2013

Power-Efficient and Low-Latency Memory Access for CMP Systems with Heterogeneous Scratchpad On-Chip Memory

Zhi Chen

University of Kentucky, zch229@uky.edu

Recommended Citation

This Master's Thesis is brought to you for free and open access by the Electrical and Computer Engineering at UKnowledge. It has been accepted for inclusion in Theses and Dissertations--Electrical and Computer Engineering by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.
STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained and attached hereto needed written permission statements(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine).

I hereby grant to The University of Kentucky and its agents the non-exclusive license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless a preapproved embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student’s advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student’s dissertation including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Zhi Chen, Student

Dr. Meikang Qiu, Major Professor

Dr. Zhi David Chen, Director of Graduate Studies
Power-Efficient and Low-Latency Memory Access for CMP Systems with Heterogeneous Scratchpad On-Chip Memory

Thesis

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in the Electrical Engineering in the College of Engineering at the University of Kentucky

By

Zhi Chen

Lexington, Kentucky

Director: Dr. Meikang Qiu, Assistant Professor of Electrical Engineering
Lexington, Kentucky 2013

Copyright© Zhi Chen 2013
ABSTRACT OF THESIS

Power-Efficient and Low-Latency Memory Access for CMP Systems with Heterogeneous Scratchpad On-Chip Memory

The gradually widening speed disparity between CPU and memory has become an overwhelming bottleneck for the development of Chip Multiprocessor (CMP) systems. In addition, increasing penalties caused by frequent on-chip memory accesses have raised critical challenges in delivering high memory access performance with tight power and latency budgets. To overcome the daunting memory wall and energy wall issues, this thesis focuses on proposing a new heterogeneous scratchpad memory architecture which is configured from SRAM, MRAM, and Z-RAM. Based on this architecture, we propose two algorithms, a dynamic programming and a genetic algorithm, to perform data allocation to different memory units, therefore reducing memory access cost in terms of power consumption and latency. Extensive and intensive experiments are performed to show the merits of the heterogeneous scratchpad architecture over the traditional pure memory system and the effectiveness of the proposed algorithms.

KEYWORDS: Heterogeneous memory, magnetic random access memory (MRAM), Zero-capacitor random access memory (Z-RAM), scratchpad memory, scheduling

Zhi Chen

May 2, 2013
Power-Efficient and Low-Latency Memory Access for CMP Systems with Heterogeneous Scratchpad On-Chip Memory

By

Zhi Chen

Zhi David Chen
Director of Graduate Studies

May 2, 2013
Date
To my wife and my family
I’d like to thank my advisor, Dr. Meikang Qiu, for his invaluable guidance and support. His encouragement and advice has helped me in many matters including writing academic papers and the success of this project. Other faculties in the ECE department of the University of Kentucky have also helped me a lot in these two years.

I would like to thank Prof. Hank Dietz for his countless priceless advices on both teaching and research. I’ve been the TA of one of his classes for 2 years. I appreciate the teaching opportunities he gave me in this period. I would also thank Prof. J. Robert Heath for his teaching, enlightenment, and very helpful advices on the course and research.

I thank numerous other professors who I have taken classes with and offered me guidance on research and personal life over the years, including Zhi David Chen, Liga Abolins, Lawrence E. Holloway, and Grzegorz Wasiakowski. I was greatly enlightened by observing the teaching style from some of them.

Thanks to Shuo Pang, Zhiyang Zhang, and Ying Yu for their friendship. Thanks to Hai Su and Jiayin Li, my officemates, for their numerous impromptu discussions and insights. Their valuable suggestions help me a lot on both of my academic research and personal life.

Most importantly, I dedicate this thesis to my wife Jing, who always gives her utmost supports for me to pursue my degree in the United States. Over the past many years, she changed her roles as my girlfriend, fiance and wife, but the only thing has been never changed is her constant support to my pursuit of academic dream. I would like to express my appreciation for her consistent patience, understanding, sacrifices, and commitment to help me reach this point. She has strongly gone through many hard years without my companion.
Thanks to my parents and parents-in-law and to my sisters for their constant financial and moral support. They have always been the constant source of inspiration during the rough times of life in a foreign country. My parents instilled in me the value of hard working and determination and encouraged me to pursue the life I long for. Words are incapable of describing my gratitude for what they have done for me.

Finally, I gratefully thank the other people who have helped, supported, and encouraged me to achieve my goal.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>Chapter 1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Cache VS. Scratchpad Memory</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Algorithms for Data Allocation in CMP Systems with SPM</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Organization</td>
<td>7</td>
</tr>
<tr>
<td>Chapter 2 Background and Related Work</td>
<td>8</td>
</tr>
<tr>
<td>2.1 SPM: Background and Problem Statement</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Related Work in SPM Architecture</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Related Work in SPM Data Allocation</td>
<td>13</td>
</tr>
<tr>
<td>2.3.1 Allocation Objects</td>
<td>13</td>
</tr>
<tr>
<td>2.3.2 Static Allocation and Dynamic Allocation</td>
<td>14</td>
</tr>
<tr>
<td>Chapter 3 Utilization of Multidimensional Dynamic Programming for Data Allocation</td>
<td>19</td>
</tr>
<tr>
<td>3.1 Definitions and Models</td>
<td>19</td>
</tr>
<tr>
<td>3.1.1 System Model</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2 Allocation Granularity</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Motivational Example</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Algorithms</td>
<td>26</td>
</tr>
<tr>
<td>3.3.1 Allocation Cost Table</td>
<td>27</td>
</tr>
<tr>
<td>3.3.2 Recursive Formulation</td>
<td>28</td>
</tr>
<tr>
<td>3.3.3 MDPDA Algorithm</td>
<td>31</td>
</tr>
<tr>
<td>Chapter 4 Genetic Solution to the Data Allocation Problem</td>
<td>36</td>
</tr>
<tr>
<td>4.1 Motivation</td>
<td>36</td>
</tr>
<tr>
<td>4.2 System Model</td>
<td>37</td>
</tr>
<tr>
<td>4.2.1 Hardware model</td>
<td>37</td>
</tr>
<tr>
<td>4.2.2 Chromosome Model</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Description of the Adaptive Genetic Algorithm</td>
<td>38</td>
</tr>
<tr>
<td>4.3.1 Problem Statement</td>
<td>38</td>
</tr>
<tr>
<td>4.3.2 Initialization</td>
<td>40</td>
</tr>
<tr>
<td>4.3.3 Fitness Function</td>
<td>40</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

1.1 Performance gap between processor and memory ........................................ 2
2.1 A typical scratchpad memory ........................................................................ 9
2.2 Elementary MRAM cell .................................................................................. 10
2.3 Elementary ZRAM cell .................................................................................. 11
2.4 A typical scratchpad memory ........................................................................ 15
3.1 System architecture ....................................................................................... 20
3.2 An example to illustrate the whole procedures of our MDPDA algorithm ...... 35
4.1 Two chromosomes in matrix structure .......................................................... 39
   (a) Chromosome C1 ...................................................................................... 39
   (b) Chromosome C2 ...................................................................................... 39
4.2 Change the chromosomes in Figure 4.1 into list structure, C1 → C3, C2 → C4. 39
   (a) Chromosome C3 ...................................................................................... 39
   (b) Chromosome C4 ...................................................................................... 39
4.3 An example of cycle crossover ...................................................................... 44
4.4 An example of adaptive cycle crossover ....................................................... 44
4.5 An example of the modified PMX algorithm .................................................. 46
4.6 An example of mutation between gene (C, 2) and (H, 5). ............................ 47
5.1 The comparison of performance of hybrid SPM and pure SPM with respect to latency and energy consumption ................................................................. 52
   (a) Latency comparison .................................................................................. 52
   (b) Energy comparison ................................................................................... 52
5.2 The comparison of the number of writes to MRAM for the greedy algorithm and the MDPDA algorithm, when the target system is a ........................................ 53
   (a) Dual-core system ..................................................................................... 53
   (b) Quad-core system ................................................................................... 53
5.3 The comparison of memory access latency for the greedy algorithm and the MDPDA algorithm, when the target system is a ........................................ 55
   (a) Dual-core system ..................................................................................... 55
   (b) Quad-core system ................................................................................... 55
5.4 The comparison of memory access power for the greedy algorithm and the MDPDA algorithm, when the target system is a ........................................ 56
   (a) Dual-core system ..................................................................................... 56
   (b) Quad-core system ................................................................................... 56
5.5 The comparison of the number of writes operations to MRAM caused by data allocation strategies of the greedy algorithm and our proposed adaptive genetic algorithm (AGADA). ......................................................... 57
5.6 The comparison of energy consumption caused by the greedy algorithm and
the adaptive genetic algorithm (AGADA) for data allocation. . . . . . . . . . . 57
5.7 The comparison of memory access latencies caused by the greedy algorithm
and the adaptive genetic algorithm (AGADA) for data allocation. . . . . . . . 58
5.8 The comparison of dynamic energy consumption caused by the multi-dimensional
programming algorithm and the adaptive genetic algorithm (AGADA) for data
allocation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Latency and energy consumption for access to different memory modules</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Latency of moving data between different memory modules</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Energy consumption of moving data between different memory modules</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>The number of data accesses for each core</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>The comparison of data allocation results for the greedy algorithm and the improved algorithm, when the size of each SRAM, MRAM, and Z-RAM is 200B, 400B, and 200B, respectively.</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>The comparison of data allocation results for the greedy algorithm and the improved algorithm, when the size of each SRAM, MRAM, and Z-RAM is 200B, 400B, and 400B, respectively.</td>
<td>26</td>
</tr>
<tr>
<td>3.7</td>
<td>Allocation cost table</td>
<td>28</td>
</tr>
<tr>
<td>3.8</td>
<td>Latency and energy consumption of allocating the first 6 of the 18 datum to corresponding memory modules when only core 1 is used.</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance parameters for the target systems and memory modules</td>
<td>50</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Over the past decades, performance of computing systems in terms of speed, power consumption, and reliability has been dramatically improved with the continuous development of silicon technology. Moore’s Law tells us that the number of transistors on a single chip is roughly doubled by every 18 months due to the enhancement of silicon technology. As a result, performance of processors has almost doubled roughly every 18 months by adding more transistors and bumping up frequencies of processors.

However, the continuous doubling integration of transistors doesn’t mean the same magnitude of CPU performance improvement. It is observed that the doubling transistor density in every technology generation can only contribute to 40% faster circuit and power consumption (with twice as many transistors) stays the same [1]. Furthermore, not all components of computer systems are able to cope with the advancements in the number of transistors and the speed of processors. For example, although CPU speed increased at an annual rate of 55% during the past decade, memory speed has been improved at most 10% annually at the same time. Dominated by this speed gap between powerful CPU and inadequate memory access, performance of many applications still has not been fully exploited. It is also measured that the frequent memory access will consume 41% of the total energy of a processor [2]. These technology barriers are termed as “Memory Wall” and “Power Wall” [3, 4, 5], which are significant deterrents to make computer system keep up with ever-increasing computational demands of applications. Given these technology trends (as shown in [1,1]), the low memory access speed severely overwhelms modern software systems. Therefore, it is of importance to design energy-aware and high performance memory architecture to sustain computation needs of different applications.
Multicore devices, a ubiquitous technology in wide computing domains including embedded systems [6,7], have emerged as a promising solution towards the heat dissipation and data synchronization limitations faced by current uniprocessor systems. Chip Multiprocessor (CMP) systems are also one type of multicore technologies, by combining a number of homogeneous and heterogeneous processors on a single chip to deal with specific real-time, low-power, and multitasking applications [8,9,10,11]. Although multicore systems contribute to the obvious benefits, they also complicate memory managements since the memory hierarchy becomes more heterogeneous in this case.

1.1 Cache VS. Scratchpad Memory

Low-power and short-latency memory access are critical to the performance of CMP devices. However, the continuous development of the current CMP systems is substantially hindered by the ever-widening processor-memory speed gap. To address this problem,
most of the mainstream processor vendors, such as IBM, Intel, and AMD, have exploited a number of techniques, including latency hiding [13][14][15] and SRAM-based hardware caches, to shrink the memory access latency. However, latency hiding often leads to a linear increase in power consumption, while yielding only a limited increase in the performance of memory systems.

Cache, storing a subset of the frequently accessed variables, has already facilitated the layered memory hierarchy for desktop systems and servers. It is predicted that the dominance of caches in desktops and servers will likely continue in the near future. The most important reason for the achievements of caches is their excellent portability since the compiled code can be fitting to different cache sizes without recompilation. In addition, caches are hardware controlled and explicitly addressed. As a result, the on-chip space is managed transparently and invisible to software which enables computing systems to exploit temporal and spatial locality and automatically handle intra-memory communications [16][17], even under the circumstance when sharing patterns are hard to capture [18]. To make more successes in using caches, Non-Uniform Cache Architecture (NUCA) is also proposed to shrink the gap between powerful CPU and inadequate memory access speed [19][20][21][22].

However, in the embedded system realms, although the binary portability of caches are still helpful, their functionality is usually overshadowed because software is co-designed with the systems and we rarely need to recompile it. Furthermore, caches impose many notorious problems to CMP systems, such as hard guarantee of predictability and high penalties in cache misses, area cost, and energy consumption. For example, caches consume up to 43% of the overall power in the ARM920T processor [23]. Unfortunately, these metrics are critical to an embedded system, thus motivating the efforts to find an alternative technology to replace hardware managed caches in embedded CMP system.

Scratch Pad Memory (SPM), a software-controlled on-chip memory, has been widely employed by key manufacturers due to two major advantages over their cache memory
counterparts. First, SPM does not have the comparator and tag SRAM, since it is accessed by direct addressing. Therefore, they don’t perform the complex decode operations to support the runtime address mapping for references. This property of SPM can save a large amount of energy. It is studied that a SPM consumes 34% smaller chip area and 40% lower energy consumption than a cache memory does [24]. Second, SPM generally guarantees single-cycle access latency, whereas accesses to cache may suffer capacity, compulsory, and conflict misses that incur very long latency [25]. Given the advantages in size, power consumption, and predictability, SPM is widely used in CMP systems, such as Motorola M-core MMC221, IBM CELL [26], TI TMS370CX7X, and NVIDIA G80. In addition, SPM is efficient in providing software with full flexibility on locality and communication management regarding addressing, granularity, and replacement policy [18].

However, scratchpad memories are not suitable for desktop processors where software may run from one version to another, rendering the on-chip memory size variable. Another shortcoming of SPM is that they cause higher software complexity due to the explicit management of on-chip address space, hence resulting in more challenges for programmers or compilers since they need to explicitly manage the address mapping of references. These problems must be carefully investigated and efficiently resolved before applying on CMP systems with hundreds or even thousands of on-chip SPM memories [27, 28, 29, 17].

Motivated by the above problems, this thesis is dedicated to investigation and development of new memory management techniques from the architecture perspective to the efficient algorithm aspect. The goal is to efficiently manage on-chip SPM resources and effectively reduce the memory access cost in terms of latency and power consumption, as well as extend their lifetimes.

1.2 Algorithms for Data Allocation in CMP Systems with SPM

Traditionally, numerous previously employed approaches for data allocation problems involve ILP [30, 31, 32, 33], dynamic programming, and heuristic approaches. ILP methods
have garnered wide interests in recent years, since it can achieve optimal solutions for the problems in consideration. However, in the data allocation context for multicore architectures, both time complexity and space complexity are critical factors. This seriously limits the applicability of ILP-based algorithms to the data allocation problem for heterogeneous SPM architectures due to the ILP formulations are always known to be nondeterministic and their solutions are NP hard in the worst case, therefore, the incurring excessive computational overhead. In addition, the intellectual property of source code is another big obstacle for the wide utilization of IPL methods. Therefore, ILP methods require high maintenance cost to combine large code which significantly constrains their extensive applications in commercial compilers.

Heuristic methods are fast and require less memory, but usually perform poorly in guaranteeing good solutions. Besides, it is possible that heuristic methods don’t converge for complicated cases when the program is very large and results in a considerably large number of blocks. Needless to mention the even more complex case that the are intricate dependencies between these blocks. To work under the tight power budget for embedded CMP systems, we need to consider more sophisticated algorithms. This thesis explores alternative strategies for heterogeneous SPM architectures to reduce energy consumption and latency incurred by frequent memory accesses.

Generally, dynamic programming algorithm can derive optimal solutions for problems at the acceptable time overhead. It is an important technique aimed at addressing optimization problems through breaking them into some subproblems which are able to be solved optimally within polynomial complexity [34]. We will design a Multi-dimensional Dynamic Programming Data Allocation (MDPDA) strategy to allocate data on different memory modules in polynomial time. Then, we make an attempt to design an adaptive genetic algorithm to further improve the space complexity of the proposed algorithm. To the best of our knowledge, this is the first paper to address the data allocation issue for CMP systems with hybrid SPMs comprising three types of memory modules. The goal
of our proposed algorithm is to minimize the overall cost (energy and latency) incurred by memory accesses. Experimental results show that our proposed algorithms can significantly reduce the number of write activities to MRAM, dynamic energy consumption, and memory access latency.

1.3 Contributions

The major contribution of this thesis are the following:

1. *Investigation of the combinations of different memory techniques:* Since different memory technologies have different properties in terms of density, duration, power consumption, access speeds, etc, it is a challenging but worth work to investigate which combination of memories is most efficient in reducing memory access latency for heterogeneous architectures. We propose a hybrid SPM architecture that consists of SRAM, MRAM, and Z-RAM. This architecture produces high access performance with low power consumption.

2. *Optimal data allocation strategies:* Static data allocation is able to achieve optimal data allocation in embedded systems [35]. There are a number of algorithms can be used to find an optimal data allocation, but most of existing data allocation techniques rely on the integer linear programming (ILP), which incurs high computation overhead. Therefore, it is critical to explore what alternative algorithms are suitable for the proposed architecture. We propose a multi-dimensional dynamic programming data allocation strategy to reduce memory access latency and power consumption, along with cutting the number of write activities on MRAM. The reduction of writes on MRAM will efficiently prolong their lifetime.

3. *Space limitation of embedded systems:* The on-chip memory capacity for embedded system is tightly constrained. How to design efficient algorithms to allocate data in applications to on-chip memories while satisfying the limited space requirement
of the embedded system is another challenge. Considering the high space demands of the multidimensional dynamic programming algorithm, we further propose an adaptive genetic algorithm with very limited sacrifice in the accuracy of solutions.

1.4 Organization

The remainder of this thesis is organized as follows. Chapter 2 presents the background materials of this thesis and overviews the related work on data allocation for CMP systems SPM. Chapter 3 introduces the multidimensional dynamic programming algorithm. In this chapter, we also present some motivational examples to illustrate our basic ideas. Chapter 4 describes the adaptive algorithms employed in the thesis, including crossover and mutation processes. Chapter 5 discusses the development and simulation methodology used to evaluate our proposed algorithms. The simulation results from different methods are also presented in this chapter. Chapter 6 concludes the thesis based on our findings.
Chapter 2

Background and Related Work

This chapter provides the background and related work to help understand this thesis. More specifically, Section 2.1 presents the basis and terminologies for CMP systems and scratch-pad memories. This information is helpful to understand our motivation and basic idea of this thesis. Section 2.2 provides the related work regarding the architecture design of SPM. Based on the investigation and analysis, we will make a proposal to describe the major idea of our heterogeneous SPM architecture. Section 2.3 introduces the related work in data allocation for computing systems with SPM.

2.1 SPM: Background and Problem Statement

Scratchpad Memory (SPM) is a software controlled on-chip memory that has been envisioned as a promising alternative to hardware caches in both uniprocessor and multiprocessor embedded systems with tight energy and timing budgets, due to its superiority in timing predictability, area and power consumption, and guarantee of single cycle access latency. Figure 2.1 shows a typical processor with a scratchpad memory, in which the SPM is implemented by direct address mapping. Particularly, the access address is always in a predetermined memory space range [36]. To efficiently use the SPM, scratchpad memory management unit (SMMU) is regularly introduced so that the programmers or compilers can explicitly manage the data allocation on it [37, 38].

Since this benefit is achieved at the cost of interference from programmer or compiler, the development of sophisticated mechanisms is a must to SPM management therefore improving the overall system performance. This thesis aims to address the data allocation
problem for the CMP embedded systems (but not just limited to CMP systems, it can be also easily applied to uniprocessor embedded system) based on the proposal of a heterogeneous architecture associated with an array of novel scheduling algorithms. The goal is to reduce the memory access cost and extend the wear-out leveling of the on-chip systems.

2.2 Related Work in SPM Architecture

Conventionally, SPMs are configured by small and fast SRAMs. SRAM, usually built by using CMOS process, is superior in providing fast memory access, making them the most widely employed on-chip memory technology. However, a SRAM cell consists of by 6 transistors, consuming large chip area therefore yielding low density. Moreover, SRAM technologies produce high standby/leakage power because the cell structure incurs complex subthreshold and gate leakage paths.

*Magnetic RAM* (MRAM) has been gathering wide interests for various appealing characteristics, such as high density, fast access speed, and excellent non-volatility [39, 40, 41, 42]. Unlike traditional RAM technologies where information is carried as electric charges, data carrier of MRAM is *Magnetic Tunnel Junctions* (MTJs). For SRAMs, in order to retain data when the power is off, a battery is required, but batteries introduce an array of
problems including replacement and frequent failures. Therefore, MRAM depends on the superposition of two orthogonal magnetic fields to perform selectivity. The most widely used structure of MRAM cell consists of one NMOS transistor as the access device and one MTJ as the storage cell [41], which is often referred to as “1T1J” structure, as shown in Figure 2.2. Although this structure makes the technology expensive, it enables MRAM have very high density and read performance. We can see in Figure 2.2 that the MTJ is connected with a NMOS transistor which is controlled by the word line (WL). Whenever a write is signaled, a high positive/negative voltage difference is introduced between the source line and the bit line for writing a “0”/“1”. This process will incur long latency and high current amplitude to reverse and retain the direction of the free layer. In sum, long write latency, high write power, as well as prohibitively expensive cost of MRAM have overwhelmed their extensive usage.

There is a new memory technology developed by Innovative Silicon, Zero-capacitor RAM (Z-RAM), to overcome the high cost of SRAM and MRAM with virtually very few performance degradation. In the past few years, AMD has licensed the second generation Z-RAM for high potential to be used in future multiprocessors. Z-RAM is manufactured with only one transistor instead of six transistors used in SRAM. Therefore, they can afford
Figure 2.3: Elementary ZRAM cell.

much higher density (usually 5x) than SRAM. Figure 2.3 presents the an elementary Z-RAM cell. We can see from this figure that the charge is stored in the floating body of the transistor instead of the separate capacitor structure of DRAM. Elimination of the capacitor used in a conventional DRAM cell benefits Z-RAM in several aspects. First, Z-RAM is able to scale to even much smaller fabrication processes than that of DRAM since the capacitor has to grow larger to retain keep sufficient charge storage. Second, it enables Z-RAM yield twice density of the conventional DRAM and even 5 times density of SRAM. Third, without the need to recharge the capacitor, read and write performance of Z-RAM is much faster than DRAM. As a result, dynamic power consumption of Z-RAM is much less than that of DRAM. Fourth, the simpler structure makes it much cheaper to manufacture than SRAM and MRAM do and consume much less die area of a chip. However, the biggest disadvantage of Z-RAM is their non-volatility as SRAM and DRAM, and their relatively long read/write latency.

In term of these concerns, we propose a hybrid SPM architecture, which incorporates SRAM, MRAM and Z-RAM as the on chip memory, to enhance the overall performance of memory systems. Prior works have investigated hybrid cache by using SRAM and MRAM and proved that the hybrid architecture can save significant amounts of energy [43, 44]. However, this is the first research to configure the hybrid SPM by using SRAM, MRAM,
Stacked 3D integration technology has been emerged as a response to the limitations of traditional ICs by vertically stacking and integrating various technologies and functional components on a die [45, 46, 47]. 3D integration is more advantageous than traditional 2D design techniques in the following aspects [48]. First, the vertical distance between two layers is usually between $10\mu m$ to $100\mu m$, resulting short interconnects therefore offering higher performance. Second, the reduction in the wire length contributes to lower interconnect power consumption. Third, 3D integration has smaller footprint. Fourth, they are able to support heterogenous technologies. This technique has facilitated the implementation of the 3D Stacked hybrid cache architecture in [43, 44, 49, 50]. These studies offer solid foundation for the feasibility of integrating SRAM, MRAM, and Z-RAM into on-chip SPM.

Hybrid architectures for processor cores and cache system design have been gathered much attention recently. In [51], Sun and Xie et al. explored the performance of MRAM and confirmed their potential to be employed as a cache, due to their advantages in access latency and power consumption. In [52], Saripalli et al. investigated the advantages of heterogeneous technologies for processor cores. They discussed the integration of Tunnel-FEL and a MRAM in a cache together with a SRAM. In [44], Wu et al. studied the inter and intra cache level hybrid cache architectures. They also explored the potential of hardware support for the intra cache data movement and power consumption management. In addition, they summarized the benefits of the hybrid cache architecture, including the increase in cache size, the decrease in power consumption, and the check of cache lines in parallel. However, these mechanisms are only proposed for hardware-controlled caches, while unsuitable for the software managed SPM. There are also other works proposed hybrid cache architectures, such as [53, 54, 55], by fabricating the cache and SPM on the same processor to either dynamically or statically capture the behaviors of caches. Unlike the previous work, we propose a hybrid SPM consisting of different memory technologies
including SRAM, MRAM, and Z-RAM.

Although a hybrid SPM can take advantages of different kinds memory technologies, a challenging problem must be addressed before utilizing them effectively, which is how to reduce energy consumption, memory access latency, and the number of write to MRAM. Targeting the benefits of each type of the involved memories, we must strategically allocate data on each memory module so that the total memory access cost can be minimized. Recall that the SPM is software-controllable, which means the datum on it can be managed by programmers or compilers. The traditional hybrid memory data management strategies, such as data placement and migration [35, 56, 57, 43], are unsuitable for hybrid SPMs, since they are mainly designed for hardware cache and unaware of write activities. Fortunately, embedded system applications can fully take the advantage of compiler-analyzable data access pattern that can offer efficient data allocation mechanisms for the hybrid SPM architecture [58].

2.3 Related Work in SPM Data Allocation

There are ample previous research on the data allocation problem for embedded systems with SPMs. Depending on different criteria, these work can be roughly classified as shown in Figure 2.4 [36].

2.3.1 Allocation Objects

Generally, the allocation objects can be program code, program data, or the mix of them with different concerns. For example, we need to manage program flow for the program code based allocation [59], and we need to consider the characteristics of different program data, such as stack, heap, and global variables, for the program data based allocation. In order to effectively exploit the memory access pattern for data allocation, data partitioning and loop scheduling mechanisms are often considered jointly [60, 61].
Most of previously proposed techniques focus on the management of program code, global variables, and stack data, and heap data for systems with SPM. Among all these program objects, heap data is the most difficult one to deal with due to two major reasons. First, heap objects are allocated by dynamic memory allocation mechanisms, such as the keywords `malloc` and `new`, to store dynamic data structures such as trees, graphs, and linked lists. The size of these variables is generally unknown at the compile stage, thus making the determination of a suitable data to SPM mapping intractable at compile-time.

Second, heap variables are allocated with dynamic methods (which will be introduced later) to move data back and forth between on-chip SPM and DRAM at runtime. This process will inevitably cause a notorious problem—invalid pointers. The reason is that heap data often is often linked to other heap data by pointers, such as the next pointers in a linked list element and the children pointers in a tree node. Whenever a heap data is moved from SPM to DRAM, all the incoming pointers become invalid. It is usually prohibitively expensive to maintain these pointers because the pointers are so frequently moved in an application. Although some techniques are developed to attack the data allocation for heap data, such as software caching, they incur significant cost code size, additional runtime, tag cost, and even power consumption, and these overheads can even make the gains from locality in pale. Therefore, we mainly focus on the program code while keeping the heap data allocation as a future work. More specifically, the basic granularity of our allocation algorithms is basic code block because it has been proved to be almost the best candidate for SPM mapping.

2.3.2 Static Allocation and Dynamic Allocation

Depending on the time when the data allocation decision is made, existing work can be categorized into static data allocation and dynamic data allocation. In static data allocation scenarios, the analysis of application program and data allocation decision is made at
Figure 2.4: A typical scratchpad memory

compile-time (offline). The required memory blocks are loaded into SPM at the system initialization stage and remain the same during the execution. The most prominent of static allocation approach is easy to implement and doesn’t need to much runtime resources.

According to the detailed comparison of on-chip to off-chip memory, while targeting scalar and array variables, Avissar et al. [35] proposed a static method for data allocation on SPMs. Padan et al. [76, 25] proposed static data partition strategies to exploit on-chip SPMs. Their major goals are to minimize total execution times for different embedded applications. To balance the workloads of the parallel processors, Ozturk et al. [77] explored loop scheduling techniques for multicore systems with SPMs. Verma et al. proposed an ILP-based approaches to allocate data for on-chip SPMs [67, 78]. Although their method can obtain optimal solution for some applications, the computation is prohibitively expensive to apply on some other applications. Differing from previous research, we mainly focus on using a Multi-dimensional Dynamic Programming Data Allocation (MDPDA) strategy to reasonably allocate data into different memory modules of the proposed hybrid SPM, and our objective is to reduce energy consumption, latency of memory access, and the number of write operations to MRAMs.

Angiolini and Menichelli, et al. [62] proposed a dynamic programming algorithm to optimally schedule a set of instruction blocks into a dedicated SPM based on their access
frequency. The goal of their work is to minimize energy consumption or program execution time. However, their work focuses on the homogeneous SPM which is configured from SRAM. In addition, although there is no need for application sources in hardware customization approach, the usage of this method is usually restricted to the architectures with the required special hardware. Unlike their work which is based on the hardware customization, we will mainly focus on using software methods.

Compared to the static allocation counterpart, program data/code to memory mapping is determined when the application is running in dynamic allocation approaches. Furthermore, data can be reloaded into SPM at some designated program points to guarantee the execution of the application. Therefore, dynamic allocation needs to be aware of the contents in SPM over time. Most of dynamic allocation approaches used in the literature commonly perform a compile-time analysis to determine the memory blocks and reloading points therefore amortizing runtime delay. In addition, good analysis of the profiled trace file or historical information of program execution is effectively beneficial to making better mapping decision. However, the most obvious shortcoming of dynamic allocation is the inexorable high cost of data mapping at runtime. To reduce this overhead, previous work depends on either pre-extracting part of program that doesn’t need runtime information \[65, 71\] or performing a compile-time analysis to find out the potential allocation sites \[79, 80\].

Udayakumaran et al. \[56\] proposed a heuristic algorithm to allocate data for a SPM, with major consideration of stack and global variables. Dominguez et al. \[66\] applied a dynamic data allocation method on heap data for embedded systems with SPMs. Three types of the program object are considered in their allocation method: global variables, stack variables, and program code. They divided a program into multiple regions, where each program region is associated with a time stamp. According to the order of time stamps, they then utilized a heuristic algorithm to determine the data allocation for each program region. Chen and Ozturk et al. presented a dynamic management method for
irregular array accesses in [81]. While they can deal with the case of an indirect indexed array, the array has to be accessed by an affine function.

[72] Kandemir et al. [57] proposed a compiler-controlled framework to manage the dynamic data for the on-chip SPM. Their algorithm is primarily oriented to array-intensive nested loops with regular data accesses. Takase et al. [82] proposed spatial, temporal, and hybrid methods for SPM partitioning and code allocation in priority-based preemptive multitask systems. Steinke et al. [59] model the data allocation problem with ILP formulations by considering the cost of placing selected program and data into SPM. Based on IBM CELL, Bai and Shrivastava proposed a method to manage heap data in the local memory by hide the programming complexity [74]. Baker and Panda et al. [83] proposed instruction mapping scheme for SPM via partitioning it into multiple regions. All these attempts are implemented with pure SPM which is configured by SRAM, without taking into account the hybrid SPM architecture.

In order to make full use of the throughput of stream applications in many-core systems, Che et al. [30] proposed two algorithms, an Integer Linear Programming (ILP) algorithm and a heuristic approach. However, the time overhead of ILP algorithm will increase exponentially with the linear increase of the number of data. The heuristic approach is not sufficient to guarantee the accuracy for solutions.

In [58], Sha et al. proposed a multi-dimensional dynamic programming strategy for the hybrid SPM architecture. Their method is able to achieve optimal allocation for each program region. Considering the efficiency of this algorithm proved in their paper, we propose a dynamic algorithm to tackle the static allocation problem for hybrid memory system. Four major differences distinguish their approach and the one proposed in this thesis.

First, while their target hybrid architecture only consists of a NVM and SRAM, this thesis investigates the features of MRAM and Z-RAM, and we proposed a more complicated architecture to attack the on-chip memory access problem.
Second, [58] targets in single processor platforms with hybrid SPM. However, we step further to focus on multicore embedded systems where each of core is attached with a hybrid on-chip memory.

Third, while their dynamic algorithm is bottom-up oriented, our method is in a top-down style. Therefore, their algorithm is always from the maximum allocation cost at hand, but our approach examines the program code/data blocks in order. Usually the blocks can be obtained by inserting special instructions at the beginning of each program block. Each step of approach is able to achieve an optimal allocation for all the data blocks in consideration.

Finally, they partitioned a program into multiple regions and aimed to manage the access to each region, our approach is more static, which concentrates on the optimal global allocation of program blocks. Hence, they focus on data allocation of each programming region while we focus on the whole on-chip memory area.

Donaldson et al. [84] presented a tool, SCRATCH, to automatically analyze SPM code for heterogeneous multicore processors. This tool can be applied on a large number of programs with the aid of the IBM Cell SDK. However, differing from their heterogeneous architecture, our platform is designed with different memory technologies. However, this method will consume a significant amount of time and space. Based on this observation, we use a genetic algorithm to allocate data on different memory units for CPMs with our novel hybrid SPM comprising SRAM and MRAM.
Chapter 3
Utilization of Multidimensional Dynamic Programming for Data Allocation

This chapter introduces the details of our multidimensional dynamic programming algorithm. We first present the system model in Section 3.1. Then we give an example, in Section 3.2 to illustrate the basic idea of the motivation. Finally, detailed descriptions of the dynamic programming algorithm is presented in Section 3.3.

3.1 Definitions and Models

3.1.1 System Model

Figure 3.1 exhibits the architecture of a target CMP system with hybrid SPMs. Each core is tightly coupled with an on-chip SPM which is composed of a SRAM, a MRAM, and a Z-RAM. We call a core accesses the SPM owned by itself as local access, while accessing a SPM held by another core is referred to as remote access. Generally, the remote access is supported by an on-chip interconnect. All cores access the off-chip main memory (usually a DRAM device) through a shared bus. CELL processor [85] is an example that adopts this architecture. In a CELL processor, there is a multi-channel ring structure to allow the communication between any two cores without intervention from other cores. Consequently, we can safely assume that the data transfer cost between cores is constant. Generally, accessing the local SPM is faster and dissipates less energy than fetching data from a remote SPM, while accessing the off-chip main memory incurs the longest latency and consumes most energy.
Figure 3.1: System architecture. A n-core with hybrid on-chip SPMs and an off-chip DRAM main memory. Core1 accesses data in SPM1 is referred to as local access, while accessing data in other cores is regarded to as remote access. All accesses to shared main memory utilize the on-chip interconnect.

In order to make sure a hit for an access to the memory modules on the heterogeneous memory, we need to move the data from the memory unit holding this data preliminarily. However, this movement will inevitably incur much higher overhead, since it needs to access a remote SPM or the main memory. In this case, the data transfer overhead is composed of two major parts: reading the memory module of a remote SPM or main memory owning the data and writing the data to the target memory module.

Therefore, the memory access cost (either latency or energy) of a specific data block $B_i$ consists of the local access cost, the remote access cost, and the data move cost. It can be calculated as Equation (3.1).

$$C_{Mem}(B_i) = NL(B_i) \times CL(B_i) + NR(B_i) \times CR(B_i) + CM(B_i)$$ (3.1)

where $NL(B_i)$ and $NR(B_i)$ represent the number of local access and remote access to block $B_i$, respectively. $CL(B_i)$ and $CR(B_i)$ represent the cost of local access and remote access to block $B_i$, respectively. $CM(B_i)$ represents the data move cost for block $B_i$. According to different memory technologies, we have the following lemma.

**Lemma 3.1.** Power consumption of different kinds of memory is proportional to time lat-
tency. By minimizing memory access latency, energy consumption can be reduced at the same time. Therefore, the memory access cost in this thesis can be referred to as either latency or energy consumption.

The cost of processing a data block $B_i$, $C(B_i)$, generally involves two parts: computation cost and memory access cost.

\[
C(B_i) = C_{\text{Compu}}(B_i) + C_{\text{Mem}}(B_i)
\]

(3.2)

where $C_{\text{Compu}}(B_i)$ is the computation cost and $C_{\text{Mem}}(B_i)$ is the memory access cost. However, we only consider the memory part in this work for two reasons. First, the memory part is the bottleneck of the whole processing, since it accounts for the most time and energy overheads. Second, the computation cost of specific data block is usually constant or changes very little.

Therefore, the total allocation cost $C_{\text{total}}$ of a set of $N$ data blocks can be computed as following:

\[
C_{\text{total}} = \sum_{i=1}^{N} C(B_i) \approx \sum_{i=1}^{N} C_{\text{Mem}}(B_i)
\]

(3.3)

3.1.2 Allocation Granularity

The granularity of a data is critical to the SPM allocation problem. Usually, there are three types of basic granularity for a data: a variable, a block (a series of program code without instructions to jump into it except the entry or jump out of it except the exit), and a page. The advantages of different granularity vary with different programs. Generally, the finer the granularity of the data/code objects, the higher benefits can be probably achieved by SPM allocation. Most of work in the literature focused on the variables and blocks, since these two kinds of granularity are easier to partition and handle by inserting programming points. However, the biggest issue of variable-based and block-based allocation is the memory fragmentation incurred by their nonuniform sizes [86].
Moreover, the too fine granularity might introduce a large number of branch instructions which complicate code generation and make the implementation by the direct use of existing linker technology very hard. In addition, the finer the granularity, the harder profiling will be. Since almost virtually all of the current data allocation techniques for SPM inherently depends on profiling, specified by compiler or programmers. This is mainly because profiling can effectively help determine the usage frequency of each data which is critical to data allocation. Another reason for the popularity of profiling is that it only requires the re-use trends of variables profiled from programs to be similar with actual data, but they don’t have to be exactly the same.

While the page oriented data allocation can overcome the fragmentation problem due to the effectiveness of memory management unit (MMU), it suffers the locality problem. Our hybrid SPM architecture can enlarge the on-chip memory space with the benefits of high density of MRAM and Z-RAM. Therefore, the locality problem outweighs the fragmentation problem, and we use the data block as the basic allocation granularity. Here, we assume the data blocks of a program are partitioned from profile tools before execution and they can be mapped to every memory block with different latency and energy consumption.

### 3.2 Motivational Example

The objective of our algorithm is to minimize memory access latency, energy consumption, as well as the number of write operations to MRAM for CMP systems with the hybrid SPM consisting of SRAM, MRAM, and Z-RAM. In this section, we present an example to illustrate the rationale behind the proposed algorithm.

For demonstration purpose, we normalize latency and energy consumption of memory access to MRAM, SRAM, Z-RAM, and off-chip main memory as Table 3.1. In this table, the columns of “LS”, “RS”, “LM”, “RM”, “LZ”, “RZ”, and “MM” represent the memory access cost to local SRAM, remote SRAM, local MRAM, remote MRAM, local Z-RAM, remote Z-RAM, and off-chip DRAM, respectively. “La” and “En” represent latency and
Table 3.1: Latency and energy consumption for access to different memory modules. “LS”, “RS”, “LM”, “RM”, “LZ”, “RZ”, and “MM” represent local SRAM, remote SRAM, local MRAM, remote MRAM, local Z-RAM, remote Z-RAM, and off-chip DRAM, respectively. “La” and “En” represent latency and energy consumption, respectively.

<table>
<thead>
<tr>
<th>Op</th>
<th>LS</th>
<th>RS</th>
<th>LM</th>
<th>RM</th>
<th>LZ</th>
<th>RZ</th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>La</td>
<td>En</td>
<td>La</td>
<td>En</td>
<td>La</td>
<td>En</td>
<td>La</td>
</tr>
<tr>
<td>Read</td>
<td>1</td>
<td>0.1</td>
<td>2</td>
<td>0.18</td>
<td>5</td>
<td>0.36</td>
<td>3</td>
</tr>
<tr>
<td>Write</td>
<td>1</td>
<td>0.1</td>
<td>3</td>
<td>0.25</td>
<td>10</td>
<td>0.98</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 3.2: Latency of moving data between different memory modules.

<table>
<thead>
<tr>
<th>Type</th>
<th>SRAM</th>
<th>MRAM</th>
<th>ZRAM</th>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>3</td>
<td>12</td>
<td>7</td>
<td>62</td>
</tr>
<tr>
<td>MRAM</td>
<td>11</td>
<td>20</td>
<td>15</td>
<td>70</td>
</tr>
<tr>
<td>ZRAM</td>
<td>9</td>
<td>18</td>
<td>13</td>
<td>68</td>
</tr>
<tr>
<td>Main</td>
<td>61</td>
<td>70</td>
<td>65</td>
<td>0</td>
</tr>
</tbody>
</table>

energy consumption, respectively. During the execution of an application, a data can be allocated to any memory module and moved back and forth among all memory modules in SPMs.

Similar to the mechanism used in [80], we assume data moving latency and energy consumption between different memory modules are given in Table 3.2 and Table 3.3, respectively. In these two tables, the column of “Type” indicates different types of memory, and the other columns represent latency and energy consumption of data movement between different memory modules. For example, the column of “SRAM” represents the cost of moving data from other kinds of memory modules to SRAM.

We assume the target system has 2 cores, and each of them equips with hybrid SPM consisting of SRAM, MRAM, and Z-RAM. The off-chip shared memory is a DRAM. In order to demonstrate the viability of our data allocation strategy, we assume a simple program which has 18 data blocks obtained from a program, namely $B_1$, $B_2$, …, and $B_{18}$. Initially, only data block $B_{18}$ is stored in the core2’s SRAM, and all others blocks are stored
Table 3.3: Energy consumption of moving data between different memory modules.

<table>
<thead>
<tr>
<th>Type</th>
<th>SRAM</th>
<th>MRAM</th>
<th>ZRAM</th>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>0.28</td>
<td>1.16</td>
<td>0.62</td>
<td>6.38</td>
</tr>
<tr>
<td>MRAM</td>
<td>0.95</td>
<td>1.83</td>
<td>1.29</td>
<td>7.05</td>
</tr>
<tr>
<td>ZRAM</td>
<td>0.86</td>
<td>1.74</td>
<td>1.20</td>
<td>6.96</td>
</tr>
<tr>
<td>Main</td>
<td>6.30</td>
<td>7.18</td>
<td>6.64</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 3.4: The number of data accesses for each core. The column of “Data” refers to the 15 data blocks, the columns of “R” and “W” represent the number of reads and writes to the corresponding data block, respectively.

<table>
<thead>
<tr>
<th>Data</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>B6</th>
<th>B7</th>
<th>B8</th>
<th>B9</th>
<th>B10</th>
<th>B11</th>
<th>B12</th>
<th>B13</th>
<th>B14</th>
<th>B15</th>
<th>B16</th>
<th>B17</th>
<th>B18</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 R</td>
<td>18</td>
<td>17</td>
<td>14</td>
<td>10</td>
<td>14</td>
<td>10</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>17</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C1 W</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>8</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>C2 R</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>1</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>C2 W</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>9</td>
<td>5</td>
<td>7</td>
<td>6</td>
<td>18</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>18</td>
<td>0</td>
</tr>
</tbody>
</table>

in off-chip DRAM. In order to illustrate the example, we assume the number of accesses for each data by each core is given in Table 3.4. In this table, the column of “DATA” indicates the data blocks used in this example. The rows of “Read” and “Write” represent the number of reads and writes to each data block incurred by each core.

To illustrate the efficiency of our approach, we compare it with a greedy algorithm proposed in [56]. The basic idea of this algorithm is as follows: it greedily selects the most frequently accessed data and allocates it to a memory unit of the core that most frequently accesses the data. If all memory modules of this core cannot provide a room for the data, the data will be allocated to the SPM of the core that accesses it with the second most times. Due to the very high overhead of main memory access, this algorithm does not allocation any data to the off-chip DRAM, unless all on-chip SPMs are occupied. Although their target system has SPM, the SPM is configured by a pure SRAM.

The total memory access cost of a specific data involves local reads, local writes, remote
Table 3.5: The comparison of data allocation results for the greedy algorithm and the improved algorithm, when the size of each SRAM, MRAM, and Z-RAM is 200B, 400B, and 200B, respectively.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Core1</th>
<th></th>
<th>Core2</th>
<th></th>
<th>Main</th>
<th>Latency</th>
<th>Energy</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SRAM</td>
<td>MRAM</td>
<td>ZRAM</td>
<td>SRAM</td>
<td>MRAM</td>
<td>ZRAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy</td>
<td>$B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}$</td>
<td>$B_{16}, B_{17}$</td>
<td>$B_{3}, B_{4}, B_{5}$</td>
<td>$B_{6}, B_{7}$</td>
<td>$B_{8}, B_{9}$</td>
<td>6928</td>
<td>677.99</td>
<td>99</td>
</tr>
<tr>
<td>Improved</td>
<td>$B_{11}, B_{12}, B_{1}, B_{2}, B_{5}, B_{13}$</td>
<td>$B_{8}, B_{15}$</td>
<td>$B_{14}, B_{16}, B_{3}, B_{4}, B_{7}, B_{17}$</td>
<td>$B_{10}, B_{18}$</td>
<td>$B_{6}, B_{9}$</td>
<td>6071</td>
<td>588.4</td>
<td>45</td>
</tr>
<tr>
<td>Improvement</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>12.37% 13.21% 54.54%</td>
</tr>
</tbody>
</table>

reads, remote writes, and data movement between different memory units. Based on the initialization of data blocks, we can use Equation (3.1) to calculate the latency and energy consumption of each data block when it is allocated to the different memory modules of different cores. For example, if we allocate data $B_1$ to core1’s SRAM, according to Table 3.1, the memory access latency of it can be calculated as: $18 \times 1 + 1 \times 1 + 0 \times 2 + 0 \times 3 + 61 = 80$. The memory access latency of allocating block $B_{18}$ to core2’s SRAM can be computed as: $1 \times 1 + 18 \times 1 + 1 \times 2 + 18 \times 3 + 0 = 75$.

In this example, for simplicity, we first assume that each core has 800B on-chip SPM space, including a 200B SRAM, a 400B MRAM, and a 200B Z-RAM. We also assume that each data block is 100B, which means a MRAM can accommodate 4 data blocks, while each SRAM and each Z-RAM can only provide rooms for 2 data blocks. By using the greedy algorithm, one possible solution allocates data blocks as shown in Table 3.5. The total latency and energy consumption are 6928 and 677.99, respectively. Meanwhile, this allocation needs 99 writes to MRAMs.

However, an improved algorithm can reduce the number of writes to MRAMs significantly, along with the reduction in latency and energy consumption. The allocation result of the improved algorithm is shown in the "Improved" row of Table 3.5. By applying this algorithm, the total latency, energy consumption, and the number of writes to MRAMs are 6071, 588.4, and 45, respectively. Compared to the greedy algorithm, the improved strategy can reduce the total latency by 12.37%, energy consumption by 13.21%, and the
Table 3.6: The comparison of data allocation results for the greedy algorithm and the improved algorithm, when the size of each SRAM, MRAM, and Z-RAM is 200B, 400B, and 400B, respectively.

<table>
<thead>
<tr>
<th>Methods</th>
<th>Core1</th>
<th>Core2</th>
<th>Main</th>
<th>Latency</th>
<th>Energy</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SRAM</td>
<td>MRAM</td>
<td>ZRAM</td>
<td>SRAM</td>
<td>MRAM</td>
<td>ZRAM</td>
</tr>
<tr>
<td>Greedy</td>
<td>$B_{10}, B_{11}$</td>
<td>$B_{12}, B_{13}, B_{14}, B_{15}$</td>
<td>$B_{1}, B_{2}, B_{3}, B_{17}$</td>
<td>$B_{4}, B_{5}, B_{6}, B_{7}$</td>
<td>$B_{8}, B_{9}$</td>
<td>5005</td>
</tr>
<tr>
<td>Improved</td>
<td>$B_{6}, B_{7}$</td>
<td>$B_{1}, B_{2}, B_{8}$</td>
<td>$B_{3}, B_{12}, B_{14}$</td>
<td>$B_{3}, B_{4}, B_{15}$</td>
<td>$B_{10}, B_{16}, B_{17}, B_{18}$</td>
<td>4199</td>
</tr>
<tr>
<td>Improvement</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

With the consideration of the high density property of Z-RAM, we can easily use an even larger one to enlarge the size of an on-chip SPM. For example, if we use a 1000B SPM which is composed of a 200B SRAM, a 400B MRAM, and a 400B Z-RAM. One possible allocation for the greedy algorithm and an improved algorithm are shown in Table 3.6. In this table, we can see that the total memory access latency is reduced by 16.10%, energy consumption is reduce by 17.11%, and the number of writes to MRAMs is reduced by 69.16%.

From the above example, we can see that the data allocation scheme is of significance to the whole memory access performance and durability of a memory hierarchy. The "improved" algorithm illustrated in the example is intrinsically the MDPDA algorithm that we will discuss in the next section.

### 3.3 Algorithms

In this section, we present our MDPDA algorithm in detail. We will first build an allocation cost table. Then, with the help of this table, we give the procedures of MDPDA algorithm. In order to illustrate the employed dynamic programming algorithm, we will exhibit a simple example.
3.3.1 Allocation Cost Table

Assume there are $N$ data blocks need to be allocated to a system with $P$ cores. Each core has a proposed hybrid SPM configured from a SRAM, a MARM, and a Z-RAM. In order to calculate latency and energy consumption for each data conveniently, we build an allocation cost table to represent the cost of allocating each data block to different memory modules, as shown in Table 3.7. In this table, we compute the 18 data blocks given in Section 3.2, with the assumption that the target CMP system is a dual-core platform with the proposed hybrid SPM memory. The column of "Data" represent the involved data introduced in the motivational example. The columns of "Core1" and "Core2" represent the 2 cores. "SRAM", "MRAM", and "ZRAM" indicate the SRAM, MMRAM, and ZRAM of the corresponding SPM. The columns "La" and "En" indicate the latency and energy consumption of allocating each data to each memory module.

We use a function $Map(b_i, x)$ to represent the cost (either latency or energy consumption) of mapping data block $b_i$ to memory module $x$, and the value of the function can be read from the allocation cost table directly. Let $C_{ij}$ represent the memory $j$ of the SPM in core $i$, where $\forall i, j, \{i, j| i < P, j \in \{MM, S, M, Z\}\}, P$ is the number of cores, MM, S, M, Z are short for the main memory, SRAM, MMRAM, and Z-RAM, respectively. For example, $Map(B_1, C_{1S}) = 80$ indicates the latency of mapping data block $B_1$ to Core1’s SRAM is 80 units. $Map(B_{18}, C_{2M}) = 75$ indicates the latency of allocating block $B_{18}$ to Core2’s MRAM is 75 units. It is much lower than the latency of allocating other blocks to Core2’s MRAM, because it is originally stored in there. Therefore, there is no data moving latency and energy consumption to allocate block $B_{18}$ to Core2’s MRAM. Since the latency cost is proportional to energy consumption, the reduction of latency will also contribute to reduction in energy consumption. Therefore, for simplicity, we just use latency for demonstration.
Table 3.7: Allocation cost table. Assume the target CMP system is a dual-core device, where each core is coupled with a proposed hybrid on-chip SPM. The columns of “La” and “En” represent latency and energy consumption for allocating a data to a corresponding memory module.

<table>
<thead>
<tr>
<th>Data</th>
<th>Core1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Core2</th>
<th></th>
<th></th>
<th></th>
<th>Main</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SRAM</td>
<td>MRAM</td>
<td>ZRAM</td>
<td>SRAM</td>
<td>MRAM</td>
<td>ZRAM</td>
<td>SRAM</td>
<td>MRAM</td>
<td>ZRAM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1</td>
<td>80</td>
<td>8.2</td>
<td>170</td>
<td>14.64</td>
<td>124</td>
<td>13.2</td>
<td>100</td>
<td>9.79</td>
<td>270</td>
<td>24.58</td>
<td>221</td>
</tr>
<tr>
<td>B2</td>
<td>80</td>
<td>8.2</td>
<td>175</td>
<td>15.26</td>
<td>126</td>
<td>13.3</td>
<td>101</td>
<td>9.86</td>
<td>280</td>
<td>25.83</td>
<td>225</td>
</tr>
<tr>
<td>B3</td>
<td>88</td>
<td>8.81</td>
<td>220</td>
<td>20.22</td>
<td>159</td>
<td>16.16</td>
<td>94</td>
<td>9.32</td>
<td>250</td>
<td>22.74</td>
<td>198</td>
</tr>
<tr>
<td>B4</td>
<td>85</td>
<td>8.6</td>
<td>210</td>
<td>18.95</td>
<td>155</td>
<td>15.6</td>
<td>98</td>
<td>9.6</td>
<td>275</td>
<td>25.88</td>
<td>208</td>
</tr>
<tr>
<td>B5</td>
<td>88</td>
<td>8.2</td>
<td>190</td>
<td>17.12</td>
<td>132</td>
<td>13.6</td>
<td>104</td>
<td>10.07</td>
<td>310</td>
<td>29.58</td>
<td>237</td>
</tr>
<tr>
<td>B6</td>
<td>83</td>
<td>8.44</td>
<td>210</td>
<td>19.21</td>
<td>149</td>
<td>14.96</td>
<td>102</td>
<td>9.9</td>
<td>305</td>
<td>29.36</td>
<td>226</td>
</tr>
<tr>
<td>B7</td>
<td>86</td>
<td>8.65</td>
<td>230</td>
<td>21.72</td>
<td>157</td>
<td>15.72</td>
<td>100</td>
<td>9.76</td>
<td>300</td>
<td>28.72</td>
<td>224</td>
</tr>
<tr>
<td>B8</td>
<td>80</td>
<td>8.2</td>
<td>205</td>
<td>18.98</td>
<td>138</td>
<td>13.9</td>
<td>107</td>
<td>10.28</td>
<td>340</td>
<td>33.33</td>
<td>249</td>
</tr>
<tr>
<td>B9</td>
<td>84</td>
<td>8.5</td>
<td>230</td>
<td>21.84</td>
<td>154</td>
<td>15.28</td>
<td>104</td>
<td>10.05</td>
<td>330</td>
<td>32.34</td>
<td>239</td>
</tr>
<tr>
<td>B10</td>
<td>128</td>
<td>12.35</td>
<td>490</td>
<td>46.74</td>
<td>329</td>
<td>31.56</td>
<td>128</td>
<td>12.36</td>
<td>490</td>
<td>46.6</td>
<td>332</td>
</tr>
<tr>
<td>B11</td>
<td>117</td>
<td>11.49</td>
<td>430</td>
<td>40.3</td>
<td>286</td>
<td>27.72</td>
<td>135</td>
<td>12.87</td>
<td>520</td>
<td>49.96</td>
<td>358</td>
</tr>
<tr>
<td>B12</td>
<td>125</td>
<td>12.11</td>
<td>485</td>
<td>46.36</td>
<td>326</td>
<td>30.98</td>
<td>130</td>
<td>12.46</td>
<td>510</td>
<td>49.51</td>
<td>336</td>
</tr>
<tr>
<td>B13</td>
<td>124</td>
<td>12.04</td>
<td>480</td>
<td>45.73</td>
<td>324</td>
<td>30.76</td>
<td>131</td>
<td>12.53</td>
<td>515</td>
<td>50.14</td>
<td>338</td>
</tr>
<tr>
<td>B14</td>
<td>136</td>
<td>12.88</td>
<td>605</td>
<td>61.35</td>
<td>374</td>
<td>34.7</td>
<td>132</td>
<td>12.6</td>
<td>585</td>
<td>58.83</td>
<td>366</td>
</tr>
<tr>
<td>B15</td>
<td>114</td>
<td>11.3</td>
<td>410</td>
<td>37.51</td>
<td>284</td>
<td>27.36</td>
<td>137</td>
<td>12.99</td>
<td>525</td>
<td>50.88</td>
<td>354</td>
</tr>
<tr>
<td>B16</td>
<td>121</td>
<td>11.83</td>
<td>465</td>
<td>43.84</td>
<td>318</td>
<td>30.1</td>
<td>134</td>
<td>12.74</td>
<td>530</td>
<td>52.03</td>
<td>344</td>
</tr>
<tr>
<td>B17</td>
<td>124</td>
<td>12.06</td>
<td>405</td>
<td>36.15</td>
<td>300</td>
<td>29.66</td>
<td>116</td>
<td>11.46</td>
<td>365</td>
<td>31.67</td>
<td>272</td>
</tr>
<tr>
<td>B18</td>
<td>78</td>
<td>6.86</td>
<td>567</td>
<td>57.81</td>
<td>324</td>
<td>29.08</td>
<td>75</td>
<td>6.58</td>
<td>567</td>
<td>57.81</td>
<td>324</td>
</tr>
</tbody>
</table>

3.3.2 Recursive Formulation

The most critical part of a dynamic programming algorithm is the construction of the recursive formulation which breaks down the target problems. First, we define a memory allocation function $\text{AllocMem}(n, x)$, which represents the total cost of the first $n - 1$ blocks when the $n$th block is allocated to memory $x$. Then, we define a total cost function $f(n, x)$ to represent the total allocation cost of the first $n$ data blocks when the $n$th block is allocated to memory $x$. For example, $f(4, C1_S)$ indicates the total allocation cost of the first 4 data blocks when block 4 is allocated to Core1’s SRAM. We define a multi-dimensional matrix, $\vec{AllC}$, to store the total cost for data allocation. The dimension of $\vec{AllC}$ is $N \times \text{size}(C1_S) \times \text{size}(C1_M) \times \text{size}(C1_Z) \times \cdots \times \text{size}(CP_Z)$, where $P$
is the number of input data blocks and size($x$) is the size of the memory $x$. For example, $AllC[4, 1, 2, 1, \ldots, 1]$ indicates the total cost for allocating the first 4 data blocks to on-chip hybrid SPMs, when the available space of the Core1’s SRAM, Core1’s MRAM, Core1’s Z-RAM, $\ldots$, and Core P’s Z-RAM is 1, 2, 1, $\ldots$, 1, respectively. Then, we can compute the total allocation cost by allocating block $i$ to different memory modules as Equation (3.5).

$$
\begin{align*}
\text{AllocMem}(b_i, MM) &= AllC[b_i - 1, s_1, m_1, z_1, \ldots, z_n] \\
\text{AllocMem}(b_i, C1_S) &= AllC[b_i - 1, s_1 - 1, m_1, z_1, \ldots, z_n] \\
\text{AllocMem}(b_i, C1_M) &= AllC[b_i - 1, s_1, m_1 - 1, z_1, \ldots, z_n] \\
\text{AllocMem}(b_i, C1_Z) &= AllC[b_i - 1, s_1, m_1, z_1 - 1, \ldots, z_n] \\
\vdots \\
\text{AllocMem}(b_i, CP_Z) &= AllC[b_i - 1, s_1, m_1, z_1, \ldots, z_n - 1]
\end{align*}
$$

(3.4)

$$
\begin{align*}
f(b_i, MM) &= \text{AllocMem}(b_i, MM) + \text{Map}(b_i, MM) \\
f(b_i, C1_S) &= \text{AllocMem}(b_i, C1_S) + \text{Map}(b_i, C1_S) \\
f(b_i, C1_M) &= \text{AllocMem}(b_i, C1_M) + \text{Map}(b_i, C1_M) \\
f(b_i, C1_Z) &= \text{AllocMem}(b_i, C1_Z) + \text{Map}(b_i, C1_Z) \\
\vdots \\
f(b_i, CP_Z) &= \text{AllocMem}(b_i, CP_Z) + \text{Map}(b_i, CP_Z)
\end{align*}
$$

(3.5)

$$
AllC[b_i, s_1, m_1, z_1, \ldots, z_n] = \min(f(b_i, MM), f(b_i, C1_S), f(b_i, C1_M), \ldots, f(b_i, CP_Z))
$$

(3.6)

Equation (3.5) shows that the minimum allocation cost is always preserved, since the total allocation cost by adding the current data block is always selected from the best allocation scheme of the previous blocks and the current one. Equation (3.4) to Equation (3.6) jointly exhibit the recursive formulation to derive the minimum total allocation cost.
for the target problem. In this equation, $AllC[bi, s_1, m_1, \ldots, z_n]$ records the minimum allocation cost when the available memory block for each of on-chip memory module are $s_1, m_1, \ldots, z_n$, respectively. Initially, if all memory blocks in SPMs (including SRAM, MRAM, and Z-RAM) are unavailable (which means $s_1 = m_1 = \cdots = z_n = 0$), then all the blocks will be assigned to the shared off-chip main memory. The allocation of a specific block is always determined by the optimal allocation of the previous data block. For example, the latency for assigning block $B_1$ to the main memory is $Map(B_1, MM) = 1140$. If there are no available on-chip memory to accommodate block $B_2$, the total allocation latency for blocks $B_1$ and $B_2$ is $Map(B_1, MM) + Map(B_2, MM) = 1140 + 1140 = 2280$.

For any other item in the matrix, the total cost is determined by both the allocation of the previous blocks and the cost of allocating this block to different memory modules. There are totally $3 \times P + 1$ choices to assign a data block, where $P$ is the number of cores in the target CMP system. The dynamic programming algorithm always selects the combination that can achieve the minimum total cost for all present data blocks.

**Theorem 3.1.** Every element in the total cost matrix $AllC[bi, s_1, m_1, z_1, \ldots, z_n]$ obtained by the recursive function is the minimum total allocation cost for data block $b_i$ when the available space of Core1’s SRAM, Core1’s MRAM, Core1’s Z-RAM, \ldots, and Core P’s Z-RAM is $s_1, m_1, z_1, \ldots, z_n$, respectively.

**Proof.** It can be proved by induction as follows.

**Basis:** When $b_i = 1$, there is only one data block. If $s_1 = m_1 = z_1 = \cdots = z_n = 0$, since there is no on-chip memory space available, the block will be assigned to the main memory. The total allocation cost is $Map(b_1, MM)$ in this case. Otherwise, the total allocation cost is $min(Map(b_1, MM), Map(b_1, x_1), \ldots, Map(b_1, x_k))$, where $x_1, \ldots, x_k$ represent the memory modules that have memory space to store data block $b_1$. Then, we can always get the minimum allocation cost for data block $b_1$. Therefore, Theorem 3.1 holds for $b_i = 1$.

**Inductive step:** we show that for $\forall i > 1$, if $AllC[bi, s_1, m_1, z_1, \ldots, z_n]$ is the minimum total
allocation cost for block $b_1$ to block $b_i$, then $AllC[b_i + 1, s_1, m_1, z_1, \ldots, z_n]$ is the minimum total allocation cost by adding data block $b_{i+1}$, when the available on-chip memory resources of Core1’s SRAM, Core1’s MRAM, Core1’s Z-RAM, \ldots, and Core P’s Z-RAM are $s_1, m_1, z_1, \ldots, z_n$, respectively.

In Equation (3.5), all the allocation schemes of data block $b_{i+1}$ are searched and their results are preserved in the total allocation cost function $f(n, x)$. Since the minimum total allocation cost for block $b_1$ to block $b_i$ are obtained from previous step, Equation (3.6) get the minimum cost by adding block $b_{i+1}$ from all possible allocation schemes. It has now been proved by mathematical induction that Theorem 3.1 holds for all data blocks.

### 3.3.3 MDPDA Algorithm

According to the built recursive formulations, we describe the *Multi-dimensional Dynamic Programming Data Allocation* (MPPDA) algorithm in Algorithm 3.1. The input of the MDPDA algorithm is $N$ data blocks obtained by profiling tools, the constructed allocation cost table, and the total cost table. The output of the algorithm is the minimum total cost (latency or energy consumption) for the $N$ data blocks.

We initialize the algorithm in Line 1 to Line 3. When there is no on-chip memory space available, we have to assign all the data blocks to the shared off-chip main memory, which is the worst case of the algorithm. In this case, the total cost of the first $b_i$ tasks is the summation of the cost of allocating them to the main memory. In order to compute the total cost validly, we add a boundary for the matrix from Line 4 to Line 14. Line 15 to Line 26 are used to recursively compute the total cost from the first data block to the last one, according to the formulations in Equation (3.5) and Equation (3.6). For the target system with $P$ cores, there are $3 \times P + 1$ layers of loops. The first loop specifies a data block in consideration, the second loop to the $(3P + 1)$th loop are employed to determine the best allocation for the first data block to the current data block. We only give several loops because of space limitations. In Algorithm 3.2, we backtrack the path that is able to
Algorithm 3.1 Multi-dimensional Dynamic Programming for Data Allocation (MDPDA).

Require: Allocation cost table $C$, total cost table $AllC[d, s_1, \ldots, m_n, z_n]$. 
Ensure: The minimum total allocation cost.

1: for $i \leftarrow 1$ to $N$ do
2: $AllC[i, 0, \ldots, 0, 0] = \sum_{j=1}^{i} \text{Map}(d_j, MM)$; /*When there is no on-chip memory available, all data blocks are allocated to shared main memory*/
3: end for
4: for $s_1 \leftarrow 0$ to $\text{size}(C_{1S})$ do
5: /*$\text{size}(C_{1S})$ represents the size of core1’s SRAM*/
6: for $m_1 \leftarrow 0$ to $\text{Size}(C_{1M})$ do
7: /*$\text{Size}(C_{1M})$ represents the size of core1’s MRAM*/
8: . . .
9: for $z_n \leftarrow 0$ to $\text{size}(CP_Z)$ do
10: /*$\text{size}(CP_Z)$ represents the size of core P’s Z-RAM*/
11: $AllC[0, s_1, m_1, \ldots, z_n] = 0$; /*Add boundaries for the matrix*/
12: end for
13: end for
14: end for
15: for $bi \leftarrow 1$ to $N$ do
16: for $s_1 \leftarrow 0$ to $\text{size}(C_{1S})$ do
17: for $m_1 \leftarrow 0$ to $\text{Size}(C_{1M})$ do
18: . . .
19: for $z_n \leftarrow 0$ to $\text{size}(CP_Z)$ do
20: Apply Equation (3.4) to get minimum memory allocation cost for the first $b_i - 1$ data blocks when $b_i$ is allocated to different memory modules.
21: Apply Equation (3.5) to calculate the cost of allocating block $b_i$ to different modules.
22: Apply Equation (3.6) to get the minimum total allocation cost for block $b_1$ to block $b_i$.
23: end for
24: end for
25: end for
26: end for
27: /*Backtrack to get the data allocation*/
28: Backtrack($AllC$);
29: return $AllC[N, \text{size}(C_{1S}), \text{Size}(C_{1M}), \ldots, \text{size}(CP_Z)]$:
derive the minimum total cost. Since there are \( N \) data blocks, we need to perform \( N \) traces to determine the allocation for all blocks.

**Algorithm 3.2** Backtrack the allocation total cost table and find out the data to memory module mapping.

**Require:** total cost table \( AllC[d, s_1, \ldots, m_n, z_n] \), total cost function.

**Ensure:** Data to memory mapping results.

```
1: \( bi \leftarrow N \);
2: while \( bi > 0 \) do
3: \( \text{min} \leftarrow \text{AllC}[bi, s_1, m_1, \ldots, z_n]; \)
4: if \( \text{min} = f(b_i, MM) \) then
5: \( bi \rightarrow MM; \) /*Allocate block \( bi \) to main memory*/
6: end if
7: if \( \text{min} = f(b_i, C_1S) \) then
8: \( bi \rightarrow C_1S; \) /*Allocate block \( bi \) to core1’s SRAM*/
9: \( s_1 \leftarrow s_1 - 1; \)
10: end if
11: if \( \text{min} = f(b_i, C_1M) \) then
12: \( bi \rightarrow C_1M; \) /*Allocate block \( bi \) to core1’s MRAM*/
13: \( m_1 \leftarrow m_1 - 1; \)
14: end if
15: \ldots
16: if \( \text{min} = f(b_i, CP_Z) \) then
17: \( bi \rightarrow CP_Z; \) /*Allocate block \( bi \) to core P’s Z-RAM*/
18: \( z_n \leftarrow z_n - 1; \)
19: end if
20: \( bi \leftarrow bi - 1; \)
21: end while
```

We employ a simple example to demonstrate the whole executing processes of the MDPDA algorithm. For simplicity, we only consider the blocks \( B_1, B_2, B_3, B_4, B_5, \) and \( B_6 \) introduced the motivational example. We assume the target CMP system is a single core system with a hybrid on-chip SPM consisting of a 100B SRAM, a 200B MRAM, and a 200B Z-RAM. We also assume the size of each data block is 100B. The allocation cost table is presented as Table [3.8]. In this case, the dimension of the total allocation cost matrix, \( AllC \), is \( 6 \times 1 \times 2 \times 2 \). Figure [3.2] illustrates the data allocation procedure for the targeted problem with 6 blocks and 3 memory modules. In this figure, \( s, m, \) and \( z \) represent the available number of memory blocks in SRAM, MRAM, and Z-RAM, respectively. \( b \)
Table 3.8: Latency and energy consumption of allocating the first 6 of the 18 datum to corresponding memory modules when only core 1 is used.

<table>
<thead>
<tr>
<th>Data</th>
<th>SRAM</th>
<th>MRAM</th>
<th>ZRAM</th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>La</td>
<td>En</td>
<td>La</td>
<td>En</td>
</tr>
<tr>
<td>$B_1$</td>
<td>80</td>
<td>8.2</td>
<td>170</td>
<td>14.64</td>
</tr>
<tr>
<td>$B_2$</td>
<td>80</td>
<td>8.2</td>
<td>175</td>
<td>15.26</td>
</tr>
<tr>
<td>$B_3$</td>
<td>75</td>
<td>7.7</td>
<td>140</td>
<td>12.22</td>
</tr>
<tr>
<td>$B_4$</td>
<td>75</td>
<td>7.7</td>
<td>160</td>
<td>14.7</td>
</tr>
<tr>
<td>$B_5$</td>
<td>80</td>
<td>8.2</td>
<td>190</td>
<td>17.12</td>
</tr>
<tr>
<td>$B_6$</td>
<td>77</td>
<td>7.9</td>
<td>180</td>
<td>16.66</td>
</tr>
</tbody>
</table>

represents the considering data block (Note the data before it has already been assigned).

“$m = x$” means the available remaining on-chip MRAM space is $x$. Similarly, “$s = y$” indicates that the available remaining on-chip SRAM space is “$y$”. For example, the value 296 in the cell where $b = B_2$, $s = 0$, $m = 1$, $z = 1$ indicates that the energy consumption of allocating data blocks $B_1$ and $B_2$ is 296 when the available memory space for SRAM, MRAM, and ZRAM is 0, 1, and 1, respectively. The allocation result is by assigning $B_1$ to MRAM and $B_2$ to ZRAM.

According to the recursive formulation and the MDPDA algorithm, we can determine that the minimum allocation cost (latency) for these 6 data blocks is 1481. The solution for the allocation is to assign $B_1$ and $B_3$ to MRAM, $B_4$ to main memory, $B_5$ to SRAM, and $B_2$ and $B_6$ to Z-RAM. With this allocation scheme, the total energy consumption is 147.84. As shown in Figure 3.2, the solid lines represent the data to memory module mapping, while the dash lines represent the unselected candidates. It can be verified that both latency and energy consumption are optimal for this set of data blocks with the assumed parameters.

**Time complexity:** We can see that the time complexity of this algorithm is determined by the recursive part, which is $O(N \times \text{size}(C1_S) \times \text{size}(C1_M) \times \text{size}(C1_Z) \times \cdots \times \text{size}(C1_N))$. Due to the limited on-chip memory space of the CMP system, the size of each hybrid memory is generally small. Assuming the size for each memory is $K$, for the
system with $P$ cores, the time complexity approximates $O(N \times K^{3P})$. Since $K$ and $P$ are constant for the given architecture, the algorithm can be solved in polynomial time.
Chapter 4  

Genetic Solution to the Data Allocation Problem

In this chapter, we will focus on the design of an adaptive genetic algorithm for the data allocation problem. The organization of this chapter is as follows: Section 4.1 gives the fundamental motivation for the development of this algorithm. Section 4.2 presents the basic system model and chromosome model used to describe the genetic algorithm. Section 4.3 introduces the adaptive genetic algorithm in detail.

4.1 Motivation

From the multi-dimensional dynamic programming algorithm in the last chapter, we can see that it needs to maintain a multi-dimensional “total allocation cost” matrix. For N core systems, the dimensional of the matrix will be $3 \times N + 1$ because these are three different memory technologies. In this case, while this algorithm is still very efficient both in time and space for small applications, the space overhead will be a critical problem for the memory limited embedded systems when the data block increase quickly in large applications. Therefore, how to design an efficient data allocation algorithm to reasonably utilize the memory space of embedded systems is rising as an important problem.

Awareness of this issue, we will design a genetic algorithm in this chapter for the data allocation problem in consideration, since it is able to yield near-optimal solutions with moderate time and space overhead. Genetic Algorithms (GAs), stemmed from the evolutionary theory, are a class of computational models which is able to achieve sub-optimal solutions for problems. These algorithms organize a solution candidate of a problem in a specific data structures (often referred to as chromosome), such as linear binary, tree,
linked list, and even matrix, and apply some operations on these structure to produce new candidates by preserving good features [87]. To achieve it, our well-developed genetic algorithm will inherit the prominent merits of traditional ones, such as accurate solutions and fast convergence. In general, a genetic algorithm always involves the following basic elements: chromosome, initialization, selection, reproduction, and termination. Targeting the data allocation problem for the heterogeneous on-chip SPM memory with SRAM, MRAM, and Z-RAM, we develop corresponding algorithms for these 4 stages.

4.2 System Model

4.2.1 Hardware model

The hardware model used for our genetic algorithm is the same as the one used for the multidimensional programming algorithm, as shown in Figure 3.1. Therefore, the memory access energy requirement and latency will be also the same.

4.2.2 Chromosome Model

A chromosome for the data allocation problem is a set of defined parameters which is able to represent a solution. The parameters here are the data blocks and the size of each memory module including all on-chip memory modules and the off-chip main memory. Therefore, we define a gene in a chromosome as a pair of these two parameters. That is, a chromosome represents an allocation scheme. There are numerous ways to represent a chromosome. Intuitively, we can use a matrix to represent a chromosome, where the rows indicate the main memory and all on-chip memory units of a SPM in each processor core. The columns indicate data allocation on the corresponding memories. For example, Figure 4.1 shows two randomly generated chromosomes, A and B. These two chromosomes are constructed in matrix structure according to the size of each memory unit, where $C_1S$, $C_1M$, $C_1Z$, $C_2S$, $C_2M$, $C_2Z$, and $MM$ represent SPM1’s SRAM, SPM1’s MRAM, SPM1’s ZRAM, SPM2’s SRAM, SPM2’s MRAM, SPM2’s ZRAM, and the main memory,
respectively. Each row of data given in the chromosome matrix is a gene sequence, which represents the data allocation on the corresponding memory module.

However, this form of chromosome is inconvenient to perform genetic operations, particularly for crossover, because it is hard to maintain the space constraint of each memory module. Hence, we modify the chromosome and organize it as a list structure where each gene in the list is defined to be a data item and a memory unit pair: \((d, MT)\). Each gene cell shows that the data item \(d\) is allocated to the memory unit \(MT\). In this method, all the memory units are numbered uniquely. Suppose that the target CMP system has \(N\) cores, where each core has an on-chip heterogeneous memory configured from MRAM and SRAM, we need at most \(2 \times N + 1\) numbers to label these memory units. For the purpose of simplicity, we use number \(3 \times i - 2\), \(3 \times i - 1\), and \(3 \times i\) \((1 \leq i \leq N)\) to represent the SRAM, MRAM, and ZRAM of the SPM associated with core \(i\), respectively. Number \(3 \times N + 1\) represents the main memory. Two chromosomes in this structure are shown in Figure 4.2 and they are transformed from the chromosomes A and B in Figure 4.1, respectively. In Figure 4.2, we use 1, 2, 3, 4, 5, 6, and 7 to correspondingly represent SPM1’s SRAM, SPM1’s MRAM, SPM1’s ZRAM, SPM2’s SRAM, SPM2’s MRAM, SPM2’s ZRAM, and the main memory. For example, the gene \((B_1, 4)\) represents data \(B_1\) is allocated to SPM2’s SRAM.

### 4.3 Description of the Adaptive Genetic Algorithm

In this section, we will discuss the details of the adaptive genetic algorithm. Typically, a genetic algorithm involves three major steps: initialization, evaluation of fitness function, and genetic operations. First, we formally define the problem of data allocation in a CMP system.

#### 4.3.1 Problem Statement

The cost optimization problem of memory access incurred by data allocation in a CMP with \(P\) processors (each of these processors is integrated with a SPM which consists of a SRAM
and a MRAM) can be defined as: Given the number of data \( N \), the initial data allocation on the on-chip memory units of all processor cores and the off-chip main memory, the capacity of each core’s SRAM and MRAM, the number of cores \( P \), the number of reading and writing references to each data of each core, the cost of each memory unit access, and the cost of moving data between different memory units, how to allocate each data to the hybrid memory units of each core so that the total memory access cost can be minimized and the write activities on MRAMs can be reduced? In this problem, we assume each core can access the off-chip main memory, the SRAM and MRAM in its local SPM, and every
remote SPM with different cost. The cost of access to each memory unit is given in Table 3.1.

The objective function of the target problem is described as: given the number of local reads $N_{LR}$, local writes $N_{LW}$, remote reads $N_{RR}$, remote writes $N_{RW}$, the cost of local read $C_{LR}$, local write $C_{LW}$, remote read $C_{RR}$, remote write $C_{RW}$, and the cost of data movement $C_{Move}$ exhibited in Table 3.2 and Table 3.3, the cost of memory access ($CM$) for a specific data can be formulated as Equation (4.1).

$$CM = N_{LR} \times C_{LR} + N_{LW} \times C_{LW} + N_{RR} \times C_{RR}$$

$$+ N_{RW} \times C_{RW} + C_{Move}$$

(4.1)

4.3.2 Initialization

The population size $PopSize(PS)$ usually depends on the proposed problem and is determined experimentally [88]. To accelerate the process of data allocation and the implementation of genetic operations, we will use the greedy algorithm in [56] to generate the initial population. A whole population will be generated from these initial individuals by randomly swapping the memory positions of genes.

4.3.3 Fitness Function

In general genetic algorithms, the fitness function is typically obtained from the objective function that needs to be optimized. The fitness of an individual $u$ is regarded to be better than the fitness of another individual $v$ if the solution corresponding to $u$ is closer to an optimal solution than $v$. According to Darwin’s principle of survival of the fittest, the individual with a greater fitness value will have higher likelihood to survive in the next generation than the counterpart with a lower fitness value. We define the fitness function as Equation (4.2).

$$FT(i) = M - \text{Total Cost}(i);$$

(4.2)
where $M$ represents maximum total cost have observed by this generation and $FT(i)$ represents the fitness value of chromosome $i$. $Total\_Cost(i)$ is the total cost of memory access to the chromosome $i$. Essentially, it equals to the total memory access cost of each gene (data) in this chromosome. We calculate the total cost by using Equation (4.3).

$$Total\_Cost(i) = \sum_{j=1}^{N} CM(j), \text{ for chromosome } i;$$ (4.3)

where $N$ is the number of data items and $CM(j)$ is the memory access cost of data $j$ that is defined as Equation (4.1).

### 4.3.4 GA Operations

Generally, the genetic operations include selection, crossover, and mutation. We describe each of them as follows.

1) **Selection.**

The selection process is carried out to form a new population, through strategically choosing some chromosomes from the old population with respect to the fitness value of each individual. It is utilized to enhance the overall quality of the population. Based on the natural selection rule, many methods are exploited to select the fittest chromosomes, such as roulette wheel selection, Boltzman selection, rank selection, and elitism, etc. In our genetic algorithm, we will use a *rank based roulette wheel selection scheme* with elitism to select chromosomes. In this method, an imaginary wheel with total 360 degrees is applied, on which all chromosomes in the population are placed, and each of them occupied a slot size according to the value of the corresponding fitness function.

Let $PS$ denote the population size and $A_i$ represent the angle of the sector occupied by the $i^{th}$ ranked chromosome. The chromosome-to-sector mapping is consistent to the fitness of each chromosome, and the $1st$ ranked chromosome has the highest fitness value, therefore allocating to the sector 1 with the largest angle $A_1$. The $(PS)th$ ranked chromosome has the lowest fitness value and is allocated to the sector $PS - 1$ with smallest angle.
Equation 4.4 to Equation 4.6 hold for the angles. Therefore, the fitter an individual is, the more area of it will be assigned on the wheel, and thus the more possible that it will be selected when the biased roulette wheel is spun. The algorithm to implement it is shown as Algorithm 4.3.

\[
\rho = \frac{A_i}{A_{i+1}} \tag{4.4}
\]

\[
A_1 = \frac{1 - \rho}{1 - \rho^{PS}} \tag{4.5}
\]

\[
A_i = \frac{(1 - \rho)}{1 - \rho^{PS}} \times \rho^{i-1} \tag{4.6}
\]

where \( A_i < 1, \rho < 1, \) and \( 0 \leq i < PS \).

**Algorithm 4.3 Algorithm for Genetic Selection**

**Require:** An old population \( Old_{Pop} \) and the size of the population \( PS \).

**Ensure:** A selected chromosome \( k \).

1: Define the total fitness \( SumFit \) as the sum of fitness values of all individuals in the current population;
2: for \( i = 1 \rightarrow PS \) do
3: \( SumFit = SumFit + Old_{Pop}(i).FT; \)
4: end for
5: Generate a random number \( RanN \) between 1 to \( SumFit; \)
6: for \( k = 1 \rightarrow PS \) do
7: \( \text{if } \sum_{i=1}^{k} Old_{Pop}(i).FT \geq RanN \) then
8: \( \text{break;} \)
9: end if
10: end for
11: return chromosome \( k; \)

2) Crossover.

Crossover is a crucial step after selection. Generally, it is employed to more broadly explore the search space. We can find the individual with higher fitness function with this operation. Conventionally, crossover operation includes signal point crossover, two point crossover, and uniform crossover. The rationale is that the “good” characteristics of the parents should
be well preserved and passed down to children. However, the rational selection may lead to the local optimal problem. To avoid this problem, the crossover operations are carried out with a specific probability, which is often referred to as *crossover rate*, denoted by $PC$. We randomly select pairs of chromosomes as parents to generate new individuals. In this section, we will use an *adaptive cycle crossover strategy* to perform the crossover operation with a tunable crossover rate which is proposed in [89], which is calculated as Equation (4.7). This method is modified from the *cycle crossover* proposed in [90]. The basic idea of cycle crossover works as follows.

$$PC = \frac{\varrho_c(F_{T_{max}} - F_{T_{bestC}})}{(F_{T_{max}} - F_{T_{avg}})}$$  \hspace{1cm} (4.7)

where $F_{T_{max}}$ is the maximal fitness value in the current population, $F_{T_{bestC}}$ is the fitness value of the parent with higher fitness value between the two crossover parents, $F_{T_{avg}}$ is the average fitness value of the current population, and $\varrho_c$ is a positive constant less than 1.

We start at the first allele of parent 1 and copy the gene to the first position of the child. Then, we look at the allele at the same position in parent 2. We cannot copy this gene to the first position of the child because it has been occupied. We will go to the position with the same gene in the parent 1 and suppose it is at the position $i$. We copy the gene in parent 2 to the position $i$ of the child. We then apply the same operation on the gene in position $i$ of parent 2. The cycle is repeated until we arrive at a gene in parent 2 which has already been in the child. The cycle started from parent 1 is complete. The next cycle will be taken from parent 2. This crossover mechanism enables the child to efficiently inherit the characteristics from both parents.

However, this approach is possible to generate invalid alleles for our data allocation problem, due to the size constraint of each memory unit. An example of such scenario is exhibited in Figure 4.3, where “Parent 1” and “Parent 2” indicate the parents chromosomes, and “Child” is generated by this two chromosomes. In this example, because of the space limitation, we assume that there are 11 data blocks, $A, B, C, D, E, F, G, H, I, J,$ and $K$. 

43
Figure 4.3: An example of cycle crossover. The solid line, dashed line, and dotted line represent the first, second, and third iteration, respectively. The gene (B, 1) is invalid, since the core1’s SRAM has already been full before the allocation of data B.

Figure 4.4: An example of adaptive cycle crossover. The solid line, dashed line, and dotted line represent the first, second, and third iteration, respectively. The circled numbers indicate the adaptive allocation for genes.

needs to be allocated to a dual-core system with hybrid on-chip SPMs configured from SRAM and MRAM. We also assume that the size of SRAM and MRAM are 4KB and 6KB respectively, while the size of each data block is 2KB. Therefore, each SRAM is able to accommodate 2 data blocks and each MRAM can store 3 data blocks. As we can see from the child chromosome, allocating data B to core1’s SRAM will exceed the maximum capacity of the SRAM. This is because the SRAM can only hold 2 data items, but it is assigned 3 data.

Because of the limitation of directly applying the cycle crossover method to our data allocation problem, we propose an adaptive cycle crossover strategy to guarantee valid data allocation. The critical idea of our approach is that we use a variables to keep the currently available space of each memory unit. For each genetic operation of data allocation, we will check if there is enough room for assigning the gene to the specific memory unit. If it is true, the data will be directly allocated. Otherwise, we will adaptive check the memory...
units of the neighboring processor cores and find a space for it. However, if all on-chip memory units, including SRAMs and MRAMs, are full, the data will be assigned to the off-chip main memory. An example of the adaptive cycle crossover operation is shown in Figure 4.4. In these figure, the circled numbers indicate the adaptive adjustments of data allocation to memory units at corresponding steps. The detailed algorithm is shown as Algorithm 4.4.

**Algorithm 4.4 Adaptive cycle crossover algorithm**

**Require:** Two parent chromosomes $P_1$ and $P_2$.

**Ensure:** A new chromosome.

1. Assume the length of each chromosome is $L$.
2. **while** Child chromosome has empty position **do**
3. **for** $i = 1 \rightarrow L$ **do**
4. **if** Gene $i$ in $P_1$ has not been copied to the child chromosome **then**
5. Keep the gene and break;
6. **end if**
7. **end for**
8. **if** The memory unit associated with gene $i$ is full **then**
9. Adaptively search an available position from neighboring memory units;
10. **else**
11. Copy gene $i$ to the same position of the child;
12. **end if**
13. Get a gene $Ge$ at position $i$ in $P_2$;
14. **while** $Ge$ has already existed in the child **do**
15. Locate the gene $Ge$ in $P_1$, suppose its position is $j$;
16. Copy the gene $Ge$ to the position $j$ of the child;
17. Get a new gene $Ge$ at position $j$ in $P_2$;
18. **end while**
19. Apply the same process on $P_2$ to copy genes to the child chromosome;
20. **end while**
21. **return** The child chromosome;

The cycle crossover is able to travel through both two parents. Therefore, it is able to examine the good features of both of them. But the downside of it is the relative long cost of checking each position of parent chromosomes. Hence, we propose another simpler crossover operation, which is a modified version of the *Partially Mapped Crossover* (PMX). The main idea of the modified PMX algorithm works as given in Algorithm 4.5.
Algorithm 4.5 Modified PMX algorithm

Require: Two parent chromosomes $P1$ and $P2$.
Ensure: A new chromosome $C$.

1: Assume the length of each chromosome is $L$;
2: Randomly generate a crossover point $0 \leq cp \leq L$;
3: for all Genes in the segment starting from the crossover point in $P1$ do
4:    Examine the gene at the same position of $P2$;
5:    if The two genes have not been copied to $C$ then
6:        Fill the positions of the child $C$ by swapping the two genes in $P1$;
7:        /*Note that here we only swap the data of two genes while keeping the memory
position unchanged*/
8:    end if
9: end for
10: Map the remaining genes in $P1$ to $C$
11: return The child chromosome $C$;

Parent 1  (A, 1)→(B, 1)→(C, 2)→(E, 2)→(G, 2)→(D, 3)→(F, 3)→(H, 4)→(I, 4)→(J, 4)→(K, 5)
Parent 2  (D, 1)→(I, 4)→(C, 2)→(J, 3)→(B, 2)→(H, 5)→(E, 2)→(F, 4)→(G, 1)→(K, 4)→(A, 3)
Child     (A, 1)→(B, 1)→(C, 2)→(E, 2)→(I, 2)→(D, 3)→(H, 3)→(F, 5)→(G, 4)→(K, 4)→(J, 5)

Figure 4.5: An example of the modified PMX algorithm.

An example, shown in Figure 4.5 is employed to illustrate the modified PMX algorithm. As shown in this figure, the gene pairs after the crossover point are swapped and copied to the child.

3) Mutation

After the crossover operation, a genetic mutation will be performed to recover some good features eliminated by the crossover and prevent the premature convergence to a local optima. It is archived by randomly flipping bits of a chromosome. Similar to the crossover, it is happened in a certain specific probability that is called mutation rate. We define it to be a tunable parameter given in Equation (4.8) and donate it as $PM$. The probability of a
mutation is much lower than that of a crossover. For every new chromosome generated by the crossover operation, we perform the genetic mutation on it with a probability of $PM$, as shown in Algorithm 4.6. Since the gene in this research is defined as a data item and a memory unit pair, the mutation operation can be performed by swapping either the data or the memory units of the selected genes. However, since the datum are independent of each other, these two mutation methods are equal. We will thus swap the number of memory units of two genes to achieve the mutation. For example, Figure 4.6 illustrate the result of our genetic mutation for a chromosome.

\[
PM = \frac{\varrho_m(FT_{\text{max}} - FT_{\text{bestM}})}{(FT_{\text{max}} - FT_{\text{avg}})}
\]  

(4.8)

where $FT_{\text{bestM}}$ is the fitness value of the chromosome to be mutated and $\varrho_m$ is a positive constant less than 1.

**Algorithm 4.6** Algorithm for Genetic Mutation

**Require:** A Chromosome in population and mutation rate $PM$.

**Ensure:** A new chromosome.

1: Randomly select two genes $i$ and $j$ in the input chromosome;
2: Generate a random number $RanN$ between 0 and 1;
3: if $RanN \leq PM$ then
4: Form a new chromosome by swapping the memory units of gene $i$ and gene $j$;
5: end if
6: return The new generated chromosome;

The whole procedure of our AGADA algorithm is described by Algorithm 4.7. First, we need to generate the initial population. In this procedure, a number of chromosomes will be generated randomly. These chromosomes are random permutations of pairs of data and
all memory units of a CMP system (line 1). After the initialization, the fitness value of each individual will be calculated according to Equation (4.3) (line 2). Then, a search process will be iteratively applied to determine the best solution for the data allocation problem until a termination condition is reached. The termination criterion includes two conditions: 1) the number of new generations exceeds a predefined maximum number of iterations, 2) after a certain number of search (typically 500 or even more), a better solution is still unreachable. In each generation, the crossover and mutation operation will be carried out in terms of the predefined crossover rate $PC$ and mutation rate $PM$ (line 6-8). Finally, based on the new population, the fitness value of each individual will be calculated and the selection operation will be employed to generate a new population (line 10).

**Algorithm 4.7 Adaptive Genetic Algorithm for Data Allocation (AGADA)**

**Require:** A set of data items, a CMP system with $P$ processor cores, each core has a hybrid SPM. Any SPM, has a SRAM with size of $SS_i$ and a MRAM with size of $SM_i$.

**Ensure:** A data allocation.

1. Generate initial population;
2. $newPoP \leftarrow \emptyset$;
3. Determine the fitness of each individual;
4. **while** Termination criterion is not met **do**
5. **for** $i = 0 \rightarrow PS$ **do**
6. Randomly select two chromosomes $i$ and $j$ from current population;
7. Optionally apply the crossover operation on chromosomes $i$ and $j$ with probability $PC$;
8. Optionally apply the mutation operation on the new chromosome with probability $PM$;
9. **end for**
10. Evaluate all individuals and perform selection;
11. **end while**
12. **return** The best allocation has obtained;
Chapter 5

Simulation and Experimental Results

This chapter describes some of the evaluation results for the heterogeneous SPM and the associated algorithms. Before presenting the results, the setups and evaluation framework are described.

5.1 Setup

We evaluate our algorithm across a host of benchmarks selected from PARSEC [91]. We run these workloads on M5 simulator [92] and obtain the memory traces for them. We implemented both of the MDPDA algorithm, the adaptive genetic algorithm, and the greedy algorithm as stand-alone programs. These programs take the memory traces we have collected as inputs. We also use a modified version of CACTI [93] to get the memory parameters, including memory read/write latency, energy consumption, and leakage power, for the simulations by using 65 nm technology.

There are two configurations for the target systems. The one is a dual-core in-order CMP system where each core has a hybrid SPM with 4KB SRAM, 16B MRAM, and 8KB Z-RAM. The other one is quad-core CMP where each core has a hybrid SPM with 4KB SRAM, 8KB MRAM, and 4KB Z-RAM. The baseline configuration is a dual-core CMP system with a pure SPM configured from an 8KB SRAM. The specifications of the hybrid memory modules and the baseline are given in Table 5.1. Then, we integrate all these parameters into our custom simulator. To verify the effectiveness of our proposed MDPDA algorithm, 10 applications are selected from PARSEC for simulations: blackscholes, bodytrack, canneal, dedup, streamcluster, facesim, fluidanimate, x264, swaptions, and ferret.
Table 5.1: Performance parameters for the target systems and memory modules.

<table>
<thead>
<tr>
<th>Device</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Number of cores: 2, frequency: 2GHz</td>
</tr>
<tr>
<td>SRAM Baseline</td>
<td>Size: 8KB, read energy: 0.319nJ, write energy: 0.319nJ, leakage power: 2.001mW, read latency: 0.626ns, write latency: 0.626ns</td>
</tr>
<tr>
<td>SRAM</td>
<td>Size: 4KB, read energy: 0.226nJ, write energy: 0.226nJ, leakage power: 1.047mW, read latency: 0.565ns, write latency: 0.565ns</td>
</tr>
<tr>
<td>MRAM</td>
<td>Size: 16KB, read energy: 0.269nJ, write energy: 1.735nJ, leakage power: 0.125mW, read latency: 0.694ns, write latency: 4.386ns</td>
</tr>
<tr>
<td>Z-RAM</td>
<td>Size: 8KB, read energy: 0.293nJ, write energy: 0.401nJ, leakage power: 0.095mW, read latency: 0.831ns, write latency: 1.290ns</td>
</tr>
<tr>
<td>Main memory</td>
<td>Size: 512MB, access energy: 18.046nJ, access latency: 20.35ns, leakage power: 102.560mW</td>
</tr>
</tbody>
</table>

1) Performance analysis of hybrid SPM: We compare data allocation performance of our proposed SRAM, MRAM, and Z-RAM hybrid memory to that of the 4K pure SRAM SPM, in terms of memory access latency and power consumption. To compare the performance of these kinds of platforms, we both use MDPDA algorithm on them. Fig. 5.1 shows the effectiveness of the hybrid SPM with the help of the MDPDA algorithm. We can observe that the hybrid SPM consumes much less power and incurs much shorter memory access latency than the pure SRAM based SPM does. On average, the power saving and memory access latency shrinking across the set of selected benchmarks are around 78.11% and 70.79%, respectively. Two major reasons contribute to the reduction in these two aspects. First, the hybrid SPM architecture benefits from the high density of MRAM and Z-RAM. As a result, it offers much more space for data allocation than that of the pure SRAM based SPM. Thus, the expensive main memory accesses can be significantly reduced in hybrid SPM systems. Second, the MDPDA algorithm is able to appropriately allocate data to different memory modules for the hybrid SPM, which further reduces the
long latency and high energy of writes on MRAM.

2) Comparison results: we compare the performance of the MDPDA algorithm to that of the greedy algorithm. Fig. 5.2 illustrate the comparisons of the number of writes to MRAM for the MDPDA algorithm and the greedy algorithm across the set of workloads, when the target platform is a dual-core system (see Fig. 5.2(a)) and a quad-core system (see Fig. 5.2(b)), respectively. It can be observed that compared to the greedy algorithm, the MDPDA algorithm can reduce the number of writes to MRAM by 35.25% and 39.67% on average for the dual-core system and quad-core system, respectively. The major reason for the reduction is that the greedy algorithm only greedily select the most frequently referenced data blocks to the the core which accesses it mostly, while it does not fully take advantages of different kinds of memory modules. Instead, the MDPDA algorithm is write-aware, and it can sophisticatedly allocate each data block to different on-chip memory units. For example, with the aid of the MDPDA algorithm, most of write activities will be assigned to SRAM and Z-RAM, while most frequently read data will be allocated to MRAM and Z-RAM.

From Fig. 5.2(a) and (b), we can also see that the reduction of the number of writes on quad-core system is more prominent than that of on dual-core system. This is mainly the more the cores, the larger the on-chip memory is available. Through the optimal allocation, the MDPDA algorithm, therefore, can more significantly reduce the number of writes to MRAM. The reduction of write activities on MRAM will efficiently contribute to the extension of their lifetime.

Fig. 5.3 and Fig. 5.4 exhibit the effectiveness of the MDPDA algorithm over that of the greedy algorithm, with respect to memory access latency and total energy consumption through the 10 workloads. We also investigate the performance for dual-core system and quad-core system for each case. Fig. 5.3(a) and (b) show that compared to the greedy algorithm, the MDPDA algorithm can reduce the total memory access latency by 16.23% on average for dual-core system and 23.43% on average for quad-core systems, respectively.
Figure 5.1: The comparison of performance of hybrid SPM and pure SPM with respect to latency and energy consumption. (a) The comparison of latency, (b) The comparison of energy consumption.
Figure 5.2: The comparison of the number of writes to MRAM for the greedy algorithm and the MDPDA algorithm, when the target system is a: (a) dual-core platform, (b) quad-core platform.
Similarly, Fig. 5.4 (a) and (b) demonstrate that the MDPDA algorithm outstrips the greedy algorithm in terms of energy efficiency. On average, the MDPDA algorithm can reduce the dynamic power consumption for dual-core system and quad-core system by 17.74% and 24.18%, respectively, compared to the greedy algorithm. In light of the results, we can see that the reduction in energy consumption is proportional to the reduction in memory access latency. The reduction is mainly because of the optimal allocation of the MDPDA for each data block at each step.

5.2 Setup for Genetic Algorithm

The following parameter specifications are used in our simulations for the AGADA algorithm. 1) Population size: 300; 2) Crossover rate: $\rho_c = 0.8$; 3) Mutation rate $\rho_m = 0.02$; 4) Selection method: rank based roulette wheel; 4) maximum generation: 1000.

We compare the performance of the AGADA algorithm to that of the greedy algorithm. Fig. 5.5, Fig. 5.6, and Fig. 5.7 illustrate comparisons between the greedy algorithm and the AGADA algorithm, with respect to the number of writes to MRAMs, dynamic energy consumption, and memory access latencies. Compared to the greedy algorithm, the average performance improvements of our AGADA algorithm are 32.96%, 15.98%, and 14.42%, respectively. By reducing the number of writes to MRAMs, the AGADA algorithm can efficiently extend the usage of MRAMs.

Performance analysis and comparison for the AGADA algorithm: First, we verify the precision of the AGADA strategy for data allocation in hybrid SPM architectures, by comparing to the optimal allocation results of the multi-dimension dynamic programming algorithm. Fig. 5.8 shows that dynamic energy consumption of the AGADA algorithm is approximate to that of the optimal dynamic programming algorithm, with respect to the 7 applications selected from PARSEC. On average, the AGADA consumes 2.21% more dynamic power than that of the multi-dimensional dynamic programming algorithm counterpart. However, considering the high time and space complexity of the multi-dimensional
Figure 5.3: The comparison of memory access latency for the greedy algorithm and the MDPDA algorithm, when the target system is a: (a) dual-core platform, (b) quad-core platform.
Figure 5.4: The comparison of memory access power for the greedy algorithm and the MDPDA algorithm, when the target system is a: (a) dual-core platform, (b) quad-core platform.
Figure 5.5: The comparison of the number of writes operations to MRAM caused by data allocation strategies of the greedy algorithm and our proposed adaptive genetic algorithm (AGADA). The AGADA algorithm reduces the number of writes 32.96% on average.

Figure 5.6: The comparison of energy consumption caused by the greedy algorithm and the adaptive genetic algorithm (AGADA) for data allocation. The AGADA algorithm reduces dynamic energy consumption by 15.98% on average.
dynamic algorithm, the AGADA algorithm is more competitive in overall performance.

For example, for a $n$-core CMP with hybrid SPMs, the multi-dimensional dynamic algorithm needs $O(N \times \prod_{i=1}^{M}(Size_{S_i} \times Size_{M_i}))$ times and spaces to get the solution and maintain the cost matrix used the algorithm, where $N$ and $M$ are the number of input data and SPMs, respectively; $Size_{S_i}$ and $Size_{M_i}$ are the size of SRAM and MRAM of SPM $i$, respectively. Instead, the AGADA algorithm organizes a chromosome in the form of the list structure, which only requires $O(G \times P \times N)$ space to maintain the entire chromosomes, where $G$ and $P$ represent the maximum number of iterations and the population size of the genetic algorithm, respectively. Moreover, $G$ and $P$ are constants, and $G \times P$ is much less than $\prod_{i=1}^{M}(Size_{S_i} \times Size_{M_i})$. 

Figure 5.7: The comparison of memory access latencies caused by the greedy algorithm and the adaptive genetic algorithm (AGADA) for data allocation. The AGADA algorithm reduces memory access latencies by 14.42% on average.
Figure 5.8: The comparison of dynamic energy consumption caused by the multi-dimensional programming algorithm and the adaptive genetic algorithm (AGADA) for data allocation. On average, the AGADA algorithm consumes 2.21% more dynamic energy consumption than the multi-dimensional programming algorithm.
Chapter 6
Conclusion and Future Work

6.1 Summary and Contributions

In the last few years, there has been a significant increase in the number and variety of research and applications in Chip Multiprocessor (CMP) systems. However, the continuously wide deployment of these systems are seriously hindered by a number of challenges including power consumption, real-time guarantee, and memory wall, etc. Given this trend, the prohibitively expensive memory access cost in terms of either energy or latency is a major limiting-factor for the advancement of CMP systems.

While cache was an effective technique to bridge the processor-memory speed gap and achieved grate success in traditional desktops, they are imposing significant performance and energy overhead for embedded CMP systems. Scratch Pad Memory (SPM), a software-controlled on-chip memory, has been gathered wide interests from both academic and industrial communities due to they superiority in area, energy consumption, and predictability of program execution over caches. While prior research investigated numerous techniques to allocate program code/data on SPMs, most of these efforts mainly focus on the pure SPM configured from a small SRAM. Few previous work proposed data allocation approaches such as dynamic programming and ILP technique for CMP systems heterogeneous SPMs. However, they either didn’t consider ZRAM and MRAM or incur large space complexity.

This thesis addressed the high energy and long latency problem in embedded CMP systems through making the following contributions:\footnote{This thesis is partially supported by CNS-1249223.}
This thesis uses and experimentally evaluates a heterogeneous SPM architecture which is configured from SRAM, MRAM, and ZRAM for embedded CMP systems. The basic idea of this heterogenous SPM is to sufficiently take advantages of different memory technologies, therefore providing a promising solution to memory wall of CMP systems.

This thesis proposes a multi-dimensional dynamic programming algorithm to obtain the optimal data allocation. There are $3 \times N + 1$ layers of loops in the algorithm, where $N$ is the number of cores. We will use a trace function to keep track of the previous position from which we obtain the best solution and a movement-recording function to record the data movement action for the current data. When we find the minimum cost, we can trace the path of the trace function and find the allocation of each data with the help of the movement-recording function.

Taking the high space complexity of the multidimensional programming algorithm, this thesis proposes an adaptive genetic algorithm, *Adaptive Genetic Algorithm for Data Allocation* (AGADA), to efficiently allocate data on each memory unit of the heterogenous SPMs. The basic idea of this algorithm is to reduce energy consumption for the proposed hybrid SPM architecture through effectively searching reasonable solutions.

### 6.2 Future Work

There are still multiple research problems aligned with this thesis that are worthy to explore in the future. We list them as follows.

1. This mainly focuses on the data allocation through static methods for embedded CMP systems with heterogeneous SPM. However, dynamic allocation mechanisms are also widely used in data allocation for pure SPMs. Most of the prior dynamic allocation methods are oblivious to the heterogeneous SPM architecture. Therefore,
in the future work, we plan to investigate a sophisticated dynamic allocation method that is able to effectively allocate heap data, stack data, and global data on heterogeneous memory.

2. How to extend wear-leveling for heterogeneous memories is of significance to embedded CMP systems because wear-leveling varies with different memory technologies. In order to enable the longer usage of on-chip heterogeneous SPM, more techniques can be exploited in the near future including data recomputation, migration, and replication.

3. We can move the heterogeneous on-chip software controlled memory to even more platforms such as many-core systems or clusters with well design. To cope with the high parallelism of these platforms, we can concentrate on a number of research spots such as how to improve the task-level parallelism and how to partition program code/data to exploit their parallelism.

4. Reliable heterogeneous on-chip and off-chip memory co-design will be also a part of our future work. Previous research mainly focuses on cache-based hierarchies, while there is virtually no work to guarantee the reliability of heterogeneous SPM. We will study the traditional mechanisms such as ECC [94] and multi-bit error protection [95]. Meanwhile, special focus on the hardware variability will be associated with the reliability ensuring methods.
Bibliography


Vita

Author’s Name - Zhi Chen

Birthplace - Changsha, China

Education

Master of Engineering in Computer Science
Hunan University, China
June - 2011

Bachelor of Science in Computer Science
Huaihua University, China
June - 2008

Research Experience

University of Kentucky
Lexington, KY
08/2011 - present
Graduate Teaching Assistant

Peer Reviewed Publications


Awards

ACM Student Research Competition Grand for the 48th Design Automation Conference (DAC), 2012, ACM

Society Memberships

ACM, IEEE