MEMORYISLANDS

MEMORYISLANDS: A FEDERATED APPROACH FOR EFFICIENT MEMORY MAPPINGS

By FATEMEH DERAKHSHANI, BS

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University

MASTER OF APPLIED SCIENCE (2024)

Hamilton, Ontario, Canada (Electrical and Computer Engineering)

TITLE:	MemoryIslands: A Federated Approach for Efficient Memory Mappings
AUTHOR:	Fatemeh Derakhshani BS (Computer Science and Engineering), McMaster University, Hamilton, Canada
SUPERVISOR:	Mohamed Hassan
NUMBER OF PAGES:	xiv, 54

Lay Abstract

The contemporary landscape of computing systems witnesses a surge in compute elements, often juggling multiple concurrent workloads. These workloads present a kaleidoscope of memory access patterns and diverse memory requirements. Despite this, main memory architectures persist in employing a uniform approach, treating all requests alike. This approach is exemplified in memory mapping, where a singular mapping strategy is applied across various parallelism levels. However, this blanket approach neglects performance nuances inherent in individual application memory access patterns. Departing from conventional methods that attempt dynamic mapping changes, this thesis proposes a novel perspective: treating main memory as a federation of independent resources, or "islands." It presents a methodology to optimize address mapping for each application, an optimization framework for defining these memory islands, and a software-aware codesign strategy to configure memory controllers accordingly. Extensive evaluation across a spectrum of workloads demonstrates substantial performance enhancements compared to other mapping approaches.

Abstract

Modern computing systems are exhibiting increasing computing elements with several co-running workloads. These workloads exhibit highly diverse memory access patterns and have different memory requirements. Nonetheless, main memory architectures are still oblivious to this diversity handling all requests with the same set of rules. Memory mapping is a clear example of this failing *one-size-fits-all* memory approach. Encompassing several parallelism levels (channels, ranks, groups, and banks), the memory performance of an application depends heavily on its particular memory access pattern and how it is mapped to these levels. In contrast, current memory controllers (MCs) deploy a fixed address mapping for all applications, which leaves significant performance opportunities if each application is serviced with the suited mapping.

Instead of following the prior approach of attempting to dynamically change the address mapping, which has significant limitations due to the need for data migrations, this thesis promotes the idea of considering main memory as an independent federated set of resources, which we call islands. Based on this idea, it introduces 1) a methodology to decide the address mapping that maximizes the performance of each application; 2) an optimization framework to statically define this federation of

islands for each set of co-running workloads; 3) and finally, a software-aware co-design methodology to configure the MC with the various memory islands and their corresponding address mappings.

Our extensive evaluation with a diverse set of more than 80 workloads and several single- and multi-core system setups show a significant performance improvement over the best compared static mapping when deploying the proposed technique.

Dedication

To my loving sister, supportive mother, and guiding father, whose unwavering encouragement has been my greatest source of strength and inspiration.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Mohamed Hasan, for his invaluable guidance, unwavering support, and continuous encouragement throughout this journey.

I am also thankful to the members of the Fanos Lab for their collaboration, insightful discussions, and camaraderie, which enriched my research experience and contributed to my growth as a researcher.

To my family and friends, whose love, understanding, and encouragement provided me with strength and motivation during both challenging and rewarding times, I am profoundly grateful.

I extend my appreciation to my previous teachers and professors for instilling in me a passion for learning, nurturing my intellectual curiosity, and imparting invaluable knowledge and skills that have shaped my academic journey.

Table of Contents

List of Figures

List of Tables

Notation, Definitions, and Abbreviations

Abbreviations

Ba Bank

Bg Bank Group

Ch Channel

Co Column

CPU Central Processing Unit

DDR Double Data Rate

DDR4 SDRAM
Double Data Rate 4 Synchronous Dynamic Random-Access Memory

DRAM Dynamic Random Access Memory

GDDR5 SDRAM Graphics Double Data Rate 5 Synchronous Dynamic Random-Access Memory

GPU Graphics Processing Unit

LPDDR4/5 Low Power Double Data Rate 4/5 Synchronous Dynamic Random-Access Memory

MC Memory Controller

MPSoC MultiProcessor System-on-Chip

PC Program Counter

OS Operating System

Ra Rank

Ro Row

SA Sense Amplifier

SRAM Static Random Access Memory

TLB Translation Lookaside Buffer

Chapter 1

Introduction

The rapid advancement of logic processing units, such as CPUs and GPUs, has significantly improved computational power in computing systems. However, the progress in memory subsystem bandwidth and latency has been comparatively limited. Consequently, the rate at which data can be transmitted within memory components may not keep up with the data consumption speed of processing units, giving rise to a phenomenon known as the "Memory wall," which hinders overall system performance and efficiency [18]. This challenge has spurred extensive research efforts aimed at enhancing memory performance within computer architecture design.

Efficient memory allocation stands as a pivotal factor in optimizing memory usage and, by extension, application performance in contemporary computing systems. Modern off-chip Dynamic Random Access Memories (DRAMs) are composed in the form of a hierarchy of segments. DRAM channels comprise one or more ranks, a rank has several bank groups, each of which has several banks. A DRAM bank comprises a matrix of 2D cells in the form of rows and columns. The address mapping module in memory controllers (MCs) is the one responsible for determining how a memory

address is mapped to these segments. Three key observations motivate this thesis.

Observation 1: The memory performance of an application depends heavily on its particular memory access pattern and how it is mapped into the aforementioned main memory segments. *Observation 2*: Modern MPSoCs execute multiple applications that exercise diverse memory access patterns. Additionally, in heterogeneous MPSoCs (like those with both GPUs and CPUs), it has been shown that typical GPU and CPU workloads exhibit different behavior that mandates different memory mappings [19]. *Observation 3*: State-of-the-art MCs deploy application-agnostic fixated techniques that are universally applied to all incoming requests. These techniques typically favor a very particular access pattern, for example, these exhibiting high locality. Nonetheless, even if they are tailored to favor another access pattern, they remain focused only on one pattern at the expense of hurting applications not exhibiting such a pattern. Taking address mapping as an example, current MCs deploy one address mapping for all applications. However, we observe that the application's performance significantly varies with different mappings. Figure 1.1 shows this observation for two of the SPEC2006 benchmarks with two different mappings. While GemsFDTD seems to have high locality and benefit from having the DRAM column bits (Co) at the least significant bits in map4, cactus exhibits better performance when the mapping is interleaving-oriented and maps channel (Ch), rank (Ra), and bank (Ba) bits to the lower address bits in the mapping in map2.

As the figure illustrates, there is a big performance variation based on the selected address mapping (*Observation 1*), and the two applications exhibit better performance using different mappings (*Observation 2*). Finally, in a multi-core SoC,

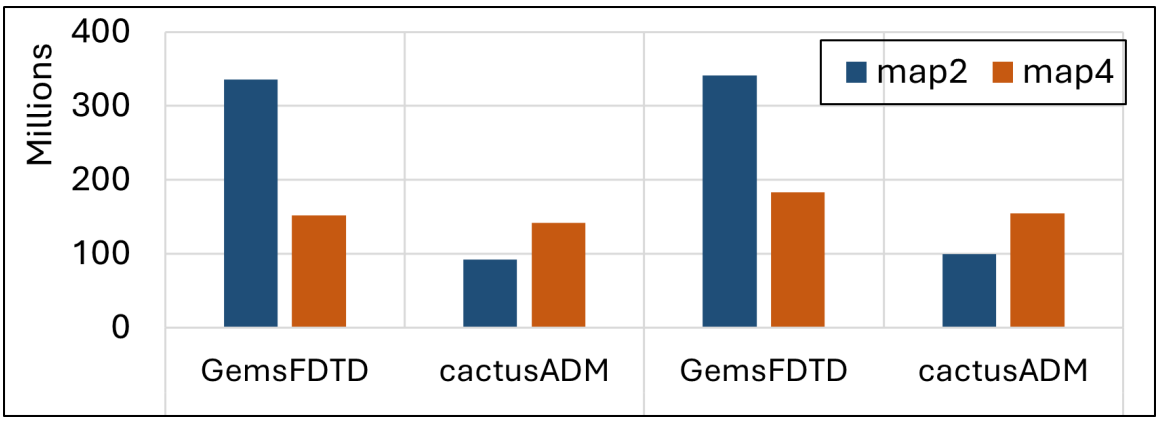


Figure 1.1: Comparing application's performance (execution cycles) with two different mappings; map2: RoCoBaRaCh, map4: RoBaRaChCo.

where the two applications execute together, the MC has to choose only one mapping. As Figure 1.1 also shows, deploying either mapping in this case, one of the two applications suffers significant performance degradation (*Observation 3*). Some efforts have been proposed to dynamically change the address mapping during run-time to adapt to this variation (e.g. [15, 18, 1]). However, dynamically changing address mapping mandates data migration in DRAM, which is a highly costly operation that diminishes the performance improvement in addition to custom circuitry to enable the migration that these works depend on. Recently, SDAM [39] proposed a software-defined approach, where it extends the *malloc* dynamic memory allocation, and the physical page OS allocator to enable programmers to define the address mapping for particular memory pages (chunks). 1) SDAM focuses on 3D memories, which entertain a large number of channels; hence, focuses mainly on mapping address bits with large bit flips to the channel bits. Compared to 3D memories, commodity DRAMs (e.g. (LP)DDR4/5) exhibit way less number of channels (usually four or fewer), which limits the applicability of SDAM to these memories. 2) SDAM requires several modifications to both software (dynamic memory allocator), and OS (virtual-to-physical

mapper), in addition to the hardware modifications. 3) its approach in determining best mapping depends on the bit-flip metric, which is an indirect metric that was found to fail to take into account DRAM timing constraints and does not generalize easily to a more complex DRAM organization hierarchy [18].

Instead of facing the complexity of dynamically changing the address mapping and its associated expensive data migration, this work looks into a modern off-chip memory as a federation of islands, where each island has its own address mapping. We believe that shifting the perception of off-chip memory in modern MPSoCs from a monolithic resource into a heterogeneous collection of resources has the potential to address many of the challenges of these systems. Toward that goal, this thesis makes the following contributions.

1) **A methodology to decide address mapping.** This methodology applies two distinct techniques. The first one is targeting applications with stream access patterns. A stream is a well-defined memory access pattern, which can be as simple as the form of `A[i]` or has level(s) of indirection such as `A[B[i]]` [36]. In addition to its widespread in multimedia[29] and data analytics [3], streams are also very common in general-purpose compute workloads [35]. This one leverages the well-defined pattern of the stream to guide the selection of the address mapping. The second one is a profile-based approach, where we propose to run the workload with several address mappings under investigation and observe the impact on performance to guide the choice of the mapping providing the best performance. Despite its simplicity, we find the profile-based approach to be more efficient than techniques aiming at indirect metrics such as analyzing the bit flip rates [40] or even using machine learning techniques [18]. This is because first, while these techniques suffer

an overlooked gap between these metrics and the actual performance of the workload with a particular address mapping, profiling provides such a direct answer. Second, from our extensive evaluations for different benchmarks, we find that generally, a handful of simple mappings prove to be effective in providing the highest performance boost for workloads without the need to try all possible address permutations. This makes profiling a reasonable task.

2) **A general approach to consider main memory as independent federated regions (we call islands).** Each island is composed of one or more memory banks and has its characteristics (e.g. size) and the MC handles it differently (e.g. deploys a different address mapping, page policy, or even scheduling techniques) without any required modifications to the DRAM devices themselves.

3) **An optimization framework to statically define this federation of islands for each set of co-running workloads** The challenge we address here is how to distribute the whole memory resources among the federated islands to optimize overall systems memory performance. As we discuss in detail in Chapter 4, we observe that applications show a drastic disparate behavior when they are assigned more memory resources. We formulate this as an optimization problem in Section 5.2.

4) **A software-aware co-design methodology to configure the MC with the various memory islands and their corresponding address mappings** This methodology does not require any changes to the software-hardware interface (i.e. Instruction Set Architecture), to the programming model, or to the OS (i.e. physical page allocators), which simplifies the adoption of the methodology. This methodology depends on two main components. 1) A software-managed scratchpad SRAM memory (SPM) at the MC that holds all the configurable registers to determine island and

mapping configurations; and 2) A reverse Translation Lookaside Buffer (rTLB) that resolves incoming requests' physical addresses to their original virtual addresses to be able to associate them to the configuration in the SPM (since it is encoded in a virtual domain). A detailed discussion about the architecture and the proposed methodology is in Chapter 5.

Our detailed evaluation in cycle-accurate simulations with $1, 2, 4$ and 8 core systems show that the proposed approach achieves a significant performance improvement (up to 50% in some cases) over the best compared static address mapping.

Chapter 2

Related Work

Address Mapping Techniques. Zhang et al. [40] introduced an address mapping scheme based on permutations. This scheme involves XOR operations between a bank bit and a row bit to enhance the flip ratio of bank address bits. Chatterjee et al.[6] extended the permutation-based address mapping by incorporating channel bits into XOR operations, particularly designed for irregular GPU applications. both of these approaches employ only trace-based application features, compared to our approach which employs application-based features. DReAM[15] monitors memory requests and dynamically adjusts the mapping based on changes in physical address bits. Adavally and Kavi[1] proposed a similar but more accurate approach for parallelism and conflict estimation compared to Dream. However, both of these approaches need data migration. none of the above papers support multiple address mappings simultaneously for different applications. Liu et al.[19] utilize an entropy-based indirect method. Their proposed method lacks support for multiple address mappings. Additionally, it focuses on indirect features, while our approach considers actual direct features of the application, especially for strided access patterns. The fact that they

also take power as a metric into account is impressive. Chen et al.'s study [8] presents an alternative dynamic entropy-based approach for general-purpose GPUs. Not only is this method primarily tailored to GPUs, but it also places a heavy emphasis on entropy-based features, which may not possess the necessary awareness to generate an appropriate address mapping. Congen[12] defines the challenge of optimizing row buffer hits as a graph-cut problem. In addition to being more complex than our approach, Congen lacks features such as partitioning for interference reduction and support for multiple address mappings. Flatfish[18] adopts an adaptive reinforcement learning method, eschewing indirect methods like bit entropy. However, in comparison to our approach, Flatfish lacks partitioning as a means to reduce interference, and it employs a single, albeit adaptive, address mapping. Moreover, its reinforcement learning method is more complex, leading to migration overhead in online mode.

Recently, SDAM [39] proposed a software-defined approach, where it extends the *malloc* dynamic memory allocation, and the physical page OS allocator to enable programmers to define the address mapping for particular memory pages (chunks). SDAM can be considered one of the closest works to this thesis; however, there are several key differences. 1) SDAM focuses on 3D memories, which entertain a large number of channels; hence, focuses mainly on mapping address bits with large bit flips to the channel bits. Compared to 3D memories, commodity DRAMs (e.g. (LP)DDR4/5) exhibit way less number of channels (usually four or fewer), which limits the applicability of SDAM to these memories. Our proposed federated islands approach considers all the complexity of a modern main memory including banks, groups, ranks, and channels, which makes it applicable to both traditional DDR memories as well as 3D ones. 2) SDAM requires several modifications to both software

(dynamic memory allocator), and OS (virtual-to-physical mapper), in addition to the hardware modifications. Other than the framework required to configure the SPM, we require no modifications at all to legacy software or OS. 3) SDM approach in determining best mapping depends on the bit-flip metric, which is an indirect metric that was found to fail to take into account DRAM timing constraints and does not generalize easily to a more complex DRAM organization hierarchy [18]. We instead adopt either a profiling-based approach for non-structured access patterns or the stream-aware approach for streaming-based patterns.

Memory Partitioning. Main memory resource partitioning has been adopted at different levels: at bank [38, 11, 28, 34], at rank [2, 41], and at channel levels [21]. It has been deployed to address a diverse set of goals including 1) minimizing inter-core interference [21, 38], providing strong real-time latency guarantees [28], and addressing side-channel security vulnerabilities [34]. Instead of obliviously mapping banks to cores or applications, we formulate an optimization problem to grant every application on the island with the number of bank resources such that overall system performance is improved.

Memory and Bank-Level Parallelism Several prior works aimed at estimating the amount of memory (e.g. [26, 33]) or bank-level parallelism (e.g. [16, 31]) an application leverages. The former usually is measured in terms of the average number of outstanding requests of the application (e.g. through Memory Status Registers (MSHRs) sizes), while the latter uses the number of requests concurrently serviced from the application as an indication of bank parallelism. Recently, [10] proposed another metric, bank parallelism utilization as a representation of the average number of

banks being concurrently used. These metrics were effectively used in runtime tracking and dynamically adapting solutions. Nonetheless, similar to the bit-flip ratio, we find all these metrics, despite their effectiveness, as an indirect way that does not fully capture the impact of how many banks are offered to an application on its performance. Since our address mapping determination is an offline process, we lean to profiling as a methodology to demystify this relationship as explained in Section 5.2.

Chapter 3

Background

3.1 DRAM Architecture

The diagram in Figure 3.1 illustrates a typical DRAM architecture, wherein DRAM cells are organized into a 2-D memory array known as a DRAM bank. Each bank incorporates a sense amplifier (SA) array, which acts as a local buffer (also called row buffer) for a DRAM row during data access. Multiple DRAM banks form a DRAM rank, and multiple ranks constitute a channel. Ranks within the same channel share a physical link to the processor. In DDR4/GDDR5 SDRAM, an additional level called "bank group" is introduced between the bank and rank[18][9][4].

3.1.1 DRAM Address Mapping

Consequently, to locate a specific DRAM cell, one must provide the corresponding Channel ID, Rank ID, Bank Group ID, Bank ID, Row ID, and Column ID. Designing

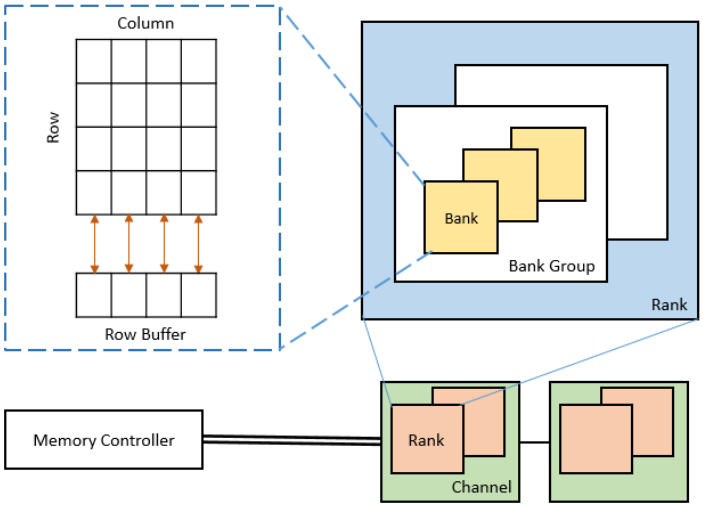


Figure 3.1: A general illustration of DRAM architecture.

an optimal scheme to map the physical address to these IDs poses a significant challenge. One of the challenges is the row buffer hit ratio. Typically, when accessing a DRAM cell, the DRAM first charges the associated DRAM row containing the target cell into the row buffer. Subsequent column accesses are then performed based on the Column ID. As a result, if subsequent requests attempt to access DRAM cells within the same row, they can directly access the desired cells within the row buffer. This scenario, commonly known as a "row buffer hit," reduces latency and power consumption by circumventing the need for DRAM precharge and charge operations. Therefore, maximizing the row buffer hit ratio can substantially enhance DRAM performance. Another challenge is the utilization of bank parallelism to enhance memory access throughput, as each bank can independently handle a distinct request. Previous studies have aimed to distribute memory requests across multiple banks to

maximize parallelism [19]. However, this simplistic distribution method proves inadequate due to the intricate timing constraints imposed by the DDR protocol, an aspect that previous works have overlooked despite its significance[18].

3.2 Streams and Strides

we can define a "stream" generally as a continuous sequence of memory accesses with a constant direction and a "stride" as the constant address delta between consecutive memory accesses[17]. Observations show that there can be a pattern between two consecutive load addresses for the same program counter (PC). Applications can have different memory access patterns, including straightforward reuse and strides, as well as more intricate calculations involving memory indirection like linked lists[5]. In our study, we categorized the applications into two distinct groups based on their memory access patterns. Strided applications are characterized by having main streams with a fixed stride size, where data accesses follow a regular pattern. In these applications, memory accesses occur at uniform intervals known as strides, facilitating predictable memory access patterns. Examples of strided applications include matrix multiplication algorithms, image processing routines, and data streaming applications. Conversely, non-strided applications exhibit irregular or unpredictable memory access patterns. In these applications, either the main streams are not easily identifiable, or their main streams do not adhere to a fixed stride size. As a result, memory accesses in non-strided applications lack regularity, posing challenges for memory management and optimization techniques. Examples of non-strided applications include certain database workloads, web servers, and multimedia applications. Understanding the distinction between strided and non-strided applications is essential for developing

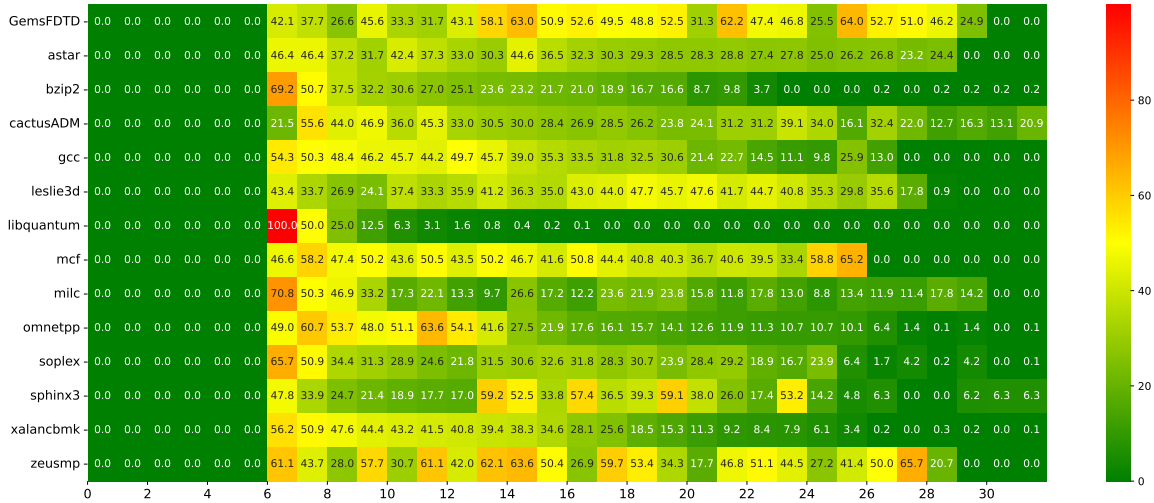
efficient memory management techniques tailored to the specific characteristics of each application category.

Chapter 4

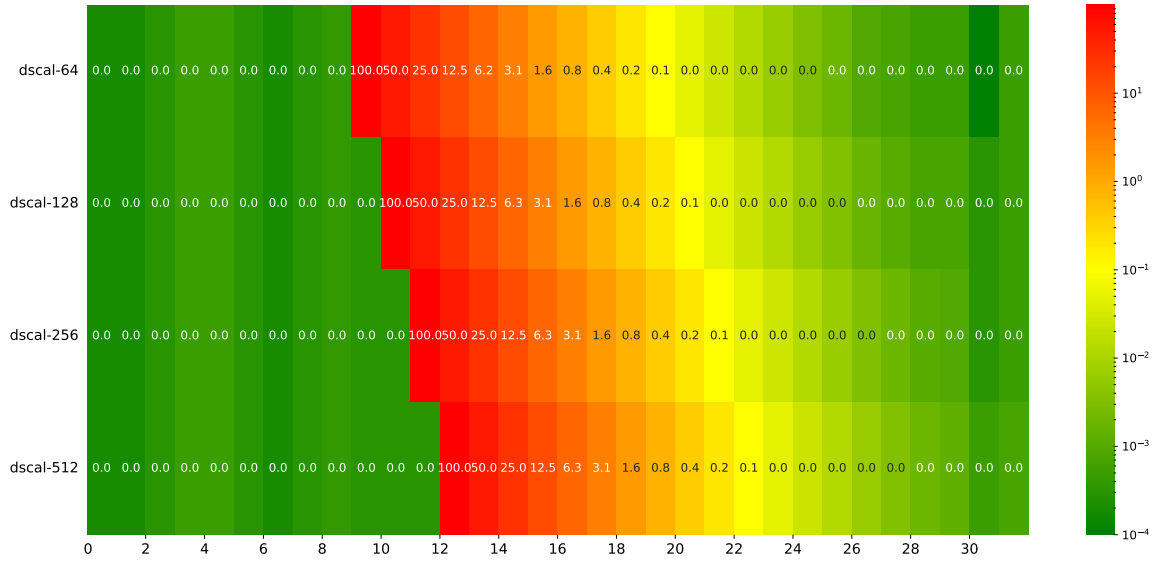
Motivation

4.1 Diverse Behavior of Applications

The selection of an appropriate mapping scheme holds paramount significance as it profoundly impacts both the system's execution time and power consumption. To validate the criticality of the mapping scheme, we conducted multiple comprehensive investigations. Figure 4.1 presents a heatmap illustrating the frequency of bitflips, calculated as the number of bitflips divided by the total number of requests, across various benchmarks.



(a) Non-strided benchmarks



(b) Strided benchmarks

Figure 4.1: Heatmap of bit-flip patterns for strided and non-strided benchmarks (The y-axis represents the benchmark)

While we emphasize that we do not rely solely on bit-flip occurrences as the sole determinant of mapping decisions, they still provide valuable insight. The heatmap reveals distinct bit-flip patterns among the benchmarks, suggesting that a single,

fixed mapping strategy cannot be adequate for all of the benchmarks. For instance, the bit-flip patterns of different benchmarks vary significantly, indicating the need for adaptable mapping strategies tailored to individual application characteristics.
Notably, while examining the heatmap of non-strided benchmarks (SPEC benchmarks [30]), depicted in Figure 4.1(a), we do not discern any specific pattern or correlation among different benchmarks. However, the strided heatmap, illustrated as in Figure 4.1(b), reveals a clear pattern between various stride sizes within a single application. This visualization adeptly showcases the relative bit-flip patterns across different stride sizes of a benchmark. Understanding these discernible patterns is imperative as we endeavor to harness this insight in crafting optimal mappings for stream-based applications.

4.2 Impact of Address Mapping

In another experiment, we utilized four distinct pre-defined mapping schemes. Our experimentation encompassed a diverse set of benchmarks as shown in Figure 4.2. Figure 4.2 presents the execution time for various applications under four different memory mappings: map1 (Ro-Ch-Ra-Ba-Co), map2 (Ro-Co-Ba-Ra-Ch), map3 (Ro-Ch-Ba-Ra-Co), and map4 (Ro-Ba-Ra-Ch-Co).

The empirical results demonstrate substantial variations in the execution time between different mapping schemes, with relative differences reaching as high as 45.57% for particularly mapping-sensitive benchmarks such as Sphinx3 from the Spec benchmark suite (on average it would be 13.53% across all of the tested benchmarks). However, it is essential to note that, in general, benchmark applications can be categorized into two distinct groups based on their sensitivity to mapping, namely

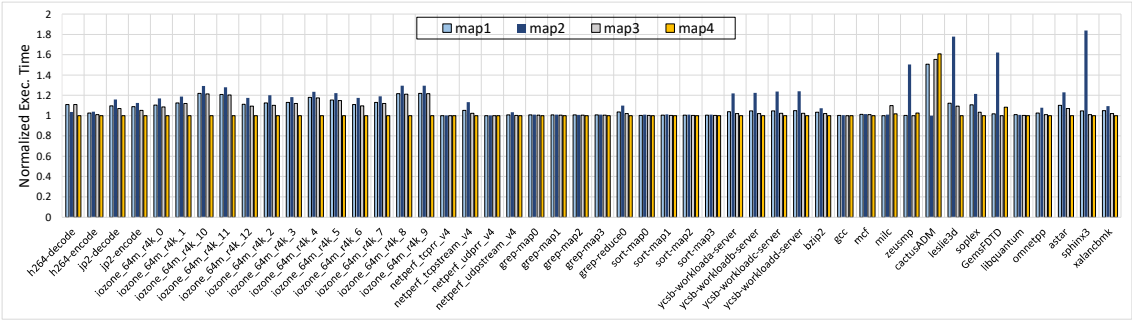


Figure 4.2: Comparison of the execution cycles for single-core systems over non-stream-based application

mapping-sensitive and mapping-insensitive benchmarks. For instance, benchmarks like Sphinx3, Leslie3d, and GemsFDTD exhibit execution time relative differences of 45.57%, 43.77%, and 38.36%, respectively. On the other hand, benchmarks such as netperf_tcprr, GCC, and Libquantum display considerably lesser mapping sensitivity, at least in our conducted tests. Furthermore, our experimental findings indicate that different applications can experience significant performance disparities under various mappings as previously highlighted for CactusADM and GemsFDTD in Figure 1.1. These experiments underscore a critical observation: an application's performance can vary significantly depending on the mapping strategy employed and a single, fixed mapping cannot serve the needs of all applications.

4.3 Partitioning Memory into Independent Federated Islands

Concurrently running applications on a multicore chip compete for access to main memory, which has a restricted bandwidth. Inadequate management of this limited memory bandwidth can lead to harmful interference among different applications,

causing significant degradation in both system and individual application performance. Various studies have been done to enhance system performance by addressing memory interference issues among applications[13, 14, 20, 22–24, 27, 37] but these proposals primarily treat the problem as a memory access scheduling challenge, focusing on the development of novel memory request scheduling policies. These policies prioritize the requests of different applications, aiming to reduce interference among them. However, implementing such application-aware scheduling algorithms necessitates non-negligible changes to the existing system. Additionally, we have another study that uses Memory channel partitioning along with Scheduling[21]. The idea is to categorize applications based on their memory intensity, to low and high memory intensity groups. Further, subdivide high memory-intensity applications into low and high row-buffer hit rate groups; then partition the available memory channels among the three application groups. Within each group, partition the assigned channels among the applications and designate a preferred channel for each application. Additionally, they combine this partitioning method with scheduling which means prioritizing low-memory-intensive applications over the others. This particular study and similar research endeavors motivate the notion of enhancing application performance and minimizing interference among different applications through the implementation of partitioning. Our innovative approach involves partitioning at a more refined level, specifically at the bank level, recognizing the potential limitation of available channels for all concurrently running applications on our multi-core (or heterogeneous) system. In our work, we integrate the partitioning concept with a static method for application-aware address mapping to further optimize performance.

4.4 Importance of Application-Aware Memory Allocation.

Another important aspect of this thesis is configuring the number of bank resources for each island based on the application behavior and requirements. Normally, all applications can use the entire memory space, but to make sure they perform well and to allow for different mapping strategies for each, we suggest dividing memory wisely. This memory division is crucial because it greatly affects how fast the applications run. A simple way to divide memory, such as fair splitting does not provide application awareness. So, one application might not benefit from its allocated resources (underutilization), while another can be greatly penalized because allocated resources are not enough. To show the importance of application-aware memory resource allocation, we show in Figure 4.3 an example of four benchmarks and how their performance is impacted when allocated a variable number of memory banks. Results

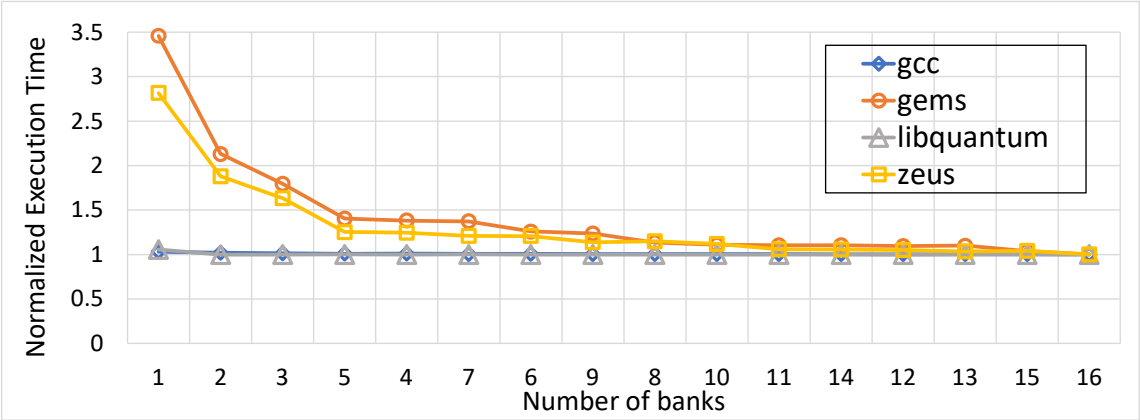


Figure 4.3: Normalized execution cycles by bank number for four benchmarks of Spec.

are execution times normalized to the case of 16 banks. As the figure illustrates, while

some applications' performance (e.g. Gcc and Libquantum) is indifferent to the allocated resources, others show significant differences with different numbers of banks: $3.5x$ for gems and $2.8x$ for Zeusmp. As a result, when these benchmarks co-execute in a system, allocating more banks to the latter two is expected to improve overall system performance without a considerable impact on the former.

Chapter 5

Methodology

5.1 Overview and Illustrative Example

The presented methodology revolves around an innovative approach to memory allocation, focusing on a static method that customizes memory mapping for individual applications and assigns a tailored partition to each application. this approach starts with the application analysis. This stage involves a detailed examination to identify key features influencing optimal memory mapping, using the stride size of the strided applications. Static profiling is employed when stride size is not evident, ensuring personalized mapping for each application's needs. By *profiling* we refer to the analysis of application characteristics during compile time as well as running it with various mappings to study its memory performance.

Following memory mapping, the approach delves into fine-grained partitioning, breaking down memory into small units (islands) and assigning them to applications

based on their requirements. This strategy prevents interference between applications and optimizes parallelization during partition assignments. Through an optimization process, the methodology determines the optimal memory division for each application, aiming to minimize execution time while efficiently utilizing resources. Overall, the methodology provides a robust and efficient solution to memory allocation optimization in heterogeneous computing systems, emphasizing static analysis for enhanced performance. Figure 5.1 illustrates a higher-level design of our proposed solution.

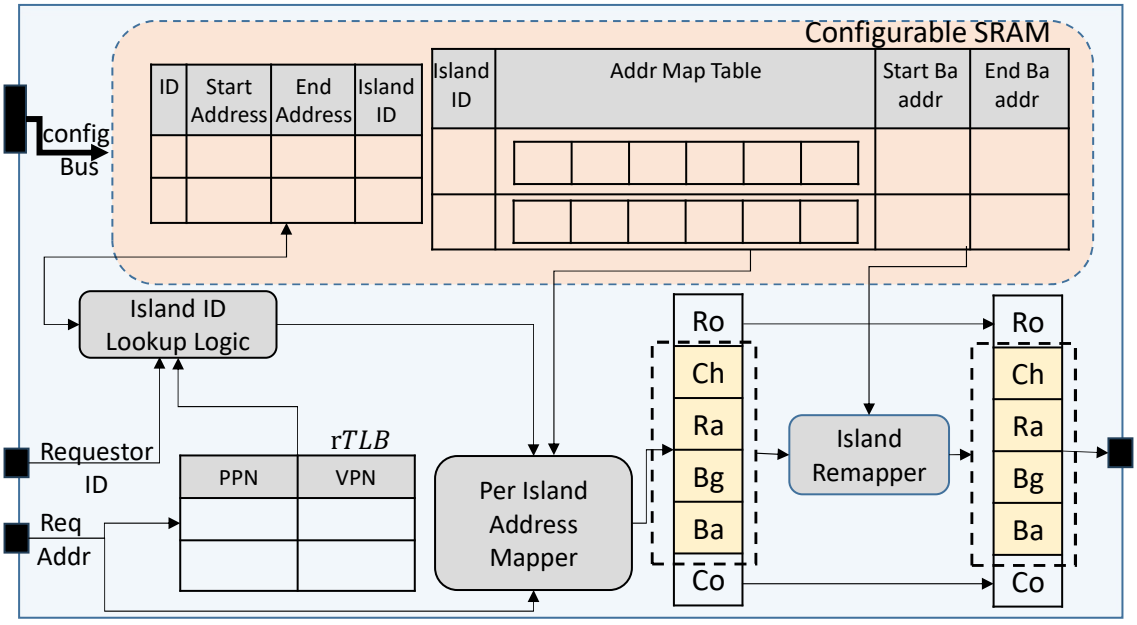


Figure 5.1: A high-level design of our island-based approach

Our approach begins by conducting a thorough analysis of the application to pinpoint features that significantly influence the selection of optimal memory mapping. These identified features play a crucial role in tailoring the memory allocation strategy for improved performance. One key feature that emerged from various experiments is the stride size of the main (memory-intensive) stream or streams. This critical

feature, representing the distance between consecutive elements accessed in the main data stream, directly influences the efficiency of memory access patterns. Building upon this insight, our approach generates a set of mapping schemes corresponding to each stream within an application. This static mapping approach allows for personalized mapping for individual streams, enhancing adaptability to diverse characteristics within a single application.

The main idea is to assign the channel bits to the address bits starting from the bit corresponding to the stride of the stream as it toggles the most as noted in Figure 4.1(b). The subsequent higher-order bits after that are assigned to ranks, then groups, then banks. For example, for a system with 2 channels, 2 ranks per channel each of which has 4 bank groups and each group encompassing 4 banks, in Figure 4.1(b) for dscal-256, bit 11 will be assigned to the channel, bit 12 is for rank, etc.

This ensures these toggles cause maximum parallelism. Another potential mapping is to assign the hot bits to columns to maximize locality. One key takeaway from understanding this pattern of the stream is to assign the lower address bits that do not toggle at all to the lower row bits (which is usually avoided by commodity address mappings). This will increase row hits, and hence improve performance. In the case of the previous dscal-256 example, these will be bits 6 to 10.

For applications where the stride size is not readily apparent or the program is not necessarily strided, static profiling is applied to the entire or a part of the program. This involves experimenting with a pool of potential base mappings to identify the best option. Incorporating partitioning (each partition would be called an island) guarantees that different applications or streams do not interfere with each other.

Figure 5.1 illustrates the architecture of our innovative proposed solution, a pivotal component integrated within the memory controller. Preceding the processing of any request, our system relies on a critical resource known as the TLB inverse (rTLB) table. This table serves as a gateway to retrieve the virtual address. The virtual address is indispensable for accessing the island ID, as all the data stored in the configurable SRAM table during compile time is intricately linked to the virtual address.

Each of the main streams and applications possesses a specific identifier, which is stored within a configurable SRAM table (denoted as the left table of Configurable SRAM in Figure 5.1). For non-strided applications, this ID corresponds to the thread ID, whereas for strided applications, it comprises a combination of the thread ID and the stream's name. Additionally, in the case of strided applications, the start and end addresses of the streams are retained to facilitate stream differentiation. Our objective is to allocate each of these applications and streams to distinct memory partitions, referred to as "islands," each assigned a unique island ID. The mappings obtained from the application analysis, along with its Island ID, are stored in another table, which is denoted as the right table of Configurable SRAM in Figure 5.1.

Upon receiving a request, the Island ID Lookup Logic utilizes the rTLB and the left table of Configurable SRAM (in Figure 5.1) to extract the corresponding island ID. This step uniquely requires the virtual address. Following this crucial stage and the subsequent extraction of the island ID, our operations will exclusively rely on the physical address.

With the island ID determined, we employ the right table of Configurable SRAM (in Figure 5.1) to identify the appropriate mapping for the given address and retrieve its

corresponding mapping in "Per Island Address Mapper".

Let's consider a scenario where we receive a request from the stream "A" with a physical address of 0x24C6A40E43F8 and the corresponding virtual address retrieved from the rTLB as 0x7FFD819DE008. Let's say in Figure 5.1, stream "A" is stored with the island ID "5" and starts from address 0x7FFD819DD010 to 0x7FFD819DE048. Using the "Island ID Lookup Logic" and then "Per Island Address Mapper" and its corresponding table in Configurable SRAM, we find the suitable mapping, which, for this example, let's say is "ROW-COL-BNK-RNK-CH".

Considering we have 15 bits for rows, 6 bits for columns, 2 bits for banks, 2 bits for bank groups, 1 bit for rank, 1 bit for channel, and 6 bits for the cache line size, we can determine the corresponding numbers for this request physical address. In this case, the channel would be 1, the rank would be 1, the bank group would be 3, the bank would be 0, the column would be 36, and the row would be 10499.

Noteworthy is our reliance on static mapping, eliminating the need for dynamic adjustments or data migration. This static nature enhances predictability and reliability in memory mapping optimization. The strength of our approach lies in its application-aware, static nature. Unlike methods based on trace features like bit flips, our approach offers superior precision and promise in optimizing memory mappings for enhanced performance. By integrating detailed application analysis with a static and adaptive memory mapping design, our methodology aims to provide a robust and efficient solution to the memory allocation optimization challenge.

Continuing from our memory mapping design, the next step in our approach involves fine-grained partitioning (bank-level partitioning). This partitioning is crucial

for dividing the entire memory into small, distinct partitions (islands), with one bank serving as the smallest possible unit in our case. Each application or stream then claims one or more partitions based on its specific requirements, thereby preventing other applications or streams from using these partitions. Our fine-grained partitioning strategy isn't just about isolation and reducing interference; it's also finely tuned to enhance parallelization across different levels, from channel to bank, when assigning banks to applications.

The number of banks required for each application is determined through an optimization process, discussed in detail in the next chapter. This optimization ensures efficient resource utilization while enhancing application performance. Our unique model of parallelization, especially during the bank assignment process, results in a high degree of parallelization. This approach assigns corresponding banks across different channels, then ranks, and then bank groups, facilitating efficient parallel processing. Interestingly, in many benchmark tests, our findings indicate that using our methodology the base mapping favoring locality often outperforms the mapping favoring parallelization, thanks to our method's inherent ability to increase parallelization levels.

As depicted in Figure 5.1, to assign each memory request to its correct location in the newly partitioned memory, the first step involves determining the bank (smallest partition) number. This is achieved by removing the row and column from the request address and rearranging the remaining memory levels in the order Ba-Bg-Ra-Ch (from Most Significant Bit to Least Significant Bit). This specific order ensures that banks are allocated across different channels, ranks, bank groups, and, if necessary, within the same bank group.

The generated bank number, however, does not account for partitioning and may correspond to any location in the entire memory. However, from previous steps, we know that this request belongs to a specific island with a unique island ID. Now, we use the right table of the Configurable SRAM which contains the start and end bank numbers, defining a region that needs to be mapped to the generated bank number. The"Island Remapper" computes the modulo of dividing the generated bank number by the corresponding island size and adds the result to the start bank number. The resulting new bank number is then rearranged to its original order based on its original mapping. Finally, in this step, we reattach the row and column, resulting in the final location of the request.

Continuing our example, using the "Island Remapper", we would compute the bank number, which in our example would be $0b001111$ or 15. Assume that the start and end bank numbers for island 5, stored in the SRAM table, are 7 to 18. Using the mathematical logic which happens in the "Island Remapper", we would compute the new bank number as $7+(15\%(18-7+1))$ or 10. Then, we would extract the original channel, rank, bank group, and bank from this new bank number; row and column remain unchanged. According to this, the channel would change to 0, rank would stay 1, bank group would change to 2, and bank would stay 0. Row would remain 10499 and column would remain 36. This is the new location that our approach maps the request to, which is more application-aware and leads to more optimized memory mapping.

Our optimization results demonstrate that this approach allows for the use of even fewer resources while achieving better execution outcomes. The success lies in the observation that increased resources can sometimes lead to decreased parallelization

and unnecessary data distribution. By carefully considering the assignment of banks and adopting a fine-tuned partitioning strategy, our methodology achieves a balance between resource usage and parallelization, contributing to enhanced application performance.

5.2 Optimization

To determine the optimal memory allocation for various streams and applications, we undertake a thorough evaluation process. Initially, each stream or application undergoes execution with varying numbers of banks, ranging from the minimum to the maximum available. The resulting data is then fitted into a function $f_i(x)$, where x_i represents the number of banks that this application island comprises from and f_i denotes the fitted function achieved from the corresponding execution cycles. Figure 5.2 illustrates the outcome of this fitting process for selected Spec and MemBen benchmarks. This fitting function represents the execution time of an application when assigned an island with a particular number of bank resources. Subsequently, we formulate an optimization problem that encompasses all the fitting functions for the available streams and applications. The objective is to minimize the total execution time(cycles) for each application. Thus, our goal is to determine the optimal number of banks $(x_1, x_2, ..., x_n)$ comprising the island for each application, minimizing the average execution time of all running applications. We use the Matlab genetic algorithm as our optimization function which is a function that finds the local minimum and can search non-continuous spaces. We need to consider the system constraints (e.g. total number of available banks) as well as constraints imposed by each application. The memory portion of each application must exceed its minimum required physical

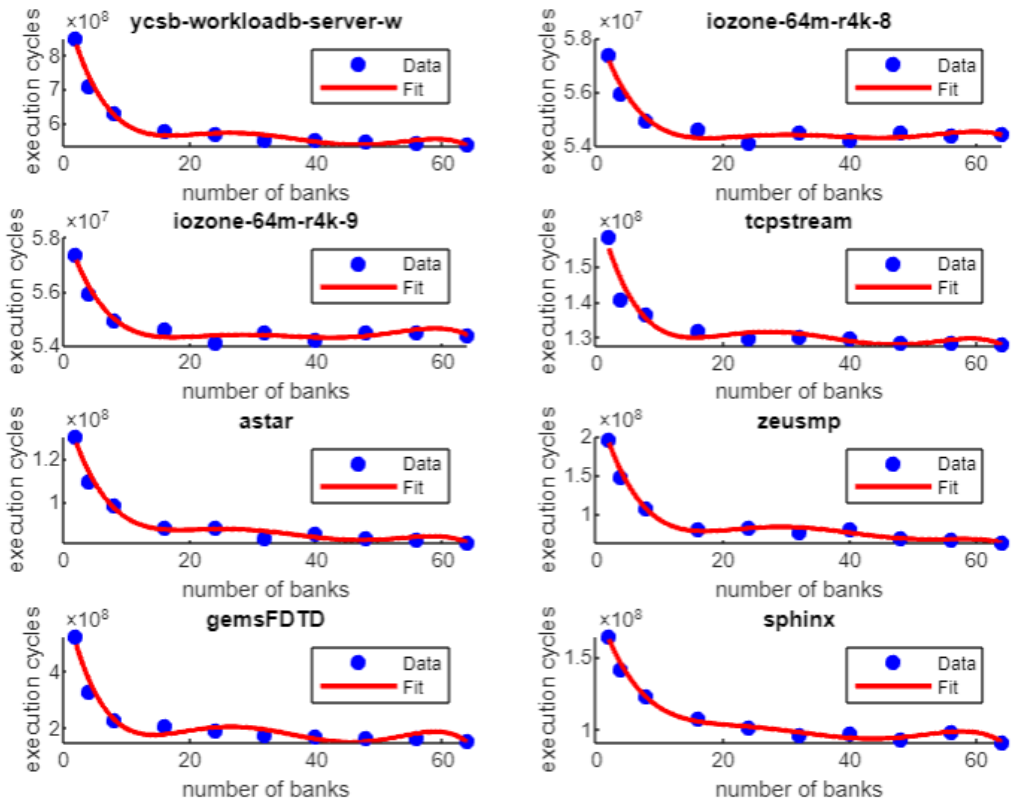


Figure 5.2: Fitting process chart for selected Spec and Memben benchmarks

memory footprint (e.g. to avoid excessive page swaps), calculated as the number of banks assigned to the application multiplied by the size of each bank. This ensures that the assigned memory portion is sufficient to meet the application's requirements.

Target Function: We formulate the optimization problem based on the fitting functions of available streams/applications:

$$\min \left(F(x) = \sum_{i=1}^{n} f_i(x_i) \right) \tag{5.2.1}$$

as Equation 5.2.1 shows, The objective is to minimize the total execution time/cycles for each application which leads to minimizing the average execution time of these applications since n is simply a constant input to the optimizer.

Input Parameters: Each stream/application (or a region of interest within that stream/application) is executed with varying numbers of banks, ranging from the minimum to the maximum. The resulting data is then fitted into the function $f_i(x)$, where x_i represents the number of banks and f_i which is our input parameter denotes the execution cycles.

Variable Parameters: In this optimization problem, the variable parameters represent the number of banks allocated to each application. The variable parameters are denoted by x_i for each application i in the range of $[1, ..., n]$, and n is the total number of applications and/or streams that will run in the system.

Constraints: To address this optimization problem, we must consider the constraints imposed.

1. **Total number of available memory banks:** We incorporate the total number of banks as a constraint, ensuring that the sum of all allocated banks remains fixed:

$$\sum_{i=1}^{n} x_i = \text{Total number of banks} \tag{5.2.2}$$

2. **Memory Allocation Constraints:** The memory allocation for each application must exceed the application's minimum required physical memory footprint:

$$x_i \times \text{size of each bank} \geq \text{footprint}_i \tag{5.2.3}$$

This ensures that the assigned memory portion is sufficient for the application's

requirements.

These parameters determine the allocation of banks to each application, directly influencing their performance within the system.

Chapter 6

Evaluation

In this chapter, we present the evaluation of our memory mapping methodology. To assess our solution, we employed Ramulator, a standalone memory simulator. Table 6.1 outlines the base configuration utilized for our evaluations. However, it's important to note that we tested our solution across various configurations to ensure comprehensive testing. Additionally, we utilized an Intel Pin-based tool integrated into Ramulator as a trace generator to produce traces for different benchmarks under evaluation. Our experiments encompass both single-core and multi-core scenarios across five distinct benchmark suites, namely Spec, MemBen, Polybench [25], Rodinia [7], and Linpack [32].

The evaluation chapter is divided into two main parts: one focusing on non-stream-based applications and the other on stream-based applications.

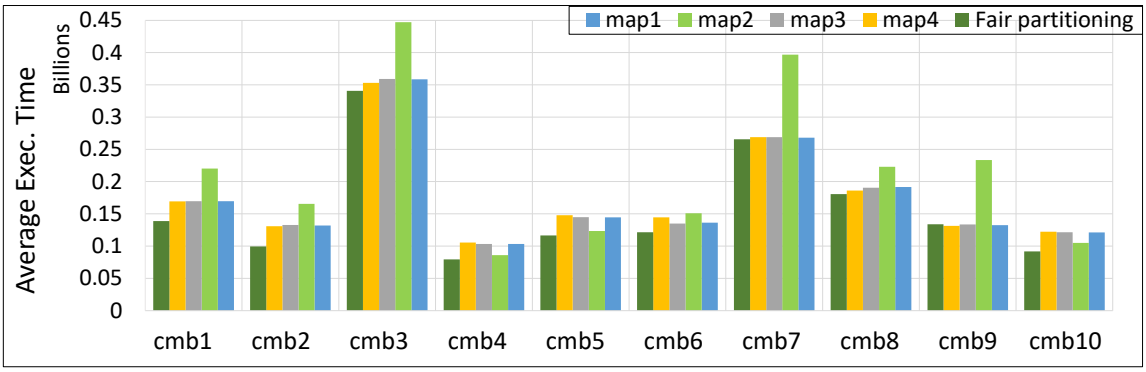
Table 6.1: Memory system overview

Description	**Value**
Memory Type	DDR4
Capacity	4Gb
Speed	2400MHz
No. of Channels	2
Ranks per Channel	2
Bank Groups per Rank	4
Banks per Bank Group	4
Rows per Bank	32768
Cache Line Size	64B

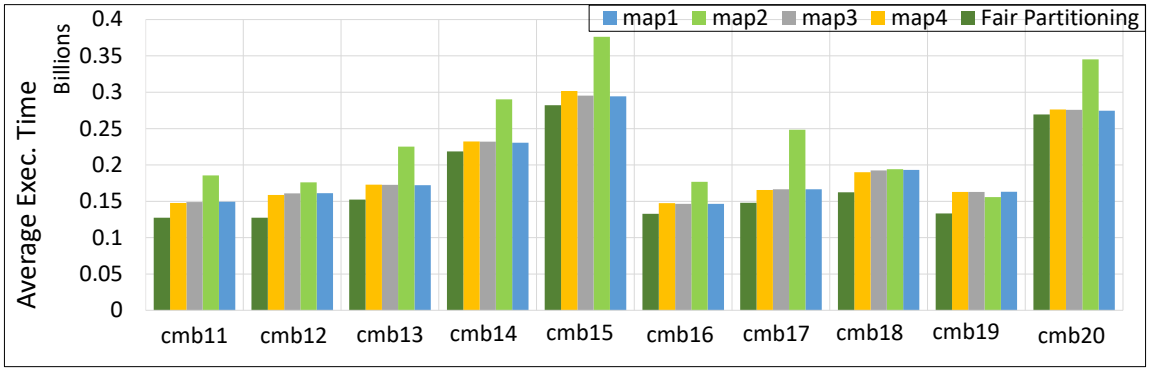
6.1 Profile-based approach

For non-stream-based applications, we conducted experiments to assess the impact of static profiling and different memory mappings on execution time. The evaluation consists of two parts: one for single-core systems (which is presented in Chapter 4) and another for multi-core systems that we present next.

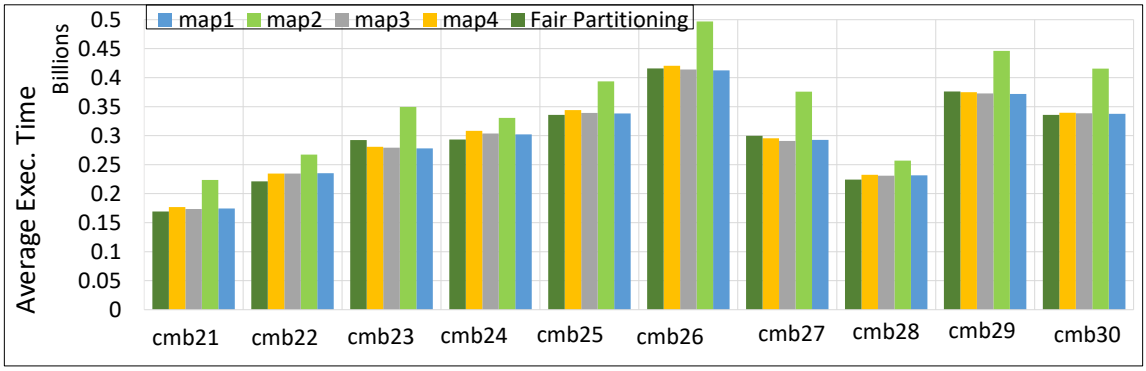
Following our assessment of suitable mapping achieved through profiling, we investigated the combined impact of mapping and partitioning on multi-core systems, encompassing 2-core, 4-core, and 8-core configurations, across Spec and MemBen benchmarks. These evaluations, illustrated in Figure 6.1, comprised 10 experiments analyzing execution times under different base mappings (mapping1, mapping2, mapping3, mapping4) without partitioning. The system we are testing on would be similar to Table 6.1.



(a) 2-Core



(b) 4-Core



(c) 8-Core

Figure 6.1: Comparison of the effect of suitable mapping on execution cycles for multi-core systems over non-stream-based application

Subsequently, we compared these results with those obtained after a fair, non-optimized division between applications using their best mapping obtained from static

profiling. For each experiment, we selected a set of benchmarks that, in their single-core tests, demonstrated the best execution results with different base mappings.

The results reveal compelling insights: when comparing the relative difference of our method against the best of base mappings for each experiment, we observed an average improvement of 8.3% for 2-core, 10.5% for 4-core, and 0.51% for 8-core configurations. Moreover, these improvements can be as high as 23.82% for 2-core, 19.61% for 4-core, and 5.81% for 8-core configurations, underlining the effectiveness of our methodology in enhancing performance across multi-core systems.

Notably, Table 6.2 and Table 6.3 offer detailed insights into the benchmarks and their utilization within the application combinations showcased in Figures 6.1, 6.2, and 6.4. While Table 6.2 provides benchmark IDs alongside their respective Application Suites, Table 6.3 presents the benchmark IDs used in each application combination. Together, these tables provide a comprehensive reference for understanding the benchmark combinations across Figures 6.1, 6.2, and 6.4.

Table 6.2: Benchmark names and the corresponding IDs

ID	Benchmark	Suite	ID	Benchmark	Suite	ID	Benchmark	Suite	ID	Benchmark	Suite
p1	2mm-256	PolyBench	p17	lu-256	PolyBench	h1	grep-map0	Hadoop	y1	ycsb-workloada-server	YCSB + Redis
p2	adi-256	PolyBench	p18	mvt-2048	PolyBench	h2	grep-map1	Hadoop	y2	ycsb-workloadb-server	YCSB + Redis
p3	atax-2048	PolyBench	p19	nussinov-512	PolyBench	h3	grep-map2	Hadoop	y3	ycsb-workloadc-server	YCSB + Redis
p4	bicg-2048	PolyBench	p20	symm-128	PolyBench	h4	grep-map3	Hadoop	y4	ycsb-workloadd-server	YCSB + Redis
p5	cholesky-512	PolyBench	p21	syrk-256	PolyBench	h5	grep-reduce0	Hadoop	s1	astar	SPEC2006
p6	correlation-256	PolyBench	l1	daxpy-128	Linpack	h6	sort-map0	Hadoop	s2	cactusADM	SPEC2006
p7	covariance-256	PolyBench	l2	ddot-128	Linpack	h7	sort-map2	Hadoop	s3	gcc	SPEC2006
p8	deriche-512	PolyBench	l3	dscal-128	Linpack	i1	iozone_64m_r4k_0	IOzone	s4	GemsFDTD	SPEC2006
p9	doitgen-64	PolyBench	l4	scusumkbn-128	Linpack	i2	iozone_64m_r4k_6	IOzone	s5	leslie3d	SPEC2006
p10	fdtd-256	PolyBench	r1	lud-512	Rodinia	i3	iozone_64m_r4k_8	IOzone	s6	libquantum	SPEC2006
p11	floyd-256	PolyBench	r2	needle-1024	Rodinia	i4	iozone_64m_r4k_9	IOzone	s7	mcf	SPEC2006
p12	gemver-512	PolyBench	r3	srad-1024	Rodinia	i5	iozone_64m_r4k_10	IOzone	s8	milc	SPEC2006
p13	gesummv-1024	PolyBench	n1	netperf_tcprr_v4	Netperf	i6	iozone_64m_r4k_11	IOzone	s9	omnetpp	SPEC2006
p14	gramshmidt-256	PolyBench	n2	netperf_tcpstream_v4	Netperf	i7	iozone_64m_r4k_12	IOzone	s10	soplex	SPEC2006
p15	heat3d-64	PolyBench	n3	netperf_udprr_v4	Netperf				s11	sphinx3	SPEC2006
p16	jacobi-2d-512	PolyBench	n4	netperf_udpstream_v4	Netperf				s12	zeusmp	SPEC2006

We conducted a series of new multi-core experiments, encompassing 2-core, 4-core, and 8-core configurations, each comprising 12 experiments. These experiments aimed to compare the performance of optimized mapping and partitioning, leveraging the

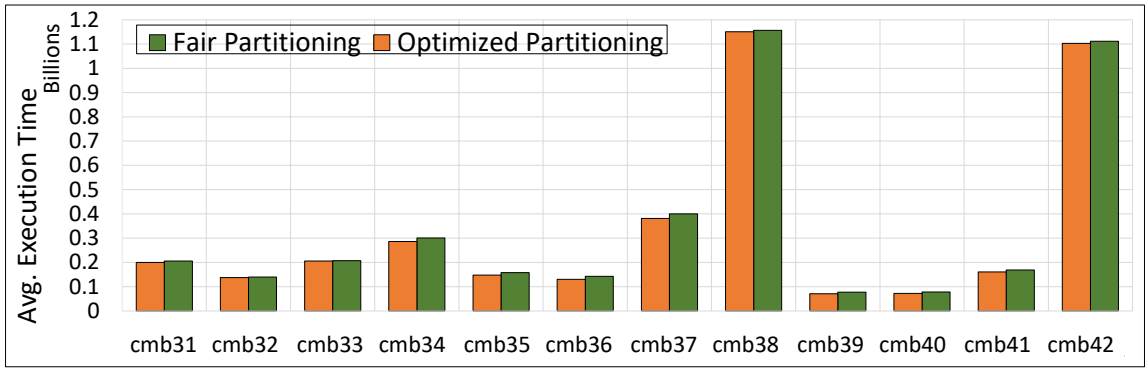
Table 6.3: Benchmark ID of combinations

Cmb	Exp	BMs	Cmb	Exp	BMs	Cmb	Exp	BMs	Cmb	Exp	BMs
cmb1	Fig 6.1(a)	s4, s2	cmb20	Fig 6.1(b)	y3, s2, s12, s4	cmb39	Fig 6.2(a)	i3, n1	cmb58	Fig 6.2(c)	s5, y3, s10, s8, i1, i4, n1, h7
cmb2	Fig 6.1(a)	s11, s2	cmb21	Fig 6.1(c)	s1, h5, i3, i4, s6 , s12, s4, s11	cmb40	Fig 6.2(a)	i5, n3	cmb59	Fig 6.2(c)	s11, s4, s2, s8, i2, h6, h1, s3
cmb3	Fig 6.1(a)	y2, s6	cmb22	Fig 6.1(c)	s1, h5, s10, s2, s6 , s12, s4, s11	cmb41	Fig 6.2(a)	s8, h7	cmb60	Fig 6.2(c)	s2, s12, y1, n2, i6, h7, h4, n3
cmb4	Fig 6.1(a)	i3, s2	cmb23	Fig 6.1(c)	s1, i3, i4, n2, y2, s12, s4, s11	cmb42	Fig 6.2(a)	i1, h4	cmb61	Fig 6.2(c)	y4, y2, s7, i1, i3, s3, n3, h6
cmb5	Fig 6.1(a)	n2, s2	cmb24	Fig 6.1(c)	s1, i2, i3, n2, y3, s2, s6 , s12	cmb43	Fig 6.2(b)	s4, s10, s8, s3	cmb62	Fig 6.2(c)	s5, s10, y3, i7, i4, h2, h3, n4
cmb6	Fig 6.1(a)	i2, s12	cmb25	Fig 6.1(c)	s1, i2, y3, s2, s6 , s12, s4, s11	cmb44	Fig 6.2(b)	y4, s2, n2, n4	cmb63	Fig 6.2(c)	s4, y1, s8, s9, i4, s3, n3, h6
cmb7	Fig 6.1(a)	y1, s11	cmb26	Fig 6.1(c)	i2, y2, y4, s2, s6 , s12, s4, s11	cmb45	Fig 6.2(b)	s7, y2, i5, h1	cmb64	Fig 6.2(c)	s10, s7, n2, i6, i7, h7, n4, n1
cmb8	Fig 6.1(a)	s10, s6	cmb27	Fig 6.1(c)	i3, i4, s5, y4, s2, s12, s4, s11	cmb46	Fig 6.2(b)	s5, s12, i1, h2	cmb65	Fig 6.2(c)	s11, y4, i2, i3, i5, h1, h3, s3
cmb9	Fig 6.1(a)	s12, s4	cmb28	Fig 6.1(c)	i3, i4, s10, n2, s2, s6 , s12, s4	cmb47	Fig 6.2(b)	s7, i6, s9, s3	cmb66	Fig 6.2(c)	s5, s12, i1, i2, i3, h2, h4, s6
cmb10	Fig 6.1(a)	s1, s2	cmb29	Fig 6.1(c)	i3, s10, y1, y2, s2, s12, s4, s11	cmb48	Fig 6.2(b)	s11, i4, i1, n1	cmb67	Fig 6.4	p13, p15, r1, p19
cmb11	Fig 6.1(b)	s1, s6, s12, s11	cmb30	Fig 6.1(c)	s5, n2, y2, s2, s6 , s12, s4, s11	cmb49	Fig 6.2(b)	s12, s8, i2, n3	cmb68	Fig 6.4	p4, l1, l4, r3
cmb12	Fig 6.1(b)	s2, s6, s12, s11	cmb31	Fig 6.2(a)	i2, s12	cmb50	Fig 6.2(b)	y1, n2, i3, h6	cmb69	Fig 6.4	p3, p18, p20, p21
cmb13	Fig 6.1(b)	s2, s12, s4, s11	cmb32	Fig 6.2(a)	n2, s2	cmb51	Fig 6.2(b)	y3, s8, s6, h7	cmb70	Fig 6.4	p2, p5, p6, p9
cmb14	Fig 6.1(b)	h5, y4, s2, s11	cmb33	Fig 6.2(a)	s11, i7	cmb52	Fig 6.2(b)	s4, i2, n3, h2	cmb71	Fig 6.4	l2, p14, p17, r2
cmb15	Fig 6.1(b)	i2, y1, s6 , s4	cmb34	Fig 6.2(a)	y1, i4	cmb53	Fig 6.2(b)	s5, i3, h3, s3	cmb72	Fig 6.4	p3, p6, p18, p19
cmb16	Fig 6.1(b)	i4, s2, s12, s4	cmb35	Fig 6.2(a)	s3, s4	cmb54	Fig 6.2(b)	s10, n2, h4, n1	cmb73	Fig 6.4	p3, p8, p14, r2
cmb17	Fig 6.1(b)	s5, s6 , s12, s11	cmb36	Fig 6.2(a)	s5, n4	cmb55	Fig 6.2(c)	s4, s10, y4, s8, i2, i6, h6, h1	cmb74	Fig 6.4	l3, p11, p10, p16
cmb18	Fig 6.1(b)	s10, s2, s6 , s12	cmb37	Fig 6.2(a)	y4, h6	cmb56	Fig 6.2(c)	s12, y1, s7, i5, s9, i3, h2, h3	cmb75	Fig 6.4	p7, p12, p14, p20
cmb19	Fig 6.1(b)	n2, s2, s6 , s12	cmb38	Fig 6.2(a)	s10, h2	cmb57	Fig 6.2(c)	s2, y2, s11, i4, i7, n2, n3, h4	cmb76	Fig 6.4	p1, p5, l2, p19

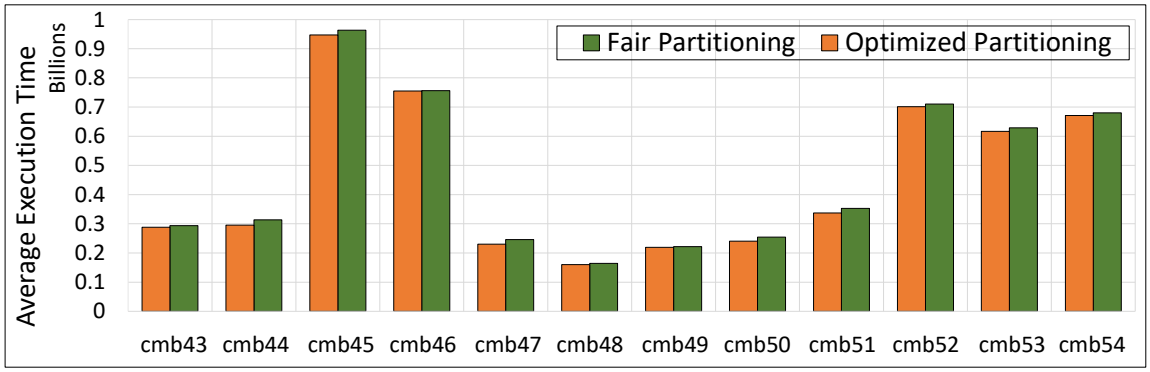
optimization aspect of our methodology, against scenarios employing only the best mapping with fair partitioning. The results of these comparisons are presented in Figure 6.2. To ensure a comprehensive evaluation, benchmarks for each experiment were carefully selected based on the criteria outlined in Table6.4. The benchmarks were categorized into three groups: Group 1 exhibited the most normalized relative difference in their single-core experiments when varying bank numbers from 1 to 16, Group 2 fell within the medium range, and Group 3 showed the least variation. To maintain fairness, different benchmarks from each group were chosen for each experiment. Our experimental setup mirrored that of Table 6.1, with a single channel and rank. The analysis revealed an average relative difference of 4.4% for 2-core, 2.84% for 4-core, and 2.2% for 8-core configurations. Notably, these differences could peak at 8.66% for 2-core, 6.36% for 4-core, and 6.74% for 8-core configurations.

Table 6.4: Classificaton of non-strided benchmarks based on the partitioning benefit

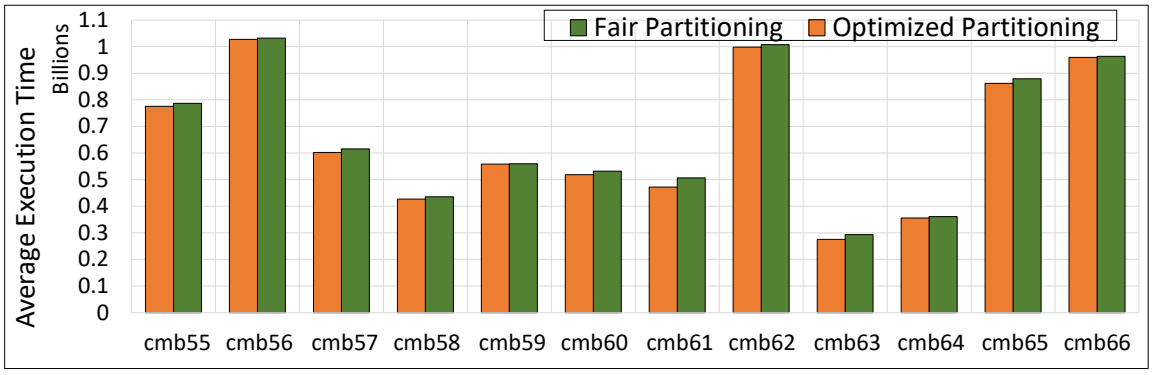
Group	**Benchmarks**
Group 1	s2, s4, s5, s7, s10, s11, s12, y1, y2, y3, y4
Group 2	i1, i2, i3, i4, i5, i6, i7, n2, s8, s9
Group 3	h1, h2, h3, h4, h6, h7, n1, n3, n4, s3, s6



(a) 2-Core



(b) 4-Core



(c) 8-Core

Figure 6.2: Comparison of the effect of optimized partitioning on execution cycles for multi-core systems over non-strided application

6.2 Stream-aware approach

6.2.1 Single-Core Systems

For our evaluation of the stream-based application, we utilized well-known benchmarks such as Poly (all), Rodinia (Hotspot, Lud, Needle, Srad), and Linpack (Daxpy, DDot, Dscal, Scusumkbn), configured with varying stride sizes, to assess the effectiveness of our approach (see Figure 6.3). In these experiments, we employed the same system configuration as described in Table 6.1. Notably, the numbers accompanying the benchmark names denote the stride size of the loops, indicating their respective configurations (before considering the data type's byte size). Similar to our analysis of non-stream-based applications, we compared the execution time across different scenarios: the best base mappings without partitioning, the best mappings obtained through analyzing application strides with fair partitioning, and the best mappings with an optimized partitioning scheme. The results from these experiments indicate that by solely focusing on finding the best mapping for each stream (based on the stride size) and implementing fair partitioning compared to the best base mappings, we can achieve an average improvement in execution time of 10.66%, reaching as high as 50.55%. Furthermore, incorporating the optimized partitioning aspect of our method yields additional improvements. When comparing optimized partitioning with fair partitioning, we observe an average improvement of 0.2%, reaching up to 9.35%. Comparing optimized partitioning with the base mappings, we see an average improvement of 11.07%, reaching as high as 47.48%. (A detailed analysis of stride sizes and suitable mappings of our tested strided benchmarks has been brought in Appendix A)

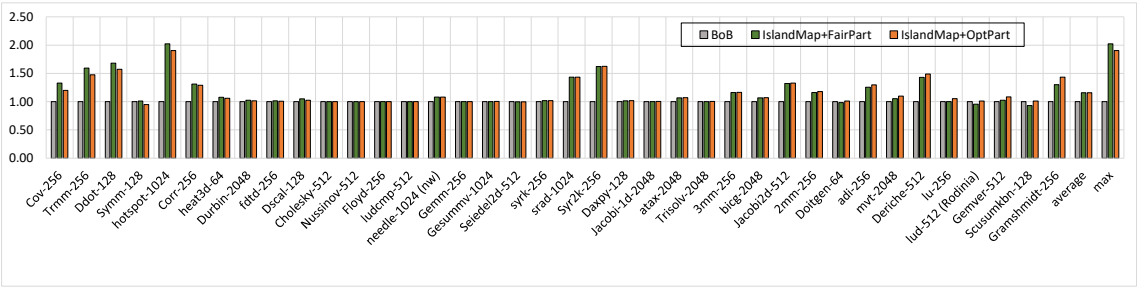


Figure 6.3: Comparison of the execution cycles for single-core systems over stream-based application

6.2.2 Multi-Core Systems

In our examination of stream-based applications on multi-core systems, we expanded our analysis to include four-core setups. Using random benchmarks from the same pool as the single-core strided benchmarks, we conducted 10 experiments to gauge the impact of our approach (refer to Figure 6.4). We compared the time taken to execute tasks across various setups: base mappings without partitioning (map2, map4), best mappings identified through analyzing application strides with fair partitioning, and best mappings with an optimized partitioning scheme using the genetics algorithm from Matlab. The findings reveal that by implementing the first part of our approach (suitable mapping) and comparing fair partitioning to the best base partitioning, we could expect an average improvement of 21.19%, reaching as high as 27.56%. Additionally, by incorporating the optimized partitioning aspect of our approach, compared to fair partitioning, we could see an average improvement of 1.12%, reaching up to 3.61%. Moreover, when comparing optimized partitioning (with suitable mapping) to the best base mappings, we could observe an average improvement of 22.10%, reaching as high as 27.79%.

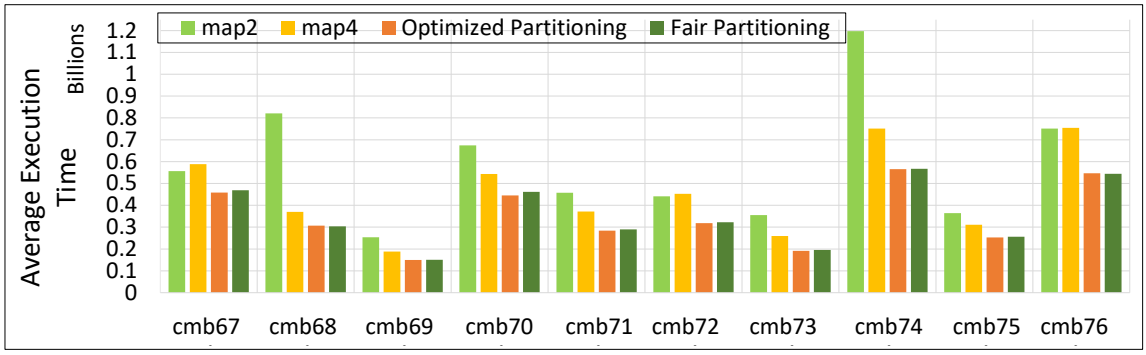


Figure 6.4: Comparison of the execution cycles for multi-core (4-core) systems over stream-based application

The evaluation aims to demonstrate the efficacy of our memory allocation methodology across different types of applications and system configurations. Through comprehensive experiments and analysis, we provide insights into the benefits of static profiling, choosing suitable memory mappings, and optimized partitioning in improving system performance.

Chapter 7

Conclusion

In this thesis, we introduce a novel methodology for optimizing memory allocation in heterogeneous computing systems. Our approach emphasizes static analysis during compile time to achieve predictability and reliability in memory mapping, thereby eliminating the need for dynamic adjustments. By conducting detailed application analysis, focusing on key features such as stride size, our methodology enables personalized memory mapping tailored to individual application requirements. Moreover, we propose fine-grained partitioning techniques to prevent interference between applications and optimize resource utilization, leading to enhanced parallelization and improved application performance.

Our evaluation results demonstrate significant improvements in execution time across diverse application scenarios and system configurations. Through comprehensive experiments, we observed average gains of up to 13.53% and peaks of 47.48% in execution time reduction. These findings underscore the effectiveness of our methodology in addressing memory allocation challenges in complex computing environments. Furthermore, our work highlights the importance of static profiling and personalized

memory mapping techniques in achieving robust and efficient memory allocation optimization. While our approach shows promising results, future research could be explored.

Appendix A

Strided Benchmark Analysis

Table A.1: Strided benchmark detailed analysis

BM	Explanation	Constants	Streams	Stride Size	Mapping
2mm-256	BM with mix of 2-D streams	NI=256 NJ=256 NK=256 NL=256	A-2D B-2D C-2D D-2D tmp-2D	DT DT*NJ DT*NJ DT DT	Base mapping DT*NJ mapping DT*NJ mapping Base mapping Base mapping
3mm-256	BM with mix of 2-D streams	NI=256 NJ=256 NK=256 NL=256 NM=256	A-2D B-2D C-2D D-2D E-2D F-2D G-2D	DT DT*NJ DT DT*NL DT DT DT	Base mapping DT*NJ mapping Base mapping DT*NL mapping Base mapping Base mapping Base mapping
adi-256	BM with single 2-D stream	N=256	u-2D v-2D p-2D q-2D	DT DT*N DT DT*N	Base mapping Base mapping Base mapping Base mapping
atax-2048	BM with mix of 2-D stream and 1-D stream	M=2048 N=2048	A-2D x-1D y-1D tmp-1D	DT DT DT DT	Base mapping Base mapping Base mapping Base mapping
bicg-2048	BM with mix of 2-D stream and 1-D stream	M=2048 N=2048	A-2D p-1D s-1D q-1D r-1D	DT DT DT DT DT	Base mapping Base mapping Base mapping Base mapping Base mapping
cholesky-512	BM with single 2-D stream	N=512	A-2D	DT	Base mapping
corr-256	BM with mix of 2-D stream and 1-D stream	M=256 N=256	data-2D corr-2D mean-1D stddev-1D	DT DT*M DT*N DT	Base mapping Base mapping Base mapping Base mapping
cov-256	BM with mix of 2-D stream and 1-D stream	M=256 N=256	data-2D cov-2D mean-1D	DT DT*M DT*N DT	Base mapping Base mapping Base mapping
daxpy-128	BM with multiple 1-D streams	arrSize=1280000000 strideX=128 strideY=128	X-1D Y-1D	DT*strideX DT*strideY	DT*strideX mapping DT*strideY mapping
Ddot-128	BM with multiple 1-D streams	arrSize=1280000000 strideX=128 strideY=128	X-1D Y-1D	DT*strideX DT*strideY	DT*strideX mapping DT*strideY mapping
deriche-512	BM with mix of 2-D stream and 1-D stream	W=512 H=512	imgIn-2D imgOut-2D y1-2D y2-2D	DT DT DT*H DT DT*H	Base mapping Base mapping Base mapping Base mapping Base mapping
doitgen-64	BM with mix of 3-D stream and 2-D stream and 1-D stream	NQ=64 NR=64 NP=64	A-3D C4-2D sum-1D	DT DT*NP DT	Base mapping DT*NP mapping Base mapping
dscal-128	BM with a single stream	arrSize=1280000000 strideX=128	X-1D	DT*stride	DT*stride mapping
durbin-2048	BM with multiple 1-D streams each has its own stride	N=2048	r-2D y-2D z-2D	DT	Base mapping Base mapping Base mapping
fdtd-256	BM with mix of 2-D stream and 1-D stream	NX=256 NY=256	ex-2D ey-2D hz-2D fict-1D	DT DT DT DT	Base mapping Base mapping Base mapping Base mapping
floyd-256	BM with single 2-D stream	N=256	path-2D	DT DT*N	Base mapping Base mapping
gemm-256	BM with mix of 2-D streams	NI=256 NJ=256 NK=256	A-2D B-2D C-2D	DT DT DT	Base mapping Base mapping Base mapping

gemver-512	BM with mix of 2-D stream and 1-D stream	N=512	A-2D w-1D x-1D y-1D z-1D u1-1D u2-1D v1-1D v2-1D	DT DT*N DT DT DT DT DT DT DT	Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping
heat3d-64	BM with a mix of 3-D streams	N=64	A-3D B-3D	DT DT	Base mapping Base mapping
hotspot-1024	BM with multiple 1-D streams	grid_rows=1024 grid_cols=1024	temp-1D power-1D result-1D	DT DT DT	Base mapping Base mapping Base mapping
jacobi-1d-2048	BM with multiple 1-D streams	N=2048	A-1D B-1D	DT DT	Base mapping Base mapping
jacobi-2d-512	BM with a mix of 2-D streams	N=512	A-2D B-2D	DT DT	Base mapping Base mapping
lu-256	BM with single 2-D stream	N=256	A-2D	DT DT*N	Base mapping Base mapping
lud-512 (Rodinia)	BM with single 1-D stream	matrix_dim=512	a-1D	DT DT*N	Base mapping Base mapping
ludcmp-512	BM with mix of 2-D stream and 1-D stream	N=512	A-2D b-1D x-1D y-1D	DT DT*N DT DT	Base mapping Base mapping Base mapping Base mapping
mvt-2048	BM with mix of 2-D stream and 1-D stream	N=2048	A-2D x1-1D x2-1D y_1-1D y_2-1D	DT DT DT DT DT	Base mapping Base mapping Base mapping Base mapping Base mapping
needle-1024 (nw)	BM with multiple 1-D streams	max_rows=1024 max_cols=1024	input_itemsets-1D referrence-1D output_itemsets-1D	DT DT DT	Base mapping Base mapping Base mapping
nussinov-512	BM with single 2-D stream	N=512	table-2D seq-1D	DT DT*N	Base mapping Base mapping
scusumkbn-128	BM with multiple 1-D streams	arrSize=1280000000 strideX=128 strideY=128	X-1D Y-1D	DT*strideX DT*strideY	DT*strideX mapping DT*strideY mapping
srad-1024	BM with mix of 2-D stream and 1-D stream	rows=1024 cols=1024	I-1D J-1D c-1D dN-1D dS-1D dW-1D dE-1D iN-1D iS-1D jW-1D jE-1D	DT DT DT DT DT DT DT DT DT DT DT	Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping Base mapping
seidel2d-512	BM with single 2-D stream	N=512	A-2D	DT	Base mapping
symm-128	BM with mix of 2-D streams	M=128 N=128	A-2D B-2D C-2D	DT DT*M DT DT*N	Base mapping Base mapping Base mapping
syr2k-256	BM with mix of 2-D streams	M=256 N=256	A-2D B-2D C-2D	DT DT*N DT DT*N	Base mapping Base mapping Base mapping Base mapping
syrk-256	BM with mix of 2-D streams	M=256 N=256	A-2D C-2D	DT DT*N DT	Base mapping Base mapping
trisolv-2048	BM with mix of 2-D stream and 1-D stream	N=2048	L-2D b-1D x-1D	DT DT DT	Base mapping Base mapping Base mapping
trmm-256	BM with mix of 2-D streams	M=256 N=256	A-2D B-2D	DT*M DT DT*N	DT*M-mapping Base mapping

Bibliography

[1] S. Adavally and K. Kavi. Towards application-specific address mapping for emerging memory devices. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '20, page 105–113, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450388993. doi: 10.1145/3422575.3422785. URL `https://doi.org/10.1145/3422575.3422785`.

[2] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber. Future scaling of processor-memory interfaces. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009.

[3] S. Amini, I. Gerostathopoulos, and C. Prehofer. Big data analytics architecture for real-time traffic control. In *2017 5th IEEE international conference on models and technologies for intelligent transportation systems (MT-ITS)*, pages 710–715. IEEE, 2017.

[4] J. S. S. T. Association. JEDEC Standard: DDR4 SDRAM, Standard JESD79-4. Technical report, Sept. 2012. JESD79-4.

[5] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, page 513–526, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371025. doi: 10.1145/3373376.3378498. URL https://doi.org/10.1145/3373376.3378498.

[6] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia. Managing dram latency divergence in irregular GPGPU applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 128–139, 2014. doi: 10.1109/SC.2014.16.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. doi: 10.1109/IISWC.2009.5306797.

[8] W. Chen, Z. Li, L. Liu, and S. Wei. Dynamically reconfigurable memory address mapping for general-purpose graphics processing unit. In *2022 IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, pages 1–2, 2022. doi: 10.1109/ICTA56932.2022.9963035.

[9] M. Ghasempour, A. Jaleel, J. D. Garside, and M. Luján. Happy: Hybrid address-based page policy in drams. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, page 311–321, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343053. doi: 10.1145/2989081.2989101. URL https://doi.org/10.1145/2989081.2989101.

[10] S. Ghose, T. Li, N. Hajinazar, D. S. Cali, and O. Mutlu. Demystifying complex

workload-dram interactions: An experimental study. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–50, 2019.

[11] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *IEEE International symposium on high-performance comp architecture*, pages 1–12. IEEE, 2012.

[12] M. Jung, D. M. Mathew, C. Weis, N. Wehn, I. Heinrich, M. V. Natale, and S. O. Krumke. Congen: An application specific dram memory controller generator. *Proceedings of the Second International Symposium on Memory Systems*, 2016. URL `https://api.semanticscholar.org/CorpusID:14005792`.

[13] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010. doi: 10.1109/HPCA.2010.5416658.

[14] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, 2010. doi: 10.1109/MICRO.2010.51.

[15] J. Kwak, Y. Kim, J. Lee, and S. Chong. Dream: Dynamic resource and task allocation for energy minimization in mobile cloud systems. *IEEE Journal on Selected Areas in Communications*, 33(12):2510–2523, 2015. doi: 10.1109/JSAC.2015.2478718.

[16] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt. Improving memory bank-level parallelism in the presence of prefetching. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 327–336, 2009.

[17] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 213–224, 2010. doi: 10.1109/MICRO.2010.44.

[18] X. Li, Z. Yuan, Y. Guan, G. Sun, T. Zhang, R. Wei, and D. Niu. Flatfish: A reinforcement learning approach for application-aware address mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41 (11):4758–4770, 2022. doi: 10.1109/TCAD.2022.3146204.

[19] Y. Liu, X. Zhao, M. Jahre, Z. Wang, X. Wang, Y. Luo, and L. Eeckhout. Get out of the valley: Power-efficient address mapping for gpus. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 166–179. IEEE Press, 2018. ISBN 9781538659847. doi: 10.1109/ISCA.2018.00024. URL `https://doi.org/10.1109/ISCA.2018.00024`.

[20] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, USA, 2007. USENIX Association. ISBN 1113335555779.

[21] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware

memory channel partitioning. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 374–385, 2011.

[22] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160, 2007. doi: 10.1109/MICRO.2007.21.

[23] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *2008 International Symposium on Computer Architecture*, pages 63–74, 2008. doi: 10.1109/ISCA.2008.7.

[24] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222, 2006. doi: 10.1109/MICRO.2006.24.

[25] L.-N. Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 437:1–1, 2012.

[26] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. *ACM SIGARCH Computer Architecture News*, 34(2):167–178, 2006.

[27] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007. doi: 10.1109/PACT.2007.4336216.

[28] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 99–108, 2011.

[29] S. Rixner. *Stream processor architecture*, volume 644. Springer Science & Business Media, 2001.

[30] SPEC. SPEC CPU2006 Benchmark Suite. `http://www.spec.org/cpu2006/`. Accessed: [Insert Access Date Here].

[31] X. Tang, M. Kandemir, P. Yedlapalli, and J. Kotra. Improving bank-level parallelism for irregular applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[32] The Stdlib Authors. stdlib. URL `https://github.com/stdlib-js/stdlib`.

[33] J. Tuck, L. Ceze, and J. Torrellas. Scalable cache miss handling for high memory-level parallelism. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 409–422. IEEE, 2006.

[34] Y. Wang, A. Ferraiuolo, and G. E. Suh. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236. IEEE, 2014.

[35] Z. Wang and T. Nowatzki. Stream-based memory access specialization for general purpose processors. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 736–749, 2019.

[36] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki. Stream floating: Enabling proactive and decentralized cache optimizations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 640–653. IEEE, 2021.

[37] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 34–44, 2009. doi: 10.1145/1669112.1669119.

[38] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, 2014.

[39] J. Zhang, M. Swift, and J. J. Li. Software-defined address mapping: A case on 3d memory. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 70–83, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392051. doi: 10.1145/3503222.3507774. URL `https://doi.org/10.1145/3503222.3507774`.

[40] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 32–41, 2000. doi: 10.1109/MICRO.2000.898056.

[41] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank:

Adaptive dram architecture for improving memory power efficiency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 210–221. IEEE, 2008.