB, 8-way
L2 Size	256kB, 8-way
LLC Size	4MB, 16-way
LLC Inclusivity	Non-inclusive
LLC Repl. Policy	LRU, TRRIP [87], Hawkeye [69]
DRAM	
DDR Type	DDR3-1600
Specification	Micron MT41J256M4

Table 5.2: System configuration.

5.3 Evaluation

In this section, we present our evaluation of CASPHAr across different applications and accelerator models.

Baseline: We use the previously calibrated MARSSx86 [21] as the cycle-accurate full-system simulator for architecture evaluation. We extend the baseline LLC model to implement CASPHAr. CASPHAr is estimated to add as little as 0.5% meta data overhead including the synchronization bits and the eviction range registers. To show CASPHAr’s compatibility with modern replacement policies, we integrate three replacement policies (LRU, TRRIP [87] and Hawkeye [69]) into CASPHAr. TRRIP extends RRIP [71] to be aware of different memory access behaviour among heterogeneous cores. Hawkeye was the winner of the most recent Cache Replacement Championship [70]. DRAMSim2 [114] is used to model DRAM delay and energy. Table 5.2 shows the detailed system configuration.

Comparison over prior art: We compare CASPHAr against prior art that employs a full/empty bit approach *FE* in main memory [46]. Full/empty bits use meta-data to help with data exchange and synchronization. In addition to the original approach from [46] that requires the synchronization to happen in the main memory (*FE-DRAM*), we compare against an approach that supports caching of full/empty bits in the shared LLC similar to CASHPHAr (*FE-Cache*).

Accelerators and benchmarks: In addition to the FMM application introduced earlier, we use six accelerated applications from the MachSuite benchmark set [111]. These applications consist of data staging and kernel execution, where the CPU performs the data staging and invokes the accelerator to carry out the kernel execution. We build a cycle-accurate model for each accelerated kernel in the MachSuite applications using the Aladdin tool flow [120]. In all cases, accelerators are integrated into the full-system model at the shared on-chip LLC.

5.3.1 Performance and Energy Results

DRAM accesses: Figure 5.7 shows the normalized DRAM accesses across different MachSuite applications. We compare a baseline acceleration (*Orig*) to execution under a basic full/empty bit approach (*FE-**), and CASPHAr with different extended replacement policies (*CASPHAr-**). Base acceleration and FE approaches use LRU policies by default. We further add a comparison to oracle execution with infinite LLC size (*Orig-Oracle*). Executions under

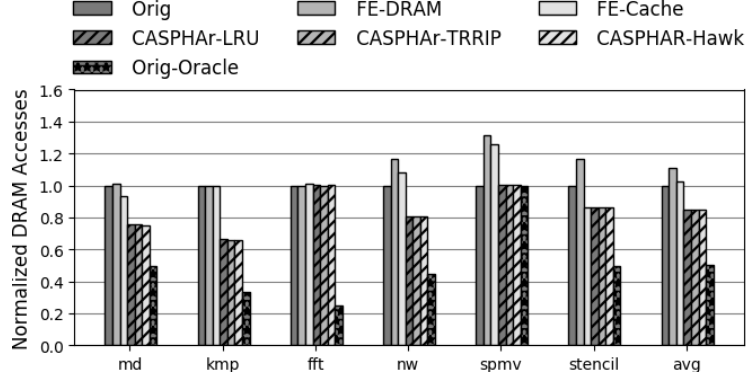


Figure 5.7: DRAM accesses.

an *FE-DRAM* approach generate an extra 16% of DRAM accesses overhead, mainly due to not only the data exchanges but additional synchronization between CPU and accelerator having to occur in main memory. Such an overhead can be reduced with *FE-Cache*, where a consumer by default inquires the LLC first for synchronization information and then possibly accesses DRAM if the targeted cache line is not resident in the LLC. However, this approach still incurs 2% more DRAM accesses than the original baseline execution on average. By contrast, with the help of eviction range registers, CASPHAr avoids unnecessary and expensive DRAM-level meta-data checking including associated cache pollution. This in turns reduces DRAM accesses by up to 35% and 18% on average. In *spmv*, the problem size is small enough to already be cache contained in baseline acceleration. Nevertheless, *FE-** still see an increase in DRAM accesses due to the extra memory traffic generated by meta-data checking as mentioned above. In case of *fft*, execution with CASPHAr does not yield significant DRAM access savings despite significant

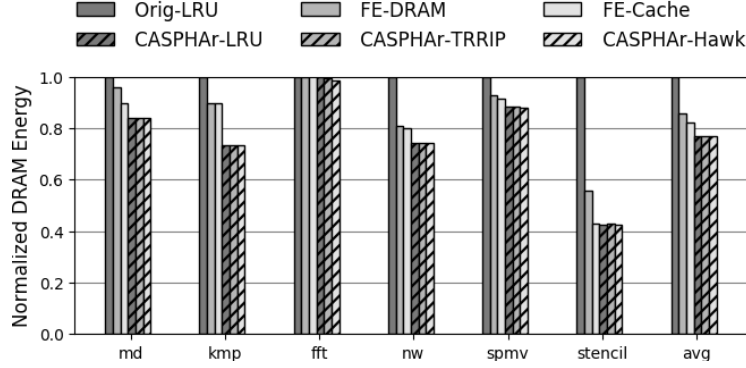


Figure 5.8: DRAM energy.

saving opportunities existing as demonstrated by the oracle case. We observe that this is caused by the opposing order of shared data accesses made by the data staging and *fft* kernel execution in the application.

As discussed in Section 5.1.2, when such a case occurs, producer-consumer transactions are restricted to a very minimal overlap. *Orig-Oracle* achieves the minimal DRAM accesses achievable among all optimizations. In *Orig-Oracle*, capacity conflicts are completely avoided, leaving only cold misses.

DRAM energy: We further compare DRAM energy consumption under original acceleration and CASPHAr, illustrated in Figure 5.8. Energy savings of up to 65% and 22% on average can be observed with *CASPHAr*-. By contrast, *FE-DRAM* and *FE-Cache* experience fewer savings, averaging just 16% and 18%, respectively.

Among the applications, *fft* has limited energy savings. As discussed above, *fft* does not experience DRAM access reductions and this directly translates into minimal energy savings.

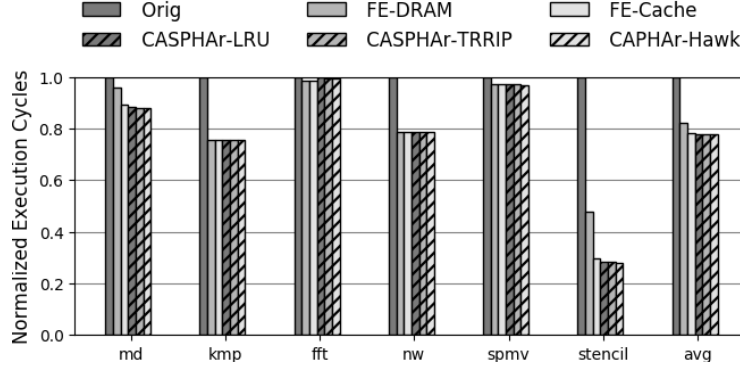


Figure 5.9: Execution cycles.

Performance: Figure 5.9 shows the normalized execution time across different MachSuite applications. Accelerations under *FE-DRAM* can improve performance by up to 52%, averaging an 18% performance boost. By contrast, CASPHAr reduces execution time by up to 70% and by 23% on average under an extended *LRU* replacement policy, outperforming both *FE-DRAM* and *FE-Cache* approaches. In *spmv*, even though the problem size already fits into the cache, CASPHAr still improves performance by 3% through fine-grain system-level pipelining and overlapping at cacheline granularity. By contrast, *fft* does not benefit from CASPHAr as the opposing order of data accesses in the applications also prevents pipelining and overlapping of dependent data.

Replacement policies: To further demonstrate the benefit of extended replacement policies, we study performance when varying the consumer access rate relative to the producer rate on a cache-sensitive *stencil* application. Figure 5.10 presents normalized runtime of CASPHAr using different unmodified or extended cache management policies under varying relative consumer rates

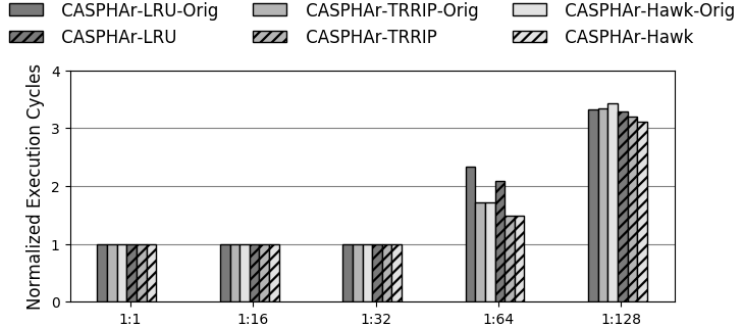


Figure 5.10: Sensitivity to consumer slowdown rate (*stencil*).

for the *stencil* application. We slow down the consumer access rate by decreasing its frequency. As the ratio of consumer to producer rate becomes smaller, CASPHAr experiences significant slowdown. A relatively lower consumer rate results in shared lines accumulating in the cache and at one point exceeding cache capacity. When such a case occurs, the replacement policy becomes critical in alleviating system-wide slowdown. For a medium slowdown of 1:64, extended replacement policies can reduce the performance degradation by up to 10% for *LRU*, and 14% for *TRRIP* and *Hawkeye*. With extended policies and better eviction decisions, CASPHAr can moderate the performance penalty due to cache capacity conflicts as much as possible.

5.3.2 Why CASPHAr Works

In the following, we show additional details about CASPHAr and the fundamental reasons and limitations in improving performance and energy over a baseline acceleration.

DRAM spills: Figure 5.11 shows the normalized breakdown of evicted shared

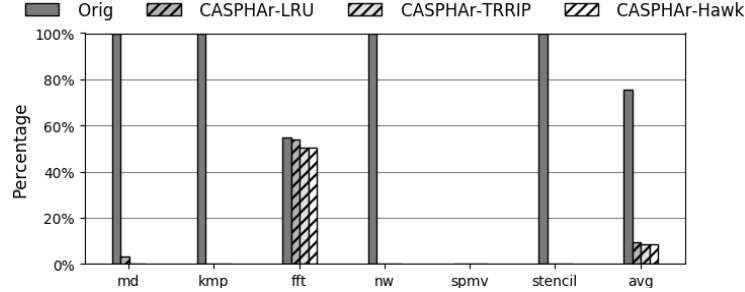


Figure 5.11: Fraction of staged lines that are evicted before they get consumed.

data across applications. We measure the fraction of data that was produced but evicted before consumed. As shown in Figure 5.11, the fraction of data that is not been consumed before it gets evicted, i.e. that has to be spilled to DRAM and eventually be brought back, is more than 50% and up to 100% in the baseline acceleration. In other words, coarse-grain data staging and synchronization experience significant data spills from LLC to DRAM. CASPHAr can avoid such unnecessary spills when coupled with any replacement policy. With CASPHAr, the cache inherently lets the consumer access the staged data in a timely fashion before it gets evicted. CASPHAr does not show significant benefits in the case of *fft* due to opposing order that makes data spills unavoidable. In *spmv*, the problem size fits into the LLC capacity, i.e. only limited data spills are observed even in the baseline case. Note that while DRAM spills directly influence energy, depending on latency hiding in the application, savings in spills do not necessarily translate into performance benefits. At the same time, fine-grain pipelining and overlapping in CASPHAr can improve performance even when there are no DRAM gains.

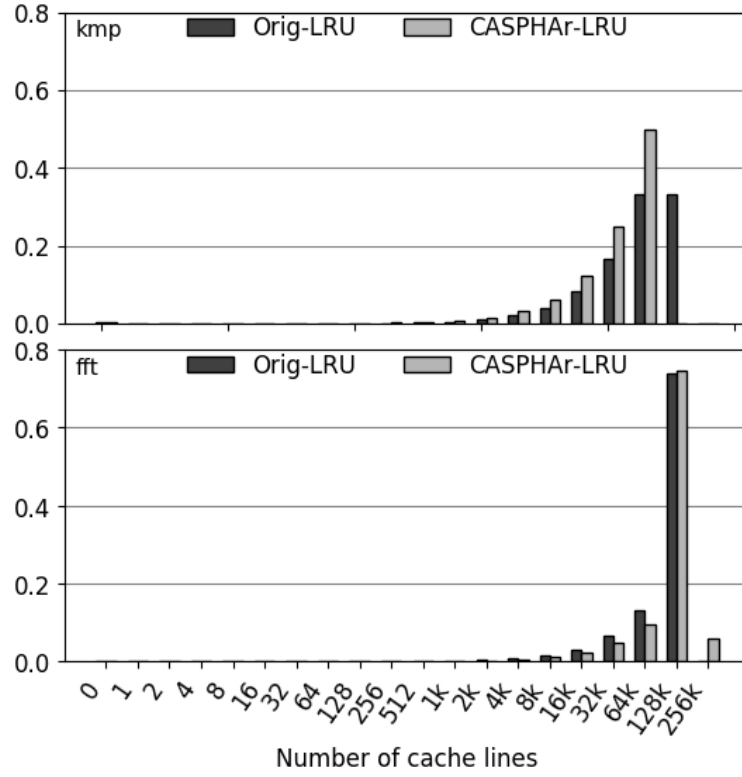


Figure 5.12: Histograms of produced but unconsumed shared cache line occupancies.

Cache occupancy: CASPHAr avoids data spills by avoiding unnecessarily long occupancy of shared data in the cache through its fine-grain producer-consumer transactions. To demonstrate the actual benefit of such a technique, we profile the distribution of shared working set in the LLC. We define shared working set as the number of produced but unconsumed cache lines over time. Here, 64K-128K is the range where the number of produced cache lines exceeds the LLC capacity. As can be seen from Figure 5.12, in the kmp case, more than 30% of the time, original execution falls into the range exceeding LLC capacity.

As a result, capacity conflicts can not be avoided. By contrast, CASPHAr conceptually "pushes" the distribution of shared working set to the lower end of the range. Consequently, CASPHAr maintains a relatively smaller average and maximum shared working set in the cache that does not cause capacity conflicts in the first place. By contrast, in the *fft* case, due to incompatible access order, CASPHAr does not significantly alter the occupancy distribution.

Shared data lifetime: We further demonstrate how CASPHAr unlocks inherent system-level pipelining and overlapping opportunities. We measure the distribution of the shared cache line lifetimes in the LLC. We define lifetime as the number of execution cycles from the moment a cache line is produced until it gets consumed. A shorter lifetime implies that fine-grain staging and synchronization are effectively achieved. Figure 5.13 illustrates that CASPHAr shortens the lifetime of cache lines by a significant margin over the baseline acceleration in the *kmp* and *fft* application. CASPHAr effectively achieves a very short waiting time and delay between producer and consumer. This enables CASPHAr to free up cache space in a timely manner to make room for other cache lines. While in *kmp*, CASPHAr was able to shift the lifetime distribution, in *fft*, baseline and CASHPHAr acceleration experience similar lifetime distributions. Again, this is due to the nature of opposing streaming order between data staging and kernel acceleration in *fft*.

Comparison to manual software optimizations: Finally, we discuss how CASPHAr compares against manually applied software blocking and pipelining optimizations. Figure 5.14 shows normalized execution cycles and DRAM

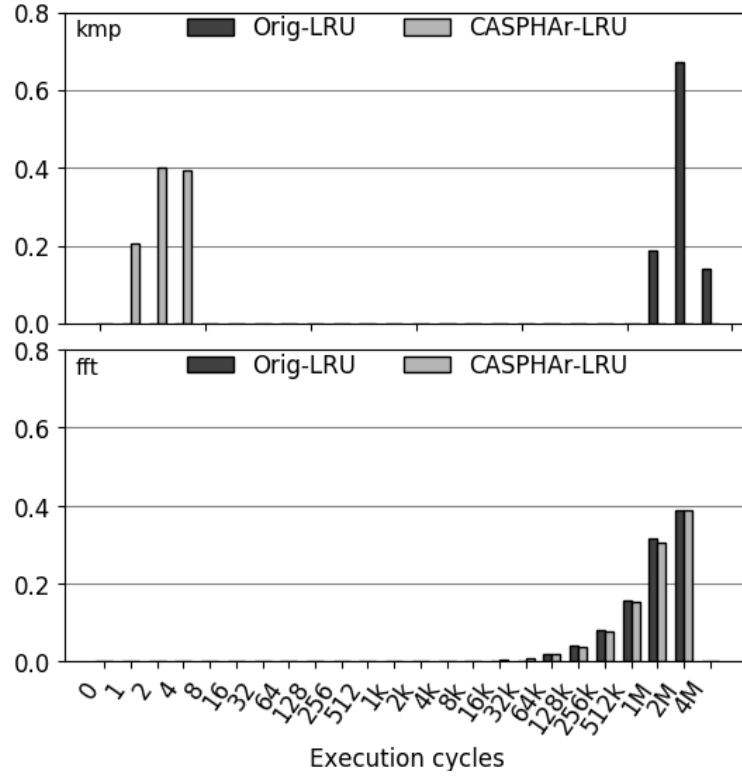


Figure 5.13: Histograms of produced shared cache line lifetimes until consumed.

accesses of CASPHAr compared to other software-centric approaches for the FMM and *spmv* applications. As mentioned before, FMM has a large data footprint while *spmv* has a relatively small footprint that natively fits into the cache. As can be seen from the figure, CASPHAr can unlock performance and off-chip access benefits similar to the manually tuned FMM. In case of the already cache-contained *spmv*, there is performance improvement from manual software optimizations. With a small data footprint and a coarse-grain accelerator, manual software optimizations are limited to only allowing accelerator

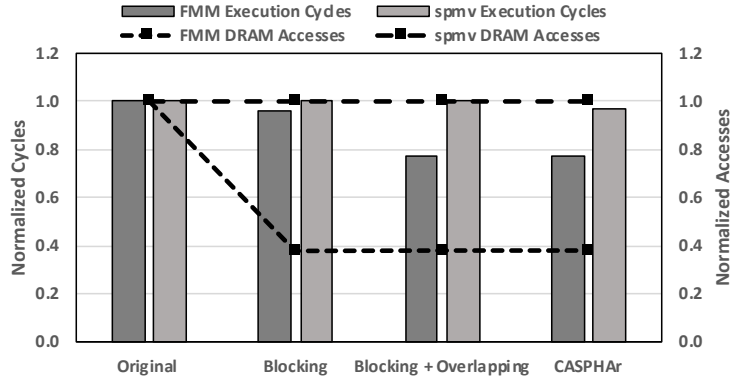


Figure 5.14: Execution cycles and DRAM accesses of CASPHAr vs. manual optimizations (*FMM* and *spmv*).

execution once the CPU finishes preparing a minimum amount of data. A too fine-grained software blocking reduces accelerator efficiency and increase synchronization overhead. This forces the application to run sequentially even though a fraction of staged data may be ready earlier. CASPHAr, by contrast, is free from such restrictions. Since staged data synchronization is taken care of by the hardware, the accelerator can be allowed to start executing and to consume data as soon as it is ready without necessarily waiting for the host CPU to fully finish staging all data. As such, CASPHAr is able to improve *spmv* performance by 3%. Overall, CASPHAr can transparently achieve the same and in some cases better performance improvements than what is possible with complex manual software optimizations.

5.4 Summary and Conclusions

We presented CASPHAr, a novel last-level cache architecture that achieves transparent and seamless fine-grain data movement orchestration and execution coordination for heterogeneous on-chip architectures. CASPHAr supports efficient accelerator staging by allowing producer and consumer accesses to synchronize at cache line granularity. In doing so, CASPHAr effectively reduces expensive spills and refetching to/from DRAM due to unnecessarily long lifetime of shared data as an effect of too coarse-grain accelerator staging and synchronization. We show that CASPHAr is able to avoid unnecessary data spills by reducing the waiting time and delay between producer and consumer. Moreover, CASPHAr inherently realizes efficient data coordination and system-level pipelining between interdependent kernels that unlock additional performance gains. All in all, CASPHAr can achieve similar and/or better results to manual system-level data movement optimizations in software, improving performance by up to 23% and reducing energy consumption by 22% compared to baseline acceleration.

Chapter 6

Cache Management for Co-Running Applications

So far we have discussed cache management techniques to optimize data movement when interdependent kernels/applications are executed on various heterogeneous components. In this chapter, we introduce Flock, an effective caching mechanism that targets an improved aggregate system-wide performance benefits when heterogeneous components execute independent applications. In particular, existing cache management solutions suffer from two fundamental drawbacks, lack of accurate performance proxy and overly rigid or too flexible sharing models, both of which limit performance in homogeneous CPU multicore systems and which get exacerbated in heterogeneous multicore systems. Flock addresses these issues by introducing novel performance proxies and sharing models for shared cache management.

6.1 Background

We first demonstrate further the need for a better performance proxy and sharing model that can lead to better shared cache management.

6.1.1 Performance Proxy

Amdahl’s Law [43] tells us that the speedup of a workload due a new optimization depends on (1) speedup of the part of the task that benefits from the new optimization and (2) the proportion of the execution time that is devoted to that task. Thus, the speedup provided by a cache depends on (1) the product of the latency savings of a cache hit and the total number of cache hits, but it also depends on (2) the proportion of time spent not accessing the cache. In the case of a last-level cache, the proportion of time spent not accessing the LLC is determined by the memory intensity and the number of hits to the L1 and L2 and the number of memory accesses.

More specifically, assume a 20-cycle LLC latency a 200-cycle DRAM latency, and consider two applications, A and B, that have the same memory intensity (number of memory accesses per thousand instructions), where application A has an L1 hit rate of 90% and application B has an L1 hit rate of 10%. Assuming 100 total memory accesses by each application, A sends 10 memory requests to the LLC, and B sends 90 memory requests to the LLC. Suppose that by changing the cache partitions, we can either give A 5 LLC hits (P_1), or we can give B 10 LLC hits (P_2).

We see in Table 6.1 that existing solutions [125] choose P_2 , because (1) 10 is greater than 5, so the LLC would provide greater latency reduction, and (2) B has many more accesses to the LLC, so it is memory-bound. However, if we look at the total memory access times (MAT) of A and B, we see that if we assume that neither application sees any LLC hits, then A spends 2,090 cycles

	L1		LLC Hits			MAT (cycles)			Speedup		Prior Work	Ours
	Accesses	Hits	Baseline	P_1	P_2	Baseline	P_1	P_2	P_1	P_2		
App A	100	90	0	5	0	2,090	1,190	2,090	43%	0%	P_2	P_1
App B	100	10	0	0	10	18,010	18,010	16,210	0%	10%		

Table 6.1: Speedup due to better cache management depends on the number of LLC hits *and* the proportion of memory accesses that reach the LLC. (MAT represents total memory access time.)

accessing memory (90 L1 hits, each with a 1-cycle latency, and 10 DRAM accesses with a 200-cycle latency), while B spends 18,010 cycles accessing memory (10 L1 hits, each with a 1-cycle latency, and 90 DRAM accesses with a 200-cycle latency). Thus, 5 extra LLC hits for A reduces this memory penalty from 2,090 cycles to 1,190 cycles (a speedup of 43%), while 10 extra LLC hits for B reduces its memory penalty from 18,010 cycles to 16,210 cycles (a speedup of only 10%).

The key insight here is that application speedup depends not only on the number of extra cache hits but also on the proportion of memory accesses that reach the LLC. Since A has a lower LLC access ratio (due to a high L1 hit rate), its performance is more sensitive to the few extra LLC hits that can completely mitigate A’s memory bottleneck. By contrast, B’s L1 hit rate is so low that a few extra hits in the LLC do not have a large impact on performance; the majority of B’s memory accesses will still go to DRAM, and B will continue to be bottlenecked by memory.

6.1.2 Sharing Model

While performance models work well in making holistic decisions about cache allocations to different cores, there remains the question of how these allocations will be realized in the shared cache. Explicit cache partitioning tends to be too rigid, so replacement-based schemes are desirable, because they adapt better to fine-grained program behavior. However, when multiple applications share a cache, replacement-based schemes are insufficient to reach desired cache allocations because applications with higher memory intensities tend to dominate cache space.

In particular, replacement-based schemes determine a line’s priority when the line is first inserted into the cache, and then during the line’s lifetime in the cache, lines are *aged* to maintain eviction priorities. However, replacement policies age at the same rate upon every cache access, and this decision is inherently based on the assumption that time should pass at the same rate for all applications. By contrast, performance models determine cache allocations of applications in isolation, recognizing that applications have different cache access rates. As a consequence, aging all lines together does not generate the sharing intended by the performance model and biases cache space towards the memory-dominant application.

This issue is especially significant when there is a large gap in memory intensity between two cores as in a CPU-GPU system. As an example, consider a CPU-GPU system sharing an LLC with a target of 6 ways and 10 ways, respectively. Assuming the GPU generates LLC accesses at a rate 10 times

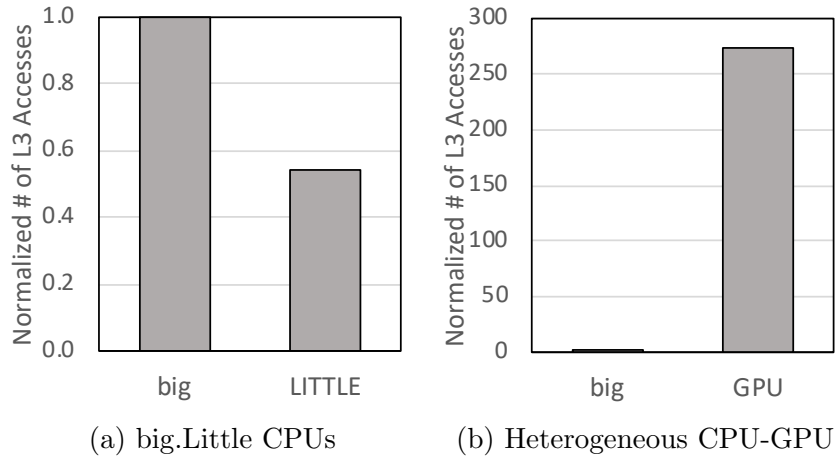


Figure 6.1: The number of LLC accesses on different evaluated systems normalized to big core accesses

higher than the CPU, CPU lines will age much faster than modeled when the performance model determined the IPC benefit of extra allocations for the CPU. Thus, it is likely that the GPU accesses will undesirably evict CPU lines, increasing the effective number of GPU share of the cache by a few ways. Furthermore, this often then creates a *feedback loop* where the GPU domination increases GPU IPC thus increasing effective memory intensity enlarging the memory intensity gap even more.

Our empirical analysis suggests that this effect is significant in any heterogeneous system: On a CPU-GPU systems, the LLC access rate for GPUs is on average $273\times$ higher than the LLC access rate for CPUs, and on a big.Little system, the LLC access rate for the big core is on average $2\times$ higher than the LLC access for the Little core (see Figure 6.1). In fact, we find that this issue is sometimes significant even in homogeneous systems when there

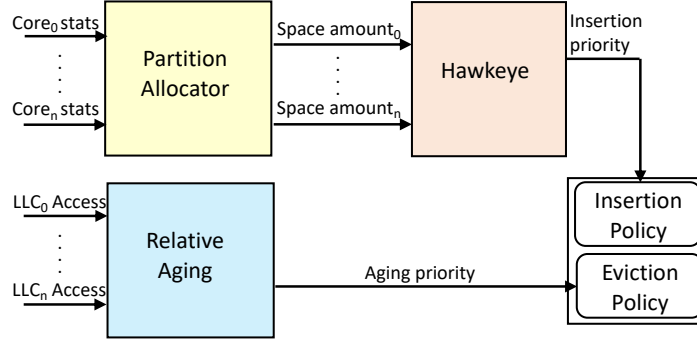


Figure 6.2: Top-level block diagram of Flock cache.

is a memory intensity gap between the applications running on the different homogeneous cores. Thus, it is important to precisely age cache lines relative to the access rate of the corresponding core.

6.2 Flock Overview

We introduce Flock, a heterogeneity-aware shared cache management. Flock uses global cache activities in the memory hierarchy as a performance proxy instead of only LLC activities. Flock then selects partition that maximizes system speedup, and uses that information to assign insertion priority. Finally, Flock protects the initially agreed partition by adjusting eviction policy adaptive to core’s access intensity, preventing domination of core with high intensity. Flock uses state-of-the art Hawkeye [69] as a base cache replacement algorithm when deciding insertion priority.

More concretely, Flock consists of two components (see Figure 6.2. The first component, the *Partition Allocator*, identifies a partition that maxi-

mizes aggregate per-core performance. This component uses a novel cache hit rate estimator and a holistic performance model [93] to compute the expected speedup for each core at different partition sizes; it employs a scalable search algorithm to arrive at the best partition. It then takes the target partition as input and learns the optimal caching solution for each individual partition—unlike Hawkeye which learns the optimal caching solution for the entire shared cache. Like Hawkeye, this optimal solution is used to predict whether data loaded by a given PC is likely to be cached or not by the optimal partition algorithm. Cache-friendly data is inserted in the cache with high priority, and cache-averse data is inserted with low priority. Finally, Flock’s partition-based replacement policy employs a novel aging scheme, namely *Relative Aging*, to ensure that memory-intensive cores do not dominate the other cores.

6.3 Flock Design

In this section, we detail Flock’s design. We first describe Hawkeye, upon which Flock is built. We then describe other two key components. *Partition Allocator* is tasked to select a partition that maximizes aggregate per-core performance while *Relative Aging* is responsible to age cache lines in proportion/relative to their core’s LLC accesses. This in turn, makes lines from dominating cores age faster and as a result alleviates space dominations from dominant cores.

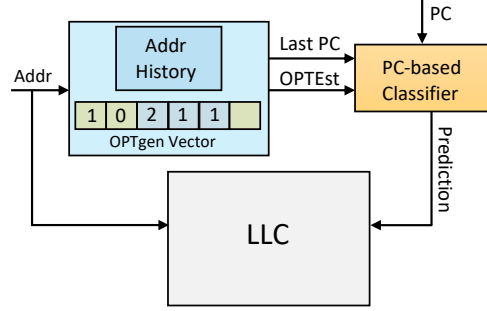


Figure 6.3: Hawkeye cache architecture.

6.3.1 Base Cache Management

Flock builds on the Hawkeye cache replacement policy [69], which won the Cache Replacement Championship in 2017 [70]. Hawkeye leverages Belady’s algorithm [23] by learning the optimal solution for past references to predict the caching behavior of future references.

Figure 6.3 shows the overall structure of Hawkeye. Its main components are: (1) OPTgen, which simulates the optimal solution for past accesses to produce training labels, and (2) the Hawkeye Predictor which is a binary classifier that is trained using OPTgen’s outcomes. On a new cache access, the Hawkeye predictor is consulted to predict whether data loaded by a given PC is likely to be cached or not by the optimal algorithm. Cache-friendly lines are inserted in the cache with high priority, and cache-averse lines are inserted with low priority.

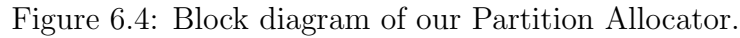
OPTgen uses an occupancy vector to record the occupied cache capacity over time. Each entry of this vector contains the number of liveness intervals that overlap at a particular time, where the liveness interval of a cache line

is defined as the interval between two consecutive uses of that line. For set-associative caches, OPTgen maintains one occupancy vector for 64 sampled cache sets, such that the maximum capacity of any occupancy vector entry never exceeds the cache associativity. Occupancy vectors measure time in terms of cache accesses to the corresponding set, and they include enough entries to model $8\times$ the size of the set (or the associativity). Thus, for a 16-way set-associative cache, each occupancy vector has 128 entries (corresponding to $8\times$ the capacity of the set), and each occupancy vector entry is 4 bits wide, as its value cannot exceed 16.

6.3.2 Partition Allocator

The *Partition Allocator* follows an efficient iterative algorithm to identify the best partitioning of shared cache resources among cores. The benefit of a given partitioning decision is evaluated by first estimating the hit rate for each core under the proposed partitioning and then estimating the corresponding IPC for each core.

Figure 6.4 shows a block diagram of the *Partition Allocator*, which takes as input the current count of cache accesses and cache hits for each core in the private (L1, L2) and shared (LLC) caches. Given a list of new *LLC* partition candidates $(P_1, P_2, \dots, P_m)$, it then gathers their expected speedups. We first describe our algorithm for estimating speedup for a given partitioning before describing the iterative algorithm for searching different partitionings.


$$CPI[\text{system}] = CPI[\text{infinite cache}] + FCP \quad (6.1)$$

107

the number of misses per instruction in an n -level cache and let T_n be the access latency of that level, then the FCP of a 3-level cache would be:

$$FCP = (mr_1 - mr_2) * T_2 + (mr_2 - mr_3) * T_3 + mr_3 * T_{dram} \quad (6.2)$$

Thus, the projected CPI for a given cache partitioning, denoted as $CPI_{projected}$, can be computed as follows:

$$CPI[\text{system}]_{projected} = CPI[\text{infinite cache}] + FCP_{projected} \quad (6.3)$$

Dividing equation 6.1 by equation 6.3, we get the following:

$$\frac{CPI[\text{system}]_{current}}{CPI[\text{system}]_{projected}} = \frac{CPI[\text{infinite cache}] + FCP_{current}}{CPI[\text{infinite cache}] + FCP_{projected}} \quad (6.4)$$

Since $CPI[\text{infinite cache}]$ remains constant for a given application, and since it can be difficult to estimate during runtime execution, we ignore it to get the following approximation for application speedup.

$$Speedup \approx \frac{FCP_{current}}{FCP_{projected}} \quad (6.5)$$

This equation tells us that the speedup, which is a ratio of the old and new CPIs, is approximately equivalent to the ratio of current and projected

FCP , so we can estimate the speedup by observing changes in FCP . To understand why this equation leads to correct partitioning decisions, consider the example shown in Table 6.1. Given the cache activity at each level, we can compute FCP_{current} , FCP_{P1} and FCP_{P2} using equation (6.2), and the FCP values evaluate exactly to the Memory Access Times described in Table 6.1.

Computing the changes in FCP is relatively simple because we already know FCP_{current} during execution and because we can compute $FCP_{\text{projected}}$ by computing LLC hit rates and DRAM latency at different LLC partition sizes. We next describe our procedure for computing LLC hit rates and DRAM latency under different partition sizes.

Hit Rate Estimation To compute FCP , we need to obtain hit rates for all levels of the cache. For private caches, we obtain hit rate by reading existing counters at the L1 and L2 caches, and for the shared LLC, we compute the projected hit rate for each core under a new LLC partition. To compute the LLC hit rates for each partition candidate, we extend Hawkeye’s OPTgen component to compute the optimal hit rate for different partition sizes. In particular, in addition to the OPTgen module that computes the optimal solution for the shared cache access stream, each core is equipped with an occupancy vector that simulates the optimal solution behavior had the core run with no interference. This per-core occupancy vector can be instantiated to any cache size, so we instantiate it to the partition size that we wish to evaluate. These per-core occupancy vectors are small, just 1 KB per core.

DRAM Latency Estimation The second component in computing FCP is the latency for accessing all levels of the memory hierarchy. While the latency of L1, L2, and LLC caches remains constant, DRAM latency will change with different LLC partitions because different partition decisions will change the off-chip traffic. Moreover, DRAM latency can also vary due to row-buffer interference, because a row-buffer hit of core i when running alone can become a row-buffer miss due to conflicting memory requests of core j .

Therefore, instead of assuming that off-chip memory latency is independent of partition decisions, we estimate changes in memory latency under different LLC partitions. Specifically, we track for each application in a given epoch, its total memory access count and its row-buffer miss rate. These quantities, denoted $TotAcc$ and $RowMissRate$, respectively, are already available in modern memory controllers. We also record the experienced average memory latency of the current epoch, namely T_{dram} . Using OPTgen vectors, we then estimate extra memory requests that would be generated for any given partition candidate and compute the estimated total memory requests $EstTotAcc$. Finally, Flock estimates the expected memory latency $\hat{T}_{dram}$ as follows:

$$\hat{T}_{dram} = RowMissRate * \frac{EstTotAcc}{TotAcc} * T_{dram} \quad (6.6)$$

$\hat{T}_{dram}$ is supplied to equation 6.2 to obtain the final FCP value, which enables Flock to have a holistic view of the entire memory hierarchy when making partition decisions.

Scalable Partitioning Algorithm Having explained how speedup can be computed for a given partitioning of the shared cache, we now present our heuristic for identifying the best partition, which is essentially the NP-hard *knapsack* problem [80], where the number of partitions grows exponentially with the number of cores. For example, with two cores sharing a 32-way cache, there are only 33 possible partitions, but for four cores sharing a 32-way cache, there are 6,545 possible partitions. Our heuristic has a worst case time complexity of $O(kn^2)$ where k is the cache associativity and n is the number of cores.

Algorithm 1 summarizes our partitioning algorithm. Each core is assigned an initial partition value of *start*, which is initialized *start* to be 80% of the ways. *total_credit* represents the sum of the ways assigned to each core, so the algorithm gradually reduces the allocation for each core until *total_credit* sums to the number of cache ways.

More specifically, the *get_min_IPC_drop* function decides which core’s allocation to reduce; the allocations are reduced at the granularity of 10% of the ways (rounded down). For example, for a 16-way cache, 10% of the ways corresponds to 1 way. The *get_min_IPC_drop* function calculates the performance loss for every core assuming that its allocation been reduced by 10%, and it selects the core with the smallest performance drop. We employ this strategy to achieve a near-optimal partitioning without paying the cost of exploring all possible partitioning combinations.

Algorithm 1 Heuristic for Scalable Partitioning

```
1: procedure DECIDE PARTITION
2:    $start \leftarrow 0.8 * ways$ 
3:    $partition[num\_cores] \leftarrow start$ 
4:    $total\_credit \leftarrow start * num\_cores$ 
5:   while  $total\_credit > ways$  do
6:      $minID \leftarrow get\_min\_ipc\_drop()$ 
7:      $partition[minID] \leftarrow partition[minID] - 0.1 * ways$ 
8:      $total\_credit \leftarrow total\_credit - 0.1 * ways$ 
9:   end while
10: end procedure
11: procedure GET_MIN_IPC_DROP
12:    $min\_core\_id \leftarrow -1$ 
13:    $min\_drop \leftarrow 0$ 
14:   for each core_id  $idx$  do
15:      $k \leftarrow partition[idx]$ 
16:      $drop \leftarrow FCP[idx][k] - FCP[idx][k - 1]$ 
17:      $drop \leftarrow drop / CPI[idx]$ 
18:     if  $min\_drop > drop$  then
19:        $min\_drop \leftarrow drop$ 
20:        $min\_core \leftarrow idx$ 
21:     end if
22:   end for
23:   return  $min\_core$ 
24: end procedure
```

6.3.3 Relative Aging

The cache partitions determined by the *Partition Allocator* are used to inform Flock’s replacement policy, whose goal is to encourage the partitioning over long periods of time, while allowing more flexibility than a strictly partitioned cache over shorter periods of time.

Flock’s replacement strategy is a modification of the Hawkeye policy [69], which uses a PC-based predictor that is trained by observing Belady’s optimal MIN policy on past cache accesses. In particular, the Hawkeye policy learns, for a given cache size, whether each PC tends to insert lines with high priority or low priority (referred to by the Hawkeye authors as Cache Friendly and Cache Averse, respectively). Once inserted, Hawkeye’s promotion policy is uniform: All lines are aged—that is, their priority is decreased—on a cache access; lines become younger—that is, their priority is increased—when they are re-referenced. Thus, Hawkeye treats a shared cache monolithically, modeling a single cache without distinguishing accesses from the various cores.

Flock instead builds a separate model of Belady’s MIN solution for each core, using that core’s target partition size and that core’s cache accesses to train the PC-based predictor for that core’s PCs.¹ Thus, Flock learns a different insertion policy for each core

To enforce partitions, Flock uses a differentiated aging policy. Instead of uniformly aging every cache line on every LLC access, Flock ages each cache

¹PCs that are shared across cores are trained by input from the various cores, but our simulation infrastructure does not model shared virtual pages).

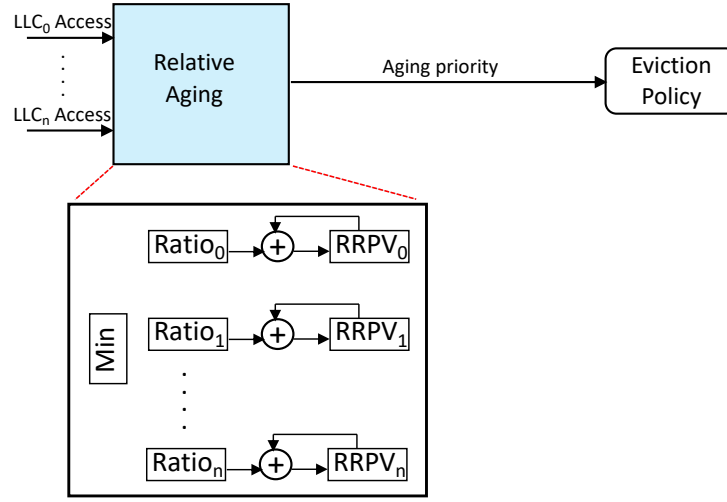


Figure 6.5: Relative aging's block diagram.

line in proportion to its corresponding core's access rate. We call this *Relative Aging*. Thus, cache lines brought in by cores with lower access rate are aged slower than lines brought in by cores with higher access rates. For example, if the access ratio between two competing cores is 1:2, then first core is aged every two cache accesses, while the second core is aged on every access.

Figure 6.5 shows how relative aging is implemented in Flock. We take as input the number of LLC accesses for each core and determine the minimum LLC access intensity among all cores. Using the minimum value, we compute the ratio of per-core LLC accesses to the maximum value; these are denoted as $Ratio_0, Ratio_1, \dots, Ratio_n$. These ratio values are then added to the RRPV register. The bigger the ratio value, meaning the more dominant a core is, the faster they are going to age. Conversely, core with less access intensity will age slower. Consequently, every cache line is guaranteed to be aged in proportion

Component	Details	Cost
OPTgen vectors	10 vectors, 32 sets, 64 5-bit entries	25KB
Hawkeye predictor	2800 4B sampler entries, 2048 pred. entries	16.4KB
Core counters	CPI (per core) proj. CPI (per core, part.)	176B
Cache counters	L1, L2 misses (per core) LLC misses proj. LLC misses (per part.) LLC miss ratio (per core) rel. age counters (per core)	27B
DRAM counters	access count cumulative latency row miss count	12B
RRPV	3-bit per LLC line	48KB
Total		89.6KB
Per core		22.4KB

Table 6.2: Flock hardware cost (4-core, 32-way 8MB LLC)

to its corresponding core’s access ratio.

6.3.4 Hardware Overhead

Table 6.2 shows the hardware overhead for a 4-core system with a 32-way 8MB LLC using Flock. While these counters can be obtained through performance counters, we assume the counters are added as extra registers/counters during design-time to ease pressure on performance counters, similar approach with previous work [125, 108]. The most costly component is the OPTgen occupancy vectors used to simulate sets of different partition sizes across the cores. Flock simulates 10 sets of different partitions configurations, of which

each set must track all 32 ways combined. Unless otherwise stated, we size counters to be 4 bytes which should be able to track any application within an epoch. Aside from the Hawkeye sampler and predictor, most of the rest of performance and cache replacement counters already exist in real systems.

6.4 Evaluation

Simulated System We evaluate our solution using the MacSim [3] simulator. We generate workload traces using Pin [6] for CPU workloads and GPUOcelot [45] for GPGPU workloads. For all simulations, if one application terminates early, we repeat it until all other applications have finished. We simulate three system architectures: homogeneous multi-core CPUs, heterogeneous big.LITTLE multi-core CPUs, and heterogeneous CPU-GPU cores. The CPU cores are modeled after a high-end out-order core and a low-end in-order core, similar to the CPU cores in Intel’s Sandy Bridge [68] and Atom, respectively. The GPU cores are modeled after NVIDIA’s Fermi [101], where each core is running in SIMD fashion with multi-threading capability. Configuration details are shown in Table 6.3.

Benchmarks We evaluate our solution by running applications from the SPEC CPU2006 benchmark suite for CPU applications and Rodinia [34] and NVIDIA CUDA SDK for GPU applications. All benchmarks are listed in Table 6.4. For CPU systems, we form multi-core workload mixes by randomly selecting two, four, and eight CPU benchmarks. For CPU-GPU applications,

CPU	big	1-4 cores, 3 GHz, 4-wide, out-of-order
	big	8-way 32KB L1 I/D (3-cycle), 64B line
	big	8-way 256KB L2 (8-cycle), 64B line
	LITTLE	1-4 cores, 3 GHz, 1-wide, in-order
	LITTLE	8-way 32KB L1 I/D (3-cycle), 64B line
GPU		6 cores, 1.5GHz, in-order, 2-wide 8-SIMD
		4-way 4KB L1 I(1-cycle), 64B line
		8-way 32KB L1 D(2-cycle), 64B line
LLC Cache		32-way 8MB, 30-cycle, 64B line
DRAM		DDR3-1333, 41.6 GB/s, FR-FCFS

Table 6.3: Simulated system configuration.

Suite	Applications
SPEC 2006	mcf, omnetpp, sphinx3, xalancbmk, cactusadm, bzip
	astar, milc, GemsFDTD, lbm, libquantum
SDK + Rodinia	bfs, btree, kmeans, pathfinder, quasirandom, convsep
	sobol, blackscholes, mergesort, histogram, convtex, fwt

Table 6.4: Applications used in our evaluation

we form workload mixes by combining one CPU benchmark and one GPGPU benchmark from Table 6.4. We run the benchmarks using the reference input set and we use SimPoint [106] to generate a single sample of 250 million instructions for each benchmark.

Performance Metrics We measure system throughput using weighted speedup [51, 124], which is calculated as follows for a mix of N applications:

$$Speedup = \sum_{i=1}^N \frac{IPC_{i,shared}}{IPC_{i,alone}} \quad (6.7)$$

In this equation, IPC_{shared} refers to the performance of an execution

where the LLC is shared among all programs, whereas IPC_{alone} refers to the performance when the program run is alone without any interference.

Baselines We compare Flock against several state-of-the-art shared cache management solutions. In particular, for CPU-CPU systems, we compare Flock against ASM [125], and for CPU-GPU systems, we compare against TAP-RRIP [87]. ASM is a cache partitioning scheme that determines cache partitions based on the estimated performance benefit of increasing an application’s partition size. ASM represents the state-of-the-art for performance-oriented partitioning policies for multi-core CPU systems. TAP-RRIP is a state-of-the-art replacement policy that is specifically designed for CPU-GPU systems.

6.4.1 Comparison With Prior Art

Figure 6.6 summarizes our main results, and we see that Flock outperforms the state-of-the-art for all three evaluated platforms. In particular, for 2-core homogeneous multi-core CPUs, Flock outperforms ASM as it improves performance by 4.7% over LRU, whereas ASM improves performance by only 1.0% (see Figure 6.7a for per-mix speedups). For 4-core heterogeneous big.Little multi-core CPUs, Flock outperforms ASM and improves performance by 6.6% on average (see Figure 6.7b for per-mix speedups). Finally, for heterogeneous CPU-GPU systems, Flock improves performance by 12.5%, whereas TAP-RRIP, which is specifically designed for CPU-GPU systems, improved

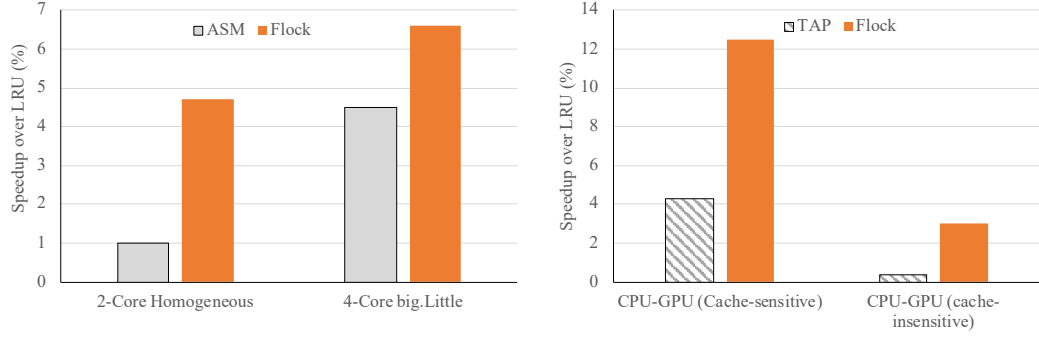


Figure 6.6: Performance improvement of evaluated replacement policies normalized to LRU on homogeneous and heterogeneous systems

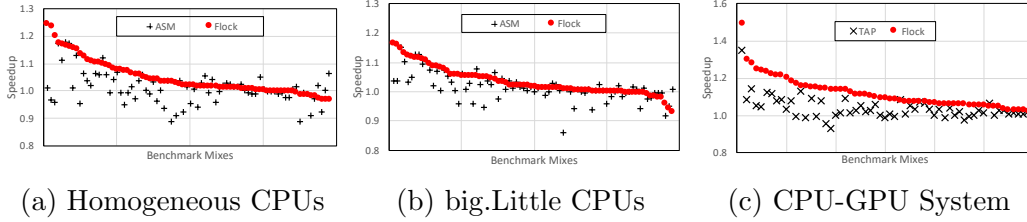


Figure 6.7: Speedup normalized to LRU for two cores sorted by Flock. Flock outperforms ASM on homogeneous CPU-CPU systems, heterogeneous CPU-CPU systems, and heterogeneous CPU-GPU systems.

performance by only 4.3% (see Figure 6.7c for per-mix speedups). These results indicate that Flock is a general cache management solution for homogeneous and heterogeneous platforms.

6.4.2 Why Flock Works

We now provide some detailed results to explain why Flock works well.

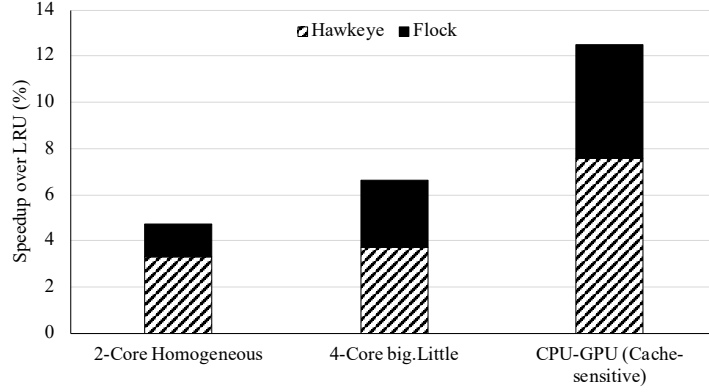
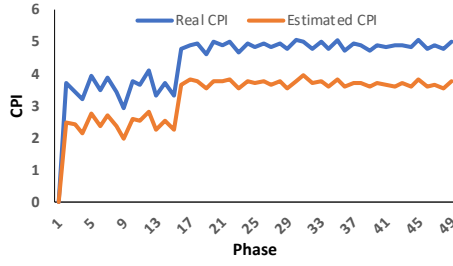


Figure 6.8: Decomposing Flock’s benefits.

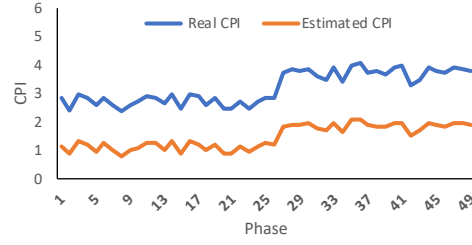
Flock vs. Hawkeye Since Flock benefits from many of Hawkeye’s strengths in reducing cache misses, Figure 6.8 decomposes Flock’s performance into benefits that come from Hawkeye and benefits that are due to our performance and sharing model. We make several observations.

First, we see that surprisingly, Hawkeye alone would have outperformed ASM for multi-core CPU systems (both homogeneous and big.LITTLE), and Hawkeye alone would beat TAP for CPU-GPU systems. This observation suggests that smart replacement policies focused on hit rates can do better than performance-oriented solutions that use less sophisticated replacement schemes.

Second, the benefit of Flock’s performance model over Hawkeye improves as we move to more heterogeneous systems. In particular, on a homogeneous CPU-CPU system, Flock contributes 1.4% extra performance, whereas on a CPU-GPU systems, Flock contributes 5.0% extra performance. Thus,



(a) Execution on big core.



(b) Execution on LITTLE core.

Figure 6.9: Comparison of Real CPI and Flock’s estimated CPI for *xalancbm*. Flock’s estimated CPI closely tracks real CPI.

Flock’s performance-driven decision making is more beneficial in heterogeneous systems because better aggregate cache hit rates—which is what Hawkeye aims for—is less likely to correspond to better performance in heterogeneous systems.

We next present results to highlight the strengths of Flock’s performance model and sharing model.

Flock’s Performance Model Figure 6.9 shows that for *xalancbm*, Flock’s estimated CPI tracks the real CPI well for both on an out-of-order CPU (big core) and an in-order CPU (LITTLE core). While the absolute values between the real CPI and the estimated CPI differ, the changes in IPC can be tracked accurately, and these precise estimations help us adjust cache allocations based on estimated speedups.

Figure 6.10 shows that Flock’s accurate performance estimation translates into better partitioning decisions—and better performance—when *xalancbm*

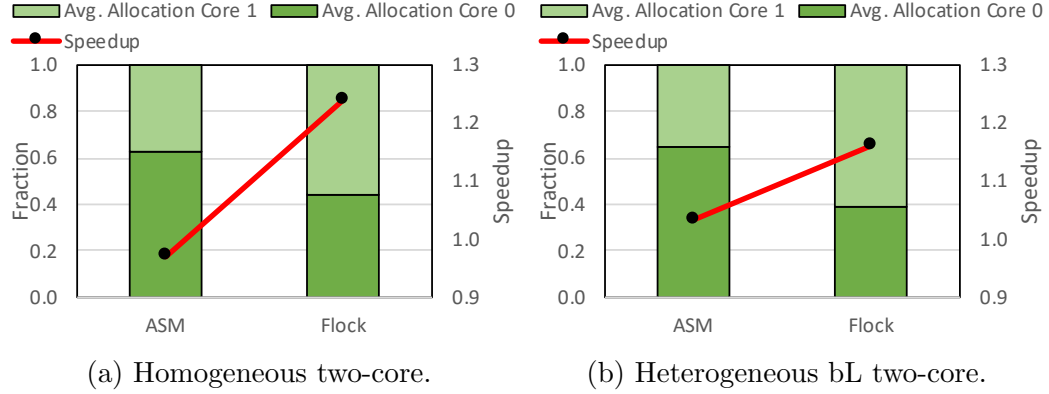


Figure 6.10: Flock’s partitioning differs from ASM because it takes a holistic view of performance and allocates more cache space to core 1. Flock achieves higher performance than ASM due to better partitioning.

shares the LLC with *mcf* on a two-core homogeneous and heterogeneous big.LITTLE system. In particular, we see that for this mix, ASM allocates 63% of the shared cache capacity to *mcf* because it has an extremely high LLC access rate. Flock, on the other hand, takes a holistic view of the memory hierarchy and allocates only 43% of the cache to *mcf*, allocating the remaining 57% of the cache to *xalancbmk*. Flock’s decision is prudent because *mcf* has a low L1 hit rate of 45% and the extra hits in the LLC mitigate only a small proportion of *mcf*’s memory accesses. By contrast, *xalancbmk* has a high L1 hit rate of 80%, so the extra LLC hits are able to mitigate a large proportion of its remaining memory bottleneck.

Thus, as we argued in Section 6.1, Flock is able to increase system throughput by taking a global view of performance instead of focusing on the LLC in isolation. On a homogeneous CPU multi-core, Flock improves perfor-

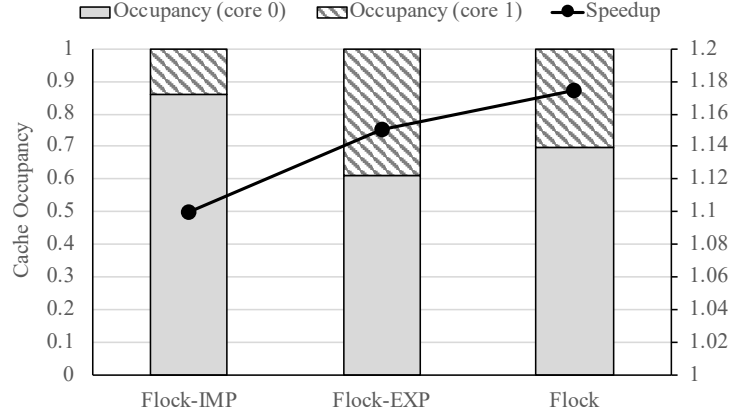


Figure 6.11: Average partition allocations of each core and their actual cache occupancy when running 470.lbm and 401.bzip2 mix with three different sharing models. Flock’s sharing model achieves the best performance.

mance for this mix by 24% over LRU, whereas ASM degrades performance by 3%. On a heterogeneous big.LITTLE system, Flock improves performance for this mix by 16%, whereas ASM improves performance by only 3.3%.

Flock’s Sharing Model To evaluate different sharing models, Figure 6.11 compares the cache occupancy and performance of three variants of Flock on a mix of *lbm* and *bzip* running on a 2-core system. The three variants include (1) Flock-EXP, which is a version of Flock that enforces the desired partitions explicitly using way partitioning, (2) Flock-IMP, which is a version of Flock that uses partition-driven replacement, but does not use relative aging, and (3) vanilla Flock, which uses both partition-driven replacement and relative aging.

Figure 6.11 shows that Flock’s sharing model achieves the best speedup

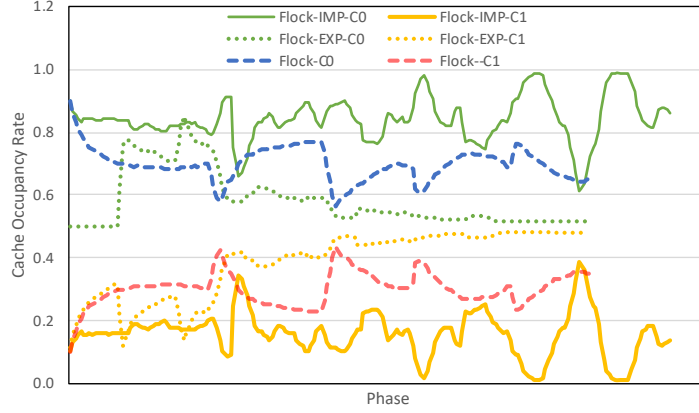


Figure 6.12: Cache Occupancy of Flock, Flock-IMP and Flock-EXP when running 470.lbm and 401.bzip2 mix on a 2-core homogeneous system. Flock strikes the right balance between explicit partitioning and implicit replacement.

of 17.4%, whereas Flock-EXP and Flock-FCP achieve speedups of 15.5% and 10.0% respectively. In particular, we see that Flock-EXP allocates 61% of the cache to core 0, which is close to the prescribed partition by the performance model. Unfortunately, Flock-EXP loses performance because of its rigidity in enforcing partitions. Flock-IMP, on the other hand, is more flexible because it is replacement-based, but with Flock-IMP, core 0 ends up occupying more than 86% of the space in the cache because lbm issues $4 - 7\times$ more LLC requests than bzip. Thus, the flexibility of replacement-driven solutions comes at the cost of this pathology. Relative aging mitigates this undesirable behavior as it steers the cache replacement policy to dedicate more space to core 1, reducing Flock-IMP's cache occupancy from 86% to 69% and improving performance from 10% to 17.4%.

Figure 6.12 shows in more detail the cache occupancy of the three

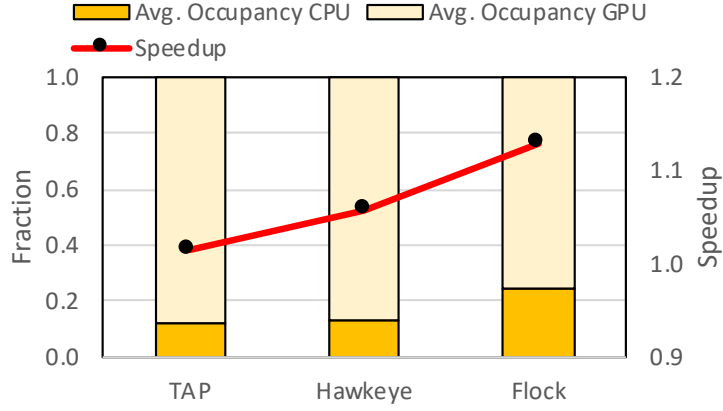


Figure 6.13: Cumulative cache occupancy and speedup on a CPU-GPU running 401.bzip2 and histogram. Flock achieves better performance than TAP and Hawkeye because it dedicates more cache to the CPU.

solutions over time, and we confirm that Flock hits the middle ground between Flock-EXP and Flock-IMP as (1) it is more adaptive than Flock-EXP, and (2) it is more balanced in cache allocations than Flock-IMP.

CPU-GPU Systems Finally, Figure 6.13 shows the Flock effectively manages the disparate cache intensities of a CPU and GPU. In particular, on a mix of bzip and histogram running on a CPU and a GPU respectively, Flock is able to allocate 25% of the cache capacity to the CPU despite the GPU issuing $78\times$ more cache accesses than the CPU; Flock’s cache allocation results in a 12.9% speedup over LRU. By contrast, TAP allocates only 12% of the cache capacity to the CPU and sees a performance improvement of only 1.5%.

Figure 6.14 shows how the cache allocations (top graph) and IPC (bottom graph) of the CPU and GPU over time for this mix. In particular, we

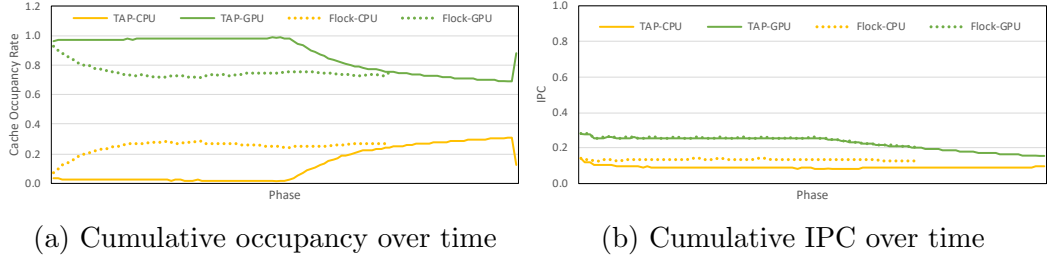


Figure 6.14: Cache Occupancy and IPC of each core over time when running bzip and histogram mix using Flock on a **heterogeneous CPU-GPU** system. Compared to TAP, Flock intelligently allocates more cache to the CPU. This results in better performance for the CPU and no performance difference for the GPU.

see that TAP allocates the entire cache to the cache to the GPU, but later it steals some cache space from the GPU to allocate it to the CPU. By contrast, Flock makes a wise decision about cache allocation much earlier: It allocates about 25% of the cache to the CPU and consequently observes a higher IPC for the CPU. Note that reducing the GPU’s cache allocation did not reduce the GPU’s performance for Flock.

Thus, we conclude that Flock is able to determine better cache allocations for the CPU and GPU with its FCP-based performance model. Moreover, it is able to better enforce these partitions with its partition-driven replacement strategy.

6.4.3 Epoch Size Sensitivity

Figure 6.15 shows Flock’s speedup on 2-core homogeneous system, as we change epoch sizes. We see that Flock’s benefits improve as we increase

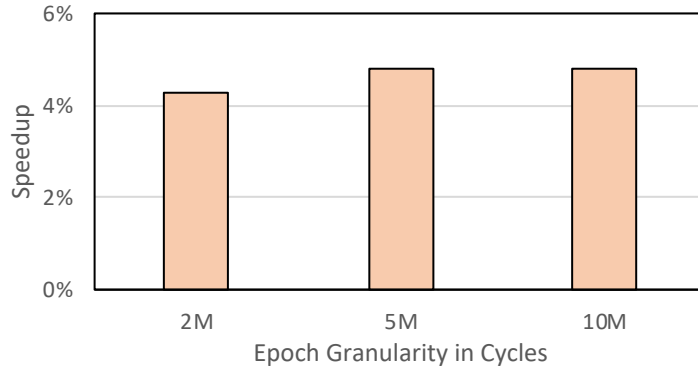


Figure 6.15: Flock’s sensitivity to epoch sizes.

epoch size from 2 million instructions to 5 million instructions, but they do not change for epoch sizes beyond 5 million instructions.

6.5 Summary and Conclusions

The trend towards increased hardware specialization points to a future with heterogeneous multi-cores that share a last-level cache. In this chapter, we have presented the first solution for managing such shared caches without making strict assumptions about core characteristics. In particular, we have presented Flock, a core-agnostic cache management scheme that (1) automatically infers for each core the performance impact of changing its allocation of the shared cache, and that (2) manages the disparity in cache pressure from different cores. The key ideas are to use a holistic—yet simple—performance model to determine cache allocation for each core, and to use a novel partition-driven replacement scheme to loosely enforce these cache allocations.

Our evaluation also reveals some interesting trends for cache replace-

ment research. First, cache replacement technology for homogeneous CPU multi-cores has improved to the point that state-of-the-art replacement policies, as embodied by the Hawkeye replacement policy, already outperform the best solutions on heterogeneous systems. Second, for maximizing aggregate per-core performance, replacement-based management of shared caches is generally more effective than rigid cache partitioning. Thus, we expect that like Flock, future research in cache management will be able to take a unified view of homogeneous and heterogeneous systems, and it will benefit from the rich literature in cache replacement.

Chapter 7

Low-Latency Task-Level Accelerator Offload

So far, we have discussed data movement optimization techniques to move and keep data as close to its compute as possible. An alternative approach is to move the actual compute closer to the data. This, however, requires efficient mechanisms for offloading computations to near-memory accelerator devices. In this chapter¹, we introduce the non-uniform compute device (NUCD) system architecture (SA) to enable generic, hardware-assisted and driver-less low-latency compute offload to task-level accelerators [20].

7.1 Background

Traditionally, interfacing hardware accelerators within a system requires a driver specific for a targeted device [5, 4, 122, 39]. Figure 7.1 shows the system-wide speedup of several benchmarks where lightweight, fine-grain kernels are offloaded to near-memory accelerators with and without device driver overhead. The speedup shown is normalized to no-offload execution. As can

¹Contents of this chapter previously appeared in:

[20] M. Asri, C. Dunham, R. Rusitoru, A. Gerstlauer, and J. Beard. The Non-Uniform Compute Device (NUCD) Architecture for Lightweight Accelerator Offload, PDP, 2020. I implemented the main ideas and collaborated with the co-authors on writing the paper.

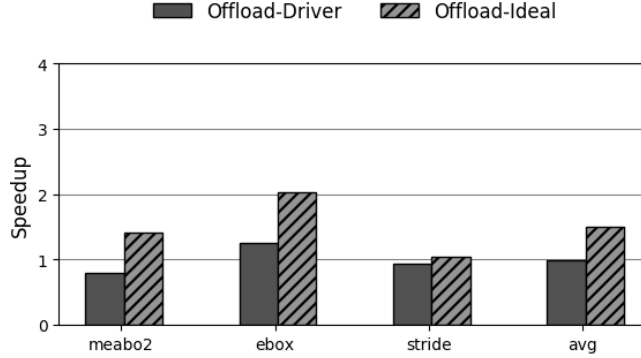


Figure 7.1: System-wide speedup with and without offload overhead.

be seen, for some applications, the actual benefit of hardware acceleration is undone by the cost of driver overhead. Instead of experiencing system-wide speedup, offloading these kernels to accelerators using the traditional device driver approach, can be worse than not offloading them in the first place.

Worse, in complex heterogeneous architectures where compute elements are sprinkled throughout the memory hierarchy, maintaining standard sequential program semantics efficiently is a challenge [83]. Data forwarding and exceptions from remote accelerators can occur naturally. This synchronization and control-flow back to the host core is normally handled by the software device driver (i.e. completion queue). However, leaving such low-level signalling mechanisms to software places a burden on the programmer, which could hamper widespread adoption of heterogeneous architectures. Therefore, a system architecture that provides portable and transparent mechanisms for accelerator offloading while maintaining standard sequential program semantics including exceptions during acceleration is essential.

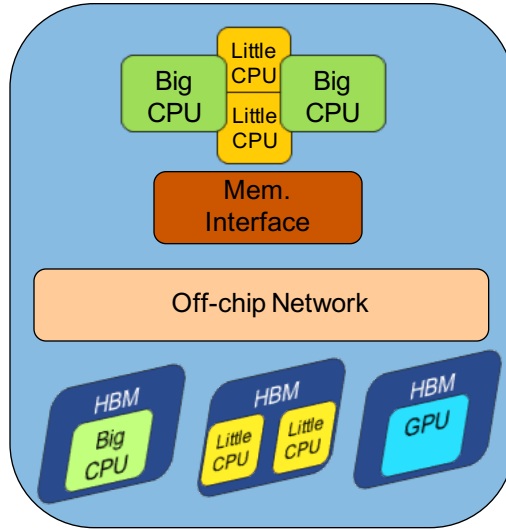


Figure 7.2: Envisioned system architecture.

In the next sections, we detail our approach to address these objectives and associated design challenges

7.2 NUCD Overview

We introduce the non-uniform compute device (NUCD) system architecture (SA) to enable generic, hardware-assisted and driver-less low-latency compute offload to task-level accelerators. NUCD assumes a number of near-memory programmable cores as accelerator devices. They are sprinkled all over the memory hierarchy and some can be integrated into the logic layers of 3D-stacked memories such high bandwidth memory (HBM) modules, as shown in Figure 7.2. Selected kernels are offloaded to these accelerators to take advantage of their greater bandwidth to memory.

To perform such offload, the NUCD SA provides a canonical programming model that adds a single instruction to indicate a region of interest as an offload candidate. This instruction provides fast identification of critical information to setup an accelerator context, such as code region size, memory footprint/granularity, and output/input registers. The NUCD microarchitecture then handles all offload, initiation, synchronization and forwarding of an accelerator context without driver involvement. NUCD tightly integrates with a general-purpose core’s out-of-order mechanisms to maintain standard sequential programming semantics while enabling transparent parallel execution between the host core and accelerators.

7.3 The NUCD Programming Model

We first discuss the NUCD programming model. The NUCD SA employs a `nucd` instruction to initiate and assist in the offloading process. The programmer identifies a region of interest (ROI) in the code by using the `nucd` instruction to mark the offloaded region. This can be done directly in assembly code or by a compiler. In both cases, necessary and relevant offload information is prepared and passed to the micro-architecture through the `nucd` instruction operands. We describe task identification and the `nucd` instruction in the following.

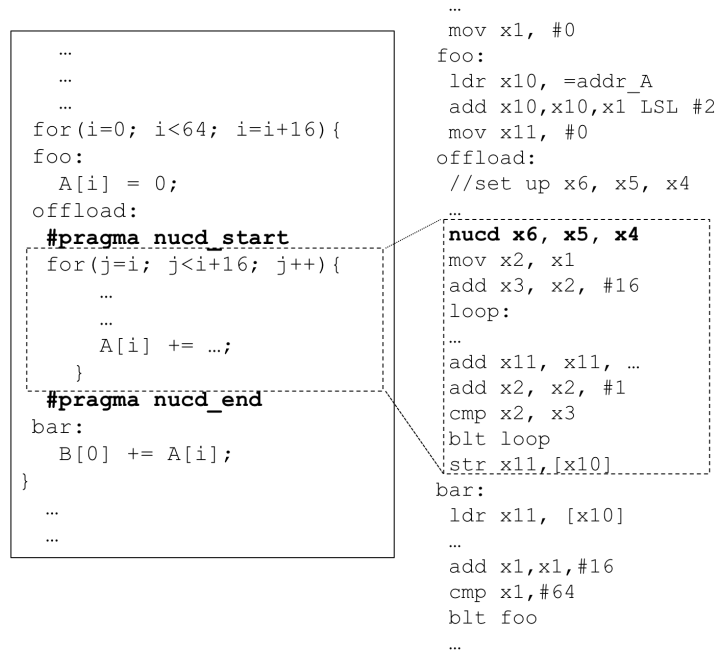


Figure 7.3: Task identification in the NUCD programming model.

7.3.1 Task Identification

Figure 7.3 illustrates the offload task identification in the NUCD architecture. The programmer marks the `offload` code block as a ROI for offload in the code using `nucd_start` and `nucd_end` pragmas. The compiler in turn inserts the `nucd` instruction before the instruction block for the `offload` ROI in the assembly code.

7.3.2 NUCD Instruction

Figure 7.4 details the format and content of the three register operands used by the `nucd` instruction. Register `Xn` indicates the base offload information, while registers `Xd` and `Xt` contain more information about the inputs and

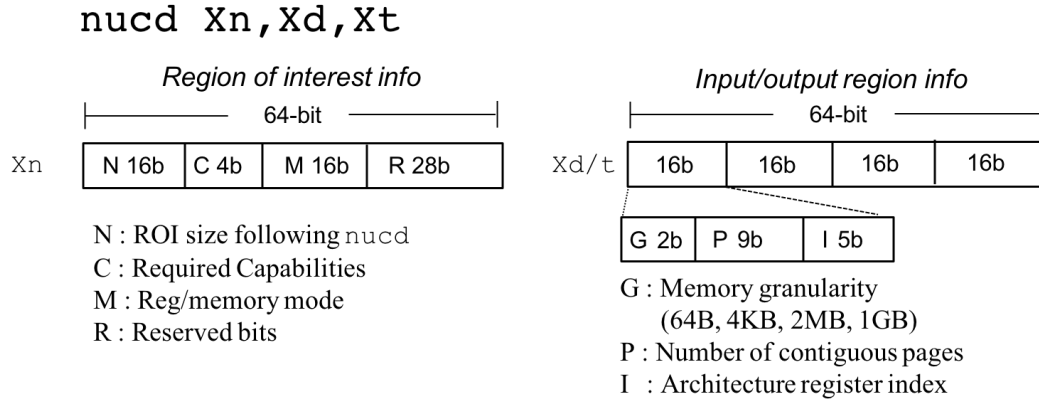


Figure 7.4: Register format of the **nucd** instruction.

outputs required by the offloaded task, respectively. **N** in register **Xn** indicates the size of the offloaded region, i.e. the number of instructions to offload following the **nucd** instruction. **C** denotes capabilities needed by the offloaded code, and **M** contains additional information such as whether the offloaded code uses registers and/or memory locations as inputs and outputs. The rest of the bits are reserved (**R**). For **Xd/Xt**, information about up to four input/output registers or memory regions can be provided. Each 16-bit segment contains 2 bits to describe the memory granule size, 9 bits to describe the size of the memory region given as the number of contiguous granules of memory, and 5 bits to point to a host architecture register index. If the offloaded task uses memory locations, this register index will point to the starting address of that memory location. With these four segments, **nucd** instructions can conceptually support up to four live input and output registers or memory locations. To simplify the register dependence logic, the return value is limited to a single dependent register in our current implementation.

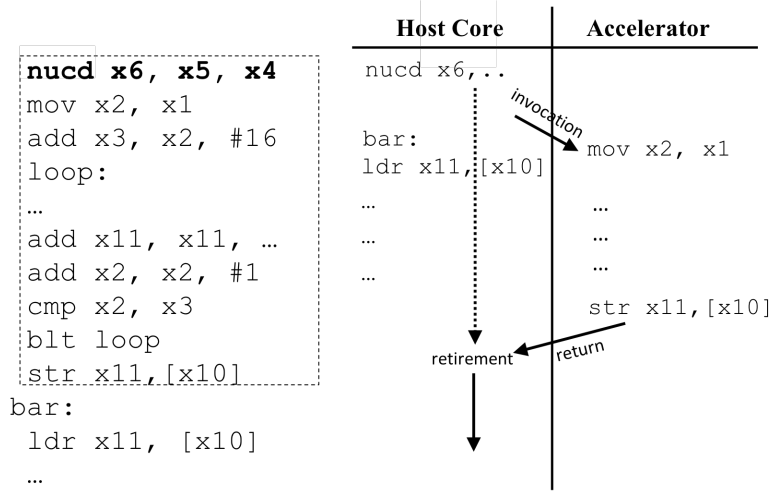


Figure 7.5: Offload flow from the host core to the accelerator and back.

7.3.3 Offload Flow

Figure 7.5 shows the general offload flow for executing the **nucd** instruction. Once the **nucd** instruction and all of its information is decoded, the micro-architecture triggers the offload process. The host core first invokes delegation of the ROI to the accelerator by transferring the context to the accelerator. Once it is sent, the host core continues fetching and executing the instructions following the NUCD sequence (**bar**). At this point, the accelerator and host core can operate in parallel. During acceleration, standard sequential program consistency is maintained through dependency checks in the NUCD microarchitecture. This allows host and accelerator to execute concurrently as long as no dependencies dictated by sequential program semantics are violated. Finally, when the accelerator finishes the offloaded task and returns, the **nucd** instruction is finally retired and the offload completes.

7.3.4 Multi-Task Offload

Note that with this support for concurrent execution with the accelerator, the host can encounter multiple `nucd` instructions in the stream and hence can offload several concurrent in-flight NUCD tasks, as shown for the example of a parallelized loop in Figure 7.3. The outer loop in the figure spawns four NUCD tasks operating independently on the regions specified by the inner loop. As discussed in the previous section, after launching the first task in the first loop iteration, the host core will continue executing instructions and additional loop iterations. Since iterations are independent, this will result in all four tasks to be successively launched and offloaded such that they can execute concurrently. We will explain how we enhance the NUCD micro-architecture to maintain consistency among in-flight tasks in the following section.

We describe the NUCD micro-architecture extensions in this section. At its core, NUCD extends the standard out-of-order (OoO) micro-architecture to support a driverless offload process where all procedures are handled by the hardware. Moreover, the extended micro-architecture also enables NUCD to maintain standard sequential program semantics during acceleration. OoO execution between the region of interest and the host core is made possible by integrating NUCD support into the host core’s existing dependency tracking mechanisms.

Figure 7.6 demonstrates the NUCD system architecture. NUCD extensions are marked in blue color. A standard OoO architecture is extended with an offload engine for accelerator invocation, and dependency and retirement

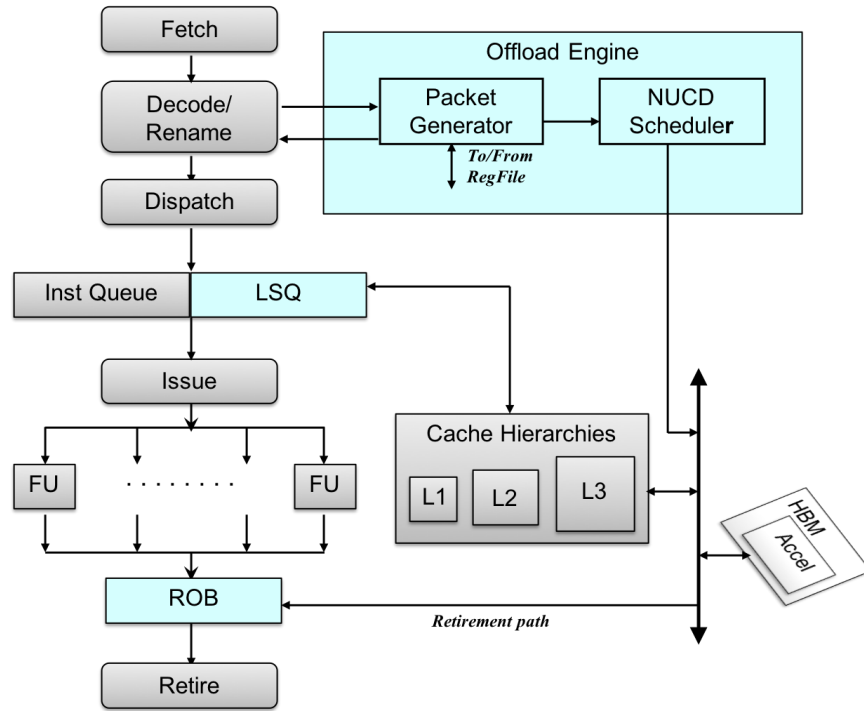


Figure 7.6: NUCD system architecture.

tracking integrated into the existing host core load-store queue (LSQ) and reorder buffer (ROB) structures. The offload engine is responsible for handling the accelerator invocation and context transfer process once the `nucd` instruction is decoded. The LSQ and ROB handle dependency tracking and retirement on the return path back from the accelerator while allowing for out-of-order execution between offloaded tasks and the host core. Further details of the offload process and relevant components will be described in the following sections.

7.4 The NUCD Micro-Architecture

7.4.1 Accelerator Invocation and Context Transfer

Once a `nucd` instruction is decoded, the host core prepares the accelerator context and performs the context transfer to the accelerator device. All instructions prior to the `nucd` instruction are retired before the `nucd` instruction is issued, effectively creating a sequential barrier. This guarantees that any dependent input registers for the NUCD offload region are ready and have the latest architectural value before the offload begins. Moreover, to preserve sequential memory consistency, the host core flushes any modified lines needed for NUCD execution to memory (i.e., from memory ranges specified through the instruction). This guarantees that the latest data is resident in memory when an accelerator core reads from locations that have been modified but not written to memory prior to offload.

Based on the register/memory information encoded in the `nucd` instruction, the host core then further arranges the context transfer by assembling a *packet* consisting of necessary information for the accelerator to execute the ROI. This includes the program counter of the first and last instruction in the region of interest as well as values of all needed live input registers. Once the *packet* is created, it will be sent to the target accelerator's work queue. After the context transfer completes, the program counter of the host core is set to the instruction following the NUCD sequence.

Figure 7.7 shows the required extension on the host core architecture to support NUCD invocation and context transfer. A standard OoO core pipeline

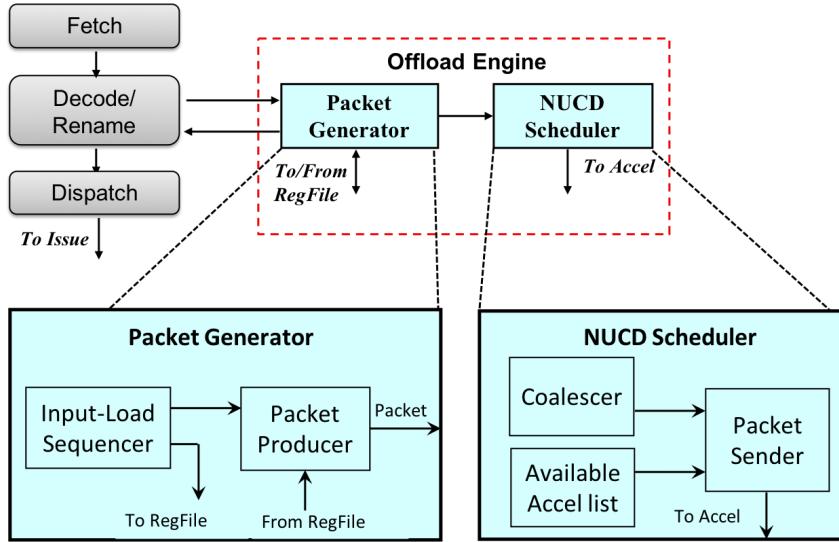


Figure 7.7: Extended host core architecture for invocation and context transfer.

is enhanced with two necessary components for hardware-offloading purposes: a *Packet Generator* and a *NUCD Scheduler*. The *Packet Generator* obtains signals from the decode stage that identify if a `nucd` instruction is in-flight. The front-end of the core is stalled until the context creation and context transfer is completed. In the beginning of the offload process, the *Input-Load Sequencer* issues a sequence of operations to read relevant registers such as the program counter identifying the beginning of the region of interest and live input registers using encoded information from the `nucd` instruction. Once the relevant information is collected from the register file, the *Packet Producer* assembles the packets for accelerator offload. Afterwards, *NUCD Scheduler* takes the packets and coalesces them if necessary. When the coalesced packet is ready, the *Packet Sender* selects a target from the pool of available accelerators

and offloads the final *packets* to the accelerator’s memory-mapped work queue.

The offload engine also handles dependency checks to maintain sequential consistency among multiple in-flight **nucd** instructions, i.e. multiple concurrent task offloads. In general, different NUCD tasks can proceed concurrently if they are independent, i.e. if their input and output data regions do not overlap, or if there is only an overlap in their input regions. In all other cases, i.e. if there is an overlap and hence data dependency or anti-dependency between an output region of one task and an input or output region of another task, the younger task is blocked and not allowed to proceed. The NUCD scheduler in the offload engine can be equipped with a dependency table that tracks input and output region dependencies among in-flight NUCD tasks. Alternatively, the memory consistency tracking logic in the load-store queue (LSQ) can be extended to provide task dependency information to the NUCD scheduler. In either case, with such information, the scheduler can delay any task offloads that violate memory consistency until any older dependent task completes.

The packet generator can be seen as a small-entry instruction/data queue for a maximum of four dependent input registers and context data. Moreover, the NUCD scheduler can be designed as a write buffer consisting of few cacheline entries for the context packet, and additional control logic for a coalescer. Such capabilities in general already exist in a standard LSQ design. Thus, the area overhead of the offload engine is similar to that of a conventional LSQ design. Moreover, the main operations performed by the offload engine

are majorly register file reads and issuing memory requests. They can run in parallel to the main pipeline and are not on the critical path.

7.4.2 Accelerator Execution and Address Translation

Once the context packet reaches the accelerator’s work queue, the accelerator loads the context and starts execution of offloaded code. During execution, the accelerator accesses conventional user-space virtual addresses. As such, translation faults can potentially occur during the execution. Any accelerators that need to interact with a virtual memory system via an input/output memory management unit (IOMMU) often can only efficiently run relatively regular applications [16, 130, 136], which are the opposite of the types of applications that benefit from near-memory acceleration. In fact, translation overheads for accelerators can degrade performance by up to 50% [22]. Hence, low-overhead translation mechanism in accelerator devices, especially near-memory ones, are a critical component for performance. The NUCD SA utilizes a simplified mechanism to mitigate overhead for address translation and memory management on the accelerator side. User-space virtual addresses accessed by offload code are allocated to a physically contiguous memory region by the OS. Translation from virtual to physical addresses on the accelerator core is accomplished using the physical base address of each set of contiguous pages (often termed *range-based* translation [55]). If a page is not loaded in memory, the fault is taken in the host core, freeing the accelerator from having to tolerate memory page faults. In the NUCD SA, the

OS memory/slab allocator is modified to allocate virtual ranges that are also contiguous in physical space, which is a common existing approach for many networking and accelerator applications and supported by the Linux kernel allocator *kmalloc* by default [115]. A different design with the ability to handle non-contiguous physical address spaces can also be provided. However, this requires virtual-to-physical translation support and overhead in NUCD devices.

7.4.3 Accelerator Retirement and Sequential Consistency

While the accelerator executes the ROI, from the host core’s perspective, the `nucd` instruction continues to execute and retires only when the ROI completes on the accelerator. This enables the host core to resume execution of the following instructions past the NUCD sequence. Once execution of the ROI on the accelerator is done, a completion signal is sent to the core from the accelerator devices. Moreover, if the acceleration produces an output, the value of the corresponding accelerator register is transferred back to the host core along with the completion signal. The host core then retires the `nucd` instruction and the entire offload process completes.

To maintain sequential program semantics during concurrent accelerator and host executions, the NUCD SA uses the support of existing out-of-order host core mechanisms for consistency management, namely scoreboarding and the load-store-queue (LSQ) for tracking of read-after-write (RAW) dependencies, and the Reorder Buffer (ROB) for handling of write-after-write (WAW),

and write-after-read (WAR) output, and anti-dependencies. We explain how different dependency tracking and sequential consistency management is supported by scoreboarding, LSQ, and ROB in the following subsections.

7.4.3.1 Scoreboarding

For consistency tracking of RAW dependencies between output registers produced by the offloaded task accessed by any following instructions executed on the host core, NUCD makes use of the standard scoreboarding mechanism. Any instructions past the NUCD sequence that are dependent on the output register of the offloaded task will be stalled in the issue stage until acceleration finishes.

7.4.3.2 Dependency Tracking with LSQ

The LSQ further tracks RAW dependencies between any subsequent memory load instructions dependent on the acceleration results. Any reads originating from the host core to NUCD output memory locations that are potentially written by the offloaded task are blocked by the LSQ while independent reads can progress to the issue stage normally. This is achieved by tracking the address range of the output region to which a NUCD task is writing results. The LSQ is augmented to store the base address and number of consecutive pages that are required by the NUCD offload task. Then, every load from the host during acceleration will be checked against the NUCD task's output region to identify whether a RAW dependency exists. In case of

a dependency, the LSQ blocks the instruction and schedules it only once the acceleration completes. Any independent load/store instructions can progress without waiting for the NUCD task to retire. This mechanism maintains memory dependency-tracking between the accelerators and the host.

7.4.3.3 Sequential Retirement with ROB

Finally, the ROB ensures that instruction retirement and hence committing of data to registers and memory follows the program order to guarantee sequential consistency of memory and register writes and thus avoid any WAW and WAR hazards between the host CPU and accelerator. While any younger and independent instructions after the `nucd` sequence can be scheduled to issue, their retirement in the ROB has to wait until the acceleration finishes execution. Once the NUCD kernel has finished, a signal is sent back to the ROB to indicate that the operation completed or threw an exception. If an exception is thrown, the operation is handled based on the offset from the NUCD program counter. If no exception has occurred, the return register is committed as the ROB head reaches the `nucd` instruction. Note that the near-memory accelerator will have committed all its memory data by the time it finishes execution and notifies the ROB. The `nucd` instruction is then retired, deallocated and the operation completes.

An alternative design choice can opt for a less conservative synchronization mechanism. It is possible to speculatively let the `nucd` instruction commit from the ROB such that it will not block any younger instructions

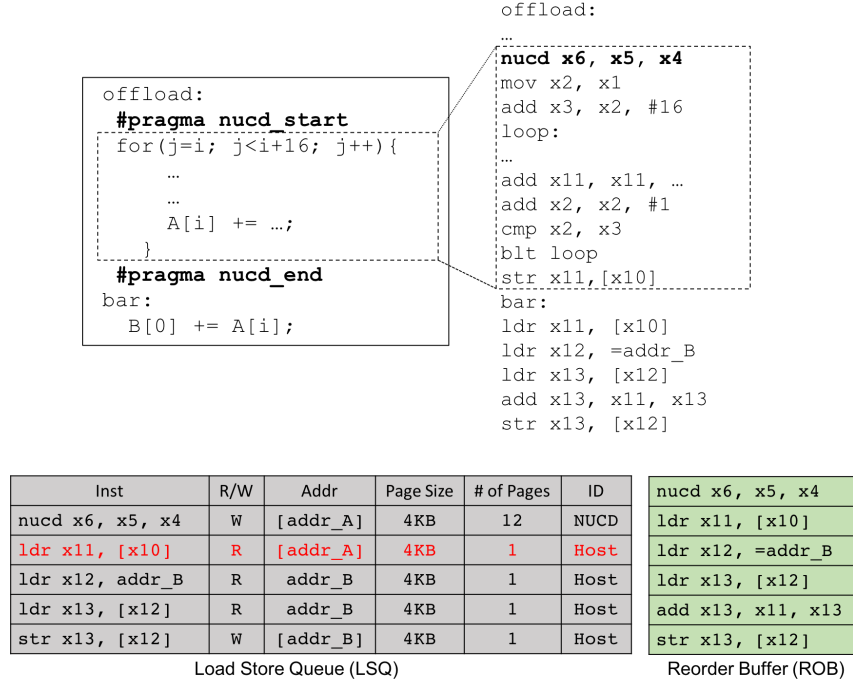


Figure 7.8: LSQ and ROB snapshot of the host core during acceleration. An instruction marked as red in the LSQ is blocked until NUCD acceleration completes to satisfy dependencies. Other instructions can execute subject to sequential ROB retirement.

from committing. However, this requires a checkpoint mechanism to detect violations and safely roll back the architectural state. Prior work has used eviction timing of any modified lines under speculative execution to detect violations and roll back accordingly [27]. In our work, we do opt for the ROB as synchronization point due to its simpler design.

7.4.3.4 Putting It All Together

Figure 7.8 illustrates an example of LSQ and ROB operation during offload execution. Once decoded, an entry is allocated in the ROB for the

`nucd` instruction, similar to standard decoding and ROB allocation in the OoO pipeline. In addition, an entry for the `nucd` instruction is allocated in the LSQ. It contains information about the memory locations to which the NUCD task writes.

The offload process then begins as described in Section 7.4.1. Once the context transfer completes, the accelerator starts execution of the ROI. At the same time, the host core program counter will branch to the code following the offloaded region (*bar*, with respect to Figure 7.5). The host core then decodes instructions and allocates ROB and LSQ entries from there. On encountering a load instruction `ldr x11, [x10]` that reads a value from the address region dependent on the NUCD task, the LSQ will block the execution of the load and wait until the NUCD-offloaded task completes (marked in red), therefore avoiding memory consistency violations. By contrast, the following instruction `ldr x12, addr_B` reads a value from a memory location `addr_B`, and since there is no read-after-write dependency between the instruction and the NUCD-offloaded task, the LSQ can proceed and issue the instruction. Moreover, while the following `ldr x13, [x12]` does not depend on the NUCD-offloaded task, it has a read-after-write dependency with the previous instruction (a.k.a `ldr x12, addr_B`). Thus, it will execute once the preceding instruction completes. Afterwards, the host core performs a store to `addr_B`, which is independent from the NUCD task. As such, the LSQ will go ahead and issue it, subject to sequential retirement in the ROB. The process repeats until there are no more instructions or ROB or LSQ become

full.

On task completion, the ROB ensures that retirement from the accelerator to the host core follows sequential order. As shown in Figure 7.8, while any younger instructions after the `nucd` call are able to issue and execute, their retirement in the ROB has to wait until the acceleration finishes and the `nucd` instruction retires.

All in all, together with scoreboarding and LSQ tracking, retirement and committing of instructions in program order in the ROB enable the NUCD architecture to maintain sequential memory and register consistency while overlapping accelerator with host core executions. This guarantees standard sequential program semantics during acceleration while exploiting any additional concurrency available on the host core.

7.5 Experiments and Results

We evaluate the NUCD system architecture using the `gem5` full-system simulator [25]. A 64-bit Arm architecture was used as a baseline in a configuration to emulate an out-of-order host core with similar characteristics to an Arm A57 [10]. In-order 64-bit Arm cores (with similar characteristics to an Arm A53 [9]) in configurations with 4, 8, 16, and 32 cores are employed as near-memory accelerators on a cross-bar network interposed between the main coherent network and the HBM1 memory controller (with 8 primary channels). This configuration enables the memory controller to use a standard interleaving pattern while allowing the accelerator and host cores to access the

	Host CPUs	Near-Memory Accelerators
ISA	ARMv8 (64-bit)	ARMv8 (64-bit)
Core Configuration	1 OoO core [10]	4-32 in-order cores [9]
L1 I/D Cache	32 KB, 2-cycle	32 KB, 2-cycle
L2 Cache	1 MB, 12-cycle	N/A
HBM Config	HBM Gen1 [86]	
HBM Peak-BW	128 GB/s	

Table 7.1: Summary of simulation parameters.

Benchmark	Option	Input	Total Dyn. Inst.	ROI	
				Dyn. Inst.	Calls
Meabo2 [18]	Phase2	8,000 elements	73,235	73,100	1
Meabo5 [18]	Phase5	8,000 elements	89,234	89,102	1
Ebox [57]	Stride 8	8,000 elements	392,909	392,701	1
Stride [1]	Distance 8	4,096 elements	1,506,743	40,129	8
Spatter [103]	Distance 8	8,000 elements	83,770	83,450	1
SpMV [49]	N/A	circuit 1 (D)	277,875	277,650	1

Table 7.2: Workload parameters.

memory fabric at full bandwidth. We use HBM1 as the main memory in a standard configuration [86]. We compare accelerations under NUCD offload with a driver-based mechanism that uses traditional memory-mapped device actuation along with standard doorbell interaction mechanisms following the driver implementation described in [5]. We model NUCD offload engine and context transfer overhead assuming standard register file and memory transaction costs. All simulation parameters are summarized in Table 7.1.

We use several fine-grain benchmarks that represent key data-intensive kernels widely used across applications. Table 7.2 summarizes the parameters, inputs, and offloaded ROI characteristics of the benchmarks. Meabo [18] is a multi-phased memory-intensive benchmark. Phase2 (meabo2) and Phase5

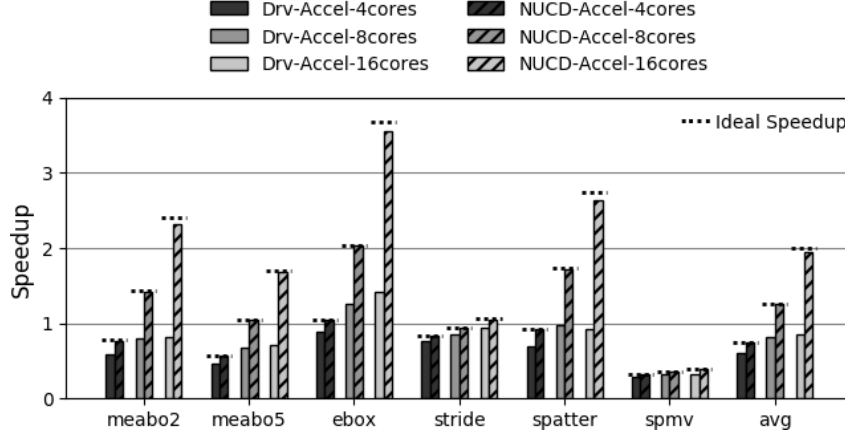


Figure 7.9: Speedup under different offload mechanisms with varying accelerators.

(meabo5) are selected for the evaluation. Both phases access memory using a random indirection vector. They employ one and two indirection vectors, respectively. Ebox [57] is an extended box filtering approximation of Gaussian convolution. Stride [1] is a benchmark stressing memory systems with several light compute kernels. SPATTER [103] is a benchmark for timing scatter/-gather kernels. Finally, SpMV [49] is the sparse matrix-vector multiplication in CSR format. We manually partition each ROI call into as many tasks as the number of accelerators, launching one task to one accelerator.

7.5.1 Speedup

Figure 7.9 shows the system-wide speedup across different applications with 4, 8 or 16 accelerator cores. We compare the original host-core execution to execution with different near-memory accelerators under driver-based vs. NUCD offload. The figure also indicates the ideal theoretical speedup that can be achieved assuming zero offload overhead for each accelerator configuration.

Results show that accelerations with NUCD offload can yield up to 3.6x speedup across different applications and accelerator configurations, averaging 21%-128% performance improvement over driver-based offload on 4-16 accelerator cores. In all cases, NUCD speedups are equal or close to the theoretically achievable maximum. Closer observation of *meabo2*, *meabo5*, *stride*, and *spatter* shows that NUCD-based offload can unlock performance benefits that do not exist with device driver offload. While driver-based offload results in performance slowdown compared to original host execution even when using 16 accelerator cores, the NUCD mechanism can achieve 1.05x-2.6x speedup for such configurations. Note that even with ideal offload, no performance benefits are observed from offloading to 4 devices across most applications due to the limited compute and reduced bandwidth available with fewer of the simpler accelerator cores. In case of *spmv*, significant slowdowns are seen that do not improve even when the number of accelerator cores increases. This is due to the data access behavior and unbalanced work distribution in *spmv* contributing to poor performance on the in-order accelerator cores and limited task-level parallelism among cores.

7.5.2 Offload Overhead

In general, driver offload of a single task to one accelerator core can take up to 40k cycles; predominantly stemming from the overhead of memory-mapping accelerator to the user program. This overhead grows proportionally with the number of tasks and cores. By contrast, in the NUCD SA, such

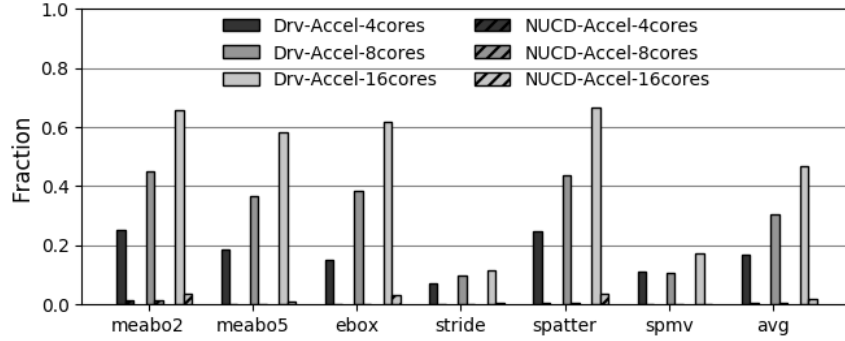


Figure 7.10: Fraction of execution time spent on offload.

overhead is minimized by delegating much of the process to the hardware, requiring only 150-200 cycles per offload with the configuration given in Table 7.1. Figure 7.10 illustrates the fraction of execution time spent on offload for different applications under different offload mechanisms. Driver-based offload can occupy up to 70% of total execution time, as seen in *spatter* and *ebox*. Driver overhead averages 18%, 35%, and 47% for 4, 8, and 16 accelerator cores respectively. By contrast, the hardware-assisted NUCD offload mechanism is able to restrict offload overhead to less than 4% of the total execution time. This in turn results in significantly greater system-wide speedup as shown in Figure 7.9. Note that due to the slowdown experienced under offloading and the longer runtime of the benchmarks, offload overhead occupies only a smaller fraction of total execution time in *spmv* and *stride*. In general, NUCD offload benefits fine-grain tasks the most. In coarse-grained tasks, higher offloading costs are more easily hidden by the longer overall execution times.

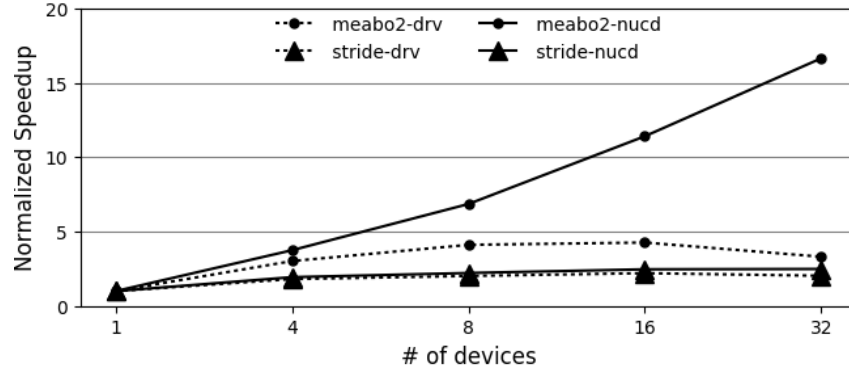


Figure 7.11: Performance scaling with number of accelerator cores on *meabo2* and *stride* applications.

7.5.3 Scaling

Figure 7.11 further shows the speedup scaling under driver-based and NUCD acceleration as a function of up to 64 accelerator devices, normalized to the speedup with one accelerator core. We observe that *meabo2* under NUCD acceleration scales very well as the number of accelerator devices increases. Acceleration under 8 and 16 cores results in 6.8x and 11x speedup gain, respectively, while utilizing 32 cores improves performance by 16x. By contrast, a driver-based offload of *meabo2* is unable to provide similar scalable performance benefits. As shown in Figure 5.9, achievable speedups only increase minimally when adding up to 16 cores. This is due to increasing driver overhead outweighing the acceleration benefits with additional cores. Acceleration with 32 cores in turn results in 20% performance slowdown over execution with 16 cores as acceleration benefits saturate while driver overheads continue to increase proportionally.

On the other hand, *stride* experiences a relatively constant speedup as

the number of accelerator devices increases for both NUCD and driver offload. While acceleration on four cores in NUCD can boost performance by 1.9x, performance benefits saturate when using 8 or more cores in both cases. This is due to a limited number of instructions within the ROI as shown in Table 7.2, where the offload overhead occupies only a smaller fraction of total execution time in *stride*.

Note that in all cases, without support for out-of-order task offload in NUCD, only one task could be in flight at any time, and speedups would be limited to a single accelerator. Out-of-order execution allows independent NUCD tasks to be simultaneously issued to achieve scalable performance.

7.6 Summary and Conclusions

We proposed the NUCD system architecture as a general and hardware-assisted accelerator offload mechanism that provides low-latency and low-overhead offload to accelerators. The NUCD system architecture incorporates a task dispatch mechanism that is tightly coupled with the core micro-architecture to perform context transfer, actuation, and synchronization integrated with out-of-order execution. This dispatch mechanism includes features that ensure memory consistency for accelerators outside of the coherence network using a common interface. The NUCD architecture allows maintaining standard sequential program semantics under concurrent acceleration coordinated in hardware from a main core. Results show that the NUCD architecture can improve performance by 21%-128% across different applications compared

to traditional driver-based offload, achieving an efficient mechanism to move the computation closer to data

Chapter 8

Summary, Conclusions, and Future Research

In this chapter, we summarize our contributions, key results, and possible future research directions.

8.1 Dissertation Summary

In this dissertation, we propose novel data movement optimization techniques for heterogeneous architectures. We show that data movement optimization on heterogeneous systems can be taken care of equally by pure software and hardware-assisted mechanisms. From the software side, we show that careful software/algorithm optimizations such as data tiling/blocking, prefetching, and pipelining can significantly help reduce data movement between accelerators and the host core and unlock further performance improvement. From the hardware side, we demonstrate several mechanisms to optimize data movement overhead including cache-managed fine-grain data staging and pipelining, shared cache management for co-running applications, and a low-cost task-level accelerator offload mechanism. Results show that: 1) applying hardware-assisted fine-grain data staging pipelining can boost performance by up to 23% and achieve energy savings of up to 22% over baseline accelera-

tions; 2) Intelligently managing shared cache for co-running applications can improve performance by up to 12.5% over state-of-the-art solutions; And 3), hardware-assisted low-latency task-level accelerator offload can achieve an average performance improvement of 21%-128% over a conventional driver-based offload mechanism.

8.2 Future Work

We discuss possible future research directions. In CASPHAR, the system software through the programmer has to explicitly mark if a cacheline is ready for consumption with the help of `clflush` instructions. In the future, this can be automated using a compiler support. Moreover, while CASPHAR assumes a non-coherent protocol, future evaluation can include the quantitative impact of coherency protocol on CASPHAR’s results. With coherence, the cache energy consumption would likely increase as a result of snoop/lookup overheads generated by the coherence/snooping protocol. Similarly, further investigations on cache size sensitivity and how it influences the quantitative benefits of CASPHAR can also be considered. In general, if the ratio of problem size and the cache size is small, the application is cache-contained and CASPHAR’s degree of performance benefits would be expected to reduce. Finally, for programs where data is read or written more than once before being discarded, or in case of multiple full-empty scenarios, marking a cacheline “produced” could be performed by a special instruction that is automatically inserted by the compiler leveraging data-flow analyses.

Flock can be improved further to accommodate more components in the memory hierarchy. Future work can investigate an improved partitioning mechanism that accommodates prefetcher activity in the memory hierarchy. Further, alternative designs that can dynamically re-configure explicit partitioning schemes to implicit partitioning can be explored further. A run-time identification of when explicit or implicit partitioning performs better per phase can be investigated. Finally, while Flock considers only shared cache management, future research directions can employ synergistic co-ordination among relevant components. For example, the memory controller can be made aware of priorities enforced in the shared cache level. This includes prioritizing memory requests that are deemed to be more critical than others.

In NUCD, future work can include extended evaluation of a wider range of applications. Particularly, applications that are known to have poor locality with fine-grain tasks repetitively offloaded to accelerators such as graph applications can be further evaluated. Moreover, while NUCD assumes that there is no communication between multiple accelerator devices, synchronization techniques between accelerators sharing data are required for many such applications. Finally, future work can include an automated mechanism to perform run-time profiling to identify potential offload tasks considering their estimated speedup and associated data movement overhead.

Bibliography

- [1] The coral-2 benchmark suite. <https://asc.llnl.gov/coral-2-benchmarks/>.
- [2] Intel software manual. <https://software.intel.com/en-us/forums/intel-isa-extensions/topic/291765>.
- [3] Macsim. <http://comparch.gatech.edu/hparch/macsim.html>.
- [4] Nouveau: Accelerated Open Source driver for nVidia cards. <http://nouveau.freedesktop.org>.
- [5] Open Source Mali GPUs UMP User-Space Drivers. <https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/ump-user-space>.
- [6] Pin tool. <http://www.pintool.org>.
- [7] Single Root I/O Virtualization. http://www.pcisig.com/specifications/iov/single_root/. Accessed October 2019.
- [8] Virtual I/O Device (VIRTIO) Version 1.1. <https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html>. Accessed October 2019.
- [9] ARM Cortex-A53 MPCore Processor : Technical Reference Manual. Arm Inc., 2013.
- [10] ARM Cortex-A57 MPCore Processor : Technical Reference Manual. Arm Inc., 2013.

- [11] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute Caches. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.
- [12] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiawicz, Beng-Hong Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *International Symposium on Computer Architecture (ISCA)*, 1995.
- [13] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *International Symposium on Computer Architecture (ISCA)*, 2015.
- [14] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *International Symposium on Computer Architecture (ISCA)*, 2015.
- [15] G. Alverson, P. Briggs, S. Coatney, S. Kahan, and R. Korry. Tera Hardware Software Cooperation. In *International Conference on Super Computing (SC)*, 1997.
- [16] N. Amit, B. Muli, and B. Yassour. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *International Symposium on Computer Architecture (ISCA)*, 2010.
- [17] R. Anderson, M. Bond, J. Clulow, and S. Skorobogatov. Cryptographic Processors: A Survey. *Proceedings of the IEEE*, 94(2):357–369, January

2006.

- [18] ARM. Meabo. <https://github.com/ARM-software/meabo>, 2018.
- [19] Arm. Revere-amu system architecture. https://pages.arm.com/rs/312-SAX-488/images/ARM-IHI-0078A_ALP-03_b_architecture_external.pdf, 2019.
- [20] M. Asri, C. Dunham, R. Rusitoru, A. Gerstlauer, and J. Beard. The Non-Uniform Compute Device (NUCD) Architecture for Lightweight Accelerator Offload. In *EuroMicro Conference on Parallel and Distributed Processing (PDP)*, 2020.
- [21] M. Asri, A. Pedram, L. K. John, and A. Gerstlauer. Simulator Calibration for Accelerator-Rich Architecture Studies. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2016.
- [22] J.C. Beard and J. Randall. Eliminating Dark Bandwidth: A Data-Centric View of Scalable, Efficient Performance, Post-Moore. In *International Conference on High Performance Computing (HPC)*, 2017.
- [23] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [24] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

- [25] 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, S. Korey, M. Shoaib, N. Vaish, M. Hill, and D.A. Wood. The gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.
- [26] J. Board and K. Schulten. The Fast Multipole Algorithm. *Computing in Science and Engineering*, 2(1):76–79, February 2000.
- [27] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *IEEE Computer Architecture Letters (CAL)*, 16(1):46–50, Jan 2017.
- [28] J. Bradbury, R. Frostig, P. Hawkins, M.J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. JAX: Composable Transformations of Python+NumPy Programs. 2018.
- [29] Nathan Brookwood. AMD Fusion Family of APUs: Enabling a Superior Immersive PC experience. In *AMD White Paper*, 2012.
- [30] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.
- [31] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli. Accuracy Evaluation of GEM5 Simulator System. In *International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2012.

- [32] T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *International Conference on Super Computing (SC)*, 2011.
- [33] C. Cascaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik. A Taxonomy of Accelerator Architectures and Their Programming Models. *IBM Journal of Research and Development*, 54(5):5:1–5:10, September 2010.
- [34] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.
- [35] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
- [36] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, 2018.
- [37] J. Choi and R.W. Vuduc. Analyzing the Energy Efficiency of the Fast Multipole Method Using a DVFS-Aware Energy Model. In *International Symposium on Parallel and Distributed Systems (IPDPS)*, 2016.

- [38] L. Choi and Pen-Chung Yew. Hardware and Compiler-Directed Cache Coherence in Large-Scale Multiprocessors: Design Considerations and Performance Study. *IEEE Transactions on Parallel and Distributed Systems*, 11(4):375–394, April 2000.
- [39] C.J. Rossbach and J. Currey and M. Silberstein and B. Ray and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Symposium on Operating Systems Principles (SOSP)*, 2011.
- [40] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. An Analysis of Accelerator Coupling in Heterogeneous Architectures. In *Design Automation Conference (DAC)*, 2015.
- [41] E.G. Cota, P. Mantovani, and L.P. Carloni. Exploiting Private Local Memories to Reduce the Opportunity Cost of Accelerator Integration. In *International Conference on Supercomputing (ICS)*, 2016.
- [42] D. Lee, J. Choi, J. Kim, S. H. Noh, S. Min, Y. Cho, and C.S. Kim. LRFU: A Spectrum of Policies That Subsumes The Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 50(12):1352–1361, Dec 2001.
- [43] J. Hennessy D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

- [44] R. Desikan, D. Burger, and S. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *International Symposium on Computer Architecture*, 2001.
- [45] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 353–364, 2010.
- [46] D. Lustig and M. Martonosi. Reducing GPU Offload Latency via Fine-grained CPU-GPU Synchronization. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.
- [47] D. Malthora and G. Biros. PVFMM: A Parallel Kernel Independent FMM for Particle and Volume Potentials. *Communications in Computational Physics*, 18(3):808–830, September 2015.
- [48] J. Dongarra and F. Sullivan. The Top 10 Algorithms. *Computing in Science and Engineering*, 2(1):22–79, February 2000.
- [49] A. Elafrou, G.I. Goumas, and N. Koziris. Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors. <http://arxiv.org/abs/1711.05487>, 2017.
- [50] H. Esmailzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In *International Symposium on Microarchitecture (MICRO)*, 2012.

- [51] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro*, 28(3):42–53, 2008.
- [52] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication. *International Conference on Graphics Hardware (EUROGRAPHICS)*, 2004.
- [53] A. Fog. Lists of Instruction Latencies, Throughputs and Micro-operation breakdowns for Intel. www.agner.org/optimize/instruction_tables.pdf.
- [54] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. Range Translations for Fast Virtual Memory. *IEEE Micro*, 36(3):118–126, 2016.
- [55] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal. Range Translations for Fast Virtual Memory. *IEEE Micro*, 36(3):118–126, 2016.
- [56] M. Gao, G. Ayers, and C. Kozyrakis. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *International Conference on Parallel Architecture and Compilation (PACT)*, 2015.
- [57] P. Getruer. A Survey of Gaussian Convolution Algorithms. 2013.
- [58] D. Giri, P. Mantovani, and L. P. Carloni. Accelerators and Coherence: An SoC Perspective. *IEEE Micro*, 38(6):36–45, 2018.

- [59] M. Gokhale, B. Holmes, and K. Iobst. Processing In Memory: The Terasys Massively Parallel PIM Array. *Computer*, 28(4):23–31, 1995.
- [60] K. Goto and R. Van De Geijn. High-Performance Implementation of the Level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1), 2008.
- [61] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver. Sources of Error in Full-System Simulation. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–22, 2014.
- [62] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-Managed Cache Design. In *International Symposium on Computer Architecture (ISCA)*, 2000.
- [63] T.J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *International Symposium on Microarchitecture (MICRO)*, 2016.
- [64] S. Haria, M.D. Hill, and M. Swift. Devirtualizing Memory in Heterogeneous Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.
- [65] J.L. Henning. SPEC CPU2006 benchmark descriptions. 34(4):1–17, September 2006.

- [66] J. Hestness, S. W. Keckler, and D. A. Wood. GPU Computing Pipeline Inefficiencies and Optimization Opportunities in Heterogeneous CPU-GPU Processors. In *International Symposium on Workload Characterization (IISWC)*, 2015.
- [67] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation. In *International Conference on Computer Design (ICCD)*, 2016.
- [68] Intel. Intel Sandy Bridge Microarchitecture. <http://intel.com>.
- [69] A. Jain and C. Lin. Back to the Future: Leveraging Belady’s Algorithm for Improved Cache Replacement. In *International Symposium on Computer Architecture (ISCA)*, 2016.
- [70] A. Jain and C. Lin. Hawkeye: Leveraging Belady’s Algorithm for Improved Cache Replacement. In *The 2nd Cache Replacement Championship*, 2017.
- [71] A. Jaleel, K. Theobald, B. Kevin, Jr. Steely, C. Simon, and J. Emer. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *International Symposium on Computer Architecture (ISCA)*, 2010.
- [72] X. Jian, P. K. Hanumolu, and R. Kumar. Understanding and Optimizing Power Consumption in Memory Networks. In *International Confer-*

ence on High Performance Computer Architecture (HPCA), 2017.

- [73] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer. Architecture Support for Improving Bulk Memory Copying and Initialization Performance. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.
- [74] D. A. Jimenez. Insertion and Promotion for Tree-Based PseudoLRU Last-Level Caches. In *International Symposium on Microarchitecture (MICRO)*, 2013.
- [75] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture (ISCA)*,

June 2017.

- [76] T.S. Karkhanis and J.E. Moreira. IBM Power Architecture. *IBM Research Report*, April 2011.
- [77] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John. A Bandwidth-Aware Memory-Subsystem Resource Management Using Non-Invasive Resource Profilers for Large CMP systems. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.
- [78] D. Kaseridis, J. Stuecheli, and L.K. John. Bank-Aware Dynamic Cache Partitioning for Multicore Architectures. In *International Conference on Parallel Processing (ICPP)*, 2009.
- [79] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, Sept 2011.
- [80] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.
- [81] S. Khan, Y. Tian, and D.A. Jiménez. Sampling Dead Block Prediction for Last-Level Caches. In *International Symposium on Microarchitecture (MICRO)*, pages 175–186, 2010.
- [82] M. Harris L. Durant, O. Giroux. Inside Volta: The World’s Most Advanced Data Center GPU. In *Parallel Forall: NVIDIA Developer Blog*, September 2017.

- [83] L. Lamport. On Programming Parallel Computers. In *ACM Sigplan Notices*, volume 10, pages 25–33, 1975.
- [84] M. LeBeane, B. Potter, A. Pan, A. Dutu, V. Agarwala, W. Lee, D. Majeti, B. Ghimire, E. V. Tassell, S. Wasmundt, B. Benton, M. Breternitz, M. L. Chu, M. Thottethodi, L. K. John, and S. K. Reinhardt. Extended Task Queuing: Active Messages for Heterogeneous Systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.
- [85] D. Lee, P. Sao, R. Vuduc, and A. Gray. A Distributed Kernel Summation Framework for General-dimension Machine Learning. *Statistical Analysis and Data Mining*, 2013.
- [86] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong. 25.2 A 1.2V 8Gb 8-Channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Effective Microbump I/O Test Methods Using 29nm Process and TSV. In *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014.
- [87] J. Lee and H. Kim. TAP: A TLP-Aware Cache Management Policy for a CPU-GPU Heterogenous Architecture. In *International Conference on High Performance Computer Architecture (HPCA)*, 2012.

- [88] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec 2009.
- [89] Z. Li, L. Ju, H. Dai, X. Li, M. Zhao, and Z. Jia. Set Variation-Aware Shared LLC Management for CPU-GPU Heterogeneous Architecture. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018.
- [90] X. Lin and R. Balasubramonian. Refining The Utility Metric for Utility-Based Cache Partitioning. 2011.
- [91] G.H. Loh, N. Jayasena, M.H. Oskin, M. Nutter, D. Roberts, M. Meswani, D.P. Zhang, and M. Ignatowski. A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM. 2013.
- [92] D. Malhotra and G. Biros. A Distributed-Memory Fast Multipole Method for Volume Potentials. *ACM Transactions on Mathematical Software*, 43(2), 2016.
- [93] R. E. Matick, T. J. Heller, and M. Ignatowski. Analytical Analysis of Finite Cache Penalty and Cycles per Instruction of a Multiprocessor Memory Hierarchy Using Miss Rates and Queuing Theory. *IBM Journal of Research and Development*, 45(6), 2001.

- [94] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
- [95] D.E. Maydan, S.P. Amarasinghe, and M.S Lam. Array-Data Flow Analysis and Its Use in Array Privatization. In *International Symposium on Principles of Programming Languages*, 1993.
- [96] V. Mekkat, A. Holey, P. Yew, and A. Zhai. Managing Shared Last-Level Cache in a Heterogeneous Multicore Processor. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.
- [97] M. Moreto, F. Cazorla, A. Ramirez, and M. Valero. MLP-Aware Dynamic Cache Partitioning. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 337–352. Springer, 2008.
- [98] P.J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *Department of Defense HPCMP Users Group Conference*, 1999.
- [99] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. Graph-PIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

- [100] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. Stream-Dataflow Acceleration. In *International Symposium on Computer Architecture (ISCA)*, pages 416–429, June 2017.
- [101] NVIDIA. Nvidia project denver. <http://goo.gl/Sbjbb>.
- [102] E.J. O’Neil, P.E. O’Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *International Conference on Management of Data*, 1993.
- [103] R. Vuduc J. Young P. Lavin, J. Riedy. Spatter: A Benchmark Suite for Evaluating Sparse Access Patterns. 2018.
- [104] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference (DAC)*, 2011.
- [105] A. Pedram, A. Gerstlauer, and R. van de Geijn. Codesign Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures. *IEEE TC*, 61(12):1724–1736, December 2012.
- [106] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. Association for Computing Machinery, 2003.
- [107] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M.A. Horowitz. Convolution Engine: Balancing Efficiency and Flexibil-

- ity in Specialized Computing. In *International Symposium on Computer Architecture (ISCA)*, 2013.
- [108] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *International Symposium on Microarchitecture (MICRO)*, 2006.
- [109] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way Cache: Demand-based Associativity via Global Replacement. In *International Symposium on Computer Architecture (ISCA)*, 2005.
- [110] M.K Qureshi, A. Jaleel, Y.N. Patt, S.C. Steely, and J. Emer. Adaptive Insertion Policies for High Performance Caching. In *International Symposium on Computer Architecture (ISCA)*, 2007.
- [111] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. In *International Symposium on Workload Characterization (IISWC)*, 2014.
- [112] J.T. Robinson, John T., and V.M. Devarakonda. Data Cache Management Using Frequency-based Replacement. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1990.
- [113] M. Rosenblum, E. Bugnion, S. Devine, and S.A. Herrod. Using The SimOS Machine Simulator to Study Complex Computer Sytems. *ACM TMCS*, 7(1):78–103, July 1997.

- [114] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE CAL*, 10(1):16–19, 2011.
- [115] A. Rubini and J. Corbet. *Linux device drivers*. " O'Reilly Media, Inc.", 2001.
- [116] D. Sanchez and C. Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousands-Core Systems. In *International Symposium on Computer Architecture (ISCA)*, 2013.
- [117] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache Aware Optimization of Stream Programs. *SIGPLAN Not.*, 40(7), 2005.
- [118] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Wei, and D. Brooks. Co-designing Accelerators and SoC Interfaces Using gem5-Aladdin. In *International Symposium on Microarchitecture (MICRO)*, 2016.
- [119] Y. S. Shao, S. L. Xi, V. Srinivasan, G. Wei, and D. Brooks. Co-designing Accelerators and SoC Interfaces Using gem5-Aladdin. In *International Symposium on Microarchitecture (MICRO)*, 2016.
- [120] Y.S. Shao, B. Reagen, G.Wei, and D. Brooks. Aladdin: A Pre-RTL, Power-performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *International Symposium on Computer Architecture (ISCA)*, 2014.

- [121] Z. Shi, X. Huang, A. Jain, and C. Lin. Applying Deep Learning to the Cache Replacement Problem. In *International Symposium on Microarchitecture (MICRO)*, 2019.
- [122] S.Kato, M. McThrow, Carlos. Maltzahn, and Scott. Brandt. Gdev: First-class GPU Resource Management in the Operating System. In *USENIX Annual Technical Conference*, 2012.
- [123] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1999.
- [124] A. Snaveley and D.M Tullsen. Symbiotic Job Scheduling for a Simultaneous Multithreaded Processor. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.
- [125] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu. The Application Slowdown Model: Quantifying and Controlling The Impact of Inter-Application Interference at Shared Caches and Main Memory. In *International Symposium on Microarchitecture (MICRO)*, 2015.
- [126] G. Suo, X. Yang, G. Liu, J. Wu, K. Zeng, B. Zhang, and Y. Lin. IPC-Based Cache Partitioning: An IPC-Oriented Dynamic Shared Cache Partitioning Mechanism. In *International Conference on Convergence and Hybrid Information Technology*, pages 399–406. IEEE, 2008.

- [127] S. Thoxiyoor, J. Brockman, and D. Rinzler. PIM Lite: A Multithreaded Processor-in-memory Prototype. In *Great Lakes Symposium on VLSI (GSVLSI)*, 2005.
- [128] F. van Zee and R. van de Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality . *ACM TOMS*, 41(3):14:1–14:33, June 2015.
- [129] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M.B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.
- [130] J. Vesely, A. Basu, M. Oskin, G. H. Loh, and A. Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.
- [131] W. A. Wong and J. . Baer. Modified LRU Policies for Improving Second-Level Cache Behavior. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2000.
- [132] C.J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, C. Simon Jr. Steely, and J. Emer. SHiP: Signature-based Hit Predictor for High Performance Caching. In *International Symposium on Microarchitecture (MICRO)*, pages 430–441, 2011.

- [133] Y. Xie and G.H Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. *SIGARCH Computer Architecture News*, 2009.
- [134] S. Xu, X. Chen, Y. Wang, Y. Han, and X. Li. CuckooPIM: An Efficient and Less-blocking Coherence Mechanism for Processing-in-Memory Systems. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2019.
- [135] S. Xu, Y. Wang, Y. Han, and X. Li. PIMCH: Cooperative Memory Prefetching in Processing-in-Memory Architecture. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018.
- [136] Y. Hao and Z. Fang and G. Reinman and J. Cong. Supporting Address Translation for Accelerator-Centric Architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.
- [137] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. CPU-Assisted GPGPU on Fused CPU-GPU Architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2012.
- [138] A. Yarkhan, J. Kurzak, P. Luszczek, and J. Dongarra. Porting the PLASMA Numerical Library to the OpenMP Standard. *Int. J. Parallel Program.*, 45(3), 2017.

- [139] T.T. Yeh, A. Sabne, P. Sakdhnagool, R. Eigenmann, and T.G. Rogers. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.
- [140] J. Zhai J. Cheng D. Xiang J. Li Y. Chai Z. Feng, H. Lin and X. Du. An Adaptive Breadth-first Search Algorithm on Integrated Architectures. *The Journal of Supercomputing*, 74(11):6135–6155, Nov 2018.
- [141] D. Zhang, N. Jayasena, A. Lyashevsky, J.L. Greathouse, L. Xu, and M. Ignatowski. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *International Symposium on High-Performance Parallel and Distributed computing (HPDC)*, 2014.
- [142] L. Zhao, L. N. Bhuyan, R. Iyer, S. Makineni, and D. Newell. Hardware Support for Accelerating Data Movement in Server Platform. *IEEE Transaction on Computers (TC)*, 56(6):740–753, 2007.
- [143] Y. Zhu and V.J. Reddi. Webcore: Architectural Support for Mobile Web Browsing. In *International Symposium on Computer Architecture (ISCA)*, 2014.

Vita

Mochamad was born in Jakarta, Indonesia, the son of Ridwan S.H and Siti Nuraini. Prior to joining UT, Mochamad received his bachelor degree in Computer Science from Tokyo Institute of Technology, Japan. As much as he enjoys research, Mochamad likes to spend his time playing guitar, soccer, singing, dancing and scooter riding.

Permanent address: 2501 Lake Austin Blvd
Austin, Texas 78703

This dissertation was typeset with $\text{\LaTeX}^\dagger$ by the author.

[†] $\text{\LaTeX}$ is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's $\text{\TeX}$ Program.