2012

ENERGY-AWARE OPTIMIZATION FOR EMBEDDED SYSTEMS WITH CHIP MULTIPROCESSOR AND PHASE-CHANGE MEMORY

Jiayin Li
University of Kentucky, lijiayin1983@gmail.com

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.

Jiayin Li, Student

Dr. Meikang Qiu, Major Professor

Dr. Zhi David Chen, Director of Graduate Studies

ENERGY-AWARE OPTIMIZATION FOR EMBEDDED SYSTEMS WITH CHIP MULTIPROCESSOR AND PHASE-CHANGE MEMORY

DISSERTATION

A dissertation submitted in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy in the
College of Engineering at the
University of Kentucky

By
Jiayin Li
Lexington, Kentucky

Director: Dr. Meikang Qiu, Professor of Electrical and Computer Engineering
Lexington, Kentucky 2012

ABSTRACT OF DISSERTATION

ENERGY-AWARE OPTIMIZATION FOR EMBEDDED SYSTEMS WITH CHIP MULTIPROCESSOR AND PHASE-CHANGE MEMORY

Over the last two decades, functions of the embedded systems have evolved from simple real-time control and monitoring to more complicated services. Embedded systems equipped with powerful chips can provide the performance that computationally demanding information processing applications need. However, due to the power issue, the easy way to gain increasing performance by scaling up chip frequencies is no longer feasible. Recently, low-power architecture designs have been the main trend in embedded system designs.

In this dissertation, we present our approaches to attack the energy-related issues in embedded system designs, such as thermal issues in the 3D *chip multiprocessor* (CMP), the endurance issue in the *phase-change memory*(PCM), the battery issue in the embedded system designs, the impact of inaccurate information in embedded system, and the cloud computing to move the workload to remote cloud computing facilities.

We propose a real-time constrained task scheduling method to reduce peak temperature on a 3D CMP, including an online 3D CMP temperature prediction model and a set of algorithm for scheduling tasks to different cores in order to minimize the peak temperature on chip. To address the challenging issues in applying PCM in embedded systems, we propose a PCM main memory optimization mechanism through the utilization of the *scratch pad memory* (SPM). Furthermore, we propose an MLC/SLC configuration optimization algorithm to enhance the efficiency of the hybrid DRAM + PCM memory. We also propose an energy-aware task scheduling algorithm for parallel computing in mobile systems powered by batteries.

When scheduling tasks in embedded systems, we make the scheduling decisions based on information, such as estimated execution time of tasks. Therefore, we design an evaluation method for impacts of inaccurate information on the resource allocation in embedded systems. Finally, in order to move workload from embedded systems to remote cloud computing facility, we present a resource optimization mechanism in heterogeneous federated multi-cloud systems. And we also propose two online dynamic algorithms for resource allocation and task scheduling. We consider the resource contention in the task scheduling.

KEYWORDS: Embedded system, CMP, memory, battery, cloud computing

Author's signature: Jiayin Li

Date: April 25, 2012

ENERGY-AWARE OPTIMIZATION FOR EMBEDDED SYSTEMS WITH CHIP MULTIPROCESSOR AND PHASE-CHANGE MEMORY

By
Jiayin Li

Director of Dissertation: Meikang Qiu

Director of Graduate Studies: Zhi David Chen

Date: April 25, 2012

Dedicated to my family

ACKNOWLEDGMENTS

This dissertation cannot be completed without help from many people. All of them are very important to me.

First of all, I would like to thank Professor Qiu for his guidance, encouragement, and support. He allowed me to explore various topics while seeking for my own topic. We closely collaborated on all the work presented in the body of this dissertation. His advice goes beyond research work and prepares me to take on bigger challenges in my career.

I am also grateful to my family. Their endless love and support always encourage me to deal with obstacles in the Ph.D. journey. In particular, I want to express my deepest gratitude to my dear wife, Ying, for her enduring love, encouragement, and understanding during my study.

I am deeply thankful to my dissertation committee, Professor Henry G. Dietz, Professor J. Robert Heath, and Professor Dakshnamoorthy Manivannan, for spending their time reviewing my dissertation and providing suggestions to improve my work. I would also like to thank Professor Wolfgang Korsch for his suggestions and comments to strengthen the dissertation.

My colleagues at our lab also enriched my study and my life. I would like to thank Hai Su and Zhi Chen for their suggestions and helps.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1 Introduction

Over the last two decades, functions of embedded systems have evolved from simple real-time control and monitoring to more complicated services running on smartphones, such as multi-media streaming, on-line shopping, and banking. Embedded systems have high influence on both the system industry and our daily life. Embedded systems equipped with powerful chips, such as multi-core processors, high-capacity memories, and high-speed I/O interfaces, can provide the performance that computationally demanding information processing application need. Designs from Nvidia already have demonstrated the power of a quad-core processor for smartphones.

Meanwhile, computer architectures have been evolved rapidly in the last five decades, in terms of computational power and architecture complexity, thanks to the fast development of semiconductor fabrication techniques. The transistor density doubles every eighteen months. However, due to the power issue, the easy way to gain increasing performance by scaling up chip frequencies is no longer feasible. Recently, low-power architecture designs have been the main trend in computer architecture research, especially in embedded system designs.

The major energy consuming components in embedded systems are the processor and the memory. Therefore, extra research efforts should be focused on the energy-aware optimization in processors and memory architectures in embedded systems. Meanwhile, since most of the embedded systems, such as wireless sensors and mobile devices, are powered by batteries, the battery-aware optimization is another method in low-power embedded system designs.

1.1 Power related issues in the embedded system architecture

Chip multiprocessors (CMP) have been widely used in Embedded Systems due to tremendous computation requirements in the modern embedded processing. The primary goals for microprocessor designers are to increase the integration density and achieve higher performance without correspondingly increases in frequency. However, traditional *two dimensional* (2D) planar CMOS fabrication processes are poor at communication latency and integration density. The *three dimensional* (3D) CMOS fabrication technology is one of the solutions for faster communication and more functionalities on chip. More functional units can be implemented while stacking two or more silicon layers in a CMP. Meanwhile, the vertical distance is shorter than the horizontal distance in a multi-layer chip [1, 2], which makes the systems more tight. The concern with regard to the on-chip temperature is increasing in CMP design. Higher power consumption leads to higher on-chip temperature. Meanwhile, high on-chip temperature impacts circuit reliability, energy consumption, and system cost. Research shows that a 10 to 15°C increase of operation temperature reduces the lifetime of the chip by half [3].

Memory architecture is another key track in low-power embedded system designs. In the last three decades, dynamic RAM (DRAM), as the major technique of the main memory, has become one of the primary energy consuming parts of the embedded systems [4,5]. For example, 2GB of DRAM consumes 3W to 6W, which is equivalent to the total power consumption of the Atom processor [6]. Meanwhile, it has also been reaching its scalability limits [7]. As the memory demands of applications keep increasing, the size of DRAM equipped in a system needs to be larger and larger. However, DRAM requires some specific architecture solutions to address some drawback issues [6]. These specific architecture solutions cause extra costs that are the major reason of the scalability limit in DRAM.

Phase-change memory (PCM) is emerging as a promising DRAM alternative technique, featuring many attractive advantages, such as high density, non-volatility, positive response to increasing temperature, zero standby leakage, and excellent scalability [5, 8–11]. PCM

switches its chalcogenide material between the amorphous and the crystalline states. Detecting the resistances of different states, data is stored in PCM devices. The application of heat that is required by the switch between states can be provided by using the electrical pulses. Researchers have stated that PCM has more robust scalability beyond 40 nm than DRAM does [12]. And a 32-nm device prototype has been demonstrated [13].

Even though PCM is alternative to DRAM as the main memory, large efforts are needed to surmount the disadvantage of PCM. PCM access latencies, especially in writes, are slower than those of DRAM. In the read access, PCM is 2x-4x slower than DRAM. Moreover, PCM displays asymmetric timings for reads/writes, which means writes in PCM need 5x-10x more time than reads do. Due to the fact that phase changes in PCM are induced by injecting current into the chalcogenide material and heating it, writes are the primary wear mechanism and the most energy-consuming mechanism in the PCM. The number of writes performed before the cell is not able to perform reliably ranges from 10^8 to 10^9. Writes in PCM limits both the performance and the lifetime of PCM. Therefore, reducing the number of writes can both increase the lifetime of the PCM and decrease the energy consumption in the memory architecture.

Another attracting property of PCM is that multiple bits can be stored in one single PCM cell, called *Multi-Level Cell* (MLC). PCM can provide four times more density than DRAM [10]. Recently, several studies [8, 14–16] have advocated for the MLC PCM memory architecture. The difference of resistance between the two states of the chalcogenide material is usually 3 orders of magnitude [16]. By precisely dividing this gap into several levels, one PCM cell can store more than one bit data. Therefore, the scalability of the PCM memory is four times higher than that of DRAM.

When the MLC technique can enhance the scalability of the PCM memory, this improvement comes at a high price. The degradation of performance and endurance of the PCM memory as well as the increase in energy consumption are the major drawbacks of the MLC techniques [16]. As the number of bits stored a single PCM cell increases, the

number of levels divided in this cell increases exponentially. For example, a 4 bits/cell MLC has total sixteen levels of resistance values. In this case, due to the 8 times smaller resistance difference between two consecutive levels, a more precise resistance detection method is required in this MLC, compared to the one used in the *single-level cell* (SLC). In the write operation in the MLC, the "program and verify" procedure is applied repeatedly until the resistance is programmed correctively in the target level [4, 14]. The repeated programming current pulses in the "program and verify" cause high power consumption in the PCM memory. In addition, these repeated pulses applied in the MLC make the already poor endurance of the PCM memory even worse [16]. Thus, the SLC PCM provides higher performance with less power consumption and longer lifetime, while the MLC PCM enhances the memory capacity without increasing the number of PCM cells.

Due to the increasingly energy consuming processor and memory in the embedded system, the lifetime of battery in the embedded system has also become a significant challenge in the embedded system design. In the recent two decades, the increase of processor speed is much bigger than the increase of energy density of battery. At the distributed embedded system point of view, scheduling tasks across different embedded devices with the consideration of battery behaviors can provide the balance between the performance of the whole system and the lifetime of the battery in different embedded devices.

When scheduling tasks in embedded systems, we make the scheduling decisions based on information, such as estimated execution time of tasks. However, when estimated task execution time is calculated by using inaccurate information, estimated tasks execution times may be different from actual ones. Therefore, decisions generated by estimated task execution times may not be robust and the resource allocation is not able to guarantee the given level of *Q*uality of ServiceQoS. Therefore, we need to measure the impacts of inaccurate information on the robustness of the system.

Another approach to reduce the energy consumption of embedded systems is to move computation tasks to remote computing facilities. Cloud computing is a promising method,

in which energy constrained embedded systems rent virtual machines from cloud providers or data centers. The energy constrained embedded system simply works as a terminal, and virtual machines in the remote cloud provider are rented to actually execute tasks. In this case, the embedded system, as a terminal, does not require a significant amount of energy. And a number of virtual machines can be rented based on the computational demand of tasks. As embedded systems are widely used in various fields, the demand of cloud computing for embedded systems may increase exponentially. Therefore, the resource capacity of a single cloud provider may not be enough when a number of embedded system clients submit their tasks to the cloud. Thus, to collaborate more than one cloud in a cloud platform, we need to investigate the resource allocation mechanism in multi-cloud platform and provide optimization methods for the cloud services.

1.2 Contributions

In this dissertation, we present our approaches to attack energy-related issues in embedded system designs, such as thermal issues in the 3D CMP chip, endurance issues in PCM, the battery issue in the embedded system design, the impact of inaccurate information in embedded system, and the cloud computing to move the workload to remote cloud computing facilities. The contributions are listed as the following:

- We propose a real-time constrained task scheduling method to reduce peak temperature on a 3D CMP. First of all, we develop an online 3D CMP temperature prediction model. Based on this model, we further design a set of algorithms for scheduling tasks to different cores in order to minimize the peak temperature on chip.

- We propose a PCM main memory optimization mechanism through the utilization of the *Scratch Pad memory* (SPM). The SPM is a small size on-chip memory mapped into the memory address space disjoint from the off-chip memory, such as the PCM main memory. We design an *Integer Linear Programming* (ILP) algorithm for schedul-

ing memory activities among the SPMs and the PCM main memory. In our ILP algorithm, unnecessary writes are eliminated. Instead, the data copies are shared among the SPMs.

- We propose an MLC/SLC configuration optimization algorithm to enhance the efficiency of the hybrid DRAM + PCM memory. Embedded systems are designed to execute specific applications. Optimizing the PCM configuration based on the characteristics of applications can further enhance the efficiency of the main memory in embedded CMP systems. We present a set of algorithms for both task scheduling and MLC/SLC PCM mode configuration.

- We further propose a energy-aware task scheduling algorithm for parallel computing in mobile systems powered by batteries. With a model of battery behaviors, we develop a energy-aware task scheduling algorithm to optimize the performance while satisfying the lifetime constraint of batteries.

- We design an evaluation method for impacts of inaccurate information on resource allocation in embedded systems. We propose a systematic way of measuring the robustness degradation and evaluate how inaccurate probability parameters affect the robustness of resource allocations. Furthermore, we compare the performance of three widely used greedy heuristics when using the inaccurate information with simulations.

- We present a resource optimization mechanism in heterogeneous federated multi-cloud systems. And we also propose two online dynamic algorithms for resource allocation and task scheduling. We consider the resource contention in the task scheduling.

1.3 Outline

The rest of the dissertation is organized as follows: Chapter 2 propose an online thermal prediction model for 3D chips. Novel task scheduling algorithms based on rotation scheduling is proposed to reduce the peak temperature on chip. In Chapter 3, we present the SPM based memory mechanism and an ILP memory activities scheduling algorithm to prolong the lifetime of the PCM memory in embedded systems. We also design four optimization algorithms for embedded systems equipped with the MLC/SLC PCM + DRAM hybrid memory in Chapter 4. In our proposed algorithms, we not only schedule and assign tasks to cores in the CMP system, but also provide a hybrid memory configuration that balances the hybrid memory performance as well as the efficiency. Chapter 5 discusses battery behaviors in embedded systems. We present a systematic system model for task scheduling in embedded system equipped with Dynamic Voltage Scaling (DVS) processors and energy harvesting techniques. We propose the three-phase algorithms to obtain task schedules giving shorter total execution time while satisfying the lifetime constraints. Chapter 7 proposed a resource optimization mechanism in heterogeneous federated multi-cloud systems and two online dynamic algorithms for resource allocation and task scheduling. We discuss how inaccurate probability parameters affect the robustness of resource allocations in the distributed embedded system network in Chapter 6. We propose a systematic way of measuring the robustness degradation and comparing the performance of three widely used greedy heuristics when using the inaccurate information with simulations. We conclude this dissertation in Chapter 8.

Chapter 2 Thermal-Aware Task Scheduling in CMP

Chip multiprocessor (CMP) techniques have been implemented in embedded systems due to tremendous computation requirements. The three-dimension (3D) CMP architecture has been studied recently for integrating more functionalities and providing higher performance. The high temperature on chip is a critical issue for the 3D architecture. In this chapter, we propose an online thermal prediction model for 3D chips. Using this model, we propose novel task scheduling algorithms based on the rotation scheduling to reduce the peak temperature on chip. We consider data dependencies, especially inter-iteration dependencies that are not well considered in most of the current thermal-aware task scheduling algorithms. Our simulation results show that our algorithms can efficiently reduce the peak temperature up to 8.1°C.

2.1 Introduction

Chip multiprocessors (CMP) have been widely used in *Embedded Systems for Interactive Multimedia Services* (ES-IMS) due to tremendous computation requirements in modern embedded processing. The primary goals for microprocessor designers are to increase the integration density and achieve higher performance without correspondingly increases in frequency. However, traditional *two dimensional* (2D) planar CMOS fabrication processes are poor at communication latency and integration density. The *three dimensional* (3D) CMOS fabrication technology is one of the solutions for faster communication and more functionalities on chip. More functional units can be implemented while stacking two or more silicon layers in a CMP. Meanwhile, the vertical distance is shorter than the horizontal distance in a multi-layer chip [1, 2], which makes the systems more tight.

In CMPs, high on-chip temperature impacts circuit reliability, energy consumption, and system cost. Research shows that a 10 to 15°C increase of operation temperature reduces

the lifetime of the chip by half [3]. The increasing temperature causes the leakage current of a chip to increase exponentially. Also, the cooling cost increases significantly, which amounts to a considerable portion of the total cost of the computer system. The 3D CMP architecture magnifies the thermal problem, due to the fact that the cross-sectional power density increases linearly with the number of stacked silicon layers, causing more serious thermal problems.

To mitigate the thermal problem, *Dynamic Thermal Management* (DTM) techniques, such as *Dynamic Voltage and Frequency Scaling* (DVFS), have been developed at the architecture level. When the temperature of the processor is higher than a threshold, DTM can reduce the processor power and control the temperature of the processor. With DTM, the system performance is degraded inevitably. Another way to alleviate the thermal problem of the processor is to use the operation system level task scheduling mechanism. They either arrange the task execution order in a designated manner, or migrate "hot" threads across cores to achieve thermal balance. However, most of these thermal-aware task scheduling methods focus on independent tasks or tasks without inter-iteration dependencies. Applications in modern ES-IMS often consist of a number of tasks with data dependencies, including inter-iteration dependencies. Therefore, it is important to consider the data dependencies in the thermal-aware task scheduling.

In this chapter, we propose real-time constrained task scheduling algorithms to reduce the peak temperature in the 3D CMP. The proposed algorithms are based on the rotation scheduling [17], which optimizes the execution order of dependent tasks in a loop. The main contributions of this chapter include:

1. We present an online 3D CMP temperature prediction model.

2. We also propose task scheduling algorithms to reduce the peak temperature. The data dependencies, especially inter-iteration dependencies in the application are well considered in our proposed algorithms.

The organization of this chapter is as follows. In Section 2.2, we discuss works related to this topic. Then, models for task scheduling in 3D CMPs are presented in Section 2.3. A motivational example is given in Section 2.4. We propose our algorithms in Section 2.5, followed by experimental results in Section 2.6. Finally, Section 2.7 conclude the chapter.

2.2 Related work

Energy-aware task scheduling has been widely studied in the literature. Weiser et al. first discussed the problem of task scheduling to reduce the processor energy consumption in [18]. An off-line scheduling algorithm for task scheduling with variable processor speeds was proposed in [19]. But tasks considered in these papers are independent tasks. Authors in [20] proposed several schemes to dynamically adjust the processor speed with slack reclamation based on the DVS technique. A scheme for the processor speed management at branches was presented in [21] based on the ratio of the longest path to the taken paths for the branch statement to the end of the program. However, the studies above only consider the uniprocessor system.

Recently, energy reduction has become an important issue in parallel systems. Research in [22, 23] focused on heterogeneous mobile ad hoc grid environments. Authors in those works studied the static resource allocation for the application composed of communicating subtasks in an ad-hoc grid. However, the goal of the allocation in those works is to minimize the average percentage of energy consumed by the application to execute across the machines, while meeting an application execution time constraint. This goal may lead to some cases in which some machines may consume much more energy than the others, even though the average consumption is minimized. Therefore, approaches proposed in those works cannot guarantee the satisfaction of the temperature constraint.

Authors in [24] proposed two task scheduling algorithms for embedded system with heterogeneous functional units. One of them is optimal and the other is near-optimal heuristic. The task execution time information was stochastically modeled. In [25], the

authors proposed a loop scheduling algorithm for voltage assignment problem in embedded system. The research in [26] focused on modeling task execution time as a probabilistic random variable. Two optimal algorithms, one for uniprocessor and one for multiprocessor system, were presented to solve the voltage assignment with probability problem. The goal of these algorithms is to minimize the expected total energy consumption while satisfying the timing constraint. However, none of them consider thermal issues on processors.

In chip design stage, several techniques are implemented for thermal-aware optimization. Authors in [27, 28] proposed different thermal-aware floorplanning algorithms. For floorplanning on 3D chips, several other approaches are proposed recently [29–32]. The authors in [33] proposed the controlling *Thin-Film Thermoeletric cooling* (TFTECs) from the microarchitecture for an enhanced DTM in multi-core architectures. Research in [34] focuses in improving the efficiency of heat removal.

Job allocation and scheduling is another approach to reduce temperature on-chip. Several temperature-aware algorithms were presented in [35–42] recently. The Adapt3D approach in [37] assigns the upcoming job to the coolest core to achieve thermal balance. The method in [41] is to wrap up aligned cores into super core. Then the hottest job is assigned to the coolest super core. The power and thermal management framework is proposed in [38] for memory subsystem. In [39], a thermal management scheme incorporates temperature prediction information and runtime workload characterization to perform efficient thermally aware scheduling. A scheduling scheme based on mathematic analysis is proposed on [40]. Authors in [42] present a slack selection algorithm for thermal-aware dynamic frequency scaling. But none of these approaches considers data dependencies in an application.

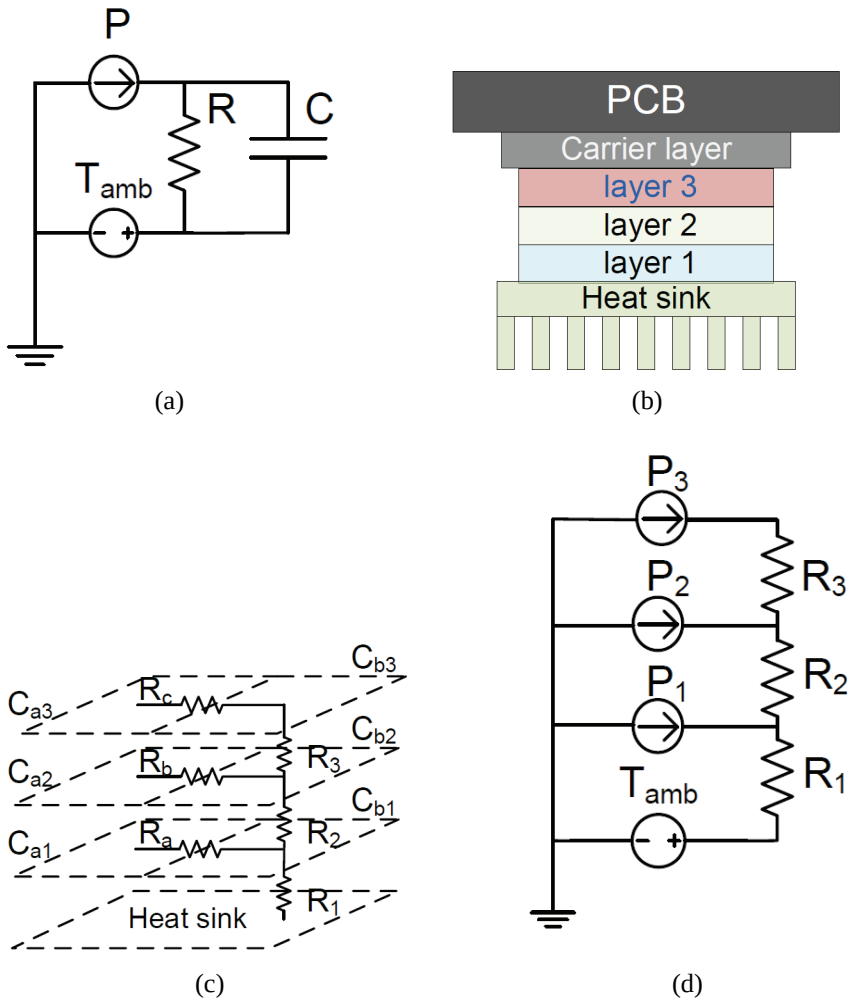


Figure 2.1: Thermal model for the 3D chip. (a) A Fourier thermal model of a single block. (b) The cross sectional view of a 3D chip. (c) The horizontal and vertical heat model, where the C_{a1} to C_{b3} are the IDs of the six cores in this example, the R_a to R_c are the vertical heat conductances, and R_1 to R_3 are the horizontal heat conductances. (d) The corresponding Fourier thermal model.

2.3 Model and Background

Thermal model

The Fourier heat flow analysis is the standard method of modeling heat conduction for circuit-level and architecture-level IC chip thermal analysis [40]. It is analogous to George Simon Ohm's method of modeling electrical current. A basic Fourier model of heat conduction in a single block on a chip is shown in Fig. 2.1(a). In this model, the power dissipation is similar to the current source and the ambient temperature is analogous to the voltage source. The heat conductance of this block is a linear function of conductivity of its material and its cross-sectional area divided by its length. It is equivalent to the electrical conductance. And the heat capacitance of this block is analogous to the electrical capacitance. Assuming there is a block on a chip with heat parameters as shown in Fig. 2.1(a). The Fourier heat flow analysis model is

$$C\frac{d(T(t)-T_{amb})}{dt} = P - \frac{T(t)-T_{amb}}{R} \tag{2.1}$$

C is the heat conductance of this block. $T(t)$ is the temperature of that block at time t. T_{amb} is the ambient temperature, P is the power dissipation, and R is the heat resistance. By solving this differential equation, we get the temperature of that block as follows:

$$T(t) = P \times R + T_{amb} - (P \times R + T_{amb} - T_{init})e^{-t/RC} \tag{2.2}$$

T_{init} is the initial temperature of that block.

Considering there is a task a running on this block and the corresponding power consumption is P_a, we can predict the temperature of the block by equation (2.2). Assuming that the execution time of a is t_a, we get the temperature of the block when a is finished:

$$T(t_a) = P_a \times R + T_{amb} - (P_a \times R + T_{amb} - T_{init})e^{-t_a/RC} \tag{2.3}$$

When the execution of task a goes infinite, the temperature of this block reaches a stable

state, T_{ss}, which is shown as follows:

$$T_{ss} = P_a \times R + T_{amb} \tag{2.4}$$

Substituting equation (2.4) in equation (2.3), we can get an alternative way of predicting the finish temperature of task a running on that block:

$$T(t_a) = (T_{ss} - T_{init})(1 - e^{-t_a/RC}) + T_{init} \tag{2.5}$$

We can further simplify equation (2.5) as follows:

$$T(t_a) = (T_{ss} - T_{init})(1 - e^{-bt_a}) + T_{init} \tag{2.6}$$

where $b = 1/RC$.

The 3D CMP and the core stack

A 3D CMP consists of multiple layers of active silicon. On each layer, there exist one or more processing units, which we call cores. Fig. 2.1(b) shows a basic multi-layer 3D chip structure. A heat sink is attached to the top of the chip to remove the heat from the chip more efficiently. The horizontal lateral heat conductance is approximately 0.4 W/K (i.e. "R_a" in Fig. 2.1(c)), much less the conductance between two vertically aligned cores (approximately 6.67 W/K, i.e. "R_2" in Fig. 2.1(c)) [40]. The temperature values of vertically aligned cores are highly correlated, compared with the temperatures of horizontally adjacent cores.

Therefore, for the online temperature prediction model used in our scheduling algorithms, we ignore the horizontal lateral heat conductance. Note that, even though we ignore this heat conductance in our model, the simulator used in our experiment is a general thermal simulator that considers both the horizontal lateral heat conductance and the vertical conductance. The efficiency of our low-computation model is tested through this general thermal simulator in our experiment. We call a set of vertically aligned cores as a *core stack*. Cores in a core stack are highly thermal correlated. The high temperature of a core

caused by heavy loading will also increase the temperatures of other cores in the core stack. For cores in a core stack, the distances from them to the heat sink are different. Considering a number k of cores in a core stack, where core k is the furthest from the heat sink and core 1 is the closest to the heat sink; the stable state temperature of the core j ($j \leq k$) can be calculated as,

$$T_{ss}(j) = \sum_{i=1}^{j}(\sum_{l=i}^{k} P_l \times R_i) + T_{amb} \tag{2.7}$$

where P_l is the power consumption of the core l and R_i is the inter-layer thermal conductance between cores $i-1$ and i (see Fig. 2.1(d)).

In order to predict the finish temperature of task a running on core j online, we approximate this finish temperature $T_j(t_a)$ by substituting equation (2.7) in equation (2.5) as

$$\begin{aligned} T_j(t_a) &= (\sum_{i=1}^{j}(\sum_{l=i}^{k} P_l \times R_i) + T_{amb} - T_{init\_j}) \\ &\quad \times(1 - e^{-t_a/R_jC_j}) + T_{init\_j} \end{aligned} \tag{2.8}$$

Application model

A *Data-Flow Graph* (DFG) is used to model an embedded system application. A DFG typically consists of a set of vertices V, each of which represents a task in the application, and a set of edges E, showing the dependencies among the tasks. The edge set E contains edges e_{ij} for each task $v_i \in V$ that task $v_j \in V$ depends on. The weight of a vertex v_i represents the task type of task i. In our model, the number of tasks may be larger than the number of task types. And the tasks with the same task type have the same execution time. Also the weight of an edge e_{ij} means the size of data which is produced by v_i and required by v_j.

We use a cyclic DFG to represent a loop of an application in this chapter. In a cyclic DFG, a delay function $d(e_{ij})$ defines the number of delays for edge e_{ij}. For example,

assuming $d(e_{ab}) = 1$ is the delay function of the edge from task a to b, which means the task b in the i^{th} iteration depends on the task a in the $(i-1)^{th}$ iteration. In a cyclic DFG, edges without delay represent the intra-iteration data dependencies, while the edges with delays represent the inter-iteration dependencies. An example of a cyclic DFG is shown in Fig. 2.2(a) where one delay is denoted as a bar. There is a real-time constraint L, which is the deadline of finishing one period of the application. To generate a schedule of tasks in a loop, we use the static *direct acyclic graph* (DAG). A static DAG is a repeated pattern of an execution of the corresponding loop. For a given cyclic DFG, a static DAG can be obtained by removing all edges with delays.

Retiming is a scheduling technique for cyclic DFGs considering inter-iteration dependencies [17]. Retiming can optimize the cycle period of a cyclic DFG by distributing the delays evenly. For a given cyclic DFG G, the retiming function $r(G)$ is a function from the vertices set V to integers. For a vertex u_i of G, $r(u_i)$ defines the number of delays drawn from each of the incoming edges of node u_i and pushed to all of the outgoing edges. Let a cyclic DFG G_r be the cyclic DFG retimed by $r(G)$, then for a edge e_{ij}, $d_r(e_{ij}) = d(e_{ij}) + r(v_i) - r(v_j)$, where $d_r(e)$ is the new delay function of edge e_{ij} after retiming and $d(e_{ij})$ is the original delay function.

Energy model

We consider the CMP in which each core is featuring the DVFS technique. In order to reduce the energy consumption, the DVFS technique jointly decreases the processor speed and the supply voltage. Research in [43] shows that the decrease in processor voltage causes nearly linear increase in execution time and approximately quadratic decrease in energy consumption. Without loss of generality, we assume that each core has three DVFS modes, denoted as L_1, L_2 and L_3, respectively. L_1 has the slowest frequency and the lowest supply voltage, while the L_3 has the fastest frequency and the highest supply voltage. Note that our approach is general enough for the number of DVFS modes larger than four. Our

algorithms are not limited by the assumption of the DVFS modes numbers in the system.

Assume we know the power consumption and the execution time of different tasks running on different cores. We use a two-dimensional matrix EP to represent this information. We assume the CMP system has heterogeneous cores, which is a more general assumption compared to the homogeneous CMP. When applying our approach in the homogeneous CMP system, we only need to set execution time of a given task on every core as the same. There are two values in each entry of the EP matrix, one is execution time and the other is power consumption. For example, $ep_{ij} = \{e_{ij}, p_{ij}\}$ is one entry of the EP matrix. e_{ij} is the execution time of task i running on core j, while p_{ij} is the power consumption.

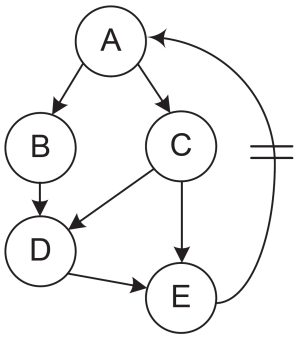
2.4 Motivational Example

An example of task scheduling in CMP

We first give an example of task scheduling in a multi-core chip. We schedule an application (see Fig. 2.2(c)) in a two-core embedded system. A DFG representing this application is shown in Fig. 2.2(a). There are two different cores in one layer. The execution times (t) and the stable state temperatures (T_{ss}) of each task in this application running on different cores are shown in Fig. 2.2(b). For simplicity, we provide the stable state temperatures instead of power consumptions in this example, and we assume the value of b (see equation (2.6)) in each core is the same: 0.025. We also assume the initial temperatures and the ambient temperatures are 50°C.

List scheduling solution

We first generate a schedule through the list-scheduling algorithm. Fig. 2.3(b) shows a static DAG, which is transformed from the DFG (see Fig. 2.3(a)) by removing the delay edge. For the DAG of this example, we can get the assigning order as {A, B, C, D, E}. For a task, we can calculate the peak temperatures when it is executed on different cores based on equation (2.5). Then tasks are assigned in a specific order to the core that can finish it



(a)

Task	P1		P2	
	t	Tss	t	Tss
A	95	84	65	92
B	70	80	60	78
C	60	95	65	82
D	80	82	80	70
E	50	75	75	82

(b)

```
for(i=2;i<N;i++) {
 A[i]=TaskA(E[i-2]);
 B[i]=TaskB(A[i]);
 C[i]=TaskC(A[i]);
 D[i]=TaskD(B[i],C[i]);
 E[i]=TaskE(C[i],D[i]); }
```

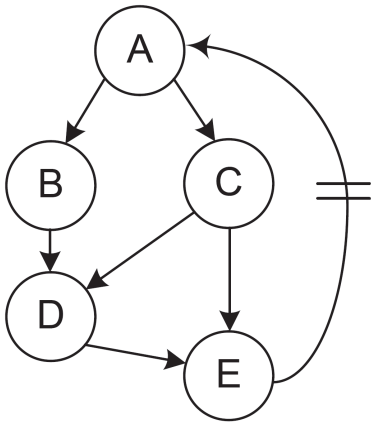
(c)

Figure 2.2: An example of task scheduling in a multi-core chip. (a) The DFG of an application. (b) The characteristics of the tasks. (c) The pseudo code of this application.

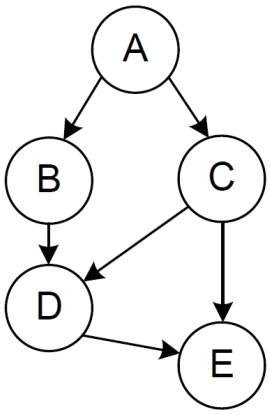
at the coolest temperature. In the list scheduling, a task assigning order is generated based on the node information in the DAG, and the tasks are assigned to the "coolest" cores in that order. A schedule is generated as Fig. 2.3(c). With the equation (2.5), we can get the peak temperature of each task as Fig. 2.3(d). Task A has the highest peak temperatures in the first two iterations. In the first iteration, task A starts at the temperature of 50°C and ends at the temperature of 80.84°C. In the second iteration, task A starts immediately after the first iteration of task E finishes, which means it starts at the temperature of 67.89°C. Since it has a higher initial temperature, the peak temperature (82.50°C) in this iteration is higher.

Our solution

Our proposed algorithm uses rotation scheduling to further reduce peak temperature. From the schedule in Fig. 2.3(c), we can find that Task A is the first tasks executed in core P0, and Task A has inter-iteration data dependency with Task E. In this case, we can implement the rotation scheduling and Task A is the proper candidate for rotation. In Fig. 2.4(a), we transform the original DFG into a new DFG by moving a delay from edge e_{EA} to edges e_{AB} and e_{AC}. The new corresponding static DAG is shown in Fig. 2.4(b). In this new DAG, there are two parts: node A and the rest nodes. There is no dependency between node A and the rest nodes. The new pseudo code of this new DFG is shown in Fig. 2.4(c),



(a)



(b)

time	P0		P1	
	task	Temperature	task	Temperature
0~95	A	50~80.84	ID	50
95~155	ID	80.84~50.18	B	50~71.75
155~220			C	71.75~79.98
220~300			D	79.98~71.35
300~350	E	50.18~67.89	ID	71.35~56.11

(c)

Task	Peak temperature	
	Iteration 1	Iteration 2
A	80.84	82.50
B	71.75	71.88
C	79.98	80.01
D	71.35	71.35
E	67.89	67.89

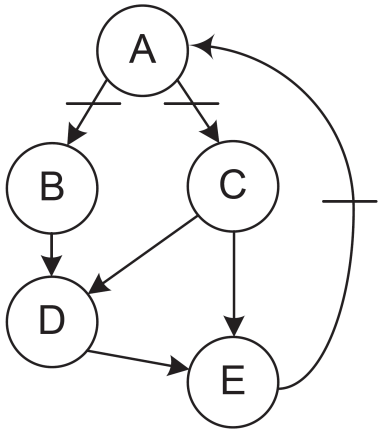
(d)

Figure 2.3: List Scheduling in a multi-core chip. (a) The DFG. (b) The static DAG. (c) The schedule generated by list scheduling. (d) The peak temperature (°C) of each task.

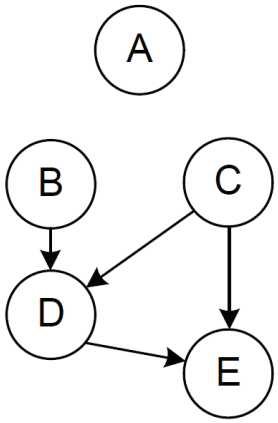
where the operation “A[i+1]=TaskA(E[i-1]);” can be placed anywhere in the loop, due to its independence. More details of the rotation scheduling are shown in Algorithm 2.7 of Section 2.5.

In this case, we can first assign the dependent nodes (B to E) to cores with the same policy used in the list scheduling. Tasks B, C and D are assigned to core P1 at the time slot of [0, 205]. And task E is scheduled to run on core P0 at [205, 255]. In this partial schedule, we discover that there are three time slots at which we can schedule task A. One is the idle gap of core P0 at [0, 205], another is the time slot after task E is done (time 255) on P0, and the last one is time slot after task D (time 205) on P1. Because the peak temperature of task A is the lowest when running in the idle gap of core P0 at [0, 205], this time slot is selected. Task A runs after the last iteration of task E, so the longer the idle gap

between them, the cooler the initial temperature at which task A starts. Thus, we schedule task A's starting time at 110. A schedule is shown in Fig. 2.4(d). In this schedule, the peak temperature is 81°C when task A is running in the second iteration (see Fig. 2.4(e)). Our approach reduces the peak temperature by 1.5°C. Moreover, the total execution time of one iteration is only 255, while the total execution time generate by list scheduling is 350.



(a)



(b)

```
A[2]=TaskA(E[0]);
for(i=2;i<N-1;i++) {
    B[i]=TaskB(A[i]);
    C[i]=TaskC(A[i]);
    A[i+1]=TaskA(E[i-1]);
    D[i]=TaskD(B[i],C[i]);
    E[i]=TaskE(C[i],D[i]); }
B[N]=TaskB(A[N]);
C[N]=TaskC(A[N]);
D[N]=TaskD(B[N],C[N]);
E[N]=TaskE(C[N],D[N]);
```

(c)

time	P0		time	P1	
	task	Temperature		task	Temperature
0~110	ID	50	0~60	B	50~71.75
			60~125	C	71.75~79.98
110~205	A	50~80.04	125~205	D	79.98~71.35
205~255	E	80.04~67.89	205~255	ID	71.35~56.11

(d)

Task	Peak temperature	
	Iteration 1	Iteration 2
A	80.84	81.00
B	71.75	73.12
C	79.98	80.25
D	71.35	71.39
E	76.67	76.72

(e)

Figure 2.4: Rotation Scheduling in a multi-core chip. (a) The retimed DFG. (b) The new static DAG. (c) The pseudo code of the retimed DFG. (d) The schedule generated by our proposed algorithm. (e) The peak temperature (°C) of each task.

In the next section, we will discuss our thermal-aware task scheduling algorithm with more details.

2.5 Thermal-aware task scheduling algorithm

In this section, we propose an algorithm, TARS (*Thermal-Aware Rotation Scheduling*), to solve *the minimum peak temperature without violating real-time constraints problem.* By

repeatedly rotating down delays in DFG, more flexible static DAGs are generated. For each static DAG, a greedy heuristic approach is used to generate a schedule with minimum peak temperature. Then the best schedule is selected among the schedules generated previously.

The TARS Algorithm

Algorithm 2.1 The TARS algorithm

Input: A DFG, the rotation times R.
Output: A schedule S, the retiming function r.

1: rot_cnt $\leftarrow$ 0 /*Rotation counter.*/
2: Initial S_{min}, r_{min}, PT_{min}, r_{cur} /*The optimal schedule, the according retiming function, the according peak temperature and the current retiming function*/
3: **while** rot_cnt $<$ R **do**
4: Transform the current DFG to a static DAG
5: Schedule tasks with dependencies. /* using the PTMM algorithm or PTLS algorithm */
6: Schedule independent tasks, using the MPTSS algorithm
7: Scale the frequencies, using the PPS algorithm /* A schedule S_{cur} for the current DFG is generated */
8: Get the peak temperature PT_{cur} of the current schedule
9: **if** $PT_{cur} < PT_{min}$ and S_{cur} meets the real-time constraint **then**
10: $S_{min} \leftarrow S_{cur}, r_{min} \leftarrow r_{cur}, PT_{min} \leftarrow PT_{cur}$
11: **end if**
12: Use RS algorithm to get a new retiming function r_{cur}
13: Get the new DFG based on r_{cur}
14: $R \leftarrow R + 1$
15: **end while**
16: Output the S_{min}, r_{cur}

In the TARS algorithm shown in Algorithm 2.1, we will try to rotate the original DFG by R times. In each rotation, we get the static DAG from the rotated DFG by deleting the delay edges in DFG. A static DAG usually consists of two kinds of tasks. One kind of tasks are the tasks with dependencies, like the tasks B, C, D, and E in Fig. 2.4(b). The other kind of tasks are the independent tasks, like the task A in Fig. 2.4(b). The independent tasks do not have any intra-iteration relation with other tasks. Below, we first present two algorithms, the PTMM algorithm and the PTLS algorithm, to assign tasks with dependencies.

The PTMM algorithm

The *Peak Temperature Min-Min* (PTMM) algorithm is designed to schedule the tasks with dependencies. Min-Min is a popular greedy algorithm [44]. The original Min-Min algorithm does not consider the dependencies among tasks. Therefore, in the Min-Min baseline algorithm used in this chapter, we need to update the assignable task set in every step to maintain the task dependencies. We define *the assignable task* as the unassigned task whose predecessors all have been assigned. Since the temperatures of the cores in a core stack are highly correlated in 3D CMP, we need to schedule tasks with consideration of vertical thermal impacts. When we consider assigning a task T_i to core C_j, we calculate the peak temperatures of cores in the core stack of C_j during the T_i running on C_j, based on the equation (2.8).

Let $T_{max}(i,j)$ be the maximum value of the peak temperatures in the core stack. When we decide the assigning of T_i, we calculate all the $T_{max}(i,j),\ for\ j = every\ core$. Due to the fact that the available times and the power characteristics of different cores in the same core stack may not be identical, the peak temperatures of the given core stack may be various when assigning the same task to different cores of this core stack respectively. Let C_{min} be the core with minimum $T_{max}(i,j)$. In each step in PTMM, we first find all the assignable tasks. Then we will form a pair $<T_i, C_{min}>$ for every assignable task. Only the $<T_i, C_{min}>$ pair which gives the minimum $T_{max}(i,j)$ will be assigned accordingly. And we also schedule the start execution time of T_i as the time when the predecessors of T_i are done and core C_{min} is ready. The PTMM is shown as Algorithm 2.2.

The PTLS algorithm

The *Peak Temperature List Scheduling* (PTLS) algorithm is another algorithm that we use to schedule the tasks with dependencies. In the PTLS, we first list the tasks in a priority list considering the data dependencies (see the Algorithm 2.3). Some definition used in the *Task Listing* (TL) algorithm is provided as following. The *Earliest Start Time* (EST)

Algorithm 2.2 The PTMM algorithm

Input: A static DAG G, m different cores, EP matrix.
Output: A schedule generated by PTMM.

```
1: Form a set of assignable tasks P
2: while P is not empty do
3:    for t = every task in P do
4:       for j = 1 to m do
5:          Calculate the peak temperatures of cores in the core stack of C_j, assuming t is
            running on C_j. And find the minimum peak temperature T_max(t, j)
6:       end for
7:       Find the core C_min(t) giving the minimum peak temperature T_max(t, j)
8:       Form a task-core pair as <t, C_min(t)>
9:    end for
10:   Choose the task-core pair <t_min, C_min(t_min)> which gives the minimum
      T_max(t, C_min(t))
11:   Assign task t_min to core C_min(t_min)
12:   Schedule the start time of t_min as the time when all the predecessors of t_min are
      finished and C_min(t_min) is ready
13:   Update the assignable task set P
14:   Update time slot table of core C_min(t_min) and the expected finish time of t_min
15: end while
```

and the *Latest Start Time* (LST) of a task are shown as in equation (2.9) and (2.10). The entry-tasks have EST equals to 0. And the LST of the exit-tasks equal to their EST.

$$EST(i) = \max_{m \in pred(i)} \{EST(m) + AT(m)\} \tag{2.9}$$

$$LST(i) = \min_{m \in succ(i)} \{LST(m)\} - AT(i) \tag{2.10}$$

where $AT(i)$ is the average execution time of task i. The critical node (CN) is a set of vertices in the DAG of which EST and LST are equal.

After a priority list is generated, we assign the tasks, in the order of the priority list, to the core with the minimum peak temperature (see the Algorithm 2.4).

The MPTSS algorithm

Using one of the PTMM and the PTLS algorithm, we can get a partial schedule, in which the tasks with dependencies are assigned and scheduled. We need to further assign the

Algorithm 2.3 The TL algorithm

Input: A static DAG, Average execution time AT of every task in the DAG.
Output: An assigning order of tasks P.

/*List tasks with dependencies*/
Calculate the EST and the LST of every task which has dependencies
Empty list P and stack S, and pull all tasks with dependencies in the list of task U
Push the CN task into stack S in the decreasing order of their LST, and remove them from U
while The stack S is not empty **do**
  if *top*(S) has immediate predecessors in U **then**
    $S \leftarrow$the immediate predecessor with least LST
    Remove this immediate predecessor from U
  else
    $P \leftarrow$ *top*(S)
    Pop *top*(S)
  end if
end while
/*List independent tasks*/
Push independent tasks in P in the decreasing order of their power consumptions.

Algorithm 2.4 The PTLS algorithm

Input: An priority list of tasks with dependencies P, m different cores, EP matrix.
Output: A schedule generated by MPT.

while The list P is not empty **do**
  $t =$ *top*(P)
  for $j = 1$ to m **do**
    Calculate the peak temperatures of cores in the core stack of C_j, assuming t is running on C_j. And find the minimum peak temperature $T_{max}(t, j)$
  end for
  Find the core C_{min} giving the minimum peak temperature $T_{max}(t, j)$
  Assign task t to core C_{min}
  Schedule the start time of t as the time when all the predecessors of t are finished and C_{min} is ready
  Remove t from P
  Update time slot table of core C_{min} and the expected finish time of t
end while

independent tasks in the static DAG. Since the independent tasks do not have any intra-iteration relations with others, they can be scheduled to any possible time slots of the cores. In the *Minimum Peak Temperature Slot Selection* (MPTSS) algorithm, we assign the independent tasks in the decreasing order of their power consumption. Tasks with larger power consumption likely generate higher temperatures. The higher assigning orders of these tasks, the better fitting cores these tasks will be assigned to, and probably the lower resulting peak temperature of the finial schedule.

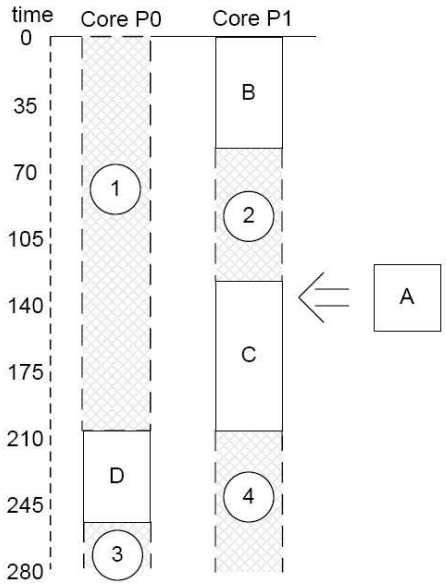


Figure 2.5: An example of time slot set for an independent task

Before we assign an independent task A, as shown in Fig. 2.5, we first find all the idle slots among all cores, forming a time slot set TS. In the example shown in Fig. 2.5, there are four time slots indicated with circled numbers for task A. Two of them, i.e., time slot 1 and 2, are among the previously scheduled tasks. And the other two, i.e., time slot 3 and 4, are at the end of cores' schedules of one iteration. The time slots that are not long enough for the execution of A will be removed from TS. Then we calculate the peak temperature of the according core stack $T_{max}(A, core)$, which is defined in the PTMM algorithm, for every time slot. One problem arise here: since the remain time slots are long enough for the execution of A, we need to decide when to start the execution.

We use two different schemes here. The first one is the *As Early As Possible* (AEAP), which means the task T_i should be scheduled to start at the beginning of that time slot. The other one is *As Late As Possible* (ALAP), which means we should schedule the start execution time of the task T_i at a certain time so that T_i will finish at the end of the time slot. These two schemes result in different impacts on peak temperature.

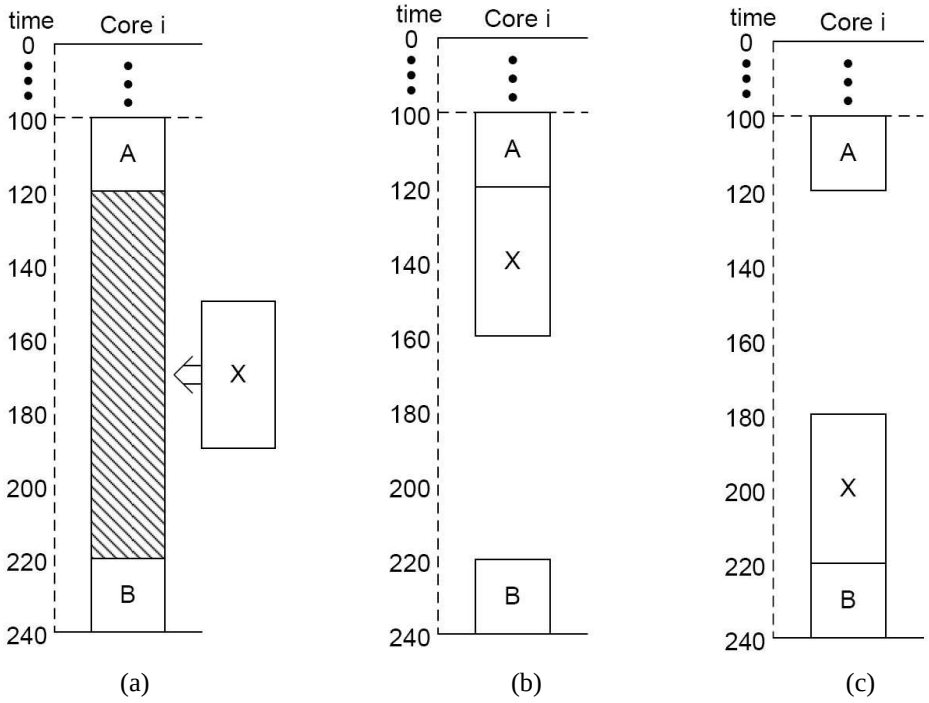


Figure 2.6: An example of the AEAP scheme and the ALAP scheme. (a) The task X is scheduled in a time slot in core i, (b) The task X is scheduled by the AEAP scheme, (c) The task X is scheduled by the ALAP scheme.

Let us assume we are considering scheduling task X to core i in the time slot, which is shown as a shadowed rectangle in Fig. 2.6(a), and tasks A and B are previously scheduled on the beginning and the end of this time slot on core i. The AEAP scheme generates a time gap between X and B, as shown in Fig. 2.6(b). The temperature of core i can be cooled down during this time gap, i.e., 160 to 220. The ALAP scheme schedules X right before B without any time gap, as shown in Fig. 2.6(c). So the initial temperature of B is lower with the AEAP scheme, i.e. the schedule in Fig. 2.6(b), than with the ALAP scheme,

i.e. the schedule in Fig. 2.6(c), due to the cooling time gap (160 to 220) between the tasks X and B.

Given a certain execution time of B, lower initial temperature leads to lower peak temperature. In addition, if the power consumption of B is higher than the power consumption of X, the peak temperature of B is likely higher than the one of X, which means we should try to cool down B rather than X in this case. Implementing the AEAP in scheduling X can cool down the X at most here. On the other hand, the ALAP can create a time gap between X and the task A that is previously scheduled right before the time slot. This time gap, e.g., the time gap 120 to 180, can reduce the initial temperature of X. So in the case where the power consumption of X is higher than the one of B, using ALAP can reduce the peak temperature of X. Thus, when we consider scheduling a task to a time slot, we will compare the power consumption of this task and the task previously scheduled right after this time slot. If the task being scheduled has more power consumption, we will use the ALAP scheme. Otherwise, the AEAP scheme will be implemented.

When we try to schedule tasks to the time slots which locates at the end of cores' schedules, we will determine which scheme, either AEAP or ALAP, will be used based on the power consumption comparison of this task and the task that will start first in the next iteration. For example, in Fig. 2.5, when we try to schedule task A to time slot 4, we will compare the power consumptions of task A and B. We will schedule a large enough time slot for cooling down the task that needs more concern, i.e., the more power consuming one between the task to be scheduled and the task starting first in the next iteration.

Another question arises: how large the cool time slot should be scheduled? We will predetermine a threshold cooling temperature T_c. Then we will create a cooling time slot large enough to let the more power consuming task cooling down to the threshold T_c, without violating the real-time constraint. The reason that we set the threshold temperature is that when the temperature of a core is cooling down, it drops dramatically at the beginning, as shown in Fig. 2.7. However, it becomes stable as the core continues to cool down. Hence, if

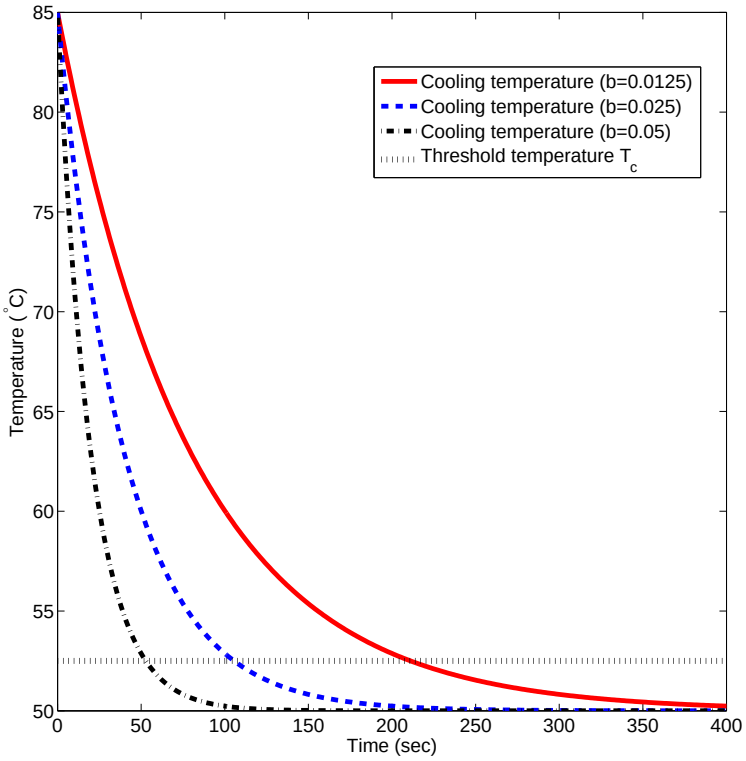


Figure 2.7: Examples of cooling temperature on-chip. All three cooling temperatures start from the initial temperature of $85°C$ to the stable temperature of $50°C$. We can observe that the cooling speeds in these three scenarios are slowing down dramatically near the threshold temperature T_C.

we try to cool down the core completely, it will take a significantly long time. As shown in Fig. 2.7, if we just need to reduce the core's temperature to the threshold, i.e., the horizontal dot line, it will be more time-efficient. We present our MPTSS algorithm in Algorithm 2.5.

The PPS algorithm

Once we get a full schedule from the previous steps, we can further reduce the peak temperature by dynamic frequency assignment. We assume that the frequencies of different cores can be different and there are several frequencies options available for each core. From a given schedule, we can predict the task which causes the peak temperature. We can further decrease the peak temperature by changing the frequency assignment of the corresponding

Algorithm 2.5 The MPTSS algorithm

Input: A partial schedule generated by PTMM, a set of independent tasks, m different cores, EP matrix.

Output: A schedule generated by MPTSS.

1: List independent tasks in a list P in the decreasing order of their power consumption
2: **while** The list P is not empty **do**
3: $t = top(P)$
4: Collect all the time slots which is long enough for t across all cores, form a time slot set TS.
5: **for** Every time slot ts_i in TS **do**
6: $j \leftarrow$ the according core of ts_i
7: Find the task t_{next} which is schedule to start right after ts_i on the core C_j.
8: **if** $Power(t) < Power(t_{next})$ **then**
9: Find the start time with the AEAP scheme
10: **else**
11: Find the start time with the ALAP scheme
12: **end if**
13: Get the $T_{max}(t, j)$ /*similar to the one in PTMM*/
14: **end for**
15: Find the time slot ts_{min} giving the minimum peak temperature $T_{max}(t, j)$
16: Assign task t to core C_{min} /*C_{min} is the core of time slot ts_{min}*/
17: Schedule the start time of t in time slot ts_{min} based on the scheme selected in the if statement (line 8)
18: Remove t from P
19: Update time slot table of core C_{min}
20: **end while**

core when that task is running.

We propose our dynamic frequency assignment algorithm, called the *Peak Point Scaling* (PPS), in Algorithm 2.6. Given a schedule, we first find the task with the highest peak temperature over all the tasks. Then the core frequency when running this task is set to one slower level. We calculate the period of this new schedule. If it meets the real-time constraint, this new schedule is acceptable. Otherwise, dynamic frequency scaling cannot reduce the peak temperature. If the new schedule is acceptable, then we find the task with the highest peak temperature in the new schedule, and repeat the frequency scaling again. This frequency scaling repeats until a schedule which violates the real-time constraint is generated. We output the last version of the acceptable schedules.

Algorithm 2.6 The PPS algorithm

Input: An initial schedule S_{init}, EP matrix, a real-time constraint TC
Output: A schedule generated by PPS.

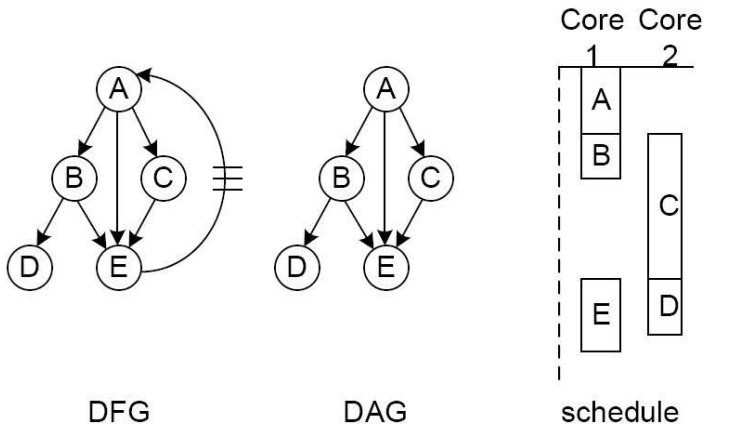
1: $S_{temp} \leftarrow S_{init}$
2: **while** $Period(S_{temp}) \leq TC$ **do**
3: $S \leftarrow S_{temp}$
4: Find the task t_{max} generating the highest peak temperature in S_{temp}, and the core C_{max} which runs t_{max}
5: **if** frequency of C_{max} when running t_{max} is the slowest level **then**
6: Break
7: **end if**
8: Set the frequency of C_{max} when running t_{max} to one slower level
9: Update S_{temp}
10: **end while**
11: Output S

The RS algorithm

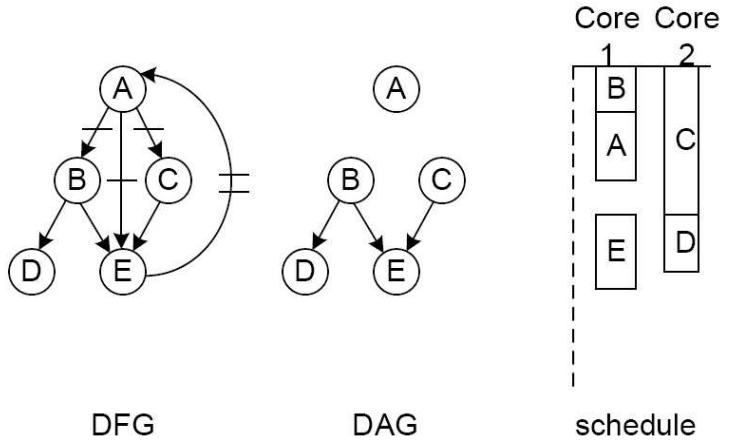
At the end of each iteration of the TARS algorithm, we create a new DFG by rotating the current DFG. First, we need to form a set of rotation tasks. If a task is the first task scheduled on a core and there is at least one delay in each of its incoming edge, this task is a rotation task. The *Rotation Scheduling* (RS) algorithm is shown in Algorithm 2.7.

Fig. 2.8 shows an example of our RS algorithm. Assuming an initial DFG shown in Fig. 2.8(a), we can transform the DFG into DAG by removing the edges with delays. Then a schedule is generated by the algorithms presented in the previous subsections.

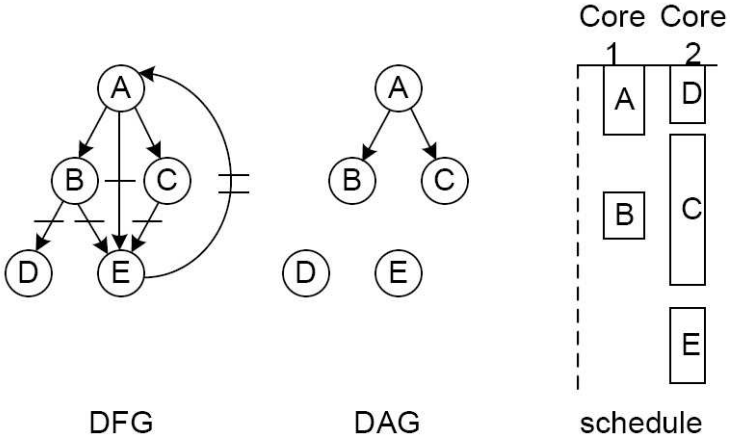
In the first rotation, we can find the task A and C are the first tasks executed in two cores. So the rotation task set includes these two tasks. Since there is none delay on the incoming edge and the outgoing edge of task C, we keep the edges of task C unchanged. For task A, there are three delays on its incoming edge, i.e. edge e_{EA}. Thus, in this rotation, we reduce one delay on edge e_{EA}, and increase the delays of all three outgoing edges of task A by one, respectively, as shown in Fig. 2.8(b). We can find that task A now becomes independent in the corresponding DAG. A new schedule is generated based on this new DAG. In this schedule, task B and C are the first tasks in two cores. These two tasks form the set of rotation tasks in the next rotation.



(a)



(b)



(c)

Figure 2.8: An example of the rotation scheduling. (a) The initial DFG, the corresponding DAG and schedule. (b) The rotated DFG in the first rotation, the corresponding DAG and schedule. (c) The rotated DFG in the second rotation, the corresponding DAG and schedule.

In the second rotation, the delays of the incoming edges of task B and C, i.e., e_{AB}, e_{AC}, are all reduced by one. The outgoing edges of task B and C, i.e., e_{BD}, e_{BE}, and e_{CE}, increase their delays by one, as shown in Fig. 2.8(c). According to this new DFG, task D and E become independent. The third schedule is created in this rotation.

As shown in this example, the RS algorithm can redistribute the delays in the DFG. Therefore, various DAGs can be reached. In these various DAGs, different tasks become independent, which leads to diverse scheduling orders of tasks and different schedules. As we implement the RS algorithm at the end of each iteration of our TARS algorithm, and we repeat the TARS algorithm for a pre-determined number of iterations, we can select the rotations with the best schedule among a number of schedules in the sense of reducing peak temperature.

Algorithm 2.7 The RS algorithm

Input: An input DFG D_{in} and a schedule S based on D_{in}, a retiming function r.

Output: An output DFG D_{out} generated by rotation scheduling, a new retiming function r_{new}.

1: Form the set of rotation tasks RT based on D_{in} and S
2: **for** Every task t_i in RT **do**
3: Reduce one delay from every incoming edges of task t_i in D_{in}
4: Increase one delay from every outgoing edges of task t_i in D_{in}
5: $r(t_i) \leftarrow r(t_i) + 1$
6: **end for**
7: $D_{out} \leftarrow D_{in}$ and $r_{new} \leftarrow r$

2.6 Experimental results

In this section, we present the experimental results of our algorithms. We develop our experiments as follows: we first use a precise microprocessor simulator, Wattch 1.0.2 [45], to get the execution and power characteristics of a set of benchmarks. Then we generate a number of random DFGs consisting of this set of benchmarks. Task schedules and power traces are created by our algorithm. We input these schedules and power traces into a thermal analysis simulator, called Hotspot 4.1 [46]. Finally, we evaluate our algorithms

with the comprehensive thermal analysis generated by Hotspot 4.1. All experiments are conducted on Linux machine equipped with an Intel Core 2 Duo E8400 CPU and 3GB of RAM.

Experiment setup

The 3D CMP architecture simulated in our experiments is a two-layer front-to-back architecture. There are eight Alpha 21264 (EV6) microprocessor cores in each layer with configuration as Table 2.1. We use per core DFVS in our simulation with three DVFS levels (3.88GHz, 4.5GHz, and 5 GHz) configured based on the parameters of Alpha 21264 [47].

Table 2.1: Configuration of Alpha cores

Processor core	Alpha 21264
Core technology	65nm
Nominal frequency	5GHz
L1 data cache	64K, 2-way
L1 instruction cache	64K, 2-way
L2 cache	2M

We choose the SPEC CPU 2000 benchmark suite and the MiBench benchmark suite in our experiment. The execution time and the power consumption of each benchmark on Alpha core are tested through the Wattch 1.0.2 simulator with the above configuration. For each benchmark, we run it under those three DFVS levels via out-of-order mode to get the task characteristic of this benchmark. We generate 10 random DFG-based applications. The tasks in these applications are randomly selected from the SPEC2000 and the MiBench benchmarks. For each application, we set the real-time constraint TC (i.e., deadline) as follows:

$$TC = \frac{\sum_{i=1}^{N} t_i}{P} \times c \tag{2.11}$$

where N is the number of tasks in this application, t_i is the execution of time of task i under the highest frequency, P is the total number of cores, i.e., 16 in our simulation, and

c is a constant which is set to 5, generating neither too tight nor too loose constraints.

Table 2.2: Thermal parameter for Hotspot

Layer	Conductivity	Capacitance per unit volumn
Silicon	100 $W/(m \cdot K)$	1.75 $\times$ 10^6 $J/(m^3 \cdot K)$
TIM	4 $W/(m \cdot K)$	4.0 $\times$ 10^6 $J/(m^3 \cdot K)$
Copper	400 $W/(m \cdot K)$	3.55 $\times$ 10^6 $J/(m^3 \cdot K)$

The thermal simulation is conducted in the Hotspot 4.1 simulator by using the power consumption traces created by our program. In the Hotspot 4.1 simulator, the lateral and vertical thermal interactions among adjacent core are all carefully considered and modeled. As we mentioned above, the architecture model used in the Hotspot simulator is a two-layer architecture, in which the thickness of the top layer (the one far from the heat sink) is 50μm, and the thickness of the bottom (the one close to the heat sink) is 300μm. There is a *Thermal Interface Material* (TIM) layer between these two layers. The core size is 4mm $\times$ 8mm. Some other parameters is listed in Table 2.2. We also set the temperature parameters as shown in Table 2.3 [48].

Table 2.3: Temperature parameter setting

Temperature parameter	Value
Ambient temperature	$35°C$
Initial temperature	$55°C$
Critical temperature	$85°C$

Peak temperature

As our algorithms are to reduce the peak temperature in 3D CMP architectures, we show the average peak temperature of all 16 cores over 10 applications in Fig. 2.9. By comparing the result of list scheduling, we find that both of our algorithms can reduce the peak temperatures. The PTLS based TARS reduces up to 7°C. And the PTMM based TARS is even

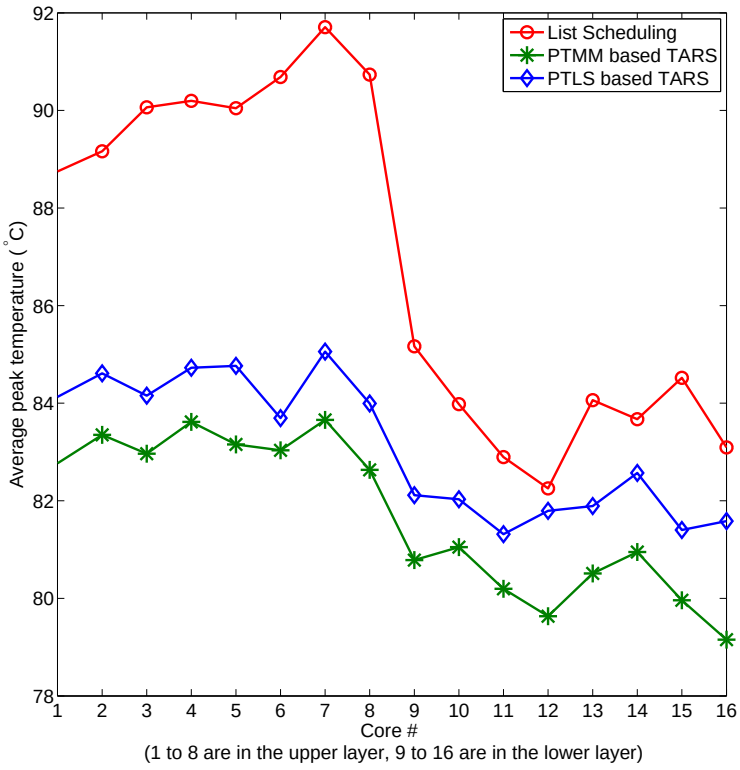


Figure 2.9: Core peak temperatures comparison. The "Core #" in the x-axle represents the IDs of the sixteen cores, where cores 1 to 8 are in the upper layer and the cores 9 to 16 are in the lower layer.

better, reducing up to 8.1°C. Both the peak reductions happen on the cores in the upper layer. For the cores in the top layer (core 1 to 8), the peak temperatures are consistently higher than the ones in bottom layer (core 9 to 16). This result is aligned to our online thermal prediction model. The peak temperatures of top layer cores is around 83°C with our PTMM based TARS algorithm, about 84.5°C with our PTLS based TARS algorithm, and about 90°C with the list scheduling. With the two phases consideration in the PTMM, i.e., the Min-Min initial scheduling algorithm, more global information is used in making the assigning decisions. Thus it generates better initial schedules leading to better performance than our PTLS based TARS algorithm does.

Larger improvements are made in the top layer cores. The reason is that in our proposed algorithm, more effort is made in reducing the temperature of the hottest core, which is

usually located in the top layer. Even though the improvements for cores in the bottom layer are not as significant as the ones in top layer, lower peak temperatures are achieved, due to the more flexible execution order explored in our algorithm and less impact from the aligned cores on the top layer. The reduction of peak temperature in the bottom layer is about 4.5°C with our PTMM based TARS algorithm, about 3.1°C with our PTLS based TARS algorithm.

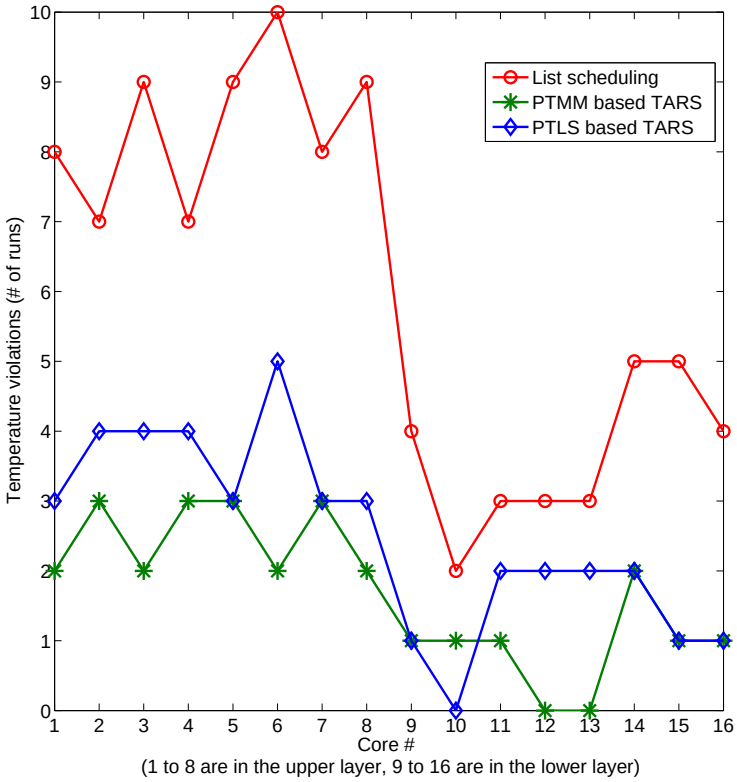


Figure 2.10: Core temperature violations comparison. The "Core #"s in the x-axle represent the IDs of the sixteen cores, where cores 1 to 8 are in the upper layer and the cores 9 - 16 are in the lower layer. Out of the 10 runs in the experiment, the temperature violations are number of runs in which the corresponding core has the peak temperature higher than the temperature constraint.

Temperature violations

In this section, we compare the schedules in the sense of avoiding or minimizing the number of temperature violations, which is shown in Fig. 2.10. We define the temperature violation as the situation where the core's temperature is higher than the critical temperature. The differences of temperature violations of cores depend on a few factors, such as the workloads of cores, the location relationship with other cores. The cores 5, 6, and 7 have more temperature violations than that of cores 10-13. The reason is that the cores 5, 6, and 7 is on the upper layer of the 3D CMP. The cores in the top layer are more likely to have higher temperature than the critical temperature. Since more efforts are made to reduce the temperature of the hottest core in our TARS algorithms, our TARS algorithms can dramatically reduce the number of times of temperature violations in the top layer cores. Up to 80% temperature violations in the list scheduling are avoided in the top layer. Aligned to the result of the above subsection, the PTMM based TARS algorithm outperforms the PTLS based TARS algorithm.

For the cores in the bottom layer, only a small number of of violations occur. In both TARS algorithms, there is one core that never has temperature higher than the critical cores. No more than two violations happen in any core in the bottom layer. In summary, both our TARS algorithms can reduce the temperature violations in both the top layer and the bottom layer.

2.7 Conclusion

In this chapter, we presented an online 3D CMP temperature prediction model for multimedia embedded systems. We also proposed our real-time constrained task scheduling algorithms, the TARS algorithms, to reduce peak temperature in a 3D CMP. By considering the the inter-iteration data dependencies and frequencies assignment collaboratively, our proposed TARS algorithms can significantly reduce the peak temperature on chip and avoid most of the temperature violations. Our simulation results showed that our TARS

algorithms can reduce peak temperature by 8.1°C, and avoid up to 80% violations in the top layer and up to 100% violations in the bottom layer.

Our future works are two-fold: 1) we will investigate the implementation of stochastic approaches in our CMP temperature prediction models; and 2) we will also further consider the priorities of tasks in our task scheduling algorithms.

Chapter 3 ILP memory activities optimization algorithm

Phase Change Memory (PCM) is emerging as one of the most promising alternative technology to the *Dynamic RAM* (DRAM) when building large-scale main memory systems. Even though the PCM is easy to scale, it encounters serious endurance problems. Writes are the primary wear mechanism in the PCM. The PCM can perform 10^8 to 10^9 times of writes before it cannot be programmed reliably. In addition, the PCM has high write latency. To prolong the lifetime of the PCM as the main memory and enhance the performance, we propose a *Scratch Pad Memory* (SPM) based memory mechanism and an *Integer Linear Programming* (ILP) memory activity scheduling algorithm to reduce the redundant write operations in the PCM. The idea of our approach is to share data copies among the SPMs, instead of writing back to the PCM main memory each time when a modify occurs. Our experimental results show that the ILP scheduling algorithm can generate the optimal schedule of memory activities with minimum write operations, reducing the number of write operations by 58% on average.

3.1 Introduction

Dynamic RAM (DRAM) has been the most widely used technology of the main memory for over three decades. However, the main memory that consists of entirely DRAM is already reaching the power and scalability limits [7]. As memory demands increase, the main memory has now become quite large. It has become one of the primary energy consuming parts of the embedded system [4, 5]. For example, 2GB of DRAM consumes 3W to 6W, which is equivalent to the total power consumption of the Atom processor [**?**]. Besides, DRAM also has the scalability issue. Due to some properties of DRAM, such as destructive reads and low retention time, some specific architecture solutions, such as, write after read operations and the refresh control, are implemented [6]. These extra costs

limit the scalability of DRAM.

New techniques, such as *Phase-Change Memory* (PCM) [10] and Magnetic RAM (MRAM) [49], have been studied for the replacement of the DRAM main memory [5]. PCM is a potential alterative of the DRAM main memory, due to its many desirable properties [6]. PCM is a non-volatile memory that switches its chalcogenide material between the amorphous and the crystalline states. By detecting the resistances of different states, data is stored in PCM devices. The application of heat that is required by the switch between states can be provided by using electrical pulses.

In the PCM write, it relies on analog currents and thermal effects, which means it does not require control over discrete electrons [12]. In addition, another attracting property of PCM is that multiple bits can be stored in one single PCM cell, called *Multi-Level Cell* (MLC). PCM can provide four times more density than DRAM [10]. Researchers have stated that PCM has more robust scalability beyond 40 nm than DRAM does [12]. In addition, a 32-nm device prototype has been demonstrated [13].

Even though PCM is alternative to DRAM as main memory, large efforts are needed to surmount the disadvantage of PCM, compared to DRAM. PCM access latencies, especially in writes, are much slower than those of DRAM. In the read access, PCM is 2x-4x slower than DRAM. Moreover, PCM displays asymmetric timings for reads/writes, which means writes in PCM need 5x-10x more time than reads do. Due to the fact that phase changes in PCM are induced by injecting current into the chalcogenide material and heating it, thermal expansion and contraction in the chalcogenide material make the programming current injection no longer reliable [12]. Writes are the primary wear mechanism in PCM. The number of writes performed before the cell is not able to perform reliably ranges from 10^8 to 10^9. Therefore, writes in PCM limits both the performance and the lifetime of PCM.

In the embedded system design field, more and more processors are equipped with the *Scratch Pad memory* (SPM), such as Motorola Mcore [50], Texas Instruments TMS370Cx [51], Motorola 68HC12 [52], etc. The SPM is a small size on-chip memory mapped into the

memory address space disjoint from the off-chip memory, such as the PCM main memory. The SPM memory is managed by the application software or automated compiler support [53]. Compared to a hardware-managed cache memory, the SPM of the same capacity are 34% smaller in term of size, and 40% lower power consumption [53]. From the memory activities optimization point of view, the SPM memory has two attracting advantages: 1) it is easier to manage without hardware modification, compared to cache memory; and 2) it guarantees the single-cycle access latency, much shorter than that of the off-chip memory.

In this chapter, we propose a PCM main memory optimization mechanism through the utilization of SPM. The major contributions of this chapter include:

- We propose a PCM main memory architecture with the SPM. Each core in the *chip multiprocessors* (CMP) is equipped with an SPM memory. All SPMs are connected to the PCM main memory controller via on-chip data buses. Data copies are shared among SPMs via on-chip data buses. The sharing copies of data can benefit the endurance of the PCM main memory by eliminating unnecessary writes

- An *Integer Linear Programming* (ILP) memory activities scheduling algorithm is proposed to minimize the number of writes in PCM. There are three major parts in our algorithm: the baseline scheduling, the ILP-based memory activities scheduling, and the post ILP procedure. The baseline scheduling generates a baseline schedule for both task executions and SPM assignments. Then, the ILP-based memory activities scheduling will find the optimal memory activities strategy to minimize the memory writes based on the baseline scheduling. Finally, the post ILP procedure will further reduce total execution time by eliminating idle slots in the schedule. Our ILP memory activities scheduling algorithm reduces the writes by 58% on average.

Memory activities optimization through the utilization of the SPM is a challenging problem. First of all, to minimize the number of write operations, data need to be shared

among SPMs by data migrations. In some cases, multi-hop data migrations, which are necessary for optimal memory activities optimization, cannot be well scheduled by greedy scheduling algorithms. Compared to greedy scheduling algorithms, our ILP method is more promising, because it explores a larger solution space. However, modeling the memory activities scheduling problem through the utilization of the SPM is more sophisticated than the existing ILP-based memory optimization problems [54, 55]. The size of the SPM is much less than the size of the main memory, resulting in the stricter SPM size constraint in the problem. Since the SPM space is limited, the optimization method should decide not only which copies of data should be kept, but also how long the SPM should keep these copies. Moreover, due to data sharing operations among SPMs, there are more kinds of memory activities to schedule than that in the existing ILP memory optimization methods. For example, to have a copy of data in a given SPM, there are three ways: loading the data from the PCM main memory to the SPM; outputting the data from the core to the SPM; and copying the data from a remote SPM via the data migration, which is either for the input requirement of the next task, or just temporary stored for future data migrations. Since copies of data are sharing among SPM via the on-chip network, data migration activities are also subject to the bandwidth of the network. Data dependencies across tasks further complicate the memory activities scheduling. Memory activities should not violate any data dependency. In this chapter, we present a comprehensive ILP format that covers different kinds of PCM memory activities when utilizing the SPMs. System and application constraints, such as the size of SPM, the on-chip network bandwidth, and data dependencies, are formulated in our ILP algorithm.

In Section 3.2, we discuss works related to this topic. In Section 3.3, the background knowledge of phase change memory is presented. An illustrating example is given in Section 3.4. We propose our algorithms in Section 3.5, followed by experimental results in Section 3.6. Finally, we conclude in Section 3.7.

3.2 Related work

For CMPs, the problem of scheduling tasks represented by a DAG is NP-complete. A number of heuristics were compared in [56]. An unbalanced thread scheduling method was proposed to fully utilize the advantage of CMP architecture, which allocates the right amount of resources to each thread [57]. Dhiman et al. presented power-aware scheduling mechanisms and policies for CMP at the operating system level, to improve the system performance per watt [58]. Another scheduling approach was introduced in [59], based on the execution phases of simultaneous threads. An operating system scheduler design was presented for CMPs, especially the network-on-chip architecture [60], which is based on the on-chip data traffic calculation of applications. Teodorescu et al. proposed a power-aware scheduling mechanism for CMP with the consideration of variation effects on the static power consumption and the maximum supported frequency [61]. However, the related works above mainly focused on the scheduling in CMP. The activity optimization in memory was not studied in these papers. In this chapter, we combine the task scheduling and the memory activity optimization for the CMP system, improving not only the performance of the system, but also the lifetime of the PCM memory.

The PCM incorporated in the memory hierarchy was studied in [62]. A DRAM based *page cache* was implemented for a large PCM memory. This page cache not only enhances the performance by buffering frequently used pages, but also improves endurance by reducing writes. Enhancement approaches, such as read-before-write, row-level rotation and segment swapping, were proposed to improve the lifetime of the PCM [9]. By rotating the cache line, the row-level rotation can distribute the row level wear evenly. In the segment swapping, the contents of the least-frequently-written page and the page being written are swapped. Lee et al. presented a PCM storage device with a bit level read-before-write loop [63]. Ferreira et al. described three lifetime enhancement methods for PCM: N-Chance victim selection replacement policy, bit level writes, and a swap management on page cache writebacks [6]. Although techniques introduced in these papers improve the en-

durance of the PCM, all of them require significant modifications in the hardware design. In this chapter, by utilizing the SPM in CMP, our optimization approach does not require hardware modifications.

A recent trend in PCM techniques has been focused on the MLC technique [8, 14–16]. In [14], multi-level programming algorithms were proposed based on the control of the tail-end of the programming pulse. A 2 bits/cell MLC PCM design was proposed in [15]. Authors in [15] also presented a programming algorithm suitable for their MLC design. A *morphable memory system* (MMS) was proposed in [16]. This MMS can switch the PCM cell between the SLC and the MLC with small hardware overheads. The adjustment is based on the statistic information of memory traffic in the running time. Another MLC/SLC PCM architecture was presented in [8]. The PCM configuration in [8] is also based on device capacity utilization in the running time. However, these MLC techniques have inherently negative impacts on the endurance of PCM, due to the iterative program-and-verify procedure applied in the MLC PCM [16].

Another major trend of techniques of improving the lifetime of non-volatile memories is the application level design. An application-specific flash memory was used as the main memory [64]. Xu et al. proposed an application-specific approach to minimize the connections by finding the minimal communication between cores in CMP [65]. The memory latency can also be hidden by optimizing the loops in the application [66]. However, these works do not consider the capacity constraint of memory, which may cause serious problems in SPM due to its limited capacity. Koc and Kandemir et al. used the recomputation in the SPM to reduce communications among different cores on chip [67], as well as between the cores and off-chip memory [68], which can reduce the number of reads in the main memory. But these recomputation techniques cannot reduce communication significantly when the application does not consist of many loops and multi-dimemsion arrays. A CMP cache management approach was presented with the idea of data migration [69]. This approach tries to keep as many pages as possible in the cache for later use. Hu et al. modeled

the data migration problem as a shortest path program and decided the best route for a given data to migrate from the source core to the destination core [70]. Nevertheless, the on-chip data traffic was not considered, which may lead to performance drawback when sharing a large amount of data simultaneously. Two different optimization approaches for memory activities in CMP were proposed [71, 72]. These two optimization approaches cannot handle the data sharing among SPMs. In our ILP-based optimization approach, we take the capacity constraint in memory, on-chip data bus bandwidth, as well as data dependencies into account. Memory operations such as load, store, and share are well scheduled in the optimal solution generated by our ILP-based optimization.

3.3 Model and Background

Phase-change memory

As one type of non-volatile memory, PCM exploits the unique characteristic of the chalcogenide to store bits. A typical PCM cell consists of a chalcogenide layer and two electrodes on both sides. Two stable states of the chalcogenide, i.e., the crystalline and the amorphous, can be switched between when different amount of heat is applied in the chalcogenide. This procedure is done by injecting current into the PCM cell. When writing the PCM cell, the SET operation heats the chalcogenide layer to temperature between the crystallization temperature (300^oC) and the melting temperature (600^oC). By this operation, the chalcogenide is in the low-resistance crystalline state, which corresponds to the logic "1". On the other hand, the RESET operation heats the chalcogenide layer above the melting temperature. The corresponding state of the high resistance is amorphous state, i.e., the logic "0". The read operation of the PCM is basically sensing the resistance level of the PCM cell. It is non-destructive and involves much less heat stress, compared to that of the write operation.

Since both the SET and the RESET write operations apply dramatic heat stress into the phase change material, write is the major wear mechanism for the PCM. A PCM cell can perform stably within 10^8 to 10^9 times of writes. Compared to the 10^{15}-time-write

endurance of the DRAM, the lifetime of the PCM becomes the major issue in implementing the PCM as the main memory.

The memory banking and memory controller

In the PCM cell array, there are several peripheral logics, such as decoders, sense amplifiers, and write drivers, to form the memory structure, which is similar to that of DRAM. The cells in the array are organized in the similar way as that of the DRAM, grouped into sub-blocks, blocks, and banks.

Among the peripheral logics, the memory controller is one of the crucial parts in the PCM. When operating a memory request, the memory controller sends a sequence of micro commands to the memory banks. In the traditional DRAM architecture, a precharge command to write back a row buffer should be issued before a new row is loaded, when the read miss occurs in the row buffer. However, this precharge is not necessary in the PCM architecture. Instead, the PCM memory controller bypasses the row buffer and writes to cells directly, in a write operation. In addition, we use the SPM as buffers, reducing the unnecessary write to the PCM memory in this chapter.

In the read operation, the controller first checks the row buffer. If the target is in the buffer, the memory controller obtains the entry without accessing the memory bank. Otherwise, the memory controller will issue an activate command to move the data to an empty row in the buffer, and a read command to get the data. In the write operation, the memory controller issues the write command and sends the data directly to the memory bank.

The multi-entry row buffer is also implemented in the PCM cell array. Replacement policies, such as *Least Recently Used* (LRU), are used to manage the entries in the row buffer. When a miss happens in the row buffer, the selected entry does not need to send back to the bank, since every write is directed to the memory bank.

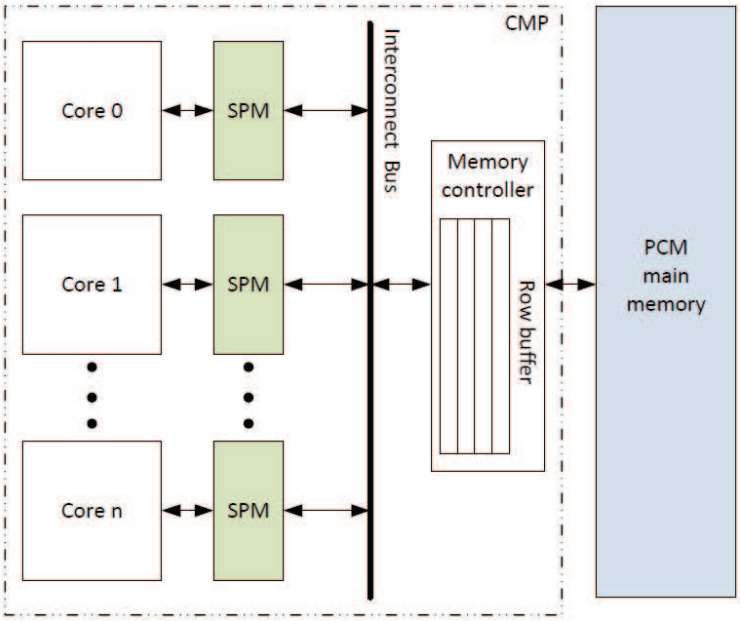


Figure 3.1: The CMP architecture with SPMs and the PCM main memory

Scratch pad memory

The SPM is an on-chip memory that can be accessed directly by processors with very low latency. The major difference between the SPM and the cache is that the data storage in the SPM is controlled by the system software, while the cache is automatically controlled by the hardware [72]. Due to the existence of the controllability on data storage in the SPM, we are able to optimize memory activities based on the characteristics of the application running in the system.

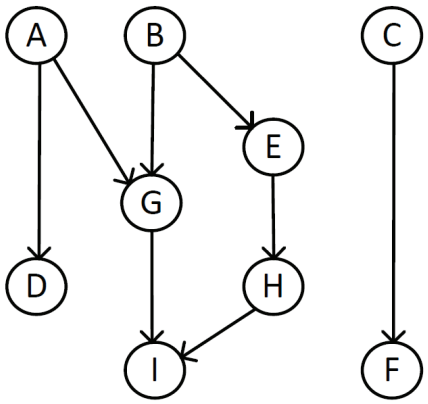
In this chapter, we focus on a CMP architecture as shown in Fig. 3.1. In this architecture, each core is connected to an SPM array. All SPMs are networked with the memory controller, which is also attached to the PCM main memory. Data are loaded or stored between the SPMs and the PCM main memory, via the memory controller. In addition, copies of data are transferred among the SPMs. When a core is executing a task, it can load data from its own SPM. The resulting data of a task can be written back to the SPM.

Application model

We model the application in this chapter as a graph $G = \langle T, E, P, R_M, W_M, E_C \rangle$. $T = \langle t_1, t_2, t_3, ..., t_n \rangle$ is the set of n tasks. $E \subseteq T \times T$ is the set of edges where $(u, v) \in E$ means that task u must be scheduled before task v. $P = \langle p_1, p_2, p_3, \ldots, p_m \rangle$ is the set of m pages that are accessed by the tasks. $R_M : T \rightarrow P$ is the function where $R_M(t)$ is the set of pages that task t reads from. $W_M : T \rightarrow P$ is the function where $W_M(t)$ is the set of pages that task t writes to. $E_C(t)$ represents the execution time of task t while all the required data are in the SPM.

3.4 Illustrating Example

An example of an application and a system



(a)

Task	Read pages	Write pages
A	1,2	3
B	4	5,6
C	7	8
D	5	9
E	6	11
F	8,12	13
G	5,14	15
H	11,16	17
I	9,17	18

(b)

Figure 3.2: An example of memory activities in the PCM. (a) The DAG of the application in the example, (b) Read pages and write pages of tasks in the application.

First we give an example that reduces the number of writes in the PCM by sharing copies across the SPM. Considering a schedule of an application represented by the DAG in Fig. 3.2(a) in a three-core system, each task in the application requires up to 2 pages that should be in the SPM before the core executes it. The required pages R_M of each task are shown in the "Read page" column of Fig. 3.2(b). Moreover, tasks also need to output

and modify up to 2 pages, i.e., W_M. The write pages W_M of each task are shown in the "Write page" column. For example, task A requests two pages, <page 1 and 2>, before its execution, and writes its result in one page, <page 3>.

Using the list scheduling, we have a baseline schedule as follows: task A, D, and G are assigned to core 0; task B, E, H, and I are assigned to core 1; and task C and F are assigned to core 2. A detailed schedule with memory activities is shown in Fig. 3.3(a). The Y axis represents the clock cycles. We assume the execution time of each task is 8 clock cycles. A core needs 2 cycles to access its own SPM, 5 cycles to a remote SPM. We also assume a read from the PCM main memory takes 80 cycles, while a write takes 800 cycles [70]. The memory activities, i.e., the shaded boxes in Fig. 3.3(a), are the major time consuming part in this schedule.

We observe that before core 0 reads page 5 in its SPM1, page 5 has been modified by the core 1, which is the output of task B. In this case, transferring pages across the SPMs reduces the write, since it is not necessary to write back page 5 before loading it again in the SPM. In addition, the time of sharing across SPMs should be much shorter than the time of writing and reading in PCM.

We modify the schedule as shown in Fig. 3.3(b). In this example, instead of writing back page 5 right after the executions of task B, we move the copy of page 5 from the SPM of core1 to the SPM of core0 before the execution of task D, which is represented as a red dotted arrow. The move occurs before the execution of task E on core1, due to the need of space in the SPM of core1 for storing the R_M of task E. After the move, a copy of page 5 is kept in the SPM of core0, until task D is executed by core0. By doing this, an unnecessary write is eliminated. Similarly, we move the copy of page 9 from the SPM of core0 to the SPM of core1 after the execution of task D, which is required by the later executed task I.

In the next section, we will discuss our ILP-based optimization algorithms with more details.

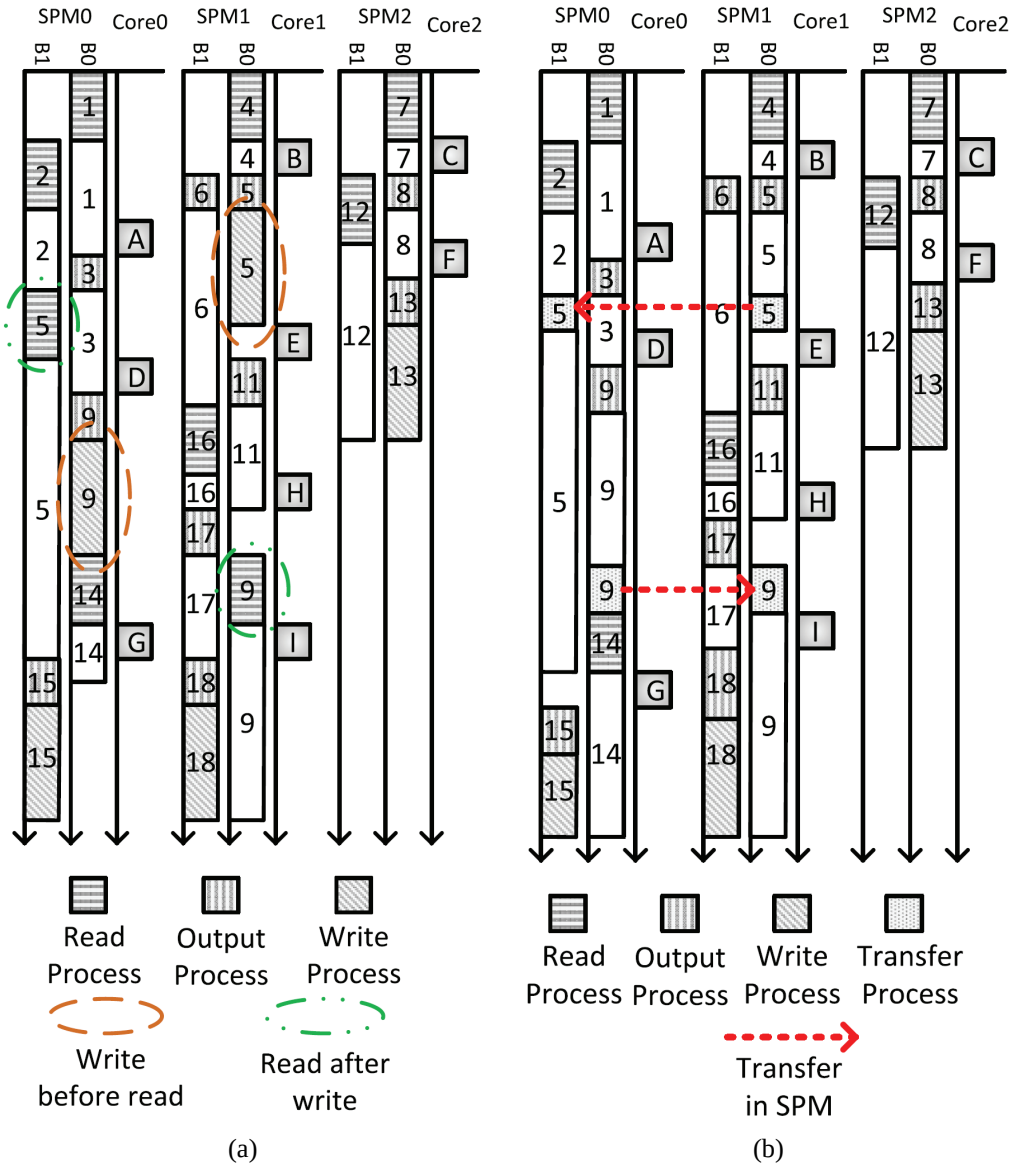


Figure 3.3: The schedules for the application in Fig. 3.2 running a three-core CMP system with two SPM blocks per core. The schedule in (a) is without data sharing in SPMs. The schedule in (b) is with data sharing in SPMs. The vertical axis represents the clock cycles. Each core has two SPM blocks, represented as the "B0" and "B1" columns. The blank box with number i in the "Bx (0 or 1)" column indicates that page i resides in SPM block "Bx" at the corresponding cycles. Since the write operation time (800 cycles) is 400 times longer than the core execution time (2 cycles), the scale of these figures does not strictly represent accurate clock cycles, only demonstrating the orders of these schedules.

3.5 ILP memory activities optimization algorithm

In this section, we present our ILP memory activities optimization algorithm. There are three major parts in our algorithm: the baseline scheduling, the ILP-based memory activities scheduling, and the post ILP procedure. The baseline scheduling generates an baseline schedule for both the task executions and the SPM assignments. Then, the ILP-based memory activities scheduling will find the optimal memory activities strategy to minimize the memory writes based on the baseline scheduling. Finally, the post ILP procedure will further reduce total execution time by eliminating the idle slots in the schedule.

Baseline scheduling

The Min-Min is a popular greedy scheduling algorithm [44, 73]. The Min-Min algorithm generates near-optimal schedule with comparatively low computational complexity [74]. In the Min-Min baseline algorithm used in this chapter, we need to update the mappable task set in every step to maintain the task dependencies. Tasks in the mappable task set are the tasks of which all the predecessor tasks are finished. Algorithm 3.1 shows the procedure of the Min-Min algorithm. Before we schedule a given task executed on a given core, we should schedule the required memory pages allocated in the SPM of the core in advance. We assume that the time of reading a memory page from the SPM is included in the execution time of this given task. We also assume that for some tasks, the output may be stored in the memory page that is different from the required pages. For example, a task may require page p_0 and p_1 as the input, and output the result in page p_2. In this case, the modified page should be loaded in the SPM before it is written back to the PCM main memory. In the case where multiple tasks on different cores need to store their results in the same page, we will schedule the SPM modifying process at different clock cycles, even though these tasks may be finished at the same time. Complicated policies for memory coherence are out of the scope of this chapter. We apply some simple policies to keep the memory content among SPMs and the PCM main memory coherent:

- Where a core initiates an SPM modifying process of a given page p, other cores that have a copy of this page in their SPM should initiates an SPM evicting process of this page. By doing this, there is no "dirty" copy of this page exists in the SPMs.

- In the baseline scheduling process, we don't consider the data sharing in SPMs. We will write back the modified page right after the modification is finished.

- In some cases, some tasks may require the page that is modified by another task previously. The read process can only be initiated after the modification is finished.

- We implement the *Least Recently Used* (LRU) replacement policy in the SPM management.

Algorithm 3.1 Min-Min algorithm

Input: A set of T tasks represented by a DAG, C different cores, E_C of tasks
Output: A schedule generated by Min-Min

Form a mappable task set MT
while Set MT is not empty **do**
  for i: task $i \in [0, T-1]$ **do**
    for j: core $j \in [0, C-1]$ **do**
      Find the earliest possible time $Tpg_{i,j}$ that all the require pages of i are available, based on dependencies
      Calculate the earliest possible task finished time $Tfin_{i,j} = Tpg_{i,j} + E_C(i)$
    end for
    Find the core $C_{min}(i)$ giving the earliest finish time of $Tfin_{i,j}, \quad \forall j \in [0, C-1]$
  end for
  Find the pair $(k, C_{min}(k))$ with the earliest finish time $Tfin_{i,C_{min}(i)}, \quad \forall \quad i \in [0, T-1]$ among the task-core pairs generated in for-loop
  Schedule the required pages of task k, $R_M(k)$, to the SPM of core $C_{min}(k)$ as soon as possible
  Assign task k to core $C_{min}(k)$
  Schedule the modification of the resulting pages, $W_M(k)$, in the SPM of core $C_{min}(k)$
  Schedule the write back process of the resulting pages
  Remove k from MT
  Update the mappable task set MT
end while

ILP formatting

Table 3.1: Symbols and acronyms used in the ILP formatting

Symbol	Description
t	Task t
c	Core c
s	Clock cycle s
p	Memory page p
T	Number of tasks
C	Number of cores
S	Total number of clock cycles
P	Number of pages
$ASM_{t,c}$	Task assignment matrix
$St_{t,c,s}$	Task start time matrix
$WL_{t,c,s}$	Core workload matrix
$Mem_{p,c,s}$	Required memory matrix
$R_M(t)$	A set of page required by task t
$R_{p,c,s}$	Read matrix
$M_{p,c,s}$	Modify matrix
$W_{p,c,s}$	Write matrix
$Ev_{p,c,s}$	Evict matrix
$Si_{p,c,s}$	SPM input matrix
$So_{p,c,s}$	SPM output matrix
$OC_{p,c,s}$	SPM occupation matrix
$PM_{p,c,s}$	SPM page available matrix
$Mo_{p,c,s}$	Move out matrix
$Mi_{p,c,s}$	Move in matrix
$Mih_{p,c,s}$	Move in indicator matrix
$Mr_{p,cs}$	SPM page modified matrix

To input the baseline schedule to the later memory activities scheduling algorithm, we define several 0-1 matrixes to indicate the task executions and the SPM memory activities. The values in these matrixes are either 0 or 1. For the convenience of the reader, we list the symbols used in the ILP formatting in Table 3.1. We give the definitions of twelve 0-1 matrixes as follows:

1. *Task assignment matrix* ASM. $ASM_{t,c} = 1$ means that task t is assigned to core c.

The matrix ASM has the characteristic as follows:

$$\sum_{c=0}^{C-1} ASM_{t,c} = 1 \qquad \forall \quad t \in [0, T-1] \tag{3.1}$$

2. *Task start time matrix* St. When $St_{t,c,s} = 1$, it means that the execution of the task t starts at clock cycle s on core c.

3. *Core workload matrix* WL. $WL_{t,c,s} = 1$ means that core c is executing task t at clock cycle s. The relationship between St and WL is:

$$WL_{t,c,s} = \sum_{i=s-E_{t,c}-1}^{s} St_{t,c,i} \qquad \forall \quad t \in [0, T-1], \quad c \in [0, C-1] \tag{3.2}$$

where $E_{t,c}$ is the execution time of task t on core c.

4. *Required memory matrix* Mem. $Mem_{p,c,s} = 1$ means page p is required by core c at clock cycle s.

$$Mem_{p,c,s} = WL_{t,c,s} \qquad \forall \quad p \in ReqMen(t) \tag{3.3}$$

where $ReqMen(t)$ is a set of pages that are required by task t.

5. *Read matrixes* R, $\widetilde{R}$, and $\bar{R}$. $R_{p,c,s} = 1$ means page p is read from the PCM memory and loaded into the SPM of core C at clock cycle s. Note that the matrix R indicates the start time of the read process, the matrix $\bar{R}$ indicates the end of the read process, and the matrix $\widetilde{R}$ represents the whole read process. The relationships among R, $\widetilde{R}$, and $\bar{R}$ are as follws:

$$\widetilde{R}_{p,c,s} = \sum_{i=s-len_r+1}^{s} R_{p,c,i} \tag{3.4}$$

$$\bar{R}_{p,c,s} = R_{p,c,(s-len_r)} \tag{3.5}$$

where len_r is the length of the read process.

6. *Modify matrixes* M, $\widetilde{M}$, and $\bar{M}$. $M_{p,c,s} = 1$ means page p is modified by the core C and loaded into the SPM of core C at clock cycle s. Here, we assume that the page

including the modified variables should be first stored in the SPM before written back. M is the start time of the modify process and the end of the modify process is indicated as $\bar{M}$, while the whole modify process is represented by $\widetilde{M}$.

$$\widetilde{M}_{p,c,s} = \sum_{i=s-len_m+1}^{s} M_{p,c,i} \tag{3.6}$$

$$\bar{M}_{p,c,s} = M_{p,c,(s-len_m)} \tag{3.7}$$

where len_m is the length of the modify process.

7. *SPM input matrixes* Si and $\bar{S}i$. $Si_{p,c,s} = 1$ means page p is loaded into the SPM of core c at clock cycle s. This page can be either read from the PCM memory or store back from the core after it is modified by that core. Thus:

$$Si_{p,c,s} = R_{p,c,s} + M_{p,c,s} \tag{3.8}$$

$$\bar{S}i_{p,c,s} = \bar{R}_{p,c,s} + \bar{M}_{p,c,s} \tag{3.9}$$

8. *Write matrixes* W, $\widetilde{W}$, and $\bar{W}$, $W_{p,c,s} = 1$ means page P is written back into the PCM memory from core C at clock cycle s. Here, we also assume the page will be evicted at the same. The differences among W, $\widetilde{W}$, and $\bar{W}$ are similar to the ones among R, $\widetilde{R}$, and $\bar{R}$.

$$\widetilde{W}_{p,c,s} = \sum_{i=s-len_w+1}^{s} W_{p,c,i} \tag{3.10}$$

$$\bar{W}_{p,c,s} = W_{p,c,(s-len_w)} \tag{3.11}$$

where len_w is the length of the write process.

9. *Evict matrixes* Ev, $\widetilde{Ev}$, and $\bar{E}v$. $Ev_{p,c,s} = 1$ means page P is evicted from core C at clock cycle s. This matrix only records the evict without write back. The differences among Ev, $\widetilde{Ev}$, and $\bar{E}v$ are similar to the ones among R, $\widetilde{R}$, and $\bar{R}$.

$$\widetilde{Ev}_{p,c,s} = \sum_{i=s-len_w+1}^{s} Ev_{p,c,i} \tag{3.12}$$

$$\bar{E}v_{p,c,s} = Ev_{p,c,(s-len_{ev})} \tag{3.13}$$

where len_{ev} is the length of the evict process.

10. *SPM output matrixes* So and $\bar{So}$. $So_{p,c,s} = 1$ means page p is evicted from the SPM of core c at clock cycle s. This page could be modified by the core or evicted after read. Thus :

$$So_{p,c,s} = W_{p,c,s} + Ev_{p,c,s} \tag{3.14}$$

$$\bar{So}_{p,c,s} = \bar{W}_{p,c,s} + \bar{Ev}_{p,c,s} \tag{3.15}$$

11. *SPM occupation matrix* OC. $OC_{p,c,s} = 1$ means page p is occupying a part of the SPM of core c at clock cycle s. The SPM occupation matrix OC holds the following equation:

$$OC_{p,c,s} = OC_{p,c,s-1} + Si_{p,c,s} - \bar{So}_{p,c,s} \tag{3.16}$$

12. *SPM page available matrix* PM, $PM_{p,c,s} = 1$ means page p is residing in the SPM of core C at clock cycle s. Note that when $OC_{p,c,s} = 1$, core c may not be able to use the page p at clock cycle s, due to the fact that it may still be in the memory transfer process. And $PM_{p,c,s} = 1$ means that core c can surely use page p at clock cycle s. The SPM page matrix PM holds the following equation:

$$PM_{p,c,s} = PM_{p,c,s-1} + \bar{Si}_{p,c,s} - So_{p,c,s} \tag{3.17}$$

We will use these 0-1 matrixes represent the baseline schedule in the following ILP-based memory activities scheduling algorithm.

ILP-based memory activities scheduling algorithm

With the baseline schedule, we will use our ILP approach to find the optimal memory activities schedule and minimize the number of the PCM activities. In some cases, a page that is needed by a task is residing in the SPM of a remote core. Instead of loading the page from the PCM memory, we can transfer the page from the SPM of the remote memory.

Additional ILP formatting for data transferring in SPMs

To represent the memory activities among the SPMs, we define three additional 0-1 matrixes as follows:

1. *Move out matrix* Mo, $\widetilde{Mo}$, and $\bar{M}o$. $Mo_{p,c,s} = 1$ means page P is moved from the SPM of core C to the SPM of another core at clock cycle s. We assume that the SPM of this core will evict this page right after. The $\widetilde{Mo}$ represents the whole moving process and the $\bar{M}o$ indicates the end of the moving.

$$\widetilde{Mo}_{p,c,s} = \sum_{i=s-len_{mi}+1}^{s} Mo_{p,c,i} \tag{3.18}$$

$$\bar{M}o_{p,c,s} = Mo_{p,c,(s-len_{mi})} \tag{3.19}$$

   where len_mi is the length of the SPM data sharing process. Remind that we set the rule in our baseline scheduling: when a page is modified by a given core, all the copies in the SPMs of the rest cores should be evicted. There is no conflict data exist in SPMs. To avoid the case that more than one different contents of the same page are copied at the same time, we still need to set a constraint in our ILP model as:

$$\sum_{c=0}^{C-1} Mo_{p,c,s} = 1 \qquad \forall \quad \begin{cases} p \in [0, P-1] \\ s \in [0, S-1] \end{cases} \tag{3.20}$$

2. *Move in matrix* Mi, $\widetilde{Mi}$, and $\bar{M}i$. $Mi_{p,c,s} = 1$ means page P is moved into the SPM of core C from the SPM of another core at clock cycle s. The $\widetilde{Mi}$ represents the whole moving process and the $\bar{M}i$ indicates the end of the moving.

$$\widetilde{Mi}_{p,c,s} = \sum_{i=s-len_{mi}+1}^{s} Mi_{p,c,i} \tag{3.21}$$

$$\bar{M}i_{p,c,s} = Mi_{p,c,(s-len_{mi})} \tag{3.22}$$

3. *Move in indicator matrix* Mih. $Mih_{p,s} = 1$ means page P is moved into the SPMs of at least one core at clock cycle s.

$$Mih_{p,s} \leq \sum_{c=0}^{C-1} Mi_{p,c,s} \qquad \forall \quad \begin{cases} p \in [0, P-1] \\ s \in [0, S-1] \end{cases} \tag{3.23}$$

When a page move out process is initiated, there also should be at least one move in process initiated for this page. In some cases, maybe multiple cores require this page simultaneously. Then multiple move in processes are initiated. So we can express this constraint as:

$$Mih_{p,s} = \sum_{c=0}^{C-1} Mo_{p,c,s} \qquad \forall \quad \begin{cases} p \in [0, P-1] \\ s \in [0, S-1] \end{cases} \tag{3.24}$$

In the previous "ILP formatting" subsection, we define the SPM input/output matrixes $Si_{p,c,s}$, $\bar{S}i_{p,c,s}$, $So_{p,c,s}$, and $\bar{S}o_{p,c,s}$ to determine whether a page is available in the SPM of a give core at clock cycle s. Now, we further modify these definitions by including the consideration of the Mi, Mo, $\bar{M}i$, and $\bar{M}o$, i.e. transferring data among SPMs. The new definition of Si, $\bar{S}i$, So, and $\bar{S}o$ as follows:

$$Si_{p,c,s} = R_{p,c,s} + M_{p,c,s} + Mi_{p,c,s} \tag{3.25}$$

$$\bar{S}i_{p,c,s} = \bar{R}_{p,c,s} + \bar{M}_{p,c,s} + \bar{M}i_{p,c,s} \tag{3.26}$$

$$So_{p,c,s} = W_{p,c,s} + Ev_{p,c,s} + Mo_{p,c,s} \tag{3.27}$$

$$\bar{S}o_{p,c,s} = \bar{W}_{p,c,s} + \bar{E}v_{p,c,s} + \bar{M}o_{p,c,s} \tag{3.28}$$

We use these new definitions of SPM input/output matrixes to calculate the SPM occupation matrix OC and the SPM page matrix PM in Equation (3.16) and (3.17).

ILP constraints for memory activities optimization

One of the most critical requirements of the memory activities is that when a task is executed by a given core, all the required memory pages should be placed in the SPM of that

core no later than the start time of the execution. This requirement can be expressed as:

$$PM_{p,c,s} \geq Mem_{p,c,s} \qquad \forall \quad \begin{cases} p \in [0, P-1] \\ c \in [0, C-1] \\ s \in [0, S-1] \end{cases} \tag{3.29}$$

Other important requirement is that no matter how the pages are transferred, the total amount of pages in the SPM of a core at every clock cycle should not be larger than the capacity of this SPM.

$$\sum_{p=0}^{P-1} OC_{p,c,s} \leq SPM(c) \qquad \forall \quad \begin{cases} c \in [0, C-1] \\ s \in [0, S-1] \end{cases} \tag{3.30}$$

where SPM(c) is the capacity of the core c's SPM.

For an eligible data sharing in SPMs, the source SPM should have the copy of the target page available when the sharing is initiated.

$$PM_{p,c,s} \geq Mo_{p,c,s} \tag{3.31}$$

Another constraint we need to set is that only one memory activity can be performed at a clock cycle, due to the arbitration of the data bus across SPMs and the PCM controller. Thus

$$\begin{aligned} \sum_{p=0}^{P-1} \sum_{c=0}^{C-1} \quad & (\widetilde{R}_{p,c,s} + \widetilde{M}_{p,c,s} + \widetilde{Mi}_{p,c,s} \\ & + \widetilde{W}_{p,c,s} + \widetilde{Ev}_{p,c,s} + \widetilde{Mo}_{p,c,s}) \quad \leq 1 \\ & \forall \quad s \in [0, S-1] \end{aligned}$$

To address the memory coherence problems, we set the rule that when a core modifies a given page in its SPM, we will evict all the "dirty" copies of this page in the SPMs of other cores.

$$Ev_{p,c,s} \geq M_{p,c_1,s} \qquad \forall \quad c_1 \neq c \tag{3.32}$$

The goal of the memory activities optimization is to reduce the number of memory writes. In the baseline scheduling, we do not consider the possible moving of the modified memory. After the page is modified, it will be written back immediately. In this case, we can get the relationship between the SPM modify matrix M and the SPM write matrix W as the following:

$$\sum_{i=0}^{S-1} M_{p,c,i} = \sum_{i=0}^{S-1} W_{p,c,i} \tag{3.33}$$

The reason why SPM data sharing can reduce the memory writes is that by moving the copy of a given page among SPMs of cores, different tasks can modified this page in serial. And the write back may be initiated after multiple modifications. In this case, Equ. (3.33) is not necessary. However, even though the number of modifies and the number of writes of a given page may not be equal, at least one write back should be scheduled for a page that had modified previously. Here, we define a 0-1 matrix Mr to indicate whether a page has been modified in the schedule before a give clock cycle. $Mr_{p,s} = 1$ means page p has been modified at least once before the clock cycle s but not written back yet.

$$Mr_{p,s} = Mr_{p,s-1} + \sum_{i=0}^{C-1} (M_{p,i,s} - W_{p,i,s}) \tag{3.34}$$

In the case that a page has been modified by a given core, but not written back yet, the following tasks that require a copy of this page can only migrate them from the SPM of that core. In other words, the following tasks cannot obtain a copy of this page by reading from the PCM main memory.

$$R_{p,c,s} \leq Mr_{p,s} \qquad \forall \quad c \in [0, C-1] \tag{3.35}$$

And for every page, it should have a newest copy in the PCM main memory at the end of the schedule. Thus

$$Mh_{p,(S-1)} = 0 \qquad \forall \quad p \in [0, P-1] \tag{3.36}$$

Finally, our objective of the memory activities scheduling is to minimize the times of write process.

$$\text{Minimize: } \sum_{i=0}^{P-1}\sum_{j=0}^{C-1}\sum_{k=0}^{S-1} W_{i,j,k} \tag{3.37}$$

Post ILP procedure

In our baseline scheduling, we schedule all writes without considering SPM data sharing. Based on this schedule, we optimize the memory activities in our ILP algorithm. Even though the number of writes in the schedule generated by our ILP algorithm is minimized, the start time of each task remains the same as the one in our baseline scheduling. Since the data sharing in SPMs is much less time consuming than the write in the PCM memory, there are a lot of idle slots in which all cores have neither task execution nor memory activities. To improve the system performance, we further eliminate these idle slots in the schedule generated by our ILP algorithm. To remain the data dependencies, we find out these idle slots and push the whole schedule of all cores forward, as long as no data dependency is violated.

3.6 Experimental results

Experiment setup

In this section, our proposed ILP algorithm is evaluated by running the DSPstone benchmarks [75] and the MiBench [76]. In our custom simulator, the CMP system has multiple cores, each of which has the similar performance as that of the CoDeL DSP [77]. We compare two different sizes of SPM, which is similar to the SPM setting in [71]. The PCM main memory parameters are set as in [63]. We use the Lingo [78] software to solve the ILP problem.

Since most of the DSPstone benchmarks are embarrassingly parallel, which means there are few data dependencies among tasks, we group multiple DSPstone benchmarks

Table 3.2: The grouping of benchmarks

Set No.	Benchmarks
Set 1	Convolution, IIR_BIQUAD_N
Set 2	FIR2D, LMS
Set 3	N_REAL_UPDATE, N_COMPLES_UPDATE
Set 4	DOT_PRODUCT, MATRIX_1x3, IIR_BIQUAD_ONE
Set 5	CRC32
Set 6	FFT
Set 7	Blowfish enc
Set 8	Mad
Set 9	PGP sign
Set 10	GSM

into four benchmark sets. In each set, we create data dependencies by sharing variables among different benchmarks. We also use another six Mibench benchmarks in our experiment, one benchmark per set. The grouping of benchmarks is shown as in Table 3.2.

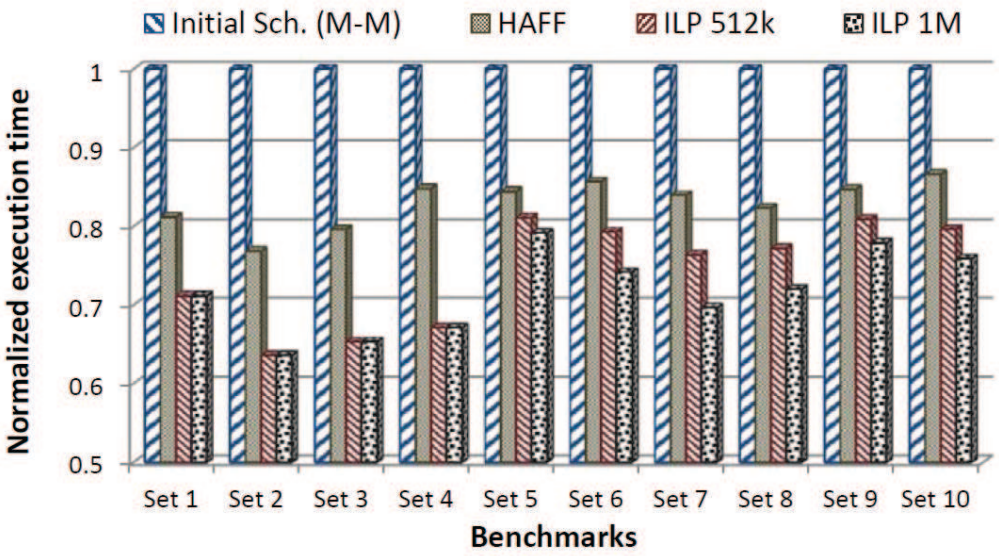


Figure 3.4: The execution time on a four-core CMP system. "Initial Sch. (M-M)" is the baseline scheduling with the Min-Min algorithm; "HAFF" is the High Access Frequency First algorithm; "ILP 512K" is our ILP-based algorithm with total 512KB SPMs; and "ILP 1M" is our ILP-based algorithm with total 1MB SPM. All columns are normalized by the corresponding execution time generated by the baseline scheduling with the Min-Min algorithm.

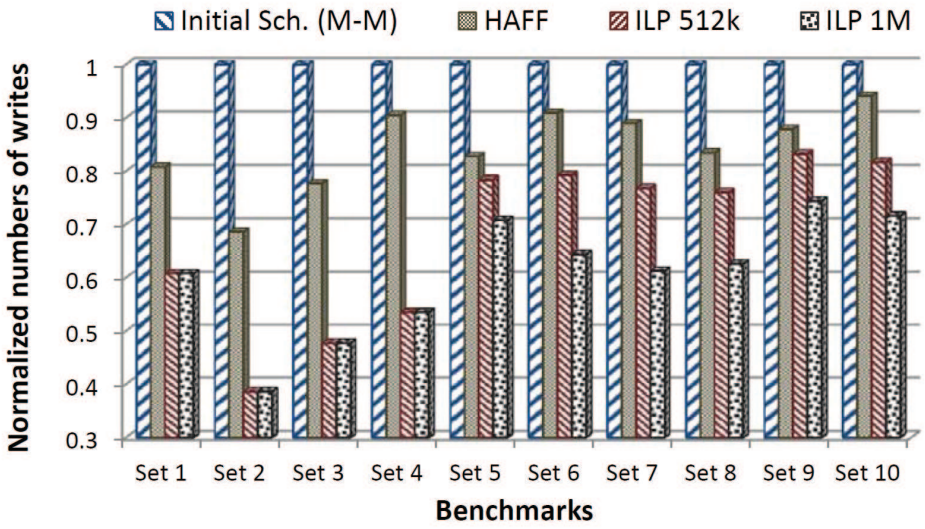


Figure 3.5: The numbers of writes on a four-core CMP system. "Initial Sch. (M-M)" is the baseline scheduling with the Min-Min algorithm; "HAFF" is the High Access Frequency First algorithm; "ILP 512K" is our ILP-based algorithm with total 512KB SPMs; and "ILP 1M" is our ILP-based algorithm with total 1MB SPM. All columns are normalized by the corresponding numbers of writes generated by the baseline scheduling with the Min-Min algorithm.

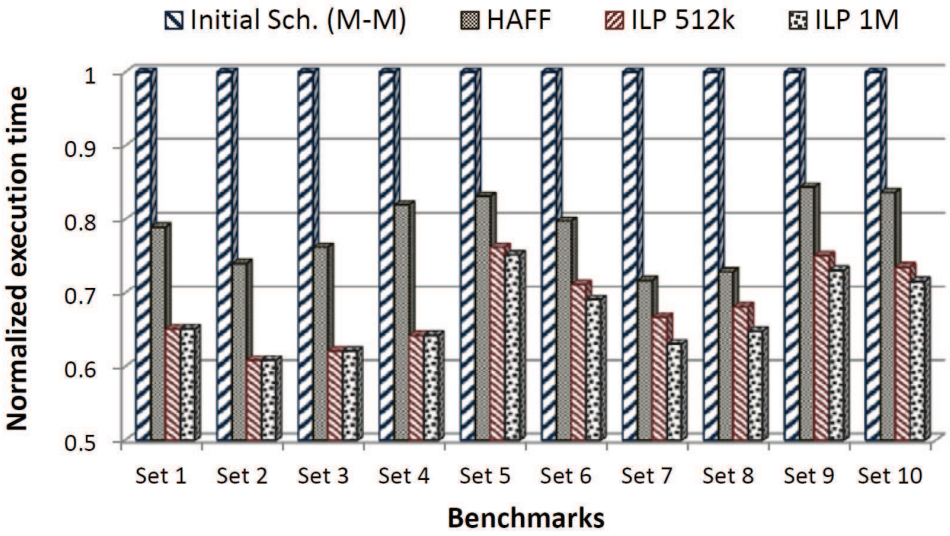


Figure 3.6: The execution time on a eight-core CMP system. "Initial Sch. (M-M)" is the baseline scheduling with the Min-Min algorithm; "HAFF" is the High Access Frequency First algorithm; "ILP 512K" is our ILP-based algorithm with total 512KB SPMs; and "ILP 1M" is our ILP-based algorithm with total 1MB SPM. All columns are normalized by the corresponding execution time generated by the baseline scheduling with the Min-Min algorithm.

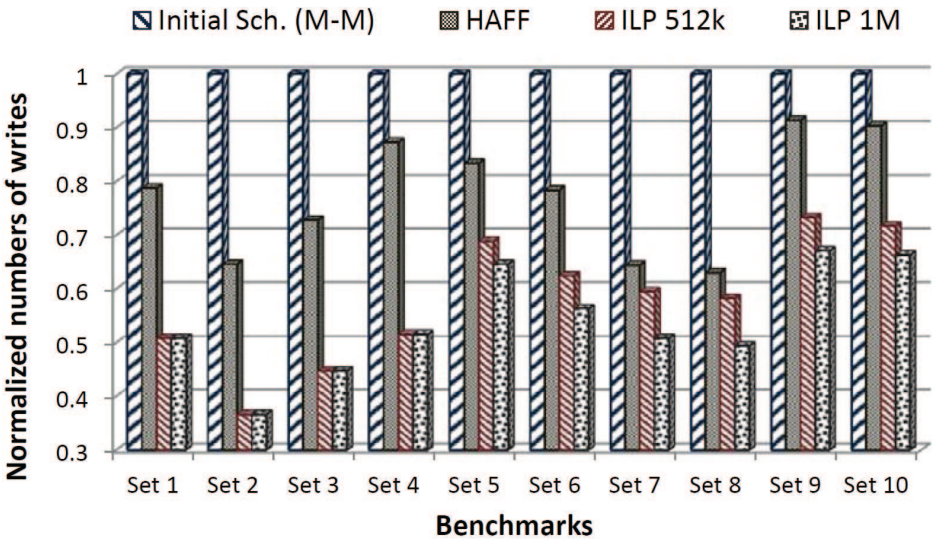


Figure 3.7: The numbers of writes on a eight-core CMP system. “Initial Sch. (M-M)” is the baseline scheduling with the Min-Min algorithm; “HAFF” is the High Access Frequency First algorithm; “ILP 512K” is our ILP-based algorithm with total 512KB SPMs; and “ILP 1M” is our ILP-based algorithm with total 1MB SPM. All columns are normalized by the corresponding numbers of writes generated by the baseline scheduling with the Min-Min algorithm.

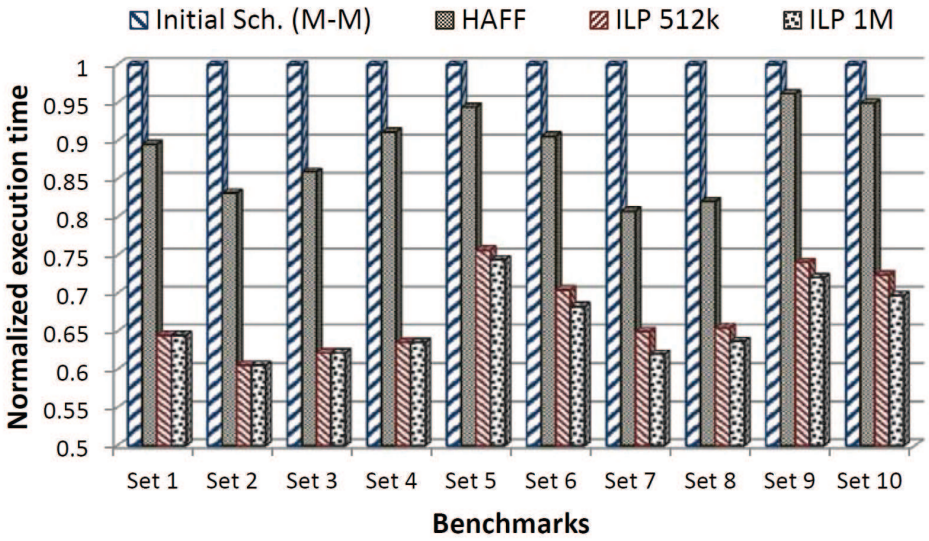


Figure 3.8: The execution time on a twelve-core CMP system. “Initial Sch. (M-M)” is the baseline scheduling with the Min-Min algorithm; “HAFF” is the High Access Frequency First algorithm; “ILP 512K” is our ILP-based algorithm with total 512KB SPMs; and “ILP 1M” is our ILP-based algorithm with total 1MB SPM. All columns are normalized by the corresponding execution time generated by the baseline scheduling with the Min-Min algorithm.

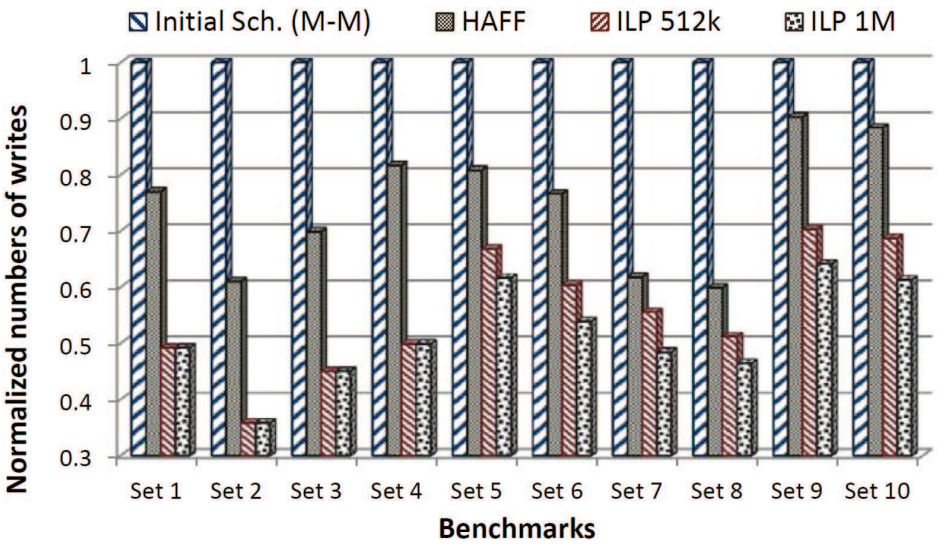


Figure 3.9: The numbers of writes on a twelve-core CMP system. "Initial Sch. (M-M)" is the baseline scheduling with the Min-Min algorithm; "HAFF" is the High Access Frequency First algorithm; "ILP 512K" is our ILP-based algorithm with total 512KB SPMs; and "ILP 1M" is our ILP-based algorithm with total 1MB SPM. All columns are normalized by the corresponding numbers of writes generated by the baseline scheduling with the Min-Min algorithm.

In Fig. 3.4, we compare the performance of our proposed ILP algorithm with that of the HAFF (High Access Frequency First) algorithm [72]. The "Initial Sch. (M-M)" columns represent the execution time of the benchmark sets by using the baseline scheduling algorithm, i.e., Min-Min, in our ILP algorithm. The "HAFF" columns demonstrate the execution time using the HAFF algorithm. The "ILP 512K" columns show the execution time optimized by our ILP algorithm with a total 512KB SPM. And the "ILP 1M" columns provide the execution time optimized by our ILP algorithm with a total 1MB SPM. In Fig. 3.5, we also compare the numbers of writes in a four-core CMP system. The HAFF has less numbers of writes than that of our baseline scheduling algorithm, although its objective is not reduce the numbers of write. Thus, the HAFF outperforms our baseline scheduling algorithm in terms of total execution time. Since our ILP algorithm targets on minimizing the number of writes in the PCM main memory, it outperforms the HAFF algorithm in reducing the numbers of writes. Due to the fact that the write operation in the PCM main

memory is the major time consuming operation in the execution of tasks, our algorithm in a four-core CMP system with 512KB SPM reduces the execution time of benchmark set by the percentages from 4.3% to 20.8%, compared to the HAFF algorithm. The performance of CMP with 1MB SPM is slightly better than the one with 512KB SPM, edging by about 5%. Since the DSPstone benchmarks have small size, the size of SPMs makes no difference when running these DSPstone benchmarks.

In the eight-core and the twelve-core CMP systems, our ILP algorithm has better speedups, compared to the ones in the four-core CMP system, as shown in Fig. 3.4, 3.6, and 3.8. In an eight-core CMP system with 1MB SPMs, our ILP algorithm can shorten the execution time by 14.9% on average, compared to the HAFF algorithm, while our ILP algorithm has 25.6% improvement on average in a twelve-core CMP system. Our ILP algorithm has smaller improvement in the four-core system, about 13.8% over that of the HAFF algorithm. The major reason of these differences is that there are more opportunities for data sharing among the SPMs in a system with more cores than that in a system with fewer cores.

We show the number of writes in Fig. 3.5, 3.7, and 3.9. The columns represent the normalized numbers of writes in the corresponding schedules. Since reducing writes in the main memory is not an objective in the HAFF algorithm, it does not reduce the numbers of writes as many as our ILP-based algorithm does. In a twelve-core system with 1MB SPM it reduces the writes by 61.3% on average, while in an eight-core and a four-core system with 1MB SPM it reduces by 58.4% and 52.3% on average, respectively.

Systems with 1MB SPM perform better than that of systems with 512kB SPM. Due to the long access time of the write operation in the PCM main memory, it is aligned with the performance improvements we analyzed above. More SPM space and more data copies on-chip lead to more opportunities of sharing copies without writing back the PCM main memory. The increasing of cores has more significant improve in the performance than that of the increasing in SPM size. The reason is that the probability, of which a data copy

exists when a remote core requires it, is higher when there are more cores inside the CMP system.

3.7 Conclusions

In this chapter, we presented an ILP-based memory activities optimization algorithm for the PCM main memory. In order to increase the lifetime of the PCM memory, we schedule and share the data in SPMs, reducing the redundant writes to the PCM memory in this algorithm. Our experimental results show that our ILP algorithm can significantly reduce the number of write by 61% on average. In addition, the performance of the system is also improved due to less writes that are time-consuming.

Chapter 4 Hyper Memory Optimization and Task Scheduling

The *Dynamic RAM (DRAM)*, as the major technique for current main memory architectures, encounters its physical limit in scalability. The *phase-change memory* (PCM) is one of the most promising alternative techniques to the DRAM. A recent research trend has focused on the *multi-level cell* (MLC) of the PCM. By precisely arranging multiple levels of resistance inside a PCM cell, more than one bit of data can be stored in this PCM cell. However, the MLC PCM suffers from the performance degradation compared to the *single-level cell* (SLC) PCM, due to the longer memory access time. In this chapter, we present four optimization algorithms for embedded *chip multiprocessor* (CMP) systems equipped with the MLC/SLC PCM + DRAM hybrid memory. In our proposed algorithms, we not only schedule and assign tasks to cores in the CMP system, but also provide a hybrid memory configuration that balances the hybrid memory performance as well as the efficiency. Our experimental results show that our genetic-based algorithm generates the best solutions. It significantly reduces the maximum memory usage by 76.8%, compared to the DRAM+ uniform SLC configuration, and improves the efficiency of memory usage by 155.6%, compared to the DRAM + uniform 4 bits/cell MLC configuration. In addition, the performance of the system, in terms of total execution time, is also improved by 101%, compared to the uniform 4 bits/cell MLC configuration.

4.1 Introduction

In the last three decades, the *dynamic RAM* (DRAM) as the major technique of the main memory has been reaching its scalability limits [63]. As memory demands of applications keep increasing, the size of DRAM equipped in a system needs to be larger and larger. However, DRAM requires some specific architecture solutions, such as the refresh control and the write after read operation, to address some drawback issues, like destructive reads

and low retention time [6]. These specific architecture solutions cause extra costs that are the major reason of the scalability limit in DRAM. Scaling DRAM beyond 40 nm sizes would be questionable in the future [63]. The *phase-change memory* (PCM) is emerging as a promising DRAM alternative technique, featured many attractive advantages, such as high density, non-volatility, positive response to increasing temperature, zero standby leakage, and excellent scalability [8,9]. A 32-nm device prototype has been demonstrated [13], showing the promising future of the PCM technique.

Recently, several studies [8,14–16] have advocated for the *multi-level cell* (MLC) PCM memory architecture. The difference of resistance between the two states of the chalcogenide material is usually 3 orders of magnitude [16]. By precisely dividing this gap into several levels, one PCM cell can store more than one bit of data, resulting in higher memory capacity density than that of the *single-level cell* (SLC) memory. However, the MLC technique enhances the scalability of the PCM memory with a high price. The degradation of the performance and the endurance of the PCM memory as well as the increase in the power consumption are the major drawbacks of the MLC techniques [16]. As the number of bits stored a single PCM cell increases, the number of levels divided in this cell increases exponentially. A more precise resistance detection method is required in the MLC memory, compared to the one used in the SLC memory. The current resistance detection method implemented in the MLC adopts multiple verify procedures, which leads to a significant degradation of the performance. Similarly, in the write operation in the MLC, the *program and verify* procedure is applied repeatedly until the resistance is programmed correctively in the target level [14], which causes high power consumption in the PCM memory. In addition, these repeated pulses applied in the MLC make the already poor endurance of the PCM memory even worse [16].

In order to avoid performance degradation caused by memory misses, a traditional computing system usually takes the larger memory capacity than the maximum capacity required by applications. However, this scheme is so pessimistic that a large portion of the

memory is not used during most of running time. As a result, the SLC/MLC PCM memory architecture is suggested in [8, 16] to improve the efficiency of the main memory, which switches the mode of PCM cells between the SLC and the MLC modes. Thus, the SLC PCM provides higher performance with less power consumption and longer lifetime, while the MLC PCM enhances the memory capacity without increasing the number of PCM cells. These existing SLC/MLC memory methods adjust the configuration based on the statistics information obtained at runtime. However, since embedded *chip multiprocessor* (CMP) systems are designed to execute specific applications, optimizing the PCM configuration based on the characteristics of applications can further enhance the efficiency of the main memory in embedded CMP systems. Furthermore, even the SLC PCM has the longer access latency, compared to that of the DRAM, especially in the writing operation. In terms of I/O performance in the embedded system, the DRAM is still a better option rather than the PCM memory. Therefore, in this research work, in order to *achieve a good balance between the memory capacity and the performance*, we suggest a hybrid memory architecture, which *integrates the DRAM and the SLC/MLC PCM memory*. With this motivation, four algorithms are presented and evaluated in this chapter, which considering both the task scheduling and the memory mode configuration. To the best of our knowledge, this chapter is the first work on the synthesis issue on PCM based embedded CMP systems.

The major contributions of this chapter can be summarized as follows:

- We propose a chromosome representation for both the task scheduling and the hybrid memory mode configuration. Our proposed chromosome representation includes three strings: the scheduling string that indicates the scheduling order; the assigning string that represents task-core assignments; and the memory mode configuration string that shows where and in which mode pages are stored in the hybrid memory. A chromosome represents a complete solution of the task scheduling with data dependencies, as well as the hybrid memory configuration optimization.

- To improve the hybrid memory efficiency, we design four algorithms for the optimization of the MLC/SLC PCM + DRAM hybrid memory. To take advantage of both the high memory capacity of the PCM memory and the fast access time in the DRAM memory, we explore solution spaces of the task scheduling and the memory configuration, and find the solution that balances the performance and the efficiency of the PCM memory utilization.

- Our experimental results show that our genetic-based algorithm generates the best solutions. It significantly reduces the maximum memory usage by 76.8% compared to the DRAM+ uniform SLC configuration, and improves the efficiency of memory usage by 155.6% compared to the DRAM + uniform 4 bits/cell MLC configuration. In addition, the performance of the system, in terms of total execution time, is also improved by 101% compared to the uniform 4 bits/cell MLC configuration.

In Section 4.2, we discuss works related to this research work. In Section 4.3, the background knowledge of the hybrid memory is presented. A motivational example is given in Section 4.4. We propose our algorithms in Section 4.5, followed by experimental results in Section 4.6. Finally, we conclude this chapter in Section 4.7.

4.2 Related work

The PCM incorporated in the memory hierarchy has been well studied in [6, 9, 10, 63]. A DRAM based page cache was implemented for a large PCM memory [10]. This page cache not only improves the performance by buffering frequently used pages, but also helps endurance by reducing writes. Enhancement approaches, such as read-before-write, row-level rotation and segment swapping, were proposed to improve the life time of the PCM [9]. By rotating the cache line, the row-level rotation distributes the row level wear evenly. In the segment swapping, the contents of the least-frequently-written page and the page being written are swapped. Lee et al. presented a PCM storage device with a bit level

read-before-write loop [63]. They verified the PCM buffer organization and proposed that partial writes are able to tolerate long latency of write. Techniques on both the hardware level and the operating system level were proposed to reduce the programming power of PCM by 50%, as well as to provide a significant improvement on the endurance over conventional designs [79]. Ferreira et al. also described three life time enhancements for PCM: N-Chance victim selection replacement policy, bit level writes, and a swap management on page cache writebacks [6]. The above works focus on device level and require hardware modifications. The hybrid memory combining the non-volatile memory and the DRAM was studied in [10, 80–82]. A combination of PCM and DRAM was proposed as an alternative architecture for the future main memory [10]. A energy efficient hybrid memory architecture, PDRAM, was proposed in [80]. An operating system supporting mechanism was designed for the NOR-flash + DRAM hybrid memory [80]. And Liu et al. proposed power-aware memory partitioning algorithms for the PCM + DRAM hybrid memory [82]. However, these papers didn't consider the SLC/MLC configuration in the hybrid memory, which limits the scalability of the PCM memory as the main memory.

Multi-level cell techniques have been widely studied in various memory platforms. An MLC *Spin-Transfer Torque Random Access Memory* (STT-RAM) implementation was provided in [83]. Chen et al. designed an access scheme for the MLC STT-RAM, at the circuit level as well as the architectural level [84]. Three different write schemes were provided based on physical principles of the resistance state transition of the MLC STT-RAM. The MLC technique has also been implemented in the flash-base memory system [85]. A multi-level address translation mechanism was proposed to accelerate the translation process in MLC flash memory storage systems [86]. Chang et al. designed a reliable memory technology device to improve the reliability of the MLC flash memory system at the device driver layer [87]. Another approach to improve the reliability of MLC flash memory, an error correcting solution concatenating *trellis coded modulation* (TCM) with an outer BCH code, was proposed by Li et al. [88]. Jung et al. presented algorithms to reduce unneces-

sary write and erase operations in the MLC flash with a buffer, to enhance the performance of the MLC flash memory [89]. Nevertheless, the above approaches focused on either improving the reliability or enhancing the performance of the MLC memory. They did not consider the efficiency of utilizing the MLC memory.

A recent trend in PCM techniques has been focused on the MLC technique [8, 14–16, 90, 91]. A number of papers focused on write techniques for a MLC PCM to obtain the tight resistance levels, by reducing the margin between two resistance levels. In [14], multi-level programming algorithms were proposed based on control of the tail-end of the programming pulse. It showed that iterative writes to program a PCM cell can provide better accuracy. A drift-tolerant MLC mechanism was proposed for the PCM memory [90]. This drift-tolerant mechanism uses the modulation coding to offer high resilience to drift. A 2 bits/cell MLC PCM cell design was proposed in [15]. An optimization design was presented for the write programming operation in the MLC PCM to improve the speed of the write [92]. A preemptable read mechanism was implemented to pause and resume iterative writes in the MLC PCM, reducing the waiting time of a read request [4]. Authors in [15] also presented a programming algorithm suitable for their MLC design. A *morphable memory system* (MMS) was proposed in [16]. This MMS switches the PCM cell between the SLC and the MLC with small hardware overhead. The adjustment is based on the statistic information of memory traffic in runtime. Another MLC/SLC PCM architecture was presented in [8]. The PCM configuration in [8] was also based on device capacity utilization in the running time. The Mercury architecture was presented to address the high-write latency and the process variation issues in the MLC PCM, by adapting different programming schemes [93]. Zhang et al. proposed the Helmet architecture to reduce the readout error rate [94]. Jagmohan et al. proposed an information-theoretic Channel Coding with Side-Information at Transmitter (CSIT) paradigm to maximize the memory capacity of the MLC PCM memory [91]. However, none of these papers considered the memory-related characteristics of applications running in the system.

4.3 Background and Model

The PCM memory

As one of non-volatile memory techniques, PCM stores data by programming the resistance of the chalcogenide, i.e., the phase-change material. When different amounts of heat are applied in the chalcogenide layer of a PCM cell, the chalcogenide material can be switched between two different states, the crystalline state and the amorphous state. Since resistances of the chalcogenide in these states are not identical, the data stored in the PCM cell can be read by simply sensing the resistance of the chalcogenide layer.

An increasing trend of research interest has been shown in the MLC operation in PCM cells. The earlier PCM techniques have been focused on the single bit operation. However, the large resistance contrast between those two states and the recent "program-and-verify" (P&V) technique enable multiple bits storing in one single cell. Assuming the resistance range of a MLC PCM device is from R_{min} to R_{max}, we can equally divide this range into 4 or 16 resistance sub-ranges for 2 bits/cell or 4 bits/cell, respectively, as shown in Fig 4.1.

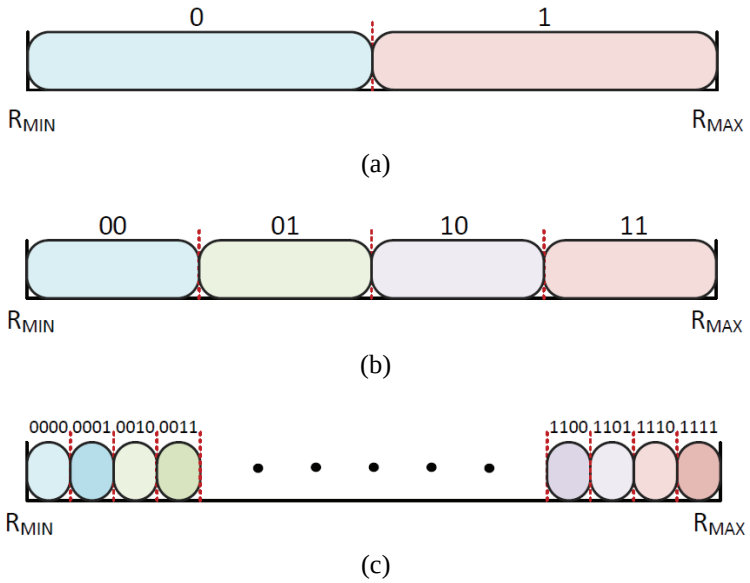


Figure 4.1: The resistance levels of a PCM cell, assuming the resistance range of the PCM cell is from R_{min} to R_{max}. (a) The SLC PCM cell, (b) the 2-bit MLC PCM cell, and (c) the 4-bit MLC PCM.

The P&V technique is widely used for the multi-bit writing in Flash memories [8]. Since the resistance distributions of multiple bit levels are non-overlapping, the P&V iteratively applies set pulse and check whether the resistance has reached the required range precisely. In details, the P&V first uses a SET-sweep pulse, which immediately followed by a RESET pulse, to program the MLC to a totally RESET state. Then a sequence of partial SET pulses is applied to the MLC, under a feedback-loop control [15]. By this approach, the MLC can be programmed to the required tight resistance range. Due to this iterative program-and-verify procedure, the write operation in MLC is more time-consuming than that in SLC [8]. Moreover, the write operation also leads to shorter endurance of the MLC.

The morphable PCM device

The advantage in the scalability of MLC has been increasingly attracting research attentions [14, 15]. However, the disadvantage in the life time and the performance has limited the implementation of MLC techniques in PCM devices [8, 16]. Since the major difference between the SLC and the MLC is the resistance ranging, the 4 bits/cell MLC can be used as a SLC or a 2 bits/cell MLC without major changes in sensing circuit. The morphable PCM cell is one of the mechanisms that can switch operation mode between SLC and MLC, based on the workload [16].

The memory capacity requirement is widely different from time to time when various applications are running. For example, the worst-case application in the SPEC CPU 2006 requires close to 1GB memory. However, most of applications in the SPEC CPU 2006 need much less memory than 1GB [16]. Thus systems with memory less than 1GB can execute most of the SPEC CPU 2006 efficiently, while they may face serious performance degradation when running the worst-case application. On the other hand, systems equipped with more than 1GB memory are not efficient at most cases. For the sake of reliability, systems are typically provisioned with more memory capacity than the required capacity for efficient executions of applications in worst-case scenarios.

The morphable PCM device can morph the memory on-the-fly [16]. By doing this, the memory runs efficiently in a low density mode, such as the SLC mode, in the common case; and switch to a high density mode, such as the 2 bits/cell MLC mode or even 4 bits/cell mode, in the worst-case scenario. The morphable memory system consists of a high-density high-latency region and a low-density low-latency region. The ratio of these two parts can be adjusted dynamically. The dynamic adjustment is decided based on the memory traffic observed by the memory monitoring circuit.

PCM + DRAM hybrid main memory

In this chapter, we focus on the optimization of the memory mode selection for system equipped with a hybrid memory architecture. This hybrid architecture consists of two parts: a DRAM array as well as a PCM memory architecture, which is similar to the morphable PCM device. The addition of the DRAM in the hybrid memory can provide better performance than that from the PCM memory. Thus, it is more realistic than the PCM-based memory architecture. We assume there are three different kinds of modes in the PCM memory: a) the SLC mode; b) the 2 bits/cell MLC mode; and c) the 4 bits/cell MLC mode.

A memory controller is the critical component to manage the PCM + DRAM hybrid main memory, as shown in Fig 4.2. In the traditional DRAM, when operating a memory request, the memory controller sends a sequence of micro commands to the memory banks. When a read miss happens in in the row buffer, a precharge command to write back a row buffer is issued before a new row is loaded. However, for the PCM, the controller always bypasses the row buffer and writes to cells directly in a write operation. Thus, the controller directly loads a row without writing back the victim row. In the PCM + DRAM hybrid main memory, we propose a memory controller with two separate sets of data and control buses, connected to the PCM and the DRAM, respectively. A multi-row buffer is equipped in the controller, loading pages from either the PCM or the DRAM. In the read operation, the

controller first checks the row buffer. If the target is in the buffer, the memory controller obtains the entry without accessing the memory bank. Otherwise, the memory controller will first decide the victim row, check whether it needs to be written back in the DRAM or it is already in the PCM. Then it will issue an activate command to move the data to an empty row in the buffer, and a read command to get the data. In the write operation, the memory controller issues the write command and sends the data directly to the memory bank, if the data address is in the PCM.

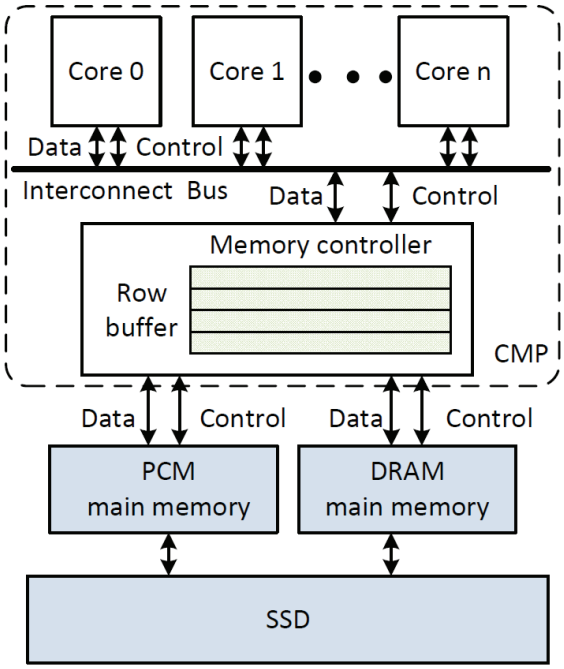


Figure 4.2: The architecture of the CMP system with PCM + DRAM hybrid main memory

Application model and problem statement

We use the *data-flow graph with pages* (DFGP) to model an application of embedded systems. A DFGP $G = \langle T, E, P, R_P, W_P, E_C \rangle$ is a *direct acyclic graph* (DAG). $T = \langle t_1, t_2, t_3, ..., t_n \rangle$ is the set of n tasks. $E \subseteq T \times T$ is the set of edges where $(u, v) \in E$ means that task u must be scheduled before task v. $P = \langle P_1, P_2, P_3, \ldots, P_m \rangle$ is the set of m pages that are required by tasks. $R_P : T \rightarrow P^*$ is the function where $R_P(t)$ is the set of

pages that task t reads. $W_P : T \rightarrow P^*$ is the function where $W_P(t)$ is the set of pages that task t writes. $E_C(t)$ represents the execution time of task t.

We consider the PCM + DRAM hybrid memory optimization for a DFGP as the combination of two parts: the task-core scheduling and the hybrid memory configuration. A task-core schedule $S_{i,j}$ is a matrix that indicates task-core assignment pairs and the execution order of tasks on each core. When $S_{i,j} \neq 0$, it represents that task i is assigned to core j, and the value is the scheduled start time of task i. Only one element in each row has a non-zero value, because each task will only be executed once. From the standpoint of the task execution, the task-core schedule tells on which core a given task will be executed and the exact start time of the execution. From the standpoint of a core, the task-core schedule indicates the task execution order of a given core and the exact start time of each task in this order. The task execution order can be obtained by sorting non-zero elements in a column of the task-core schedule S. The hybrid memory configuration $P =< R, W >$ is a pair of matrixes. $R_{i,j}$that shows in which memory mode that page i read by task j is stored in memory. $W_{i,j}$that shows in which memory mode that page i written by task j is stored in memory. In those matrixes, "0.5", "1", "2", and "4" indicate that the page is stored in the DRAM, the PCM of the SLC mode, the PCM of the 2 bit/cell MLC mode, and the PCM of the 4 bit/cell MLC mode, respectively.

Because of the parallel processing of an application, only a hybrid memory configuration is not enough for the hybrid memory optimization. Different task-core schedules lead to different memory usages at a certain time period. With the same hybrid memory configuration, some schedules may exceed the memory capacity, while some others may not. Therefore, the output of our hybrid memory optimization includes a task-core schedule S and a hybrid memory configuration P. The problem statement is given as the following:

Input: A DFGP $\langle T, E, P, R_P, W_P, E_C \rangle$, and the capacity of the DRAM and the PCM.

Output: A task-core schedule S and a hybrid memory configuration P, which subject to the following objectives:

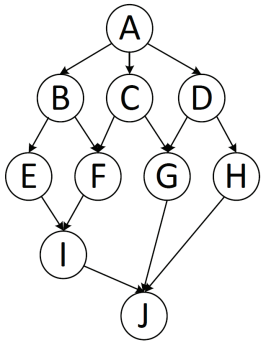
Objective 1: The memory usage should not exceed the memory capacity at any time.

Objective 2: The memory usage should be the most efficient.

The idea behind the first objective is that the exceeding memory usage results in accesses to the hard drive, which are far slower than accesses to the PCM memory, not to mention the access speed of the DRAM. And the second objective is the basic objective of our optimization. An efficient memory usage should avoid low memory usages. It should also favor the DRAM + SLC PCM mode the most, because of the low access time and the low energy consumption in this mode. And the 4 bits/cell MLC mode should be least favored, due to its long access time and high energy consumption. In the best case scenario, all pages should be stored in the DRAM all the time, which leads to the best performance and the lowest energy consumption. However, it may conflict with the first objective, where the memory capacity is not large enough for storing all pages in either DRAM or the SLC mode PCM all the time. Therefore, generating a task-core schedule S and a hybrid memory configuration P subjecting to these objectives is the key to efficiently utilize the hybrid memory. In our proposed iterative algorithms, we check the memory capacity objective for every new solution in each iteration, and only solutions that meet the memory capacity objective may be accepted. Thus, the output of our proposed iterative algorithms will satisfy the first objective, unless storing all pages in 4 bits/cell MLC mode configuration cannot meet the first objective. In addition, by evaluating solutions by our proposed fitness function, the output of our proposed algorithm favors he DRAM + SLC PCM mode the most, and configures the 4 bits/cell MLC mode as few as possible.

4.4 Motivational Example

In this section, we first give an example to show that configuring the hybrid memory can improve the performance of the CMP system and the efficiency of the hybrid memory. Considering a schedule for an application represented by the DFGP in Fig. 4.3(a) in a three-core CMP system, each task in the application needs to read pages from a shared



(a)

Task	READ	WRITE
A	P1~P5	P6~P16
B	P3,P4,P6,P11,P14	P11
C	P4,P7,P8,P12,P15	P12
D	P3,P9,P10,P13,P16	P13
E	P11,P14	P14
F	P3,P11,P12	P2
G	P6,P7,P12,P13,P15	P15
H	P2,P7,P11,P13,P16	P16
I	P2,P14	P14
J	P14,P15,P16	P16

(b)

	READ	WRITE
DRAM	250	250
PCM(SLC)	500	2000
PCM(4-MLC)	1000	4000
PCM(4-MLC)	2000	8000
SSD	10000	10000

(c)

Figure 4.3: An example of configuring the hybrid memory. (a) The DFGP of the application in the example, (b) read pages and write pages of tasks in the application, (c) read and write latency of the hybrid memory and the SSD

hybrid memory and write pages in the hybrid memory, as shown in Fig. 4.3(b). The system is also equipped with a SSD as the secondary storage. The data stored in the hybrid memory are mainly from the SSD or write operations of previous tasks. When a required page is not in the hybrid memory, it is read from the SSD.

STEP	Core 0	Core 1	Core 2	DRAM	PCM
0	SSD P1~P5				
1	R P1~P5			P3,P4	P1,P2,P5
2	EXE A			P3,P4	P1,P2,P5
3	W P6~P16			P3,P4	P6~P16
4	R P3,P4,P6,P11,P14	R P4,P7,P8,P12,P15	R P3,P9,P10,P13,P16	P3,P4	P6~P16
5	EXE B	EXE C	EXE D	P3,P4	P6~P16
6	Move P3			P3,P4	P3,P6~P16
7	W P11	W P12	W P13	P11,P12	P3,P6,P7,P13~P16
8	R P11,P14	R P3,P11,P12	R P6,P7,P12,P13,P15	P11,P12	P3,P6,P7,P13~P16
9	EXE E	EXE F	EXE G	P11,P12	P3,P6,P7,P13~P16
10	W P14	W P2	W P15	P2,P14	P3,P6,P7,P13,P15,P16
11	R P2,P14	R P2,P7,P11,P13,P16		P2,P14	P7,P11,P13,P15,P16
12	EXE I	EXE H		P2,P14	P7,P11,P13,P15,P16
13	W P14	W P16		P14,P16	P15
14	R P14,P15,P16			P14,P16	P15
15	EXE J			P14,P16	P15
16	W P16			P14,P16	P15

Figure 4.4: A task-core schedule for the applicant in this example. “SSD P1 P5” means that the content of pages P1 P5 is stored in the hybrid memory from the SSD. “R P1 P5” shows that the core reads pages P1 P5 from the hybrid memory. “EXE A” indicates that the core executes task A. “W P6 P16” represents the write operation on pages P6 P16. “Move P3” is the operation that copy the content of page P3 from DRAM to PCM in the hybrid memory. The “DRAM” and “PCM” columns show pages that need to be in the hybrid memory in each step.

In this example, we refer the number of SLC cells for storing one page as a page block, while one page block in the 4 bits/cell MLC mode can store up to four pages. The DRAM in the hybrid memory is enough for two page blocks. The settings on the memory capacity and the number of cores in this example are small, for the sake of simplicity. The settings in our experiments are highly related to the real system, as we will mention in Section 4.6. We assume that all tasks in this application require the same execution time, 1000 cycles. Reading a page from the PCM requires 500 cycles in the SLC mode, 1000 cycles in the 2 bits/cell mode, and 2000 cycles in the 4 bits/cell mode, respectively [82]. The read/write latencies of the hybrid memory and the SSD are shown in Fig. 4.3(c).

Using a simple list-scheduling algorithm, we can get a task-core schedule shown in Fig. 4.4. Note that, even though we show the schedule in step-wise, three cores do not necessarily start and end the same step at the same time. As we mentioned in the previous section, if the required page are not in the PCM memory, the system needs to request the page block back from the SSD. In this case, the SSD requires significant access overheads compared to the PCM memory accesses. Therefore, to avoid unnecessary performance degradation, the system should be equipped with the large enough size of PCM memory for the maximum memory requirement. As shown Fig. 4.4, high memory requirement occurs in step 6, where twelve page blocks are needed to be in the PCM memory.

Since memory accesses in the DRAM are both significantly shorter than that of the PCM, all DRAM blocks are used in every step in this schedule. Configuring all PCM cells in the SLC mode, the number of required page blocks is shown in the second column of Fig. 4.5. Thus, the system should have at least 12 page blocks of the PCM memory. However, we observe that the memory requirement is no more than 50% of the maximum memory requirement in 13 out of 17 steps. We show the required time of each task in Fig. 4.6. The critical path of this schedule is $\{A, D, G, J\}$. And the total execution times of this schedule are 87000, 119750, and 183750 cycles for SLC mode, 2 bits/cell MLC, and 4 bits/cell MLC, respectively.

STEP	Required page in PCM			
	SLC	2-MLC	4-MLC	Sch.
0	0	0	0	0
1	3	2	1	3(SLC)
2	3	2	1	3(SLC)
3	11	6	3	3(4MLC)
4	11	6	3	3(4MLC)
5	11	6	3	3(4MLC)
6	12	6	3	3(4MLC)
7	6	3	2	3(2MLC)
8	6	3	2	3(2MLC)
9	6	3	2	3(2MLC)
10	5	3	2	3(2MLC)
11	5	3	2	3(2MLC)
12	5	3	2	3(2MLC)
13	1	1	1	1(SLC)
14	1	1	1	1(SLC)
15	1	1	1	1(SLC)
16	1	1	1	1(SLC)
Peak	12	6	3	3

Figure 4.5: The number of page blocks required in the PCM section of the hybrid memory in each step. The "SLC" columns, the "2-MLC" columns and the "4-MLC" columns indicate that all PCM cell are configured in the SLC mode, the 2 bits/cell MLC mode, and the 4 bits/cell MLC mode, respectively. The "Sch" columns indicate the hybrid memory configuration generated by our genetic-algorithm.

Using our genetic-based algorithm presented in the next section to explore the hybrid PCM configuration space, we can find a hybrid configuration as the required page of $\{A, J\}$ are stored in the DRAM and SLC PCM mode, the required page of $\{I, H\}$ in the DRAM and 2 bits/cell MLC PCM mode, and the required page of $\{B, C, D, E, F, G\}$ in the DRAM and 4 bits/cell MLC PCM mode. This schedule has a significant improvement in the memory utilization and performance. Our schedule only needs three PCM memory blocks, which is 75% less than the SLC mode, 50% less than the 2 bits/cell MLC mode. And in 12 out of 17 steps, all three PCM blocks are used. And the total execution time is 163500, which is 11% shorter than that of the 4 bits/cell MLC mode.

Task	Total exe. time (cycles)			
	SLC	2-MLC	4-MLC	Sch.
A	74250	98500	145500	141000
B	4000	6000	10000	10000
C	3500	5500	9500	9500
D	5250	9250	17250	13250
E	2000	2500	3500	2500
F	2250	2750	3750	2750
G	5250	9250	17250	7000
H	3500	5500	9500	5500
I	1750	1750	1750	1750
J	2250	2750	3750	2250
Total	87000	119750	183750	163500

Figure 4.6: The execution time of each task, including the time of loading pages from the SSD to the hybrid memory, reading pages from the hybrid memory, executing the task, and writing pages to the hybrid memory. The red rows represent tasks in the critical path, which includes task A, D, G, and J.

4.5 Scheduling Algorithms for Hybrid Memory

In this section, we propose four different scheduling algorithms for the hybrid memory. The *Genetic Algorithms* (GA), the *Stimulated Annealing* (SA), and the *Tabu* algorithm are three iterative algorithms. In addition, we also design a heuristic algorithm to schedule the hybrid memory.

The Genetic Algorithm

The GA is a heuristic method to find the near-optimal solution in a large solution space. The GA is inspired by the process of natural evolution. In the GA, a solution is represented as a chromosome. A population, i.e., a large number of chromosomes, is generated by some low computational approaches, such as random generation or greedy heuristics. Each chromosome in the population is associated with a *fitness value*. A predefined number of iterations of evolution follow the initial population generation. In each iteration, some pairs of chromosomes are selected by a biased random selection approach. Chromosomes

with the higher fitness values are more likely selected from the population. A crossover approach is implemented on each pair of selected chromosomes to generate some new chromosomes. Some other chromosomes are also selected from the population, followed by a mutation procedure that also generates some other new chromosomes. In each iteration of the GA, the fitness values of all chromosomes in the population are evaluated, and the best chromosome is recorded. After a large number of iterations, the best chromosome in the population is translated as the selected solution. We show the genetic algorithm in Alg. 4.1. The detailed description of each step in Alg. 4.1 will be provided in the following part of this subsection.

Algorithm 4.1 The genetic algorithm

Input: A set of tasks, m different cores, PCM memory capacity MC, and DRAM memory capacity DC, predefined parameters: population size P, the number of chromosomes pairs for crossover R, the number of chromosomes for mutation Q, two threshold numbers of iterations I and G_{th}
Output: A schedule generated by the genetic algorithm

```
1: Form the initial population with the size of P
2: for i: 1 to I do
3:     Selecting R pairs of chromosomes from P_cur
4:     Create 2R new chromosomes by crossovering the R pairs of chromosomes selected above
5:     Selecting Q chromosomes from P_cur
6:     Create Q new chromosomes by mutating the Q chromosomes selected above
7:     Include the 2R + Q chromosomes in P_cur
8:     Selecting P chromosomes from P_cur for next iteration
9:     if The best chromosome has not been changed in the last G_th iteration then
10:        Break
11:    end if
12: end for
```

Representation of chromosome

In our genetic-based algorithm, we consider both the task-core scheduling and the hybrid memory configuration. We use three strings to represent a complete solution: the scheduling string, the assigning string, and the memory mode string. For a solution, these strings have the same length n, which represents the number of tasks in the application.

The scheduling string is a one dimensional representation of the DFGP. We can transform the DFGP into a string by the topological sort [95]. The scheduling string indicates

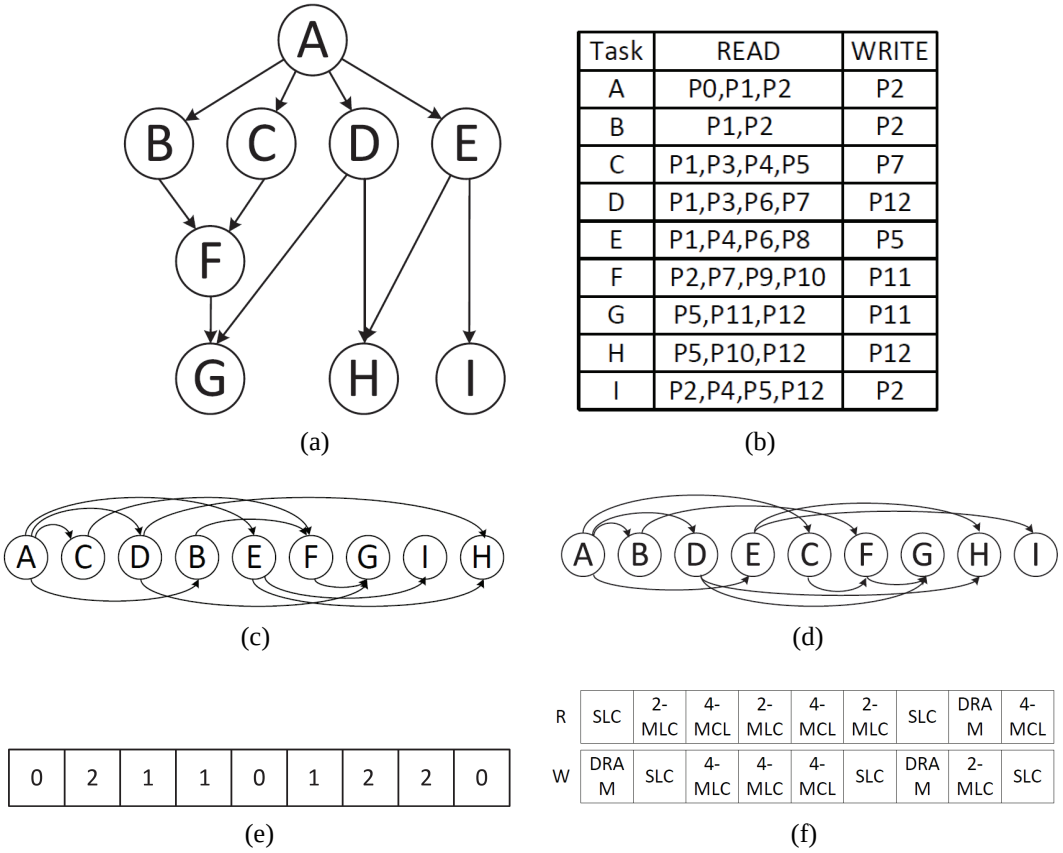


Figure 4.7: A chromosome representation of an application. (a) is the DFGP of the application. (b) the read/write pages of each task. (c)and (d) are two valid scheduling strings for the application. (e) is an assigning string for the application. (f) is a memory mode string for the application.

the scheduling order of tasks. Each task only appears once in the scheduling string. For instance, t_i placed in the fourth element of the string means that task t_i is the fourth task to be scheduled. Note that valid scheduling string representations of a given DFGP may not be unique, as long as the data dependencies are held. For example, Fig. 4.7(c) shows one valid scheduling string of the DFGP in Fig. 4.7(a). Since task A is the predecessor of tasks B, C, D, and E, task A should be placed before task B, C, D, and E in the scheduling string. In this schedule, task A is the first task to be scheduled, followed by task C, D, B, and so on. Fig. 4.7(d) shows another valid scheduling string.

The assigning string is a vector indicating task-core assignments. The value of the i-th element demonstrates the core where task t_i is assigned to in this solution. Fig. 4.7(e) is a

valid assigning string. Note that order of associated tasks is alphabetical. It is not the order indicated in the scheduling string. In Fig. 4.7(e), the first element is associated with task A, and the second element is associated with task B. Tasks A, E, and I are assigned to core 0; tasks C, D, and F are assigned to core 1; and tasks B, G, and H are assigned to core 2.

The combination of one valid scheduling string and one assigning string can be translated into a complete task-core schedule S by assigning tasks to the corresponding core in the order indicated in the scheduling string. Given a scheduling string and an assigning string, when we decide the start time of a task on a core, we set its start time as the earliest time when the core is available as well as all its predecessor tasks are finished.

The last part of the chromosome is the memory mode string, which includes strings for read and write operation. This string is also associated with tasks in alphabetical order. The value of each element represents where and in what memory mode the required pages of the corresponding task are stored. Fig. 4.7(f) shows an example of the memory mode string for the application in Fig. 4.7(a) and (b). This string indicates that the required pages of task A, i.e., $\{P0, P1, P2\}$, are stored in the SLC mode of PCM when they are read, and the written pages of task A, that is $P2$, is stored in the DRAM. In some cases, multiple tasks, which share same pages and are executed concurrently in a given schedule, may conflict in the mode string. The shared pages are stored in the mode configuration of the task appearing the earliest in the scheduling string. Therefore, in the mode configuration, pages read by the same task may not be identical. In addition, we also set a criteria for placing pages in DRAM. In the case where pages of a task are scheduled to be placed in the DRAM when the DRAM is full, we define this chromosome is not acceptable, which we will discuss later in this chapter. However, in some cases, the DRAM has some spaces available, but not enough for all pages required by the task. Therefore, we set different priorities for pages: 1) pages that are or will be written by this task, and will be read by some tasks later, have the highest priority; 2) pages that are or will be written by this task have the second highest priority; 3) pages that will be read by some tasks later have

the second lowest priority; and 4) other pages have the lowest priority. With this priority, pages with higher priorities are selected to place into the DRAM. The rest pages are placed in the PCM with the SLC mode. Based on these criteria, we can translate a memory mode string into a hybrid memory configuration P. Combining the hybrid memory configuration string and the task-core schedule, we can get a complete solution for optimizing the hybrid memory.

Initial population

In the first step of our genetic algorithm, we need to randomly generate a pre-defined number of chromosomes in the population. For the assigning string and the memory mode string, any randomly generated string is valid, as long as each element of the string is within the valid range of value. However, for the scheduling string, we have to check the data dependencies inside the string. For each task represented in the scheduling string, all its predecessor tasks should be placed before this task, and each of its successor tasks should be placed after it. Due to data dependencies, the number of valid scheduling strings may be smaller than the size of population. In this case, we can generate multiple chromosomes by combining one scheduling string with multiple pairs of assigning string and memory mode string. To ensure that there are chromosomes in the population in some extremely low memory capacity, we generate some chromosomes which all pages are stored in the DRAM + 4 bits/cell MLC mode configuration. The lowest memory usage chromosomes are the ones that schedule all tasks in one core and store all pages in the DRAM + 4 bits/cell mode, since there is only one task that requires data in the memory at a time and all data are stored in the least space-requiring mode. Thus we also include these chromosomes in the population. Finally, we need to remove multiple identical chromosomes in the population, so that every chromosome is unique. The population initialization procedure is shown in Alg. 4.2.

Algorithm 4.2 Generating initial population

Input: A set of tasks, the population size P
Output: An initial population

```
1: Initial an empty population P_int
2: while size(P_int) < P or no new valid assigning string can be created do
3:    Put all tasks in task set U
4:    Initial an empty scheduling string S
5:    while U is not empty do
6:       Put all assignable tasks in task set A
7:       Randomly select a task i in A
8:       Remove task i from U
9:       Push i into S
10:   end while
11:   Randomly form a assigning string AS
12:   Randomly form a memory mode string MM
13:   Form the chromosome C by combining S, AS, and MM
14:   Add C into P_int
15: end while
16: while size(P_int) < P do
17:   Randomly select P − size(P_int) chromosomes in P_int
18:   Modify assigning string and memory mode strings of these chromosomes
19:   Add them into P_int
20:   Remove identical chromosomes from P_int
21: end while
```

Selection

In the genetic algorithm, a small portion of chromosomes are selected from the population for the further evolution, modeling the nature's survival-of-the-fittest mechanism [96]. A proper selection procedure in a genetic algorithm should have two basic characters. First, fitter solutions should have better chances to survive, while weaker ones tend to perish. This character helps the convergence in the evolution. The other character is that the selection should be a random process. A less random selection procedure leads to small search space explored.

In our genetic-based algorithm, the first step of the selection procedure is to evaluate fitness functions of all chromosomes. The fitness function is the key to evaluate chromosomes. As we have mentioned in the previous subsection, one chromosome represents a complete task-core schedule as well as a hybrid mode configuration. Based on the schedule

and the mode configuration, we define the *fitness function* as follows:

$$Fitness = \frac{\sum_{i=1}^{n} \sum_{P_j \in R_P(t_i) \cup W_P(t_i)} size(P_j)}{\sum_{i=1}^{n} \sum_{P_j \in R_P(t_i) \cup W_P(t_i)} (MODE(i) \times size(P_j) \times I_{i,j})} \tag{4.1}$$

In the above fitness function, $MODE(i)$ relates to the i^{th} element of the memory mode string in the chromosome, where "0.5", "1", "2", and "4" represent "DRAM", "SLC", "2 bits/cell MLC", and "4 bits/cell MLC", respectively. $size(P_j)$ is the size of page P_j. $I_{i,j}$ indicates whether page P_j is stored in the hybrid memory with the mode explicated in the i^{th} element of the memory mode string. For example, assuming tasks t_1 and t_3 share the same page P_5 at the same time, and t_1 is listed before t_3 in the scheduling string, we store P_5 in the mode indicated in the 1^{st} element of the memory mode string, and we set $I_{1,5} = 1$ as well as $I_{3,5} = 0$.

This fitness function represents the average hybrid memory performance of the application, in terms of bits/cell. Since we set the definition of a valid chromosome as the one without exceeding the pre-defined maximum memory capacity, the higher the fitness function is, the less average "bits/cell" the memory is configured in the chromosome. Less average "bits/cell" in the memory leads to a better memory performance. In addition, more pages shared in the hybrid can improve the memory performance by reducing reads and writes in the memory, which is also reflected in the fitness function. Thanks to the use of "$I_{i,j}$" indicators, only one memory access is counted in the denominator of the fitness function, when there is a page shared among multiple tasks. The more pages shared, the higher the fitness function is.

After fitness functions of all chromosomes in the population are evaluated, we sort these chromosomes in the descending order of their fitness functions. The chromosomes with identical values of fitness functions are sorted arbitrarily among themselves. Then we use a *rank-based roulette wheel selection scheme* to select chromosomes [96]. In this selection procedure, the P different chromosomes are determined as the next population.

Considering the whole sorted chromosome population as a roulette wheel, each chromosome is located in a sector of this roulette wheel, based on its fitness function. To realize the "survival-of-the-fittest" of the nature evolution, we partition the roulette wheel into sectors based on fitness functions. Chromosomes with a higher value of fitness function have larger sectors in the roulette wheel. Let P denotes the population size and the S_i denote the angle of the sector representing the i^{th} rank chromosome. We also define a constant ratio $C = S_i/S_{i-1} < 1$. Thus the following equations hold:

$$S_i = C^{i-1} S_1 \tag{4.2}$$

$$\sum_{i=1}^{P} S_i = \frac{1 - C^P}{1 - C} S_1 \tag{4.3}$$

Normalizing the whole 360° of the wheel, i.e., $\sum_{i=1}^{P} S_i$ in Equ (4.3), as to 1, we can have the sector angles of the first chromosome and a given i^{th} chromosome as follows:

$$S_1 = \frac{1 - C}{1 - C^P} \tag{4.4}$$

$$S_i = \frac{1 - C}{1 - C^P} \times C^{i-1} \tag{4.5}$$

In order to keep the population size in each iteration of the evolution, we need to select P chromosomes from the population, which is usually larger than the default population due to the crossover and the mutation procedures in the last iteration. In our genetic-based algorithm, we select P random pages from the range of 0 to 1. Each of these P random pages falls in a sector mentioned above. The corresponding chromosomes are selected. Since pages are selected randomly, some of them may fall in the same sector, leading to the case that multiple identical chromosomes exist in the population. Multiple identical chromosomes do not help in improving the performance of the genetic algorithm. To avoid this, we check the P pages, and re-select any of them if they are related to the same sector. In this selection procedure, the P different chromosomes are determined as the next population.

Crossover

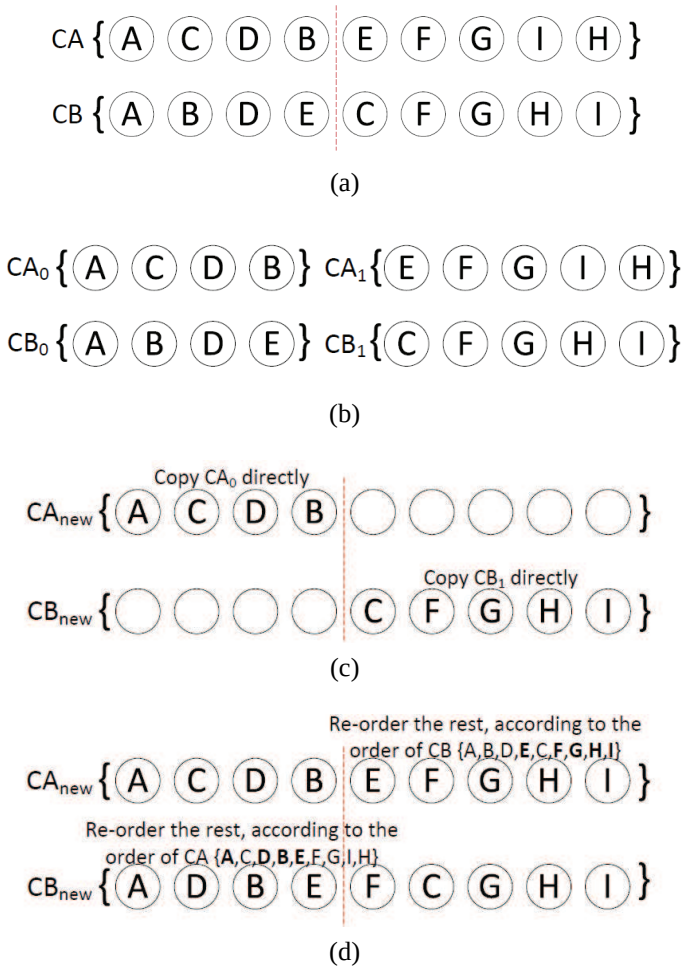


Figure 4.8: Steps of the crossover procedure on scheduling strings. (a) Two scheduling strings CA, CB, and a cutting point of 4; (b) Four strings CA_0, CA_1, CB_0, and CB_1 after cutting; (c) Forming two new scheduling strings, by copying CA_0 as the upper part of CA_{new}, and copying CB_1 as the lower part of CB_{new}; (d) Completing these two new scheduling strings by re-ordering the rest.

The traditional crossover procedure generates new chromosomes by truncating two chromosomes and jointing one part of each. Our chromosome representation consists of three strings, one of which, the scheduling string, includes the data dependencies. Hence, the crossover procedure operates differently for those three strings in a given chromosome. In the first step of the crossover procedure, we randomly select R pairs of chromosome. The pair selection is similar to the selection presented previously, by using the rank-based

roulette wheel scheme. The major difference is that the chromosomes in the population selection must be unique, while a chromosome can be selected in multiple pairs in the crossover selection, as long as no multiple pairs are identical. The implementation of the rank-based roulette wheel scheme in this selection mimics the natural fact that better individuals have better chance in reproducing offspring. Each pair of chromosomes creates two new chromosomes.

For the scheduling strings of a pair of chromosomes, we first randomly pick a cutting point, truncating each of the chromosomes into two parts. Let CA and CB denote the scheduling string of these two chromosomes, and CA_0, CA_1, CB_0, and CB_1 represent four truncated parts of these two scheduling strings. In the generation of two new chromosomes, we copy the CA_0 as the upper part of a new chromosome, and the CB_1 as the lower part of another new chromosome. For the tasks represented in the CA_1 and CB_0, we will re-order them based on the tasks order in CB and CA, respectively. In this crossover method, we keep the upper part of a string and the lower part of another string unchanged, instead of keeping the upper parts of two strings unchanged. The reason is that keeping the upper parts of two strings in crossover leads to fast convergence and poor solutions, since the upper parts of strings in the population are less likely to be changed via crossover in this case.

For example, let the scheduling string in Fig. 4.7(a) be CA, the scheduling string in Fig. 4.7(b) be CB, and the cutting is 4, as shown in Fig. 4.8(a). By truncating these scheduling strings, we have CA_0 = {A, C, D, B}, CA_1 = {E, F, G, I, H}, CB_0 = {A, B, D, E}, CA_0 = {F, C, G, H, I}, as shown in Fig. 4.8(a). To create the first new scheduling string, we copy the CA_0 as the first 4 bit of the new string, as shown in Fig. 4.8(c). Then for the tasks {E, F, G, I, H} in CA_1, we observe that their order in string CB is {E, F, G, H, I}. We place these five tasks in the last five bits of the new string in the order of {E, F, G, H, I}. Thus the first new scheduling string CA_{new} is {A, C, D, B, E, F, G, H, I}, as shown in Fig. 4.8(d). We can also get the second new string CB_{new} as {A, D,

B, E, F, C, G, H, $I\}$. By this truncate and joint procedure, we can crossover the task scheduling orders of two scheduling strings without violating data dependencies, based on Theorem 4.5.1.

Theorem 4.5.1 *Let scheduling strings $A = \{A_0, A_1\}$ and $B = \{B_0, B_1\}$ be truncated by the same cutting point. Also let $A'_1 = reorder(A_1, B)$, and $B'_0 = reorder(B_0, A)$. The reorder function $reorder(x, y)$ re-orders string x based on the order of same characters appearing in string y. If A and B maintain data dependencies, then $\{A_0, A'_1\}$ and $\{B'_0, B_1\}$ also maintain data dependencies.*

Proof: Assume $\{A_0, A'_1\}$ violates the data dependencies, which means at least one of A_0 and A'_1 strings violates data dependencies. If A_0 violates dependencies, then it contradicts to the assumption "A maintains data dependencies" in Theorem 4.5.1. If A'_1 does not satisfy the dependencies, some tasks in A'_1 are scheduled before their predecessor tasks. Since the order in A'_1 follows the order of B, the scheduling order in B does not satisfy the dependencies, which contradicts to the assumption "B maintains data dependencies" in Theorem 4.5.1. Proofing by contradiction, the new scheduling string $\{A_0, A'_1\}$ definitely maintain data dependencies. Similar proof can be applied to string $\{B'_0, B_1\}$.

Since there is no data dependency in the assigning string and the memory mode string, the crossovers in these two strings are simpler than that in the scheduling string. For two assigning strings, we randomly select a cutting point, and switch lower parts to generate new strings. The same procedure is applied to a pair of memory mode strings.

Mutation

While the crossover procedure creates two new chromosomes from two parent chromosomes, the mutation generates a new chromosome from single parent chromosome. Similar to the crossover procedure, the mutation procedure works differently on those three strings in the chromosome representation. For the assigning string or the memory mode string, we

randomly select a bit for mutation. The selected bit is changed to another randomly picked value. By switching the selected bit, a new string is generated.

(a) (b)

Figure 4.9: Steps of the mutation procedure on the scheduling string of the application in Fig. 4.3(a), assuming that task D will be the target of the mutation. (a) The flexible zone of taks D, and a random pick of replacing spot (between E and G); (b) A new scheduling string after the mutation procedure.

However, when we mutate the scheduling string, we need to consider two characteristics of the scheduling string: 1) each value (i.e. the tasks ID) should only appear once; 2) the order of the value should maintain the data dependencies. Thus, in the mutation procedure on the scheduling string, we randomly relocate the selected bit, instead of changing its value. For a given bit in the scheduling string, we define the flexible zone of this bit (corresponding to task i) as the area ranging from the corresponding bit of the last predecessor task of i, to the corresponding bit of the first successor task of i. To maintain data dependencies, a randomly relocating spot is selected with the flexible zone of the selected bit. Then we insert this bit at the relocating spot and push forward the bits between the original spot of the selected bit and the relocating spot forward. An example of the mutation procedure is shown in Fig 4.9.

Iterative evolution

In each generation of our genetic-based algorithm, we select R pairs of chromosomes for crossover, generating $2R$ new chromosomes. Q chromosomes are then picked for mutation, resulting in Q chromosomes. Therefore, there are $P + 2R + Q$ chromosomes in the population at the beginning of next generation. The selection procedure keeps the population as P. This iterative evolution stops either when the total generation reaches the

pre-defined number, or when there is no improvement in the last G_{th} generations, where G_{th} is also a pre-defined parameter.

Stimulated annealing algorithm

The SA is also an iterative optimization algorithm [97]. In our proposed SA algorithm, we use the same representation of the application as the chromosome in the GA. The basic idea of the SA is that some new generated poor chromosomes will be accepted probabilistically based on how the average "temperature" of the current generation, in order to obtain a better search of the solution space. The temperature is a metric of the population, which generally decreases in each generation. As the temperature of the populations becomes lower, the probability of accepting a poor chromosome is lower. Thus, in the beginning of the SA, poor chromosomes are more likely to be accepted, leading to a wider search in the solution space. In addition, at the end of the SA, poor chromosomes are hard to be accepted, which helps in the convergence of the search.

The initial population is created in the same way as that of our GA. The initial temperature used in our SA algorithm is the reciprocal of the average value of fitness functions, which are computed by Equation 4.1, of all chromosomes in the initial population. In an iteration, we select R pairs of chromosomes for crossover, generating $2R$ new chromosomes. Q chromosomes are then picked for mutation, resulting in Q chromosomes. For a new chromosome C_{new} generated by the mutation of chromosome C_{ori}, we compute $Rcp(C_{new}) = 1/Fitness(C_{new})$ and $Rcp(C_{ori}) = 1/Fitness(C_{ori})$. A uniform random value $r \in [0, 1)$ is selected for C_{new}. If $r > threshold(C_{new})$ and C_{new} meets the memory capacity constraint, C_{new} will be accepted and C_{ori} will be discarded. Otherwise, C_{new} will be discarded and C_{ori} will be kept. And the definition of $threshold(C_{new})$ is as Equation (4.6). For two new chromosomes generated by the crossover of two original chromosomes, we randomly pick one new chromosome and one original chromosome as a pair, and the rest as another pair. The same probabilistic accepting procedure is applied to both

pairs of chromosomes. After each iteration, the temperature is decreased by a pre-defined cooling rate, which is 90% in our design.

$$threshold(C_{new}) = \frac{1}{1 + e^{\frac{Rcp(C_{ori}) - Rcp(C_{new})}{temperature}}} \tag{4.6}$$

Tabu algorithm

The Tabu algorithm is a iterative solution search that keep track of already-searched regions so that it does not search a local space repeatedly [98, 99]. Again, we use the chromosome represents a solution.

In our proposed Tabu algorithm, we randomly generate a pre-defined number, hop_{long}, of chromosomes. Since we will start the local search with each of these chromosomes, we need to make sure that they are different from each other. For two given chromosomes C_1 and C_2, we define a long hop metric, $diff(C_1, C_2) = 1 - R_a(C_1, C_2) \times R_s(C_1, C_2) \times R_m(C_1, C_2)$, where $R_a(C_1, C_2)$ is the percentage of identical values in assigning strings of C_1 and C_2 (0.5 means half of strings are identical between them), $R_s(C_1, C_2)$ and $R_m(C_1, C_2)$ are the percentages of different values in scheduling strings and memory mode strings, respectively. When generating initial chromosomes, we accept a new initial chromosomes C_i only when $diff(C_i, C_j) > 0.5$, $\forall j \in [0, i-1]$. With this condition, we can make sure these initial chromosomes have long distance with each other in the solution space.

Starting from each initial chromosome, we conduct a local search in the solution space near this initial chromosome, which we call a region. The local search is shown in Algorithm. 4.3. At the end of the local search of a region, the best chromosome is selected. The output of this Tabu algorithm is the best chromosome among these selected chromosomes.

Algorithm 4.3 Local search in a region

Input: t tasks, m different cores, PCM memory capacity MC, DRAM memory capacity DC, a pre-defined threshold number of short hops hop_{short}, and an initial chromosome C

Output: A best schedule in the local search

```
Select random integer numbers, τ ∈ [0, t), η ∈ [0, m).
suc_hop = 0
for i = 0 to t-1 do
    ti = (τ + i) mod t
    [t_a, t_b] < − flexible zone of ti
    for j ∈ [0, m) do
        mi = (η + j) mod m
        for tk ∈ [t_a, t_b] do
            for p ∈ {0.5, 1, 2, 4} do
                for q ∈ {0.5, 1, 2, 4} do
                    Modify chromosome C by assigning task ti to core mi, insecting ti right before
                    tk, and changing its read mode to p, and write mode to q
                    Evaluate the new chromosome with Equation (4.1)
                    if The new is better and it meets the memory capacity constraint then
                        Update C as the new one, discard the old one
                        suc_hop = suc_hop + 1
                    else
                        Keep the old one, discard the new one
                    end if
                    if suc_hop ≥ hop_short then
                        BREAK
                    end if
                end for
            end for
        end for
    end for
end for
```

Hybrid memory task scheduling heuristic

To evaluate the performance of our genetic-based algorithm, we design a task scheduling heuristic for comparisons. This task scheduling heuristic is based on the Min-Min algorithm to generate task execution orders of all cores [44]. The Min-Min algorithm generates high performance schedules with comparatively low computational complexity [74]. The Min-Min algorithm schedules and assigns tasks to cores by comparing task-core pairs twice, as shown in Algorithm 4.4. A mappable task set is a set of tasks of which all predecessor tasks have been assigned. After the Min-Min task scheduling, we have task

execution orders of all cores. After scheduling a task, we use an off-line hybrid memory utilization estimator to estimate the trace of hybrid memory utilization, based on the task execution orders generated in previous steps. We define three utilization conditions: 1) the DRAM is not full; 2) the PCM utilization estimator is lower than 50% and the DRAM is full; 3) the PCM utilization estimator is between 50% and 75% and the DRAM is full; and 4) the PCM utilization estimator is higher than 75% and the DRAM is full, which is similar to the setting of the performance-aware management in [8]. When a given task is executed and it is under condition 1, all read or write pages of this task are placed in the DRAM. When DRAM is full, and condition 2 is met, all read or write pages are placed in the PCM in the SLC mode, unless the page has been loaded in the hybrid memory by any predecessor task. When it is under condition 3, pages are loaded or modified in the 2-bit MLC mode. When it is under condition 4 pages are loaded or modified in the 4-bit MLC mode. The use of the PCM utilization estimator helps in making the PCM configuration decision off-line.

4.6 Experimental results

Experiment setup

In this section, our proposed algorithms are evaluated by running benchmarks from Mibench [76] and Mediabench [100], Eight selected benchmarks are $susan$, $dijkstra$, gsm, $blowfish$, $mpeg2dec$, $mpeg4dec$, $h264dec$, and $h264enc$. We use the Simics [101] to collect the memory traces of these benchmarks, and implement them in our traced based simulator that simulates both CPU executions and memory operations. In our simulator, the CMP system has 8 cores. The details of the target CMP system is shown as Table 4.1 [63, 102, 103]. To generate applications, we create 10 groups of DAGs using TGFF [104]. Each group has 64 unique applications represented by DAGPs, and each application is composed of up to 16 tasks. We generate 32 types of tasks by scaling the memory access of eight benchmarks by 1X, 2X, 4X and 8X.

Algorithm 4.4 Hybrid Memory Task Scheduling Heuristic

Input: A set of tasks, m different cores, PCM memory capacity MC, and DRAM memory capacity DC

Output: A schedule generated by hybrid memory task scheduling heuristic

```
1: Form a mappable task set P
2: PCM utilization estimator E = 0
3: while Set P is not empty do
4:    for i: task i ∈ P do
5:       Find the core C_min(i) giving the earliest finish time of i /*The first comparison.*/
6:    end for
7:    Find the pair(k, C_min(k)) with the earliest finish time among the task-core pairs generated
      in for-loop /*The second comparison.*/
8:    Assign task k to device C_min(k)
9:    Remove k from P
10:   Update the mappable task set P, the earliest available time of core C_min(k)
11:   if DRAM is not full then
12:      Configure the read and write page of k in the DRAM,
13:   else if (E/MC) ≤ 50% then
14:      Configure the read and write page of k in the SLC mod, update E
15:   else if 50% < (E/MC) ≤ 75% then
16:      Configure the read and write page of k in the 2 bits/cell MLC mod, update E
17:   else
18:      Configure the read and write page of k in the 4 bits/cell MLC mod, update E
19:   end if
20: end while
```

To evaluate the performance of our proposed algorithms, we compare them with three different approaches. In the following part of this chapter, we use abbreviations listed in Table 4.2. In our iterative algorithms, include 1000 initial chromosomes [105], of which, 10 pairs are selected for the crossover and 10 individuals are selected for the mutation in the GA and the SA. In the GA, it ends as soon as one of the following two stopping criteria is met: 1) 1000 generations have been computed, 2) the best chromosomes have not been changed for 150 generations [106]. In the SA, it ends when the temperature is below 10^{-200}, or the best chromosomes have not been changed for 150 generations. In the Tabu, each local search ends when the threshold number of short hops is met, or the i for-loop is finished.

In Fig 4.10, we show the performance of different approaches, in terms of total execution time. The *List SLC* always has the lowest total execution time, while *List MLC*

Table 4.1: Details of the target CMP system

System	8-core CMP, 4GHz 3 GB morphable PCM memory (MLC/SLC) 1 GB DRAM memory 120 GB SSD
Memory / SSD write	DRAM: 55 ns PCM SLC: 300 ns PCM 2 bits/cell MLC: 600 ns PCM 4 bits/cell MLC: 1200 ns SSD (NAND-SLC): 200 μs
Memory / SSD read	DRAM: 55 ns PCM SLC: 80 ns PCM 2 bits/cell MLC: 160 ns PCM 4 bits/cell MLC: 320 ns SSD (NAND-SLC): 25 μs

Table 4.2: Table of Abbreviations

Abbreviation	Description
List SLC	The list-scheduling and the DRAM + uniform SLC PCM configuration
List 2 MLC	The list-scheduling and the DRAM + uniform 2 bit/cells MLC PCM configuration
List 4 MLC	The list-scheduling and the DRAM + uniform 4 bit/cells MLC PCM configuration
Heuristic	The hybrid memory task scheduling heuristic
GA	The genetic algorithm
SA	The stimulated annealing algorithm
Tabu	The Tabu algorithm

has the highest total execution time. Our proposed genetic-based algorithm has the second best performance in terms of total execution time. Since the memory access time is much longer than the task execution time, *List SLC* has the fastest speed due to the fact that it always uses the shortest access time mode. The *List 4 MLC* has the worst performance in terms of total execution time, since it always has the longest memory access time in the 4 bits/cell MLC mode. Our genetic-based algorithm reduces the total execution time by 24.5%, 101%, 10.4%, 44.0%, and 61.1%, compared to the total execution times of *List 2*

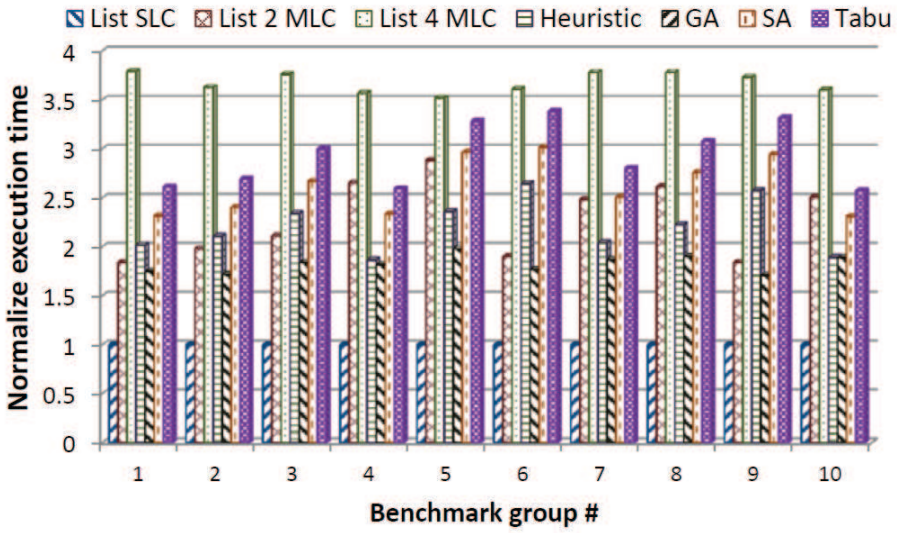


Figure 4.10: Normalized total execution times of ten groups of applications. All executions times are normalized with that of the *List SLC*.

MLC, *List 4 MLC*, *Heuristic* respectively.

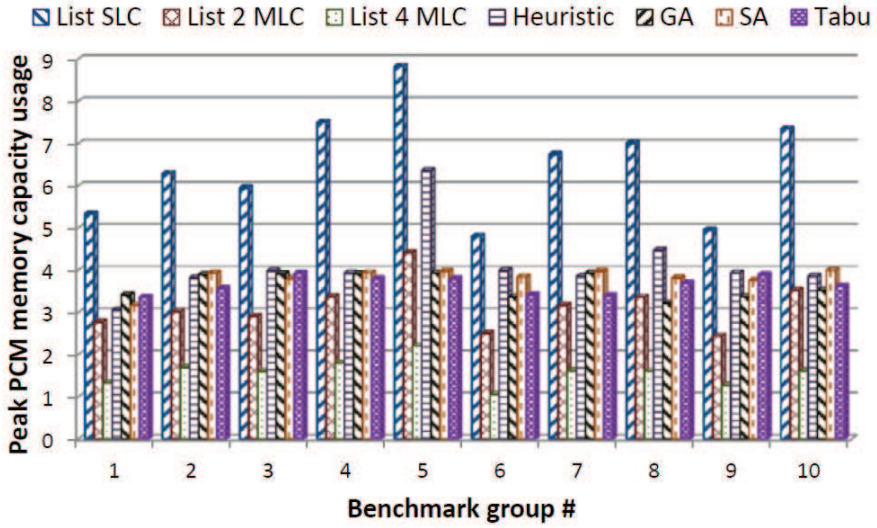


Figure 4.11: Peak memory capacity usages of ten groups of applications. The pre-defined maximum PCM memory capacity is 4 GB.

Even though *List SLC* has the fastest speed, it cannot guarantee the satisfaction of the memory capacity constraint. In our simulation, we set the size of memory as 4 GB, which is a large memory in the embedded system. For every one group of applications, as shown in Fig 4.11, ”List SLC” needs more than 4 GB memory space, exceeding from 13% to 70%.

List 4 MLC does not exceed the maximum memory capacity in all ten groups. *Heuristic* cannot guarantee that the memory capacity constraint is met. It exceeds the limit in two out of ten benchmark groups. Since we set the definition of a valid chromosome as the one without exceeding the pre-defined maximum hybird memory capacity, our three algorithms all have less than 4 GB peak memory usage in all ten groups. The *GA* achieves 76.8% and 2% average reduction of peak memory, compared to *List SLC* and *Heuristic*.

In addition, we compare the average memory usages of different algorithms, as shown in Fig 4.12. Since *List 4 MLC* always uses the high-density mode, the average usage is from 0.53 GB to 1.1 GB, averaging 19.8% of memory capacity. *Heuristic* uses 41.3% of memory capacity on average. The average memory usage of our genetic-based algorithm is from 1.6 GB to 1.96 GB, averaging 46% of memory capacity. The average memory usage of *SA* is from 1.1 GB to 2.2 GB, averaging 35% of the memory capacity. And the *Tabu* is from 0.9 GB to 1.8 GB, averaging 29.5%. Thus our genetic-based algorithm is 12.2% more efficient than *Heuristic*, 31.4% than *SA*, 58.6% than *Tabu*, and 155.6% than *List 4 MLC*.

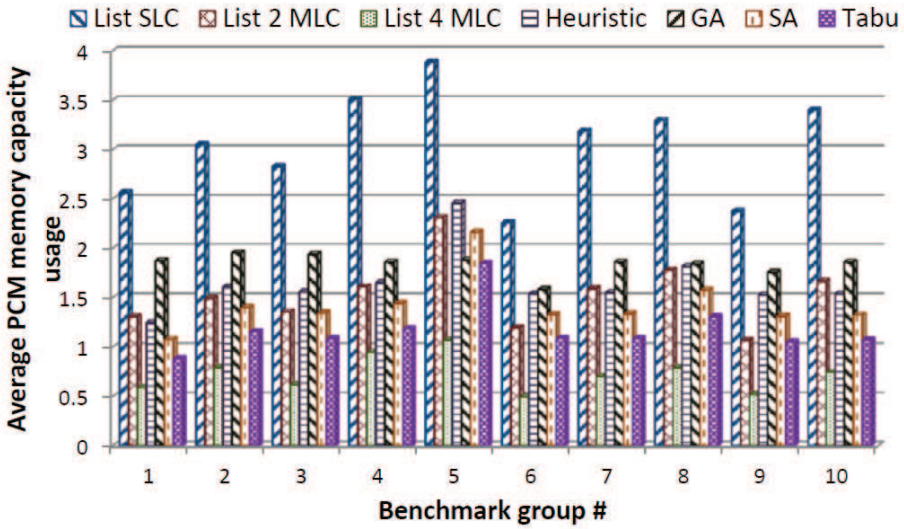


Figure 4.12: Average memory capacity usages of ten groups of applications.

To test the performance of our proposed algorithms, we compare them with the DRAM + uniform PCM mode list scheduling as well as the heuristic, in different settings of the

memory capacities. In this comparison, we set the hybrid memory capacity from 8 GB to 128 MB, where the ratio of PCM/DRAM is 3:1, as shown in Table 4.3. The memory capacity constraint has the most severe impact on the *List SLC*. Even in the largest memory capacity setting (i.e., 8 GB), the solution from this algorithm exceeds the memory capacity in one out of ten benchmark groups. The *List 2 MLC* is slightly better than the *List SLC*. However, it still fails in any benchmark group in memory capacity settings smaller than 4 GB. The *List 4 MLC* and the heuristic have similar performance in memory capacity settings smaller than 4 GB.The reason is that as the capacity setting gets smaller, pages of some single tasks require larger portions of memory. Thus, the PCM utilization estimator in the heuristic is more likely to have a value larger than 75%, resulting in more pages are stored in the DRAM + 4 bits/cell MLC mode PCM. As we set the accepting criteria as satisfying the memory capacity constraint, our three iterative algorithms successfully finds the solution that meets this constraint, in capacity settings larger than 512 MB. Our genetic-based algorithm can even successfully schedule in 512MB and 256MB. In the 128 MB setting, even the solution that sequentially executes tasks in single core and stores pages in the 4 bits/cell MLC mode, exceeds the capacity constraint. In the 8 GB, 4 GB, and 2 GB settings, the *List 4 MLC*, the heuristic, and our genetic-based algorithm can generate solutions meeting the capacity constraint in most benchmark groups. However, our genetic-based algorithm has the highest average memory usage, which is 37.2% higher than that of the *List 4 MLC*, 23.8% higher than the *Heuristic*, 18.9% higher than the *Tabu*, 11.9% higher than that of the *SA*, in the 8 GB setting. It means that our genetic-based algorithm generates solutions that utilize the hybrid memory more efficiently.

4.7 Conclusions

We present four optimization algorithms for embedded CMP systems equipped with the MLC/SLC PCM + DRAM hybrid memory. In our proposed algorithms, we not only schedule and assign tasks to cores in the CMP system, but also provide a memory configuration

Table 4.3: Comparisons of algorithms in different hybrid memory capacity settings. The "E #" columns represents numbers of solutions that exceed the hybrid memory capacity constraint. The "U %" columns indicate the average memory usage, normalized by the memory capacity. It is an average value over solutions for 10 benchmark groups. In the hybrid memory, the ratio of PCM/DRAM is 3:1.

Hybrid memory capacity	List SLC		List 2 MLC		List 4 MLC		Heuristic		GA		SA		Tabu	
	E #	U %	E #	U %	E #	U %	E #	U %	E #	U %	E #	U %	E #	U %
8 GB	1	37.95	0	19.25	0	9.11	0	42.21	0	52.71	0	46.47	0	43.75
4 GB	10	75.9	1	38.5	0	18.23	1	38.02	0	46.17	0	28.11	0	38.25
2 GB	10	151.8	10	77	1	36.45	1	51.69	0	55.97	0	43.73	0	64.02
1 GB	10	303.6	10	154	3	72.9	5	88.72	0	62.87	0	75.68	0	79.84
512 MB	10	607.2	10	308	5	145.8	8	141.71	0	58.64	2	68.16	3	86.75
256 MB	10	1214.4	10	616	10	291.6	10	308.96	0	52.19	5	126.82	7	230.63
128 MB	10	2428.8	10	1232	10	583.2	10	589.19	10	131.87	10	259.76	10	315.58

that balances the hybrid memory performance as well as the efficiency. Our experiments show that our genetic-based algorithm generates the best solutions. It significantly reduces the maximum memory usage by 76.8%, compared to the DRAM+ uniform SLC configuration, and improves the efficiency of memory usage by 155.6%, compared to the DRAM + uniform 4 bits/cell MLC configuration. In addition, the performance of the system, in terms of total execution, is also improved by 101%, compared to the uniform 4 bits/cell MLC configuration.

Chapter 5 Battery-Aware Task Scheduling in Embedded Systems

A distributed mobile DSP system consists of a group of mobile devices with different computing powers. These devices are connected by wireless network. Parallel processing in the distributed mobile DSP system can provide high computing performance. Due to the fact that most of mobile devices are battery based, the lifetime of the mobile DSP system depends on both the battery behavior and the energy consumption characteristics of tasks. In this chapter, we present a systematic system model for task scheduling in mobile DSP system equipped with Dynamic Voltage Scaling (DVS) processors and energy harvesting techniques. We propose a set of three-phase algorithms to obtain task schedules giving shorter total execution time while satisfying the lifetime constraints. The simulations with randomly generated *directed acyclic graphs* (DAG) show that our proposed algorithms generate optimal schedules which can satisfy lifetime constraints.

5.1 Introduction

The mobile computing system, which is an embedded system, has recently received tremendous attention. The interest is growing due to the benefits mobile computing brings and large number of unexplored applications. However, when applying in *digital signal processing* (DSP) area, mobile computing faces challenges which limit their usability. One of the most notable is the energy limit. Mobile devices usually are equipped with batteries. Some of them may also apply energy harvesting techniques, for example, solar cells. But in the recent two decades, the increase of processor speed is much bigger than the increase of energy density of battery. In the battery based mobile system, the loss of some mobile devices may have great impacts on the system performance. It not only leads to the loss of computation power, but also causes significant overhead of network topological re-organization. Therefore, energy consumption is important for the mobile system

application. Another limit is the computation power. Many DSP applications require considerable computation demands. Parallel processing in mobile computing system can be a solution to intensive computation requirement.

Some problems need to be solved when we apply parallel processing in mobile DSP systems: 1) how to assign tasks to the devices; 2) in what order the devices should execute the tasks assigned to them; and 3) how to schedule communication among the network. Task scheduling can solve these three problems. Task scheduling has been studied in high performance computing [107, 108]. However, a useful scheduling algorithm strongly depends on the accuracy of the model it based on. Applying task scheduling in distributed mobile DSP system, we need to develop a model for this kind of systems. Besides, task scheduling in mobile computing system should subject to some limitations, for instance, power consumption, lifetime requirement and so on.

The two major contributions of this chapter are:

- We present a complete model for task scheduling in distributed mobile DSP system, which includes application model, system model as well as energy model.
- We propose three-phase scheduling algorithms for scheduling tasks. They can generate schedules with shorter total execution time than that of traditional greedy algorithms while subject to the battery lifetime constraint.

In section 5.2, we discuss works related to this topic. In section 5.3, models for task scheduling in distributed mobile DSP system are presented. A motivational example is given in section 5.4. We propose our algorithms in section 5.5, followed by experimental results in section 5.6. Finally, we give the conclusion in section 5.7.

5.2 Related work

Task scheduling in mobile multiprocessors has been studied in the literature recently. Researches in [22, 23] focused on heterogeneous mobile ad hoc grid environments. Authors

in those works studied the static resource allocation for the application composed of communicating subtasks in an ad hoc grid. However, the goal of the allocation in those works is to minimize the average percentage of energy consumed by the application to execute across the machines, while meeting an application execution time constraint. This goal may lead to some cases in which some machines may consume much more energy than the others, even though the average consumption is minimized. So the approaches proposed in those works cannot guarantee satisfaction of the lifetime constraint in mobile DSP system. Authors in [109] proposed an energy-aware task scheduling mechanism, EcoMapS. EcoMapS incorporates channel modeling, concurrent task mapping as well as communication and computation scheduling. The scheduling algorithm in EcoMaps is based on list-scheduling, which is similar to our approach. But the WSN concerned in EcoMapS is homogenous sensor network, which means that the proposed mechanism cannot be used in the heterogeneous systems. The proposed scheduling mechanism does not consider the lifetime constraint either. In [110], the authors proposed a method of predicting the execution time of tasks based on statistics gathered from the previous instances of the same task. Authors in [24] proposed two task scheduling algorithms for embedded system with heterogeneous functional units. One of them is optimal and another is near-optimal heuristic. The task execution time information was stochastically modeled.

Weiser et al. first discussed the problem of task scheduling to reduce the processor energy consumption in [18]. An off-line scheduling algorithm for task scheduling with variable processor speed was proposed in [19]. But the tasks considered in this research are independent tasks. Authors in [20] proposed several schemes to dynamically adjust processor speed with slack reclamation based on DVS technique. A scheme for processor speed management at branches was presented in [21] based on the ratio of the longest path to the taken paths for the branch statement to the end of the program. Chandrakasan et al. showed that few voltage/speed levels can achieve almost the same energy saving as infinite levels for periodic tasks in [111]. [112] also proposed several scheduling algorithms for

periodic task. But the researches above only consider the uniprocessor system. An analytical expression to determine the optimal supply voltage under a given clock frequency was presented in [113]. In [114, 115], power constrained resource management in DVS-enable heterogeneous multiprocessors is studied. Dynamic power management in [114] used the static slack based on the degree of parallelism in the schedule. Any idle period of the processors is explored by the dynamic management. Yu et al. studied the static allocation of independent tasks in a heterogeneous system with DVS enabled in [115]. They proposed a LR-heuristic for this assignment problem. They also provided the upper bound analysis. In [116, 117], the voltage selection problem was formulated as integer programming problem. A slack allocation scheme was employed based on a conditional task graphs and resource constraints in [118]. In [25], the authors proposed a loop scheduling algorithm for voltage assignment problem in embedded system. Research in [26] focused on modeling task execution time as a probabilistic random variable. Two optimal algorithms, one for uniprocessor and one for multiprocessor DSP system, were presented to solve the voltage assignment with probability problem. The goal of these algorithms is to minimize the expected total energy consumption while satisfying the timing constraint.

Experiment conducted by Rakhmatov and Vrudhula [119] showed that the energy dissipated in the device is not equivalent to the energy consumed from a battery. When discharging, the energy consumed in battery is more than needed. In idle time, the over-consumed energy is recovered. Several analytical models on battery discharging behavior have been developed recently [119–121]. In [120], Panigrahi provided a model based on a negative exponential function. The discharging and recovery were represented as a transient stochastic process. Rakhmatov and Vrudhula [119] proposed an analytical battery model based on one-dimensional model of diffusion in a finite region. However, these two models are not suitable for task scheduling in mobile DSP system due to their high computational complexity. Ma presented an online computable battery model in [121]. The relatively low computational complexity makes it suitable for task scheduling.

5.3 Model and Background

Table 5.1: Symbols and acronyms used in Chapter 5

Name	Description
DAG	Directed Acyclic Graphs
v_i	The vertex representing the task i in a DAG
e_{ij}	The edge connecting the vertices v_i and v_j
$W(e_{ij})$	The weight of the edge e_{ij}
$T_{i\_c}(i)$	The initial communication time of the task i
$T_{exe}(i)$	The execution time of the task i
$T_{res}(i)$	The time of sending the result data of task i back to the manager node
$D(i)$	The device executing the task i in a given schedule
BW_d	The network bandwidth of device d
SP_{ij}	The speed of device j executing task i
$M_p(i)$	The size of processing data of the task i
$M_r(i)$	The size of result data of the task i
δ_i	The ith power-on period in the battery behavior model
τ_i	The ith power-off period in the battery behavior model
t_i	The beginning time of period δ_i
T	The entire lifetime of the battery when used in greedy mode
$\Delta\alpha$	The dissipated energy
β	A constant in battery behavior model
$\zeta_i(t)$	The residual discharging loss at time t in period τ_i
E_j	The initial capacity of the battery in the device j
CUR_{ij}	The discharge current of the device j when running the task i
CUR_T_j	The discharge current of the device j when communicating with others
C_{lt}	The lifetime constraint
EST	The earliest start time of a task in a DAG
LST	The latest start time of a task in a DAG
CN	The critical node in a DAG
DAT	The device available time
TAT	The task available time
LPFT	The latest predecessor-finish time
ESST	The earliest successor-start time
BITS	The backward independent task set

Application model

In this chapter, we use the *Directed Acyclic Graphs* (DAG) to represent the DSP applications. A DAG $T = (V, E)$ consists of a set of vertices V, each of which represents a task in the application, and a set of edges E, showing the dependencies among the tasks. The edge set E contains edges e_{ij} for each task $v_i \in V$ that task $v_j \in V$ depends on. The weight of a vertex v_i represents the task type of the task i. Also the weight of an edge e_{ij} means the size of data which is produced by v_i and required by v_j. For the convenience of the reader, we list the symbols and the acronyms used in the rest of this chapter in Table 5.1.

Given an edge e_{ij}, v_i is the immediate predecessor of v_j, and v_j is called the immediate successor of v_i. A task only starts after all its immediate predecessors finish. Tasks with no immediate predecessor are entry-tasks, and tasks without immediate successors are exit-tasks.

System model

In this study, we assume that a number of mobile devices are deployed in a certain area of space. All these devices and an extra task manager node are connected by a wireless network. The task manager node assigns tasks to the mobile devices and monitors the executions of those tasks. Different mobile devices have various computation power and characteristics. The network bandwidths are also different from device to device. The following assumptions are made:

- A device can compute and communicate with others simultaneously.
- Data communications are point to point. Routing is beyond the scope of this chapter. A device can only communicate with one other device at a time. The energy consumption during communication cannot be ignored.

Here is an example of how the system assigns, executes tasks and collects the result. First of all, the task manager node assigns task i to a device d. Meanwhile, the devices

where the immediate predecessors of i are executed send the required data to the device d. This initial communication time T_{i_c} can be computed as follow:

$$T_{i\_c}(i) = \sum_{k \in pred(i)} \frac{W(e_{ki})}{BW_{D(k),d}} \tag{5.1}$$

$D(k)$ is the device which runs the task k, $BW_{D(k),d}$ is the network bandwidth between $D(k)$ and d, i.e., the smaller bandwidth of these two devices. $W(e_{ki})$ means the size of data which is required by i and produced by k. Only when the predecessor task and the current task are executed on the same device, the communication is not required because of the already existing data. Once device d receives all the predecessors result data, it begins the execution of the task. The execution time T_{exe} depends on the speed of executing task i on the device d, SP_{id}, and the size of the processing data $M_p(i)$:

$$T_{exe}(i) = SP_{id} \times M_p(i). \tag{5.2}$$

After computing the result data, device sends its result back to the task manager node if the current task is the exit-task. The time of sending the result to task manager node T_{result} ideally should be proportional to the product of the size of result data $M_r(i)$ and the network bandwidth of the assigned device d BW_d:

$$T_{res}(i) = \frac{M_r(i)}{BW_d}. \tag{5.3}$$

When a non-exit-task is done, the device will communicate with device which needs data from it and start the procedure of the next task assigned to it.

Battery behavior

Nickel-cadmium and lithium-ion batteries are the most commonly used batteries in mobile devices. These kinds of batteries consist of an anode and a cathode, separated by an electrolyte. When a battery is connected to a load, a reduction-oxidation reaction transfers electrons from the anode to the cathode. Active species are consumed at the electrode surface and replenished by diffusion from the bulk of the electrolyte. However this diffusion

process cannot keep up with the consumption. A concentration gradient builds up across the electrolyte. When this concentration falls, the battery voltage drops. When the voltage is below a certain cutoff threshold, the electrochemical reaction cannot be sustained at the electrode surface anymore, so the battery stops working. But in fact, the active species which has not yet reached the electrode are not used. This unused charge is called discharging loss. Discharging loss is not physically lost but simply unavailable. If the battery current is reduced to a low value or even zero before the battery stops working, the concentration gradient flattens out after a sufficiently long time. The remaining active species reach the electrode again. Then the discharging loss is available for extraction. This procedure is called the battery recovery [121]. Experiments show that this discharging loss might take up to 30% of the total battery capacity [121].

Precise battery behavior model is essential for optimizing system performance. The battery behavior model used in this chapter is based on Ma's approach [121]. Consider the scenario where a battery is turned on for δ_i time, and turned off for τ_i time ($i = 1, 2, \ldots$). This on-off period is repeated until the battery dies. We assume that the discharging current of the battery in epoch δ_i is I_i, and the beginning time of this epoch is t_i. The energy dissipated by the battery in epoch δ_i is:

$$\Delta\alpha = I_i \times \delta_i + 2I_i \times \sum_{m=1}^{\infty}[\frac{e^{-\beta^2 m^2(T-(t_i+\delta_i))} - e^{-\beta^2 m^2(T-t_i)}}{\beta^2 m^2}] \tag{5.4}$$

The model is interpreted as follows. The first term in the right-hand side of (5.4) is simply the energy consumption during the epoch δ_i. And the second term is the discharging loss during the δ_i epoch. T is the entire lifetime of the battery when the battery is on until it dies (greedy mode). β is a positive constant, which is determined in experiment and may vary from battery to battery. An idle period τ_i follows the epoch δ_i. The battery is turned off when the device has finished the current task and is waiting for the next task. The residual discharging loss when it is t time after epoch δ_i can be computed as:

$$\zeta_i(t) = 2I_i \times \sum_{m=1}^{\infty}[\frac{e^{-\beta^2 m^2(T+t-(t_i+\delta_i))} - e^{-\beta^2 m^2(T+t-t_i)}}{\beta^2 m^2}] \tag{5.5}$$

$\zeta_i(0)$ equals to the discharging loss of δ_i. Note that this residual discharging loss is just a potential energy in the sense that it only makes sense when the battery is alive. Once the battery dies, this residual energy will not be recovered. When the battery is alive during the τ_i period, the energy recovered at the end of the τ_i period is:

$$\Delta\alpha_r(\tau_i) = \zeta_i(0) - \zeta_i(\tau_i) \tag{5.6}$$

Energy model

Energy harvesting

In mobile computing, the CPU speed increases exponentially from 90s. However, the increase of energy density in battery is much smaller than the increase of CPU speed [122]. Energy consumption becomes one of the bottlenecks of the mobile computing. New technologies such as micro fuel cells can recharge handheld devices with power plants the size of candy bar. But these technologies are only powerful enough for devices with low energy consumption, such as the wireless sensor nodes [122]. Laptop-sized handheld devices are too big to be powered by this kind of microcells. Meanwhile, energy harvesting is another approach to solve this problem. In energy harvesting, many different techniques can transfer various kinds of ambient energy to power electronics. Some common techniques include using background radio signals as power reservoir, broadcasting RF energy to power remote devices, collecting energy from ambient light or heat, and harvesting energy from vibrational excitation. Table 5.2 shows performances of various energy harvesting opportunities.

In this chapter, we assume that every mobile device in the system is equipped with a rechargeable battery connected to an energy harvester. We also assume there are three types of energy harvesters: “fast”, like the solar cell directed toward bright sun; “slow”, like the RF energy broadcasting; and “disable”, i.e., no energy harvesting. Table. 5.3 shows the details of energy harvesters.

Table 5.2: Harvesting performance of various energy sources

Energy source	Performance
Background radio signals	less than 1 μW/cm^2 [123]
RF energy	1 to 100 μW [122]
Ambient light	100 mW/cm^2(under bright sun)
	100 μW/cm^2(illuminated room) [122]
Ambient heat	60 μW/cm^2 [124]
Vibrational excitation	800 μW/cm^2(machines-kHz) [125]

Table 5.3: Harvesting power and recharge current from fast and slow harvesters

	Harvesting Power	Recharge current (voltage = 1.2v)
Fast	500 mW	416.7mA
Slow	10 mW	8.3mA

Dynamic voltage scaling modes and lifetime constraint

We consider the distributed mobile DSP system in which the mobile devices are equipped with Dynamic Voltage Scaling (DVS) processors. In order to reduce the energy consumption, DVS technique jointly decreases the processor speed and the supply voltage. Research in [43] shows that the decrease in processor voltage causes nearly linear increase in execution time and approximately quadratic decrease in energy consumption. Without loss of generality, we assume that each processor has three DVS modes, denoted as L_1, L_2, L_3. The supply voltage of L_i is half of the supply voltage of L_{i-1}. Table 5.4 shows the relationships among the DVS modes when task i is executed by device j.

Table 5.4: Parameters in DVS modes

DVS mode	Supply voltages	Processor speeds	Battery discharge current
L_1	U_j	SP_{ij}	CUR_{ij}
L_2	50%$\times U_j$	66%$\times SP_{ij}$	50%$\times CUR_{ij}$
L_3	25%$\times U_j$	57%$\times SP_{ij}$	25%$\times CUR_{ij}$

Devices in the mobile DSP system are powered by batteries. As discussed above, some of these batteries can re-gain energy from the harvesting techniques. Some definitions used

in the rest of the chapter are given as follow. E_j is the maximum and initial capacity of the battery in device j. CUR_{ij} is the discharge current of device j when running task i. When device j is transmitting data, the discharge current is CUR_T_j. When the remaining energy of a battery is lower than a threshold value (we assume it is 5% of the maximum capacity), device cannot finish the rest assigned tasks if the discharge current is larger than the harvesting current. We say the device dies at that point of time. Given a task schedule, we can calculate the energy consumption and the dead time of the devices with equation (5.4), (5.6) as well as the recharge current of their energy harvesters. The lifetime of the whole system is the time when the earliest device dies.

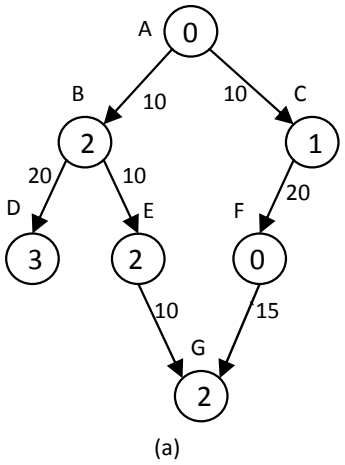
In this chapter, the objective of our schedule method is to minimize the total execution time of tasks when the system lifetime is larger than a pre-determined lifetime constraint C_{lt}. Note that if all the devices can finish the assigned tasks, we set the lifetime of the whole system as infinite.

5.4 Motivational Example

Example of application and mobile system

First we give an example for task scheduling in distributed mobile DSP system. In this chapter, we assume that applications have already been preprocessed. We already know how tasks in the applications are represented in the form of DAG. For example, a DAG of an application is shown in Figure 5.1(a).

In Figure 5.1(a), there are 7 different tasks, each of which has a weight value indicating the type of that task. For example, the task A is a task of type 0. There are 4 different types of tasks in our example. The weights of edges mean the sizes of required data for the successors. The weight 10 of the edge between task A and B means that the size of data which are required by B and generated by A is 10. More details of tasks in our example are provided in Figure. 5.1(b). Also, we assume that there are 2 mobile devices in our example. Figure. 5.1(d) shows the characteristics of the devices. As discussed previously



(a)

Type	Size of task data	Size of result data
0	20	10
1	40	30
2	80	20
3	60	15

(b)

Type of task	D0		D1	
	speed	current	speed	Current
0	20	20	35	25
1	10	100	20	30
2	40	40	15	55
3	30	20	25	60

(c)

Devices	Bandwidth	Current when communication	Mode of energy harvesting	Battery capacity	Lifetime constraint
D0	1	100	slow	100000	4500
D1	2	150	disable	200000	4500

(d)

Figure 5.1: An example of application and mobile system. (a) a DAG, (b) data sizes of task types, (c) heterogeneous characteristics of mobile devices and (d) details of two mobile devices.

in this chapter, when running a task, different devices have different speeds and require different energy consumptions (in the form of current). We show the differences in Figure. 5.1(c).

Table 5.5: EST and LST of tasks in the DAG

Task	A	B	C	D	E	F	G
EST	0	550	550	2750	2750	1150	4950
LST	0	550	3800	2750	2750	4400	4950
CN	yes	yes	no	yes	yes	no	yes

Based the list-scheduling algorithm (discussed in section 5.5), we compute the EST and the LST of each task in the DAG, shown in Table. 5.5. A priority task list of the example DAG is generated as [A, B, D, E, C, F, G]. Then we select tasks from the top of the list to bottom and assign them to devices which can finish them at the earliest time. A schedule generated by list-scheduling is shown in Figure. 5.2. Using equation (5.4), (5.6) as well

as the energy harvesting currents, we can calculate the lifetimes of these two devices. We find out that at time 4037 (mins), the battery of device D1 dies. In this case, device D1 cannot satisfy the lifetime constraint (4500). Also the whole application is not executed completely.

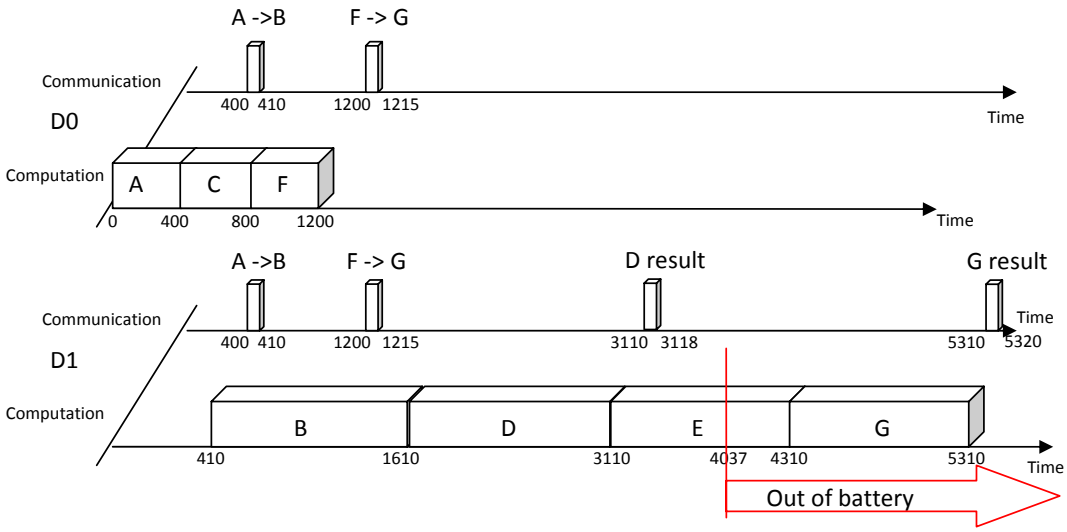


Figure 5.2: A schedule generated by list-scheduling.

Our solution

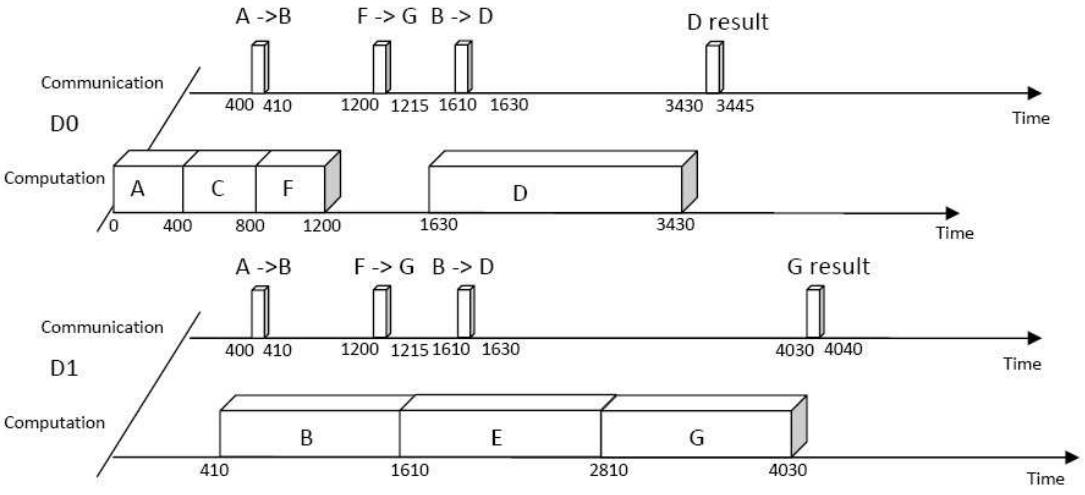


Figure 5.3: A modified schedule.

Since the list-scheduling does not consider the energy consumptions and the batteries' lifetimes, tasks are likely assigned to some machines which are generally faster and consume more energy. In our example, the schedule generated by list-scheduling assigns more tasks in device D1 than in device D0. While the D1 is out of battery, the battery of D0 still has 514205 (mAmin) left. So in this case, a proper way to find a schedule which satisfies the lifetime constraint is moving some tasks in the constraint-violating devices to some other devices with redundant energy. In our example, we move the most energy-consuming task in the device D1, the task D, to the device D0, shown as in Figure. 5.3. After calculating the energy consumptions and the lifetimes of these two devices, we find out that in this new schedule both two devices can finish their tasks without running out of batteries, which means the lifetime constraint is met. What's more, the total execution time in this schedule is surprisingly shorter than the one in the original schedule. The former is 4040 mins and the latter is 5320 mins (without considering the batteries' lifetimes). Since in this new schedule, the devices are able to finish all tasks in their full speeds, DVS adjustment is not needed in this cast. In the case where re-assignment still cannot find the suitable schedule, DVS adjustment may generate a schedule meeting the lifetime constraints.

In the next section, we will discuss our three-phase algorithms which deeply explore the solution space to find the optimal meeting the lifetime constraints.

5.5 Three-phase constraint-aware algorithm

In our proposed algorithm, a baseline algorithm generates an initial schedule without considering energy consumption and lifetime constraint. Then a re-scheduling algorithm adjusts the schedule so that the lifetime constraints are met. This re-scheduling algorithm jointly considers both re-assigning task-device pairs and switching of DVS mode of the device. Finally, in the phase three, we further explore the solution space and find a better schedule satisfying the lifetime constraint.

Phase I: Baseline scheduling

In phase I, we try to find a simple baseline schedule without considering the constraint. The greedy algorithms can solve this problem with low computational complexity. We use two kinds of baseline greedy algorithms in this chapter: list-scheduling and Min-Min algorithm. Two definitions used in rest of this section are provided as follow. *Device available time* (DAT) is the time when the device finishes all the tasks which are previously assigned to this device. *Task available time* (TAT) is the time when all the predecessors of this task are finished. These two definitions are based on the scheduling decisions made in the previous steps of the algorithm.

list-scheduling

The list scheduling used in phase I is similar to CPNT [108]. Some definitions used in listing the task are provided as follow. The *earliest start time* (EST) and the *latest start time* (LST) of a task are shown as in (5.7) and (5.8).The entry-tasks have EST equals to 0. And the LST of the exit-tasks equal to their EST.

$$EST(i) = \max_{m \in pred(i)} \{EST(m) + AT(m)\} \tag{5.7}$$

$$LST(i) = \min_{m \in succ(i)} \{LST(m)\} - AT(i) \tag{5.8}$$

CPNT in [108] targets homogeneous system. The system concerned in this chapter is heterogeneous. The execution times of a task on different devices are not the same. $AT(i)$ is the average execution time of task i. The critical node (CN) is a set of vertices in the DAG of which EST and LST are equal. Algorithm 5.1 shows a function forming a task list based on the priorities.

Once the list of task is formed, we can assign tasks to devices in the order of this list. The task on the top of the list is assigned to the device which can finish it at the earliest

Algorithm 5.1 Forming a task list based on the priorities

Input: A DAG, Average execution time AT of every task in the DAG
Output: A list of tasks P based on priorities.

```
1: Calculate the EST of every task.
2: Calculate the LST of every task.
3: Empty list P and stack S, and pull all tasks in the list of task U
4: Push the CN task into stack S in the decreasing order of their LST, and remove them
   from U
5: while The stack S is not empty do
6:    if top(S) has immediate predecessors in U then
7:       S ←the immediate predecessor with least LST
8:       Remove this immediate predecessor from U
9:    else
10:      P ← top(S)
11:      Pop top(S)
12:   end if
13: end while
```

time. Then this task is removed from the list. The procedure repeats until the list is empty. A schedule is obtained after this assigning procedure which is shown in Algorithm 5.2.

Algorithm 5.2 The assigning procedure

Input: A priority-based list of tasks P, m different devices, SP_{device} matrix
Output: A schedule generated by list-scheduling.

```
1: while The list P is not empty do
2:    T = top(P)
3:    Find the device D_min giving the earliest finish time of T
4:    Assign task T to device D_min
5:    Remove T from P
6:    Update DAT of device D_min and TAT of successors of T
7: end while
```

Min-Min algorithm

Min-Min is another popular algorithm [44]. The original Min-Min algorithm does not consider the dependencies among tasks. So in the Min-Min baseline algorithm used in this chapter, we need to update the mappable task set in every step to maintain the task dependencies. Tasks in the mappable task set are the tasks of which all the predecessor tasks are finished. Algorithm 5.3 shows the pseudo codes of the Min-Min algorithm.

Algorithm 5.3 Min-Min algorithm

Input: A set of tasks, m different devices, SP_{device} matrix
Output: A schedule generated by Min-Min.

1: Form a mappable task set P
2: **while** Set P is not empty **do**
3: **for** i: task $i \in P$ **do**
4: Find the device $D_{min}(i)$ giving the earliest finish time of i
5: **end for**
6: Find the pair$(k, D_{min}(k))$ with the earliest finish time among the task-device pairs generated in for-loop
7: Assign task k to device $D_{min}(k)$
8: Remove k from P
9: Update the mappable task set P, DAT of device $D_{min}(k)$ and TAT of successors of k
10: **end while**

Phase II: constraint-aware rescheduling

To satisfy the lifetime constraint, we need to conduct a re-scheduling if the schedule obtained in the previous phase violates the lifetime constraint. First of all, we examine the battery lifetimes of all devices. Devices violating the lifetime constraint, which are called urgent devices, will be pushed into a list. This phase II approach includes three part: DVS adjusting, task re-assigning and execution re-ordering. Some definitions are used in follows. Given a schedule, the latest predecessor-finish time of a task i LPFT(i) is the latest time when all its predecessors are finished and have all the required data sent to the device executing i. LPFT(i) is the earliest start time of i without violating the task dependencies. The earliest successor-start time of a task i ESST(i) is the earliest time when any of its successors is scheduled to start the data communication with i. The execution zone of i is the time between LPET(i) and ESST(i). Obviously, as long as i starts and completes in its execution zone, no matter how long the execution time is, the task dependencies are hold and the successor tasks of v_i are not delayed. A target task is the task to be re-scheduled. A target device is the device to which the target task is re-assigned.

The DVS adjusting in Phase II try to reduce the energy consumption while maintain the original baseline schedule. In order to avoid any impacts on the executions of other tasks,

we change the DVS mode of the device so that the device is still able to complete the target task in the task's execution zone. Algorithm 5.4 shows the function of DVS adjusting.

Algorithm 5.4 DVS(S,CS,D_u), a function of adjusting the DVS modes

Input: A schedule S, battery lifetime constraint CS, an urgent device D_u.
Output: A DVS adjusted schedule

1: Generate a list of tasks U in the order of decreasing energy consumption. These tasks was assigned to D_u in the original schedule.
2: **while** U is not empty and device D_u violates the lifetime constraint **do**
3: T = top(U)
4: **if** T can be finished in its execution zone assuming DVS mode of D_u is set to L_3 **then**
5: Set DVS mode of D_u as L_3 when running T
6: **else if** T can be finished in its execution zone assuming DVS mode of D_u is set to L_2 **then**
7: Set DVS mode of D_u as L_2 when running T
8: **else**
9: Keep original DVS mode
10: **end if**
11: Compute the lifetime of deviceD_u.
12: **end while**

If the DVS adjusting cannot provide a new schedule satisfying the lifetime constraints, the task re-assigning will reassign tasks in urgent device to another device (target device). Several criteria are used to determine target devices for a given target task:

1. Target device should not be the urgent device.

2. Target device should be idle in the execution zone of the target task.

3. The devices with predecessors and/or successors of the target task are preferred.

The idea behind 2) is that when re-assigning the target task to a device which is idle in the execution zone, the successors of target task and the following tasks in the task list of target device won't be delay. So the total finish time of this device is the same as the original one. When choosing the target device in 3), the total finishing time may be shorter, due to the fewer data to communicate. So when we fill the target devices set, we first choose the devices satisfying all three conditions. Then we select the ones satisfying condition 1) and

one of the other two. Last we choose the non-urgent devices. The urgent devices can not be the target devices. A Function of re-assigning a given target task is shown in Algorithm 5.5.

Algorithm 5.5 reassign(S,CS,D_u), A Function of reassigning tasks from an urgent device to other device

Input: A schedule S, battery lifetime constraint CS, an urgent device D_u.
Output: A reassigning schedule

1: Generate a list of tasks U in the order of decreasing energy consumption. These tasks was assigned to D_u in the original schedule.
2: **while** U is not empty and device D_u violates the lifetime constraint **do**
3: T = top(U)
4: Find a set of target devices P of T
5: **while** P is not empty **do**
6: S_{temp} = reassign T to top(P)
7: Add S_{temp} into SS, SS is a set of schedules.
8: Remove top(P) from P
9: **end while**
10: Find the best schedule S_{best} in SS which has the longest system lifetime.
11: S = S_{best}
12: Remove T from U
13: Empty SS
14: **end while**

If the urgent device still violate the lifetime constraints after reassigning, we will re-order the task execution orders. Let's assume when device d is running task v_k, the battery runs out of the energy. This device d will either complete task v_k with the energy from harvester if the recharge current from harvester is larger than CUR_{kd}, or just stop if the recharge current is not large enough.

In the latter case, as discussed in section 5.3, this device dies. However, if there is some tasks satisfying conditions listed below, we can re-order the execution order as shown in Algorithm 5.6, so that device d can execute these tasks before v_k. In this way, we can further prolong the lifetime of device d. The whole re-scheduling algorithm is provided as Algorithm 5.7.

1. The tasks are assigned the same machine as task v_k. And they are scheduled to run

Algorithm 5.6 re-order(S,D_u), a function of re-ordering the execution order in the urgent device

Input: A baseline schedule S, an urgent device D_u
Output: A re-ordering schedule

1: Find the task T which device D_u is executing when it dies.
2: In the task list of device D_u, find tasks which satisfy the conditions of re-ordering and push them in a set of task U
3: Move all the tasks in U before T in the execution order.
4: Update schedule S

Algorithm 5.7 The constraint-aware rescheduling procedure

Input: A baseline schedule S, battery lifetime constraint CS
Output: A schedule generated by The constraint-aware rescheduling.

1: A list of urgent Devices U is generated.
2: **while** The list U is not empty **do**
3: $D_u = top(U)$
4: DVS(S,CS,D_u)
5: **if** D_u violates the constraint CS **then**
6: Reassign(S,CS,D_u)
7: **end if**
8: **if** D_u still violates the constraint CS **then**
9: Re-order(S,D_u)
10: **end if**
11: **end while**

after task v_k in the original schedule.

2. The tasks are independent with task v_k.

3. The tasks are ready to run at the time when task v_k is scheduled to start.

4. The discharge currents of device d running these tasks are lower than the recharge current from harvester.

Phase III: Push-Pull algorithm

In most of the cases, the schedules generated in Phase two have longer total execution times than the baseline schedules do. So, we try to find a better schedule satisfying the lifetime constraint based on the schedule we get in phase II. We implement the Push-pull algorithm [126] in this phase III. The Push-pull algorithm is an iterative algorithm as shown

in Algorithm 5.8. It improves the schedule by repeating the push operation and the pull operation.

Algorithm 5.8 The push-pull algorithm

Input: A baseline schedule S generated in phase II, battery lifetime constraint CS
Output: A schedule generated by The push-pull algorithm.

1: count = 0
2: **while** count less than 500 AND improvements exist in last 5 iteration **do**
3: count++
4: S_{push} = PUSH(S,CS)
5: S_{pull} = PULL(S,CS)
6: S = the one with the shortest total execution time among S, S_{push} and S_{pull}
7: **end while**

Two definitions are used in the Push-pull algorithm. The critical tasks path is a path of tasks which has the biggest impact on the total execution time in a given schedule. We can find the critical tasks path by traversing the DAG. Among the exit-tasks, the one finishing at the latest time is pushed into a stack S. Then the predecessor of $top(S)$ with the latest finishing time is pushed into S. This process repeats until an entry-task is found. S is the critical tasks path of this given schedule. The length of the critical tasks path is the total execution time of the given schedule. The backward independent task set (BITS) of a task i in a given schedule is a set of tasks meeting the following conditions: 1) scheduled to the same device as D(i); 2) scheduled to execute prior to i; 3)independent with i.

The Algorithm 5.9 and 5.10 show the details of push operation and the pull operation. Target devices for re-assigning in push operation are selected in the same method as the reassigning target conditions in phase two. The “acceptable” condition in these two operations is that the new schedule should satisfy the lifetime constraint and have a shorter total execution time than the original one.

Algorithm 5.9 PUSH(S,CS), The push operation

Input: A baseline schedule S generated in phase II, battery lifetime constraint CS
Output: A schedule generated by The push operation.

```
1: Find the critical tasks path CP of S
2: while CP is not empty do
3:    trigger = Pop(CP)
4:    Find the BITS(trigger)
5:    while BITS(trigger) is not empty do
6:       target_task = top(BITS(trigger))
7:       P = taget devices for re-assigning
8:       while P is not empty do
9:          S_temp = re-assign target to top(P)
10:         if S_temp is acceptable then
11:            S_push = S_temp
12:            Re-assign target_task to top(P)
13:         end if
14:         Remove top(P) from P
15:      end while
16:      Remove target_task from BITS(trigger)
17:   end while
18:   Remove trigger from CP
19: end while
```

Algorithm 5.10 PULL(S,CS), The pull operation

Input: A baseline schedule S, battery lifetime constraint CS
Output: A schedule generated by The pull operation.

```
1: Find the device d which finishes its task list in the earliest time
2: Form T, a list of tasks executed in d
3: while T is not empty do
4:    trigger = top(T)
5:    Form P, a list predecessors of trigger, which are not executed in d
6:    while P is not empty do
7:       target_task = top(P)
8:       S_temp = re-assign target to d
9:       if S_temp is acceptable then
10:         S_push = S_temp
11:         Re-assign target_task to d
12:      end if
13:      Remove target_task from P
14:   end while
15:   Remove trigger from T
16: end while
```

5.6 Experimental results

Experiment setup

We evaluate the performance of the three-phase constraint-aware algorithms through simulations. Each simulation run (total 10 runs) has 64 unique applications, and each application is composed of up to 16 tasks. For each task, the maximum fan-in and fan-out are both 3. There are 32 devices in the mobile DSP system. We set parameters in the model randomly between the maximum and minimum values shown in Table 5.6. δ of all batteries are set to 0.1. Lifetime constraint for all devices is set to 500.

Table 5.6: Ranges of model parameters

parameter	Minimum	Maximum
SP_{ij}	10	40
CUR_{ij}	20	100
$M_p(i)$	20	148
BW_d	2	10
E_i(mAmin)	1.0×10^5	8.0×10^5
CUR_T_j	20	400

Result

Figure 5.4 shows the average total execution time over 10 runs. We find out that the schedules from the Min-Min based three-phase algorithm have the shortest total execution times. Those two three-phase algorithms all generate schedules with shorter total execution time than the ones from the original baseline schedule. As shown in the Figure 5.4, the push-pull algorithm in phase III reduces the total execution time in phase II to a lower level than the original baseline schedule.

In the aspect of satisfying lifetime constraint, our proposed algorithms do much better than original baseline scheduling. The original baseline schedules have the average 7.3 out of 32 devices violating the constraint. As shown in Figure 5.5, the minimum lifetime of

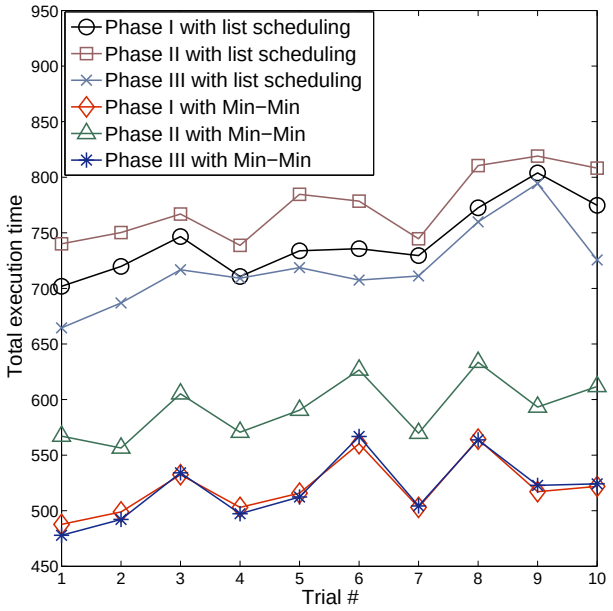


Figure 5.4: Total execution time

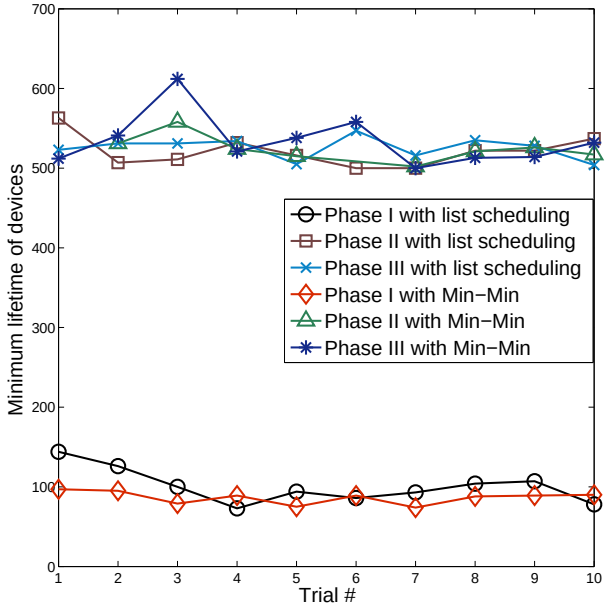


Figure 5.5: Minimum lifetime among all devices

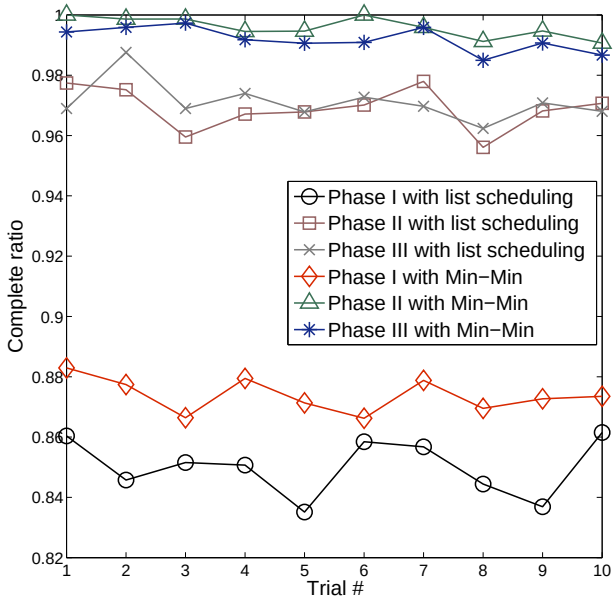


Figure 5.6: Complete ratio

the original baseline schedules are just around 100, much less than the lifetime constraint 500. The schedules generated in phase II avoid all the lifetime constraint violations. Since we set the "acceptable" condition in phase III as improving total execution time without violating the constraint, the schedules further developed in phase III satisfy the lifetime constraint in all 10 runs.

In the simulations, we set the parameters in the way that it is hard for the system to complete all the tasks given the energy setting. So in most of our simulations, the system cannot finish all the tasks. The three-phase algorithm based on Min-Min has the best performance here. It completes three of the ten runs. We define the complete ratio as the ratio of the number of complete task over the total number of task in a run. As shown in Figure 5.6, our proposed algorithms have higher complete ratios than the original baseline algorithms.

5.7 Conclusion

In this chapter, we present a complete model for task scheduling in distributed mobile DSP system, which includes application model, network model as well as energy model. Using this model, we propose our battery-aware three-phase scheduling algorithms. We show that these algorithms can generate optimal schedules while satisfying lifetime constraint, especially the one based on Min-Min algorithm. These algorithms can also improve the complete ratio of the system.

Chapter 6 Resource Allocation Robustness with Inaccurate Information

Multi-core technologies are widely used in embedded systems. Stochastic resource allocations can guarantee the certain quality of the services (QoS). In the heterogeneous embedded system resource allocation, execution time distributions of different tasks on cores are predicted before scheduling. The difference between the actual execution time and the estimated execution time may lead to allocations that are not robust. In this chapter, we present an evaluation of impacts of inaccurate information on resource allocation. We propose a systematic way of measuring the robustness degradation and evaluating how inaccurate probability parameters affect the robustness of resource allocations. Furthermore, we compare the performance of three widely used greedy heuristics when using the inaccurate information with simulations.

6.1 Introduction

Embedded multi-core technologies are represented mainly by two categories of multi-core processors [127]: 1) processors with dual, quad, and eight cores based on symmetric multiprocessing and 2) processors with the combination of heterogeneous cores. An example of the later kind of multi-core is the typical *system on chip* (SoC), which has almost unlimited combination of heterogeneous processors on the chip. As the number and the heterogeneity of cores increase, resource allocation management in the embedded multi-core system can efficiently improve the QoS.

Embedded systems usually operate in environments replete with uncertainties [24]. Meanwhile, these systems are expected to provide a given level of QoS. Stochastic resource allocation can deal with the environment uncertainties and satisfy the QoS demand. In stochastic resource allocation, uncertainties in system parameters and their impacts on system performance are modeled stochastically. This stochastic model is then used to de-

rive a quantitative evaluation of the robustness of a given resource allocation. This quantitative evaluation results in a probability that the allocation will satisfy the given constraints. A proper approach of stochastic model is using the *probability mass function* (PMF) to describe the probability distributions of execution time of tasks running on cores.

According to [128], any claim of robustness for a given system must answer three questions: (a) what behavior of the system makes it robust? (b) What uncertainties is the system robust against? (c) Quantitatively, how robust is the system? For example, some systems are robust if they are capable of finishing all the tasks within a given deadline. A resource allocation deployed in these systems must be robust against uncertainty of the task execution time. The robustness of systems can also be the makespan (total execution time) or the time slackness.

The problem of resource allocation in the field of heterogeneous multi-core systems is NP-complete (e.g., [129]). Heuristics are used to find near optimal solutions (e.g., [106, 130–135]). In static resource allocations, decisions are made based on estimated PMFs of execution time of tasks running on different cores. However, when estimated PMFs of tasks execution time are based on inaccurate information, estimated PMFs may be different from actual PMFs. Therefore, decisions generated by estimated PMFs may not be robust and the resource allocation is not able to guarantee the given level of QoS.

For example, in a surveillance sensor network, such as the Omnitrack [136], Cameras are installed across the target field, and connected to sinks. Tasks of sinks include collecting data from the cameras, compressing the images, and sending the results to the background server for further processing. After the surveillance sensor network is switched on, tasks come periodically. To better manage resources of a sink, the operating system in each sink schedules a stochastic static resource allocation before the sensors start working. The estimated PMFs can be obtained by observing previous executions of the tasks or analyzing the codes of the tasks. Using the static stochastic resource allocation, certain level of uncertainties can be tolerated, and the sensor network can maintain a given level of QoS.

However, the statistical characteristics of a task may be significantly various from period to period. For instance, when the temperature of the target field increases, processors in the sink of a sensor network may be unstable, leading to longer execution time in average. In this case, the mean of the actual PMF may increase. In another case, the frame size of the image may be reduced by the administrator in a surveillance system, which means the data size decreases and the average execution time of this task is shorter. Besides, tasks may arrive at sinks in a short period of time due to the synchronization among cameras. In this case, a lot of tasks need to wait for execution, queuing in the task buffer of the sinks. Since the order of the queue is random, the execution time of a given task may be random. The deviations of actual PMFs increase.

Some questions arise when estimated PMFs are different from actual PMFs: 1) How does the original static schedule work? Does it still maintain the required level of QoS? 2) If the performance of the original schedule degrades, how much is the degradation? 3) How much improvement can re-scheduling provide? Is re-scheduling a practical solution? The stochastic resource allocation includes a lot of convolutions, which are time consuming. Furthermore, the number of convolutions is proportional to the number of processing units, i.e., cores. The recent many-core technologies provide hundreds of cores in one processor. The re-scheduling may become a significant overhead. Our experiment shows that the Min-min algorithm takes more than an hour to schedule 1024 tasks in an eight-core system. Only when the overhead of re-scheduling is smaller than the degradation of the original schedule, the re-scheduling can be considered as a practical solution.

The major objective of this chapter is to answer above questions. In the first part of this work, a stochastic model for resource allocation is presented. The estimated task execution time information is known as a PMF. For a given task schedule, the makespan PMF of a core is generated by convoluting PMFs of all the tasks on its task list. A probability that the whole system can complete all tasks in a certain time is computed by convoluting makespan PMFs of cores. So for a given resource allocation, we find the robustness, e.g., makespan,

that system can provide with a given probability. We also propose a measurement metric for the impacts of differences between estimated PMFs and actual PMFs. In the second part of this work, we simulate the environment with inaccurate information and compare three greedy heuristics when using the inaccurate information.

In summary, two major contributions of this work include: (1) The development of a metric for measuring the impact of the inaccurate information on stochastic resource allocation. (2) The performance comparison of three greedy heuristics when using incorrect information.

In Section 6.2, we discuss related works. In Section 6.3, models for stochastic task scheduling in multi-core embedded systems are presented. We also provide the model for information inaccuracies in this section. A motivational example is provided in Section 6.4. We discuss three algorithms for stochastic task scheduling in Section 6.5, followed by experimental results in Section 6.6. Finally, we give the conclusion in Section 6.7.

6.2 Related works

A framework for robust resource allocation is provided in [128]. Authors in [128] give a robustness definition. Also, a four-step procedure is established for deriving a robustness metric. In step one, the robustness of system is described in a quantitative way, and the range of performance parameter (β_{min}, β_{max}) is given. In step two, all the system and environmental parameters that may impact the robustness of the system are modeled. In step three, the relationship between these perturbation parameters and the performance parameters is defined. Finally, the robust range of perturbation parameter is determined by substituting the perturbation parameters in the range of performance parameter (β_{min}, β_{max}).

Previous works have been reported on determining the stochastic behavior of application execution times [25, 26, 137–141]. A new approach for predicting task execution times is proposed in [142]. In [131], the authors present a derivation of the makespan

problem that relies on a stochastic representation of task execution times. In [143], the problem of robust static resource allocation for distributed computing systems under imposed QoS constraints is investigated. A stochastic robustness metric is proposed based on a stochastic model describing the uncertainty in system and its impact on system performance. Although the stochastic representation of task execution times can describe the system uncertainty, problems arise when modeling the stochastic representation. There are two conventional ways to model the stochastic representation that is usually PMFs: 1) using the statistic information from previous runs of the same task to generate the PMF directly; 2) assuming PMFs of task execution times are Gaussian distributions, and using the statistic information from previous runs to determine the expectation and the variance [143]. However when the environment is changed, these stochastic representations may not be accurate. For example, a set of PMFs are generated based on some previous runs that occur in a light-weight contention scenario. When they are applied in other heavy contention scenarios, these PMFs are not accurate in the sense that actual ones may have larger variance due to the heavy contention. So resource allocation with these inaccurate PMFs may lead to the violation of QoS requirements. The related works above does not evaluate what the relationship is between the degree of inaccurate in stochastic representation and the degradation of robustness in the system.

6.3 Model and definition

Stochastic model

In a normal heterogeneous multi-core embedded system, usually there is a set of tasks to be executed. Also, there are a number of cores with various computation power and characteristics in the system. An estimated probabilistic *estimated time to compute* (ETC) matrix P is known before scheduling. For the convenience of readers, we list acronyms used in the rest of this chapter in Table 6.1. We assume that the estimated probabilistic ETC matrix is generated using the second approach as discussed in section 6.2. The entry

Table 6.1: Acronyms used in Chapter 6

Name	Description
QoS	Quality of the service
PMF	Probability mass function
ETC	Estimated time to compute
CAT	Core available time
MCT	Minimum completion time alogrithm
M_o	Original makespan
M_n	New makespan
M_c	Correct makespan
MN_o	Normalized original makespan
MN_n	Normalized new makespan
MN_c	Normailzed correct makespan
R_n	New_ratio
R_c	Correct_ratio
R_i	Improve_ratio

$P_{i,j}$ of P represents the PMF of execution time of task i on core j. When making mapping decisions, we use the information to generate probability distributions of task completion times on different cores. For a given set of tasks and a given schedule, the *estimated makespan* distribution is the probability distribution of total execution time of the whole set of tasks based on the ETC matrix. We can calculate this probability distribution by convoluting probability distributions of task execution times. The robustness in this chapter is the minimum makespan (Λ) while maintaining a pre-determined probability θ that all cores will complete their tasks list within Λ.

As estimated PMFs of task execution times are generated with statistic information of previous runs of tasks, any environment or system changes may lead to inaccuracy. Assuming that we can get the updated information about those distribution by some methods, we are able to obtain a resource allocation that meets the QoS requirement with more confidence. We call these distributions (PMFs) updated PMFs. There are methods to obtain updated PMFs, for example, on-line profiling [144, 145]. The development of these methods is out of the scope of this chapter.

In the case that we can get updated PMFs of task execution times, whether a new resource allocation is necessary becomes another problem. Using a new resource allocation not only requires time to re-run the scheduling algorithm, but also brings the overhead of re-arranging resources in the system. However, if we can predict the degradation of robustness based on the difference between updated PMFs and estimated PMFs, i.e., the degree of inaccurate information, we can decide whether a new resource allocation is necessary. Furthermore, with knowledge of which scheduling algorithm performs the best when using inaccurate information, we can reduce the probability that a new resource allocation is necessary by using the best scheduling algorithm. We will provide some insights on these two questions in our evaluation part in the chapter.

Measurement Parameters

Since differences between estimated PMFs and updated PMFs may cause the robustness degradation, several measurement parameters are introduced to measure the robustness degradation.

- *Original Schedule*: Task Schedule generated by using estimated PMFs
- *Remapped Schedule*: Task Schedule generated by using updated PMFs
- *Makespan*: The total time taken for a system to finish all tasks with a given task schedule
- *Original Makespan (M_o)*: The makespan using estimated PMFs and the original Schedule
- *New Makespan (M_n)*: The makespan using updated PMFs and the original Schedule
- *Correct Makespan (M_c)*: The makespan using updated PMFs and the remapped Schedule

- *New_ratio (R_n)*:

$$R_n = \frac{M_n - M_o}{M_o} \tag{6.1}$$

- *Corretc_ratio (R_c)*:

$$R_c = \frac{M_c - M_o}{M_o} \tag{6.2}$$

- *Improve_ratio (R_i)*:

$$R_i = \frac{M_n - M_c}{M_c} \tag{6.3}$$

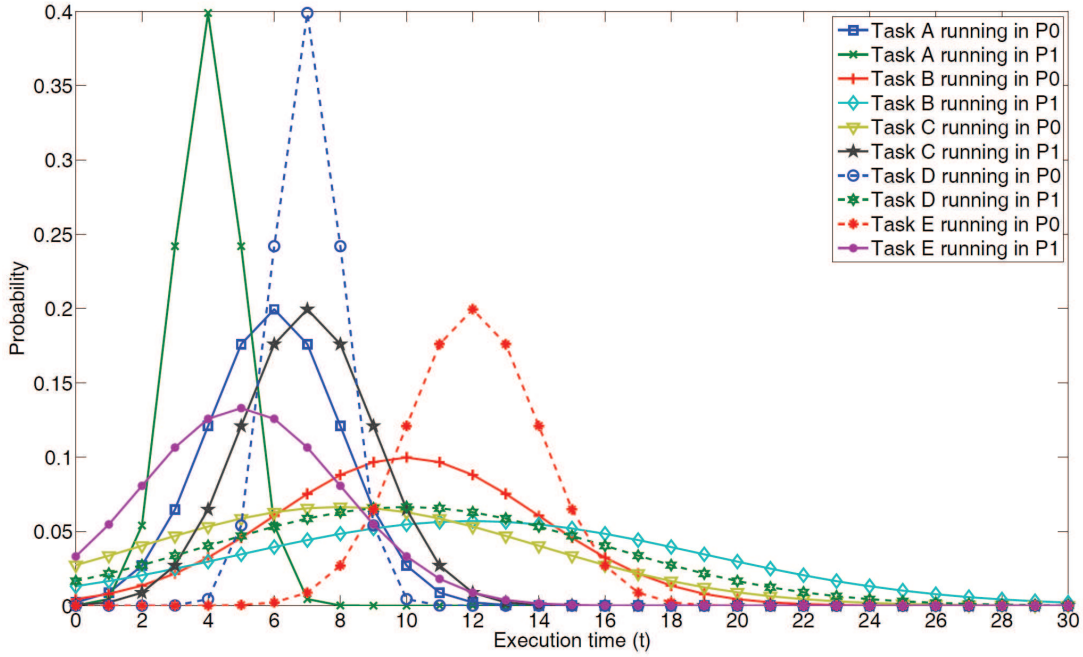
As discussed in the previous section, the robustness metric in this chapter is the minimum makespan (Λ) while maintaining a pre-determined probability θ that all cores will complete their tasks list within Λ. The smaller the makespan (Λ) is, the more robust the system is. Original makespan gives the robustness of the system assuming accurate information is used in the schedule. When inaccurate information is used in the original schedule, new makespan results in the actual robustness of the system without re-running the scheduling algorithm. Correct makespan indicates the new robustness when a new schedule is generated with updated accurate information. New_ratio shows the degradation of the robustness when using the inaccurate information. Improve_ratio reveals the improvement caused by re-running the scheduling algorithm. Correct_ratio indicates impacts of changes of environment on the system's robustness.

6.4 Motivational example

In this section, we will demonstrate how the inaccurate information impacts the robustness of a schedule. Consider a case with five independent tasks that need to be scheduled in a two cores embedded system. The estimated execution time distributions of different tasks running in these two-core are shown in Fig. 6.1(a). We assume all these distributions are normal distributions as shown in Fig. 6.1(b).

Task	P0		P1	
	Mean (μ)	Standard Deviation(σ)	Mean (μ)	Standard Deviation(σ)
A	6	2	4	1
B	10	4	12	7
C	5	3	8	6
D	7	2	7	1
E	10	1	12	2

(a)



(b)

Figure 6.1: An example of the impacts of the inaccurate information. (a) Means and standard deviations of the task execution time distributions; (b) Normal distributions of task execution time

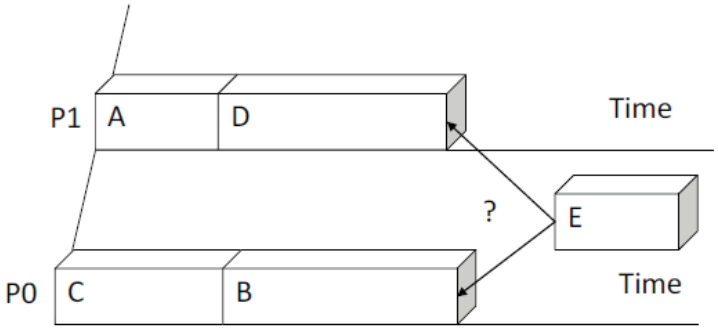


Figure 6.2: The schedule without task E

In this example, we use the Min-min heuristic, which will be introduced in the next section, to schedule these independent tasks. Task A is scheduled first in core $P1$, followed by task C in core $P0$. Then we schedule task D in core $P1$ right after task A, and task B in core $P0$, as shown in Fig. 6.2. After we schedule these four tasks in the system, we can compute the probability distributions of makespans in these two cores by convoluting task execution time distributions. Makespan distributions are shown in Fig. 6.3. For each of these two cores, we can calculate the convolution of the makespan distribution of the core and the execution time distribution of E running in the core, which is shown in Fig. 6.4. By comparing results of these convolutions, we can make a greedy decision of which core task E is scheduled to. If task E is scheduled in $P0$, all five tasks can be finished by time 34, with the probability of 90%. Otherwise, If task E is scheduled in $P1$, all tasks can be finished by time 27 with the probability of 90%. We schedule task E in $P1$.

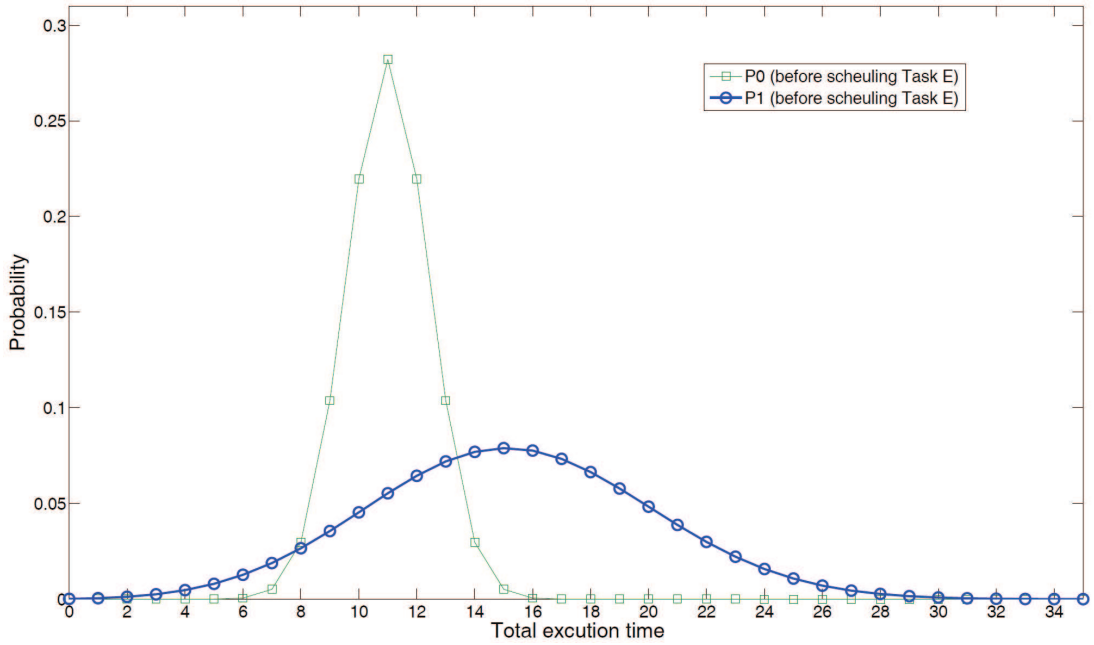


Figure 6.3: Makespan probability distributions of cores before task E is scheduled

In some cases, current statistical characteristics of the task execution time may be different from previous estimated ones. The estimated PMF cannot represent the actual distri-

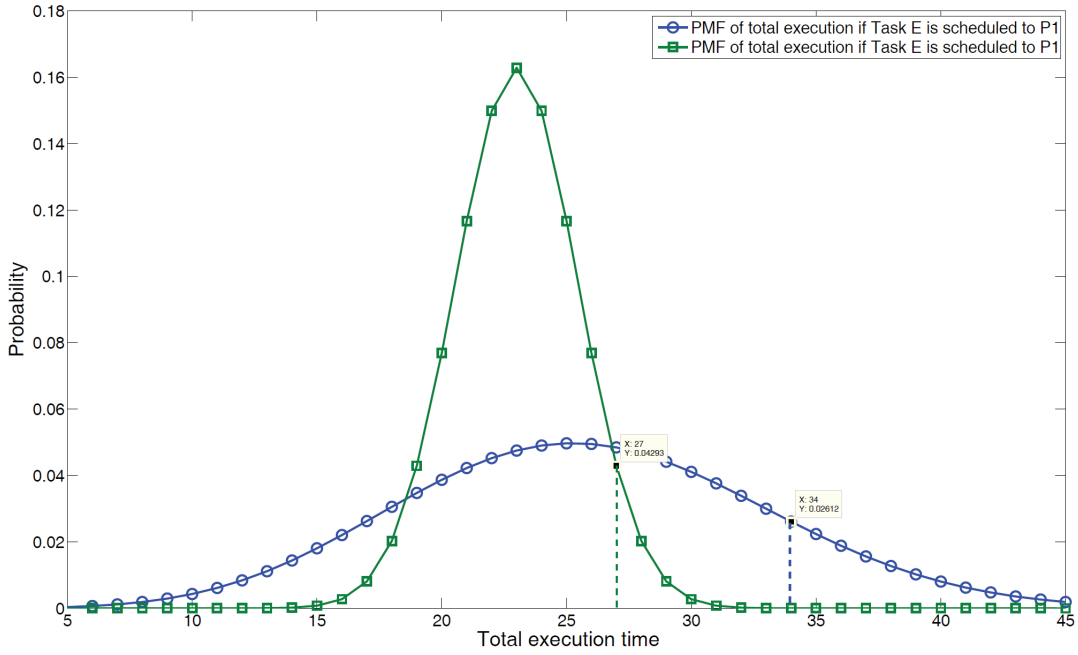


Figure 6.4: Estimated makespan probability distributions of cores after task E is scheduled

bution of the task execution time accurately. Assuming that the actual distribution of task E is different from the estimated one, the distribution of E in core $P0$ is a normal distribution with the mean of 9, and the standard deviation of 1, while the distribution in core $P1$ is another normal distribution with the mean of 14 and the standard deviation of 6. In this case, if E is scheduled in $P1$, the system will finish tasks by time 34 with 90% guarantee, about 26% robustness degradation. If E is scheduled in $P0$, all tasks will be done by time 33 with 90% guarantee, which results in a different greedy decision from the one based on estimated information as shown in Fig. 6.5.

In this example, the inaccurate information can degrade the robustness, i.e., makespan in this example. Therefore, we will investigate how different degrees of inaccurate impact the robustness and how different scheduling heuristics perform under an inaccurate information environment in following sections.

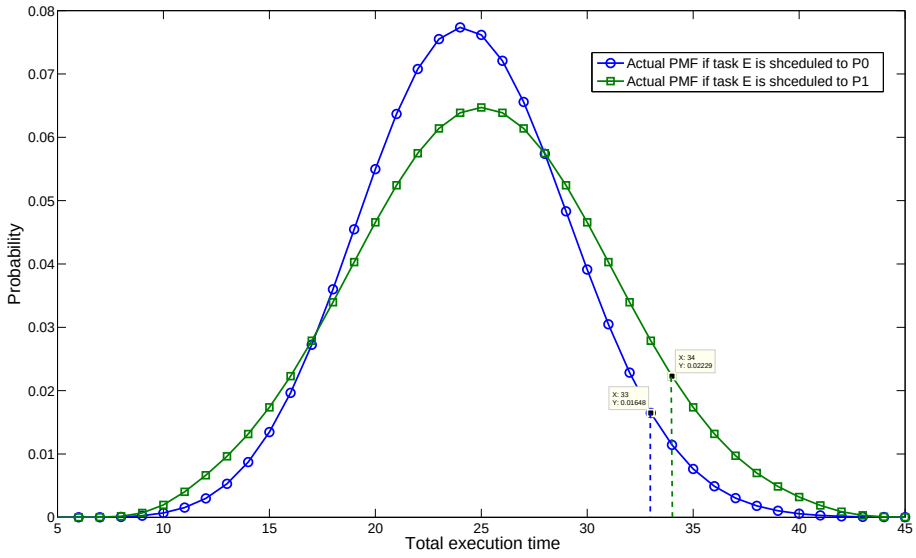


Figure 6.5: Actual makespan probability distributions of cores after task E is scheduled

6.5 Algorithms

Overview

Three static greedy heuristics are used. *Minimum completion time* (MCT) [146] is an one-phase heuristic. The output of this heuristic depends on the order in which tasks are mapped to cores. *Min-min* [146, 147] and *Max-min* [146, 147] are two-phase heuristics. These two heuristics are independent from tasks assigning order in the sense that for a given set of tasks and a system with a certain set of cores, outputs are identical no matter how many times it runs.

Greedy heuristics are widely used in heterogeneous system resource allocation. Compared to global heuristics such as *genetic algorithm* and *simulated annealing*, greedy heuristics can get a schedule much quicker than global heuristics. Previous works show that Min-min heuristics can get a schedule as optimal as the one generated by a genetic algorithm.

Definitions of these three heuristics are provided below. *Core available time* (CAT) is the probability distribution of time when the core will finish all tasks that are assigned to

this core previously. The PMF of the completion time for a new task t_i on core c_j, $ct_{i,j}$, can be calculated by convoluting the CAT of core c_j and the execution time distribution of task t_i on core c_j.

MCT

Minimum Completion Time (MCT) [146] assigns tasks in an arbitrary order to cores. For an unmapped task, MCT maps it on the core that can complete this task in the earliest time while maintaining a certain probability. The idea behind MCT is that it considers both the execution time of the task on the core as well as the load balance. Since MCT assigns tasks in an arbitrary order, the scheduling results are non-determinstic. The MCT algorithm is shown in Fig. 6.6.

Min-min

Min-min [146, 147] selects the task-core pair in two phases. In phase 1, for each unmapped task, the core that can complete it in the earliest time while maintaining a certain probability is selected to form a pair. In phase 2, among all pairs, the pair that has the minimum ct is selected, and the task in the pair is mapped to the corresponding core. The idea behind Min-min is that it does its best to keep the current load balance with the least change on it. The Min-min is provided in Fig. 6.7.

Max-min

Max-min [146, 147] is similar to Min-min. In phase 1, Max-min does exactly the same as that of Min-min. Then in phase 2, Max-min finds the task-core pairs with the maximum ct, which is different from Min-min. The idea behind is that tasks with larger execution time will likely increase the penalty if these tasks are not assigned to their best cores. Fig. 6.8 shows the Max-min algorithm.

Input: a set of tasks, m different cores, ETC PMF matrix
Output: A MCT resource allocation schedule
1: A list of unmapped tasks U is generated.
2: Reorder the list in an arbitrary order.
3: **while** the list U is not empty **do**
4: The first task i in the list U is selected; then among m cores, the core j which has the minimum $ct_{i,j}$ is also selected.
5: Assign the task to the core.
6: Remove the task from the list U.
7: Update the CAT of the selected core.
8: **end while**

Figure 6.6: MCT algorithm

Input: a set of tasks, m different cores, ETC PMF matrix
Output: A Min-min resource allocation schedule
1: A list of unmapped tasks U is generated.
2: **while** the list U is not empty **do**
3: For each task in the list U, find the core that gives the minimum ct.
4: Among task-core pairs formed in step 3, find the pair with the minimum ct.
5: Assign the task in the selected pair to the according core.
6: Remove the task from the list U.
7: Update the CAT of the selected core.
8: **end while**

Figure 6.7: Min-min algorithm

Input: a set of tasks, m different cores, ETC PMF matrix
Output: A Max-min resource allocation schedule.
1: A list of unmapped tasks U is generated.
2: **while** the list U is not empty **do**
3: For each task in the list U, find the core that gives the minimum ct.
4: Among task-core pairs formed in step 3, find the pair with the maximum ct.
5: Assign the task in the selected pair to the according core.
6: Remove the task from the list U.
7: Update the CAT of the selected core.
8: **end while**

Figure 6.8: Max-min algorithm

6.6 Simulation

Simulation Setup

To evaluate the robustness degradation caused by the inaccurate information, the following approach was used to simulate the stochastic resource allocation in a heterogeneous multi-core embedded system. A set of 1024 independent tasks was formed randomly. They consist of 28 task classes, where tasks in the same class are identical. There are 8 heterogeneous cores in a system. Each of these cores has its own computation power and characteristic. So the estimated probabilistic ETC matrix P has the size of 28×8. PMF $P_{i,j}$ is based on Gamma distribution with a mean of $m_{i,j}$ and a standard deviation of $sd_{i,j}$. In our simulation, we generate PMFs by sampling the *probability density functions* (PDF) of Gamma distributions with a start point, an end point and a fixed step. Each of the 40 simulation trials has different estimated probabilistic ETC matrix P.

Before generating PMFs of Gamma distributions, values of means and standard deviations need to be determined. We randomly generate a 28×8 mean matrix based on Gamma distribution as well as the standard deviation matrix. Here, we use the COV based method [148] with the mean of task execution time from 40 to 80, and both coefficients of variation of tasks and cores uniformly from 0.35 to 1, as shown in Fig. 6.9. When forming the PMF $P_{i,j}$, we can sample the PDF of Gamma distribution with a mean of $m_{i,j}$ and a standard deviation of $sd_{i,j}$. The objective of this method to generate PMFs for simulation. And this method can be implemented easily by a statistical computing tool R [149]. In literature, there are several low-overhead methods [139, 150, 151] to generate stochastic profiles with sufficient coverage of variances in practical applications.

To simulate the case in which updated PMFs are different from estimated PMFs, parameters (mean or standard deviation) of updated PMFs are generated by multiplying parameters of estimated PMFs with a scalar matrix S.

Input: t different tasks, m different cores, coefficient of variation of task and core V_{task}, V_{core}, mean of tasks' ETC μ_{task}
Output: A random ETC matrix based on Gamma distribution
1: Compute the shape parameter and the scale parameter of task as well as the shape parameter of core
$\alpha_{task} = 1/{V_{task}}^2 \alpha_{core} = 1/{V_{core}}^2$
$\beta_{task} = \mu_{task}/\alpha_{task}$
2: **for** i from 0 to $(t-1)$ **do**
3: $q[i] = G(\alpha_{task}, \beta_{task})$
/*$q[i]$ will be used as mean of i-th row in the ETC matrix*/
4: $\beta_{core}[i] = q[i]/\alpha_{core}$
/*scale parameter for i-th row*/
5: **for** j from 0 to $(m-1)$ **do**
6: $e[i,j] = G(\alpha_{core}, \beta_{core}[i])$
7: **end for**
8: **end for**

Figure 6.9: COV based method for generate Gamma random matrix

For example, if mean values are modified,

$$updated\_mean(i,j) = mean(i,j) \times S_{i,j} \tag{6.4}$$

The entry of scalar matrix S is based on a uniform distribution with a range of [S_{min}, S_{max}].

Simulation Results

Compare impacts on robustness when modifying different parameters

In this part, we compare impacts on robustness when using different scalar matrixes as well as modifying different parameters.

We simulate two different scenarios in which two different kinds of inaccurate information occur:

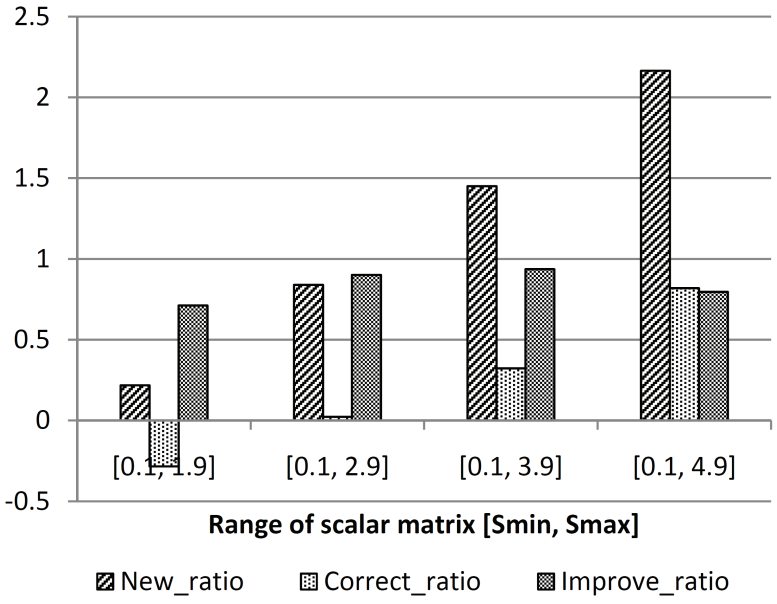
1. Keep standard deviations unchanged, and multiply means with a scalar matrix.

2. Keep means unchanged, and multiply standard deviations with a scalar.

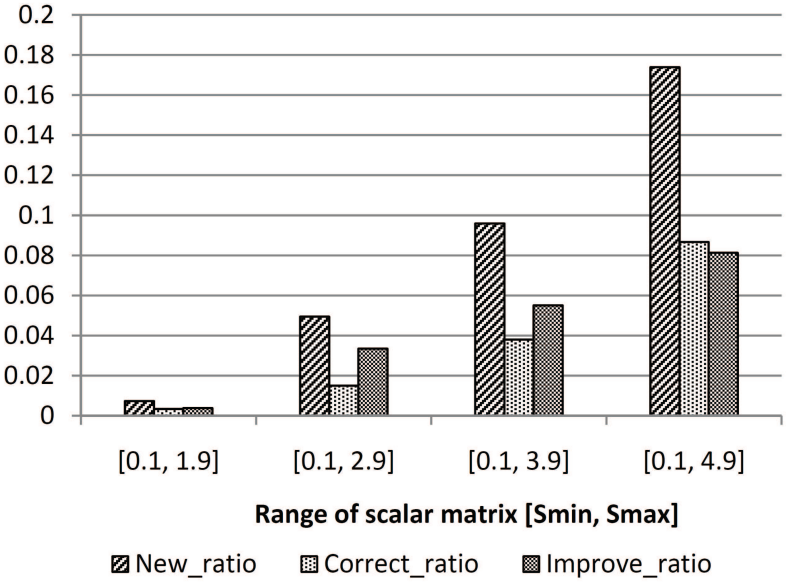
The first scenario usually happens when the embedded system is employed in a physically inconstant environment. For example, in an environment where temperature changes rapidly, cores will likely run faster in low temperature than that in high temperature. As the temperature increases, means of the probability distribution of execution times may increase. In this case, the statistic information collected previously in low temperature may not be accurate. The second scenario happens when resource contention among tasks changes. When the resource contention is light, a core likely finishes same tasks in a narrow distribution, especially around the mean of the distribution. When the contention is heavy, the distribution of a task class in a core may be wide, i.e., with larger standard deviations. In our simulation, the scalar matrixes are within the range of [0.1, 1.9], [0.1, 2.9], [0.1, 3.9], [0.1, 4.9].

MCT heuristic is used in all these four parameter modifications. The result of each trial is the average value of MCT with 25 different task mapping order.

In Fig. 6.10(a), the increase of new_ratio is proportional to the increase of the scalar matrix range with 20% to 70% penalty. Obviously, the increase of mean values of the execution time distribution leads to a longer makspan. This 20% to 70% penalty is caused by the inaccurate information used in the original schedule. We find that the improve_ratio, which indicates the improvement of re-scheduling, does not change as much as the increase of the scalar matrix range. Note that when we calculate the improve_ratio, we compare the difference between the new_makespan and the correct_makespan. In the convolution of these two distributions, we use the updated PMFs. The "improve_ratio" columns show that the level of improvement brought from the re-scheduling does not mainly depend on the inaccurate degree of the information, but depends on what the task set consists of. The correct_ratio is also proportional to the increase of the scalar matrix range. It shows that the degradation of robustness is a linear function of the degree of how the environment changes. Comparing Fig. 6.10(b) with Fig. 6.10(a), we find that the inaccurate standard deviations have much less impacts on the robustness than that of the inaccurate means.



(a)



(b)

Figure 6.10: Three ratios with different inaccurate information. (a) New_ratio, correct_ratio and improve_ratio when changing the mean; (b) new_ratio, correct_ratio and improve_ratio when changing the standard deviation

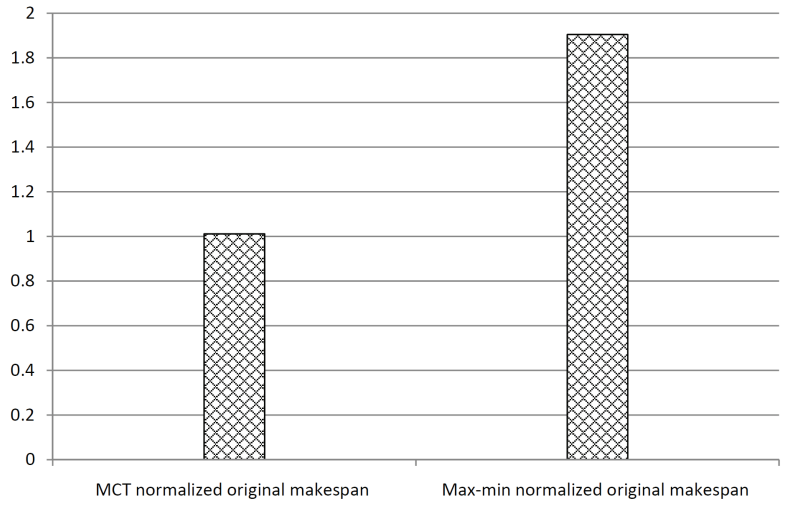


Figure 6.11: The Original makespan when changing the mean and the standard deviation with a fixed scale parameter

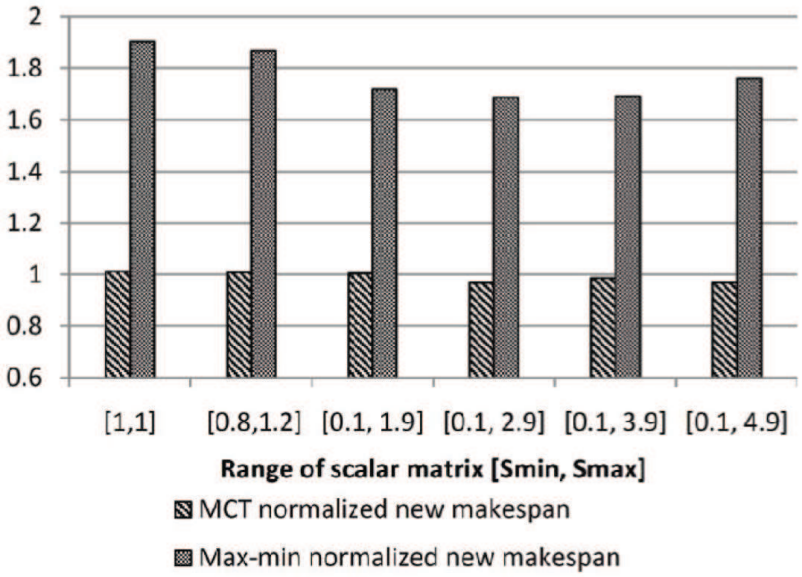


Figure 6.12: The normalized new makespan when changing the mean and the standard deviation with a fixed scale parameter

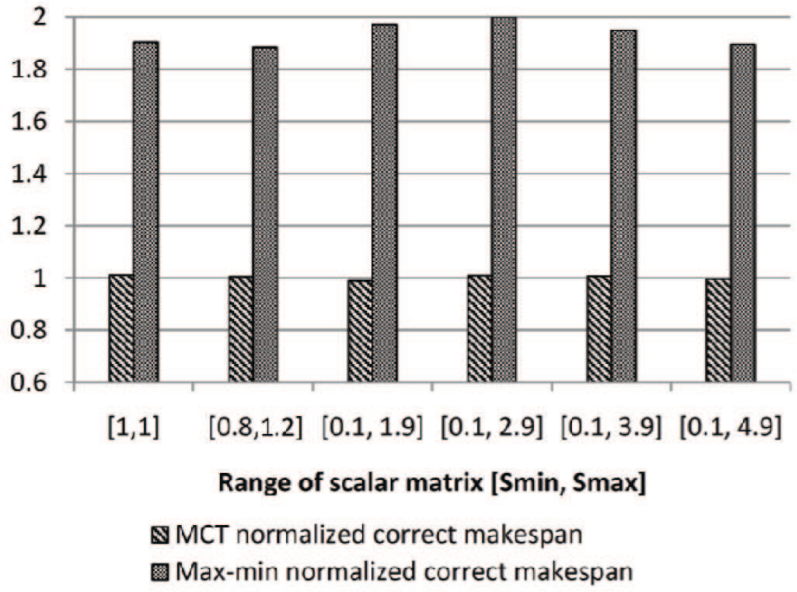


Figure 6.13: The normalized correct makespan when changing the mean and the standard deviation with a fixed scale parameter

Compare the performance of different heuristics

In this part, three different heuristics (Min-min, MCT, Max-min) are compared with their performance when using inaccurate information. In this part, we will keep the standard deviations fixed and change mean values. To compare the performance of these heuristics, normalized makespans of MCT and Max-min are introduced.

- Max-min normalized original makespan

$$MN_o(Max-min) = \frac{M_o(Max-min)}{M_o(Min-min)} \tag{6.5}$$

- Max-min normalized new makespan

$$MN_n(Max-min) = \frac{M_n(Max-min)}{M_n(Min-min)} \tag{6.6}$$

- Max-min normalized correct makespan

$$MN_c(Max-min) = \frac{M_c(Max-min)}{M_c(Min-min)} \tag{6.7}$$

- MCT normalized original makespan

$$MN_o(MCT) = \frac{M_o(MCT)}{M_o(Min - min)} \tag{6.8}$$

- MCT normalized new makespan

$$MN_n(MCT) = \frac{M_n(MCT)}{M_n(Min - min)} \tag{6.9}$$

- MCT normalized correct makespan

$$MN_c(MCT) = \frac{M_c(MCT)}{M_c(Min - min)} \tag{6.10}$$

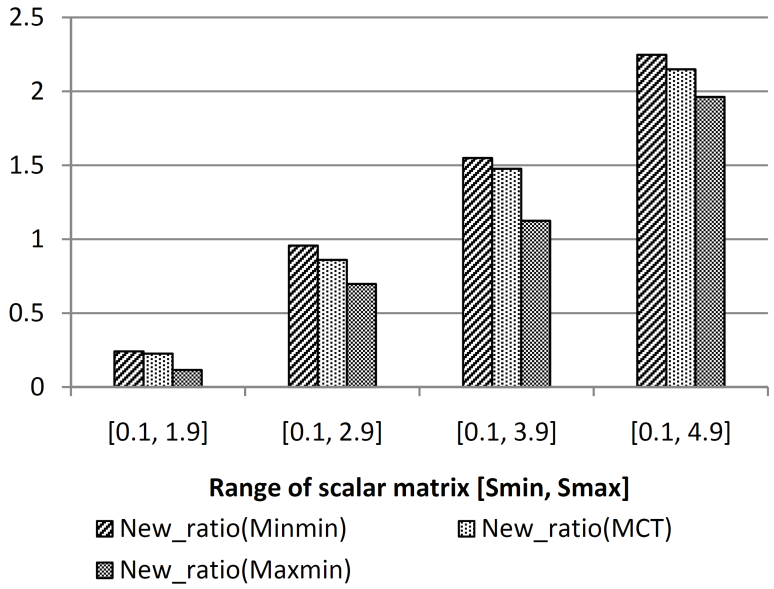


Figure 6.14: The new_ratio of three heuristics when changing the mean and the standard deviation with a fixed scale parameter

In the respect of the three ratios (Nnew_ratio, correct_ratio, and improve_ratio), Fig. 6.14, 6.15, and 6.16 show that the Max-min is least impacted by the inaccurate information. However, in Fig. 6.11, 6.12, and 6.13, Max-min has the longest new makespans and the longest correct makespans among these three heuristics. It means that the Max-min generates the least robust schedules in the environment with or without inaccurate information, even though the inaccurate information has smallest impacts in the Max-min. So

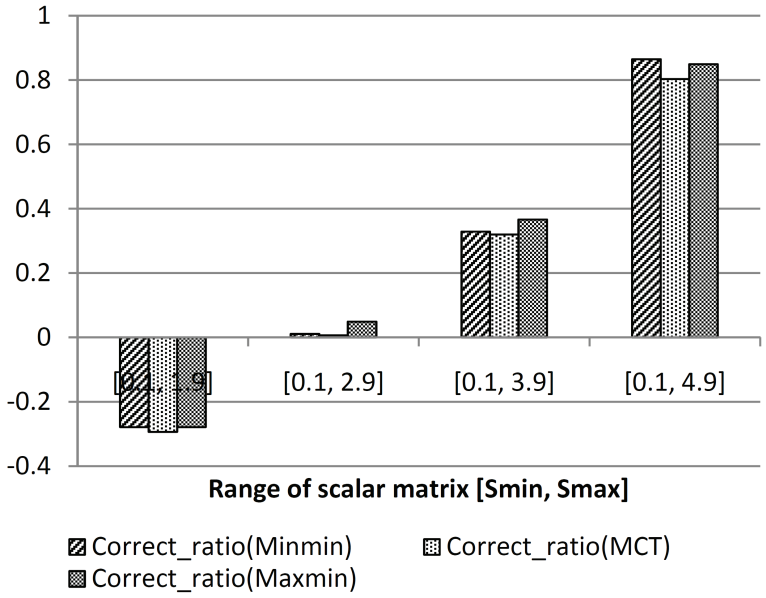


Figure 6.15: The correct_ratio of three heuristics when changing the mean and the standard deviation with a fixed scale parameter

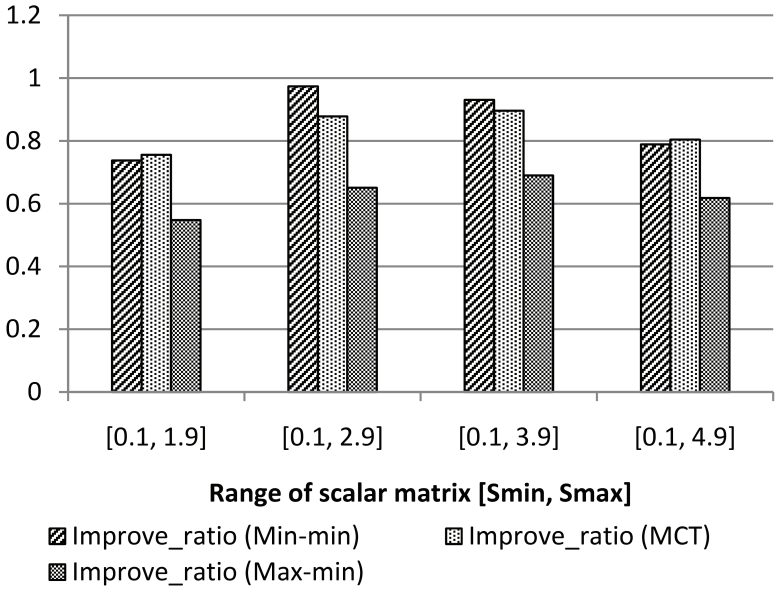


Figure 6.16: The improve_ratio of three heuristics when changing the mean and the standard deviation with a fixed scale parameter

the Max-min performance is the worst among these three heuristics. The performance of MCT is very close to the performance of Min-min with respect to the original makespan. Furthermore, MCT outperforms the Min-min in the new makespan. It means that MCT is less impacted by the inaccurate information and performs close to the Min-min in the original makespan, and it performs the best in the new makespan even though the difference between these two heuristics is not significant.

6.7 Conclusion

We propose a systematic method of measuring the robustness degradation with a stochastic approach. We evaluate impacts of inaccurate information on system robustness in two different scenarios. In our simulation, the makespan is the robustness metric. We find that the makespan with inaccurate information increases proportional to the increase of mean values of task execution time distribution caused by environment changes. Also, 20% to 70% penalty is caused by the inaccurate information used in making scheduling decisions. The impact of environment changes on the robustness is linear to the degree of how much inaccurate information (mainly the shift of means of PMFs) is generated by these environment changes. However, the improvement of re-scheduling with updated information mainly depends on how the task set consists of, not how inaccurate the information is. We also find that the impact of inaccurate means of PMFs is much larger than inaccurate standard deviations.

Among these three greedy algorithms, MCT performs the best under inaccurate information. It generates schedules that are almost as optimal as ones from Min-min where accurate information is used. And inaccurate information has less impacts on schedules from MCT than it does on Min-min. Max-min performs the worst.

Chapter 7 Online Optimization on Cloud systems

In *Infrastructure-as-a-Service* (IaaS) cloud computing, computational resources are provided to remote users in the form of leases. For a cloud user, he/she can request multiple cloud services simultaneously. In this case, parallel processing in the cloud system can improve the performance. When applying parallel processing in cloud computing, it is necessary to implement a mechanism to allocate resource and schedule the execution order of tasks. Furthermore, a resource optimization mechanism with preemptable task execution can increase the utilization of clouds. In this chapter, we propose two online dynamic resource allocation algorithm for the IaaS cloud system with preemptable tasks. Our algorithms adjust the resource allocation dynamically based on the updated of the actual task executions. And the experimental results show that our algorithms can significantly improve the performance in the situation where resource contention is fierce.

7.1 Introduction

In cloud computing, a cloud is a cluster of distributed computers providing on-demand computational resources or services to the remote users over a network [152]. In an Infrastructure-as-a-Service (IaaS) cloud, resources or services are provided to users in the form of leases. The users can control the resources safely thanks to the free and efficient virtualization solutions, e.g., the Xen hypervisor [153]. One of the advantages of the IaaS clouds is that the computational capacities providing to end-users are flexible and efficient. The *virtual machines* (VMs) in Amazon's Elastic Compute Cloud are leased to users at the price of ten cents per hour. Each VM offers an approximate computational power of a 1.2 GHz Opteron processor, with 1.7 GB memory and 160 GB disk space. For example, when a user needs to maintain a database with a certain disk space for a month, he/she can rent a number of VMs from the cloud, and return them after that month. In this case, the user

can minimize the costs. And the user can add or remove resources from the cloud to meet peak or fluctuating service demands and pay only the capacity used.

Cloud computing is emerging with growing popularity and adoption [154]. However, there is no data center that has unlimited capacity. Thus, in case of significant client demands, it may be necessary to overflow some workloads to another data center [155]. These workload sharing can even occur between private and public clouds, or among private clouds or public clouds. The workload sharing is able to enlarge the resource pool and provide even more flexible and cheaper resources. To collaborate the execution across multiple clouds, the monitoring and management mechanism is a key component and requires the consideration of provisioning, scheduling, monitoring, and failure management [155]. Traditional monitoring and management mechanisms are designed for enterprise environments, especially a unified environment. However, the large scale, heterogeneous resource provisioning places serious challenges for the management and monitoring mechanism in multiple data centers. For example, the Open Cirrus, a cloud computing testbed, consists of 14 geographically distributed data center in different administrative domains around the world. Each data center manages at least 1000 cores independently [156]. The overall testbed is a heterogeneous federated cloud system. It is important for the monitoring and management mechanism to provide the resource pool, which includes multiple data centers, to clients without forcing them to handle issues, such as the heterogeneity of resources and the distribution of the workload. Virtualization in cloud computing, such as VMs, has been intensively studied recently. However, scheduling workloads across multiple heterogeneous clouds/data centers has not been well studied in the literature. To the best of our knowledge, this is the first chapter to address the scheduling issue in the federated heterogeneous multi-cloud system.

A large numbers of applications running on cloud systems are those compute on large data corpora [157]. These “big data” applications draw from information source such as digital media collections, virtual worlds, simulation traces, data obtain from scientific in-

struments, and enterprise business databases. These data hungry applications require scalable computational resources. Fortunately, these applications exhibit extremely good parallelism [157]. Using a "map/reduce" approach in the cloud application development, large batch processes can be partitioned into a set of discrete-linked processes, which we call tasks. These tasks can be executed in parallel to improve response time [158]. In Fedex's data center, a four-hour batch process can be successfully runs in 20 minutes after the "map/reduce" [158]. When applying parallel processing in executing these tasks, we need to consider the following questions: 1) how to allocate resources to tasks; 2) in what order the clouds should execute tasks, since tasks have data dependencies; and 3) how to schedule overheads when VMs prepare, terminate or switch tasks. Resource allocation and scheduling can solve these three problems. Resource allocation and task scheduling have been studied in high performance computing [107,108] and in embedded systems [24,159]. However, the autonomic feature and the resource heterogeneity within clouds [152] and the VM implementation require different algorithms for resource allocation and task scheduling in the IaaS cloud computing, especially in the federated heterogeneous multi-cloud system.

The two major contributions of this chapter are:

- We present a resource optimization mechanism in heterogeneous IaaS federated multi-cloud systems, which enables preemptable task scheduling. This mechanism is suitable for the autonomic feature within clouds and the diversity feature of VMs.
- We propose two online dynamic algorithms for resource allocation and task scheduling. We consider the resource contention in the task scheduling.

In section 7.2, we discuss works related to this topic. In section 7.3, models for resource allocation and task scheduling in IaaS cloud computing system are presented, followed by an motivation example in section 7.4. We propose our algorithms in section 7.5, followed by experimental result in section 7.6. Finally, we give the conclusion in section 7.7.

7.2 Related works

Cloud system has been drawing intensive research interests in the recent years. A number of public clouds are available for customer and researchers, such as Amazon AWS [160], GoGrid [161], and Rackspace [162]. Some other companies also provide cloud services, such as Microsoft [163], IBM [164], Google [165], and HP [166]. To benefit the cloud research, open source cloud services are under way, such as Eucalyptus [167], Open Nebula [168], Tashi [157], RESEVOIR [169], and Open Cirrus [156]. Open Cirrus is a cloud testbed consists of 14 distributed data centers among the world. Essentially, it is a federated heterogeneous cloud system, which is similar to the target cloud system in this chapter.

Data intensive applications are the major type of applications running in the cloud computing platform. Most of the data intensive applications can be modeled by MapReduce programming model [170]. In MapReduce model, user specify the map function that can be executed independently, and the reduce function that gather results from the map function and generate the final result. The runtime system automatically parallelizes the map function and distributes them in the cloud system. Apache Hadoop is a popular framework, inspired by MapReduce, for running the data-intensive application in IaaS cloud systems [171]. Both reliability and data motion are transparently provided in Hadoop framework. MapReduce programming model and Hadoop distributed file system are implemented in the open-source Hadoop framework. All-pairs, an high level abstraction, was proposed to allow the easy expression and efficient execution of data intensive applications [172]. Liu et al. designed a programming model, GridBatch, for large scale data intensive batch applications [173]. In GridBatch, user can specify the data partitioning and the computation task distribution, while the complexity of parallel programming is hidden. A dynamic split model was designed to enhance the resource utilization in MapReduce platforms [174]. A priority-based resource allocation approach as well as a resource usage pipeline are implemented in this dynamic split model. Various scheduling methods for data-intensive services were evaluated [175], with both soft and hard *service level agree-*

ments (SLA). However, the problem of scheduling workloads in heterogeneous multi-cloud platform was not considered in the related work mentioned above.

Virtualization is an important part in cloud computing. Emeneker et al. propose an image caching mechanism to reduce the overhead of loading disk image in virtual machines [176]. Fallenbeck et al. present a dynamic approach to create virtual clusters to deal with the conflict between parallel and serial jobs [177]. In this approach, the job load is adjusted automatically without running time prediction. A suspend/resume mechanism is used to improve utilization of physical resource [178]. The overhead of suspending/resume is modeled and scheduled explicitly. But the VM model considered in [178] is homogeneous, so the scheduling algorithm is not applicable in heterogeneous VMs models.

Computational resource management in cloud computing has been studied in the literature recently. To make resource easy for users to manage collectively, CloudNet [179] provides virtual private clouds from enterprise machines and allocates them via public clouds. Computation-intensive users can reserve resources with on-demand characteristics to create their virtual private clouds [180–185]. However, CloudNet focuses on providing secure links to cloud for enterprise users, resource allocation is not an objective in CloudNet. Lease-based architecture [185, 186] is widely used in reserving resource for cloud users. In [185], applications can reserve group of resources using leases and tickets from multiple sites. Haizea [186] supports both the best-effort and the advanced reservation leases. The priorities of these two kinds of leases are different. The utilization of the whole system is improved. The model of job in these two paper is a batch job model, which mean every application is scheduled as independent. Data dependencies are not considered. Thus this method cannot be “map/reduce” and parallelized among multiple data centers. In our proposed resource allocation mechanism, we model the data dependencies among an application, and distribute the application among multiple data centers at the task level, leading to more flexible and more efficient resource allocation schedules.

Wilde et al. proposed Swift, a scripting language for distributed computing [187]. Swift

focuses on the concurrent execution, composition, and coordination of large scale independent computational tasks. A workload balancing mechanism with adaptive scheduling algorithms is implemented in Swift, based on the availability of resources. A dynamic scoring system is designed to provide an empirically measured estimate of a site's ability to bear load, which is similar to the feedback information mechanism proposed in our design. However, the score in the Swift is decreased only when the site fails to execute the job. Our approach has a different use of the feedback information. The dynamic estimated finish time of remote site is based on the previous executions on this site in our approach. Therefore, even a "delayed but successful" finish of a job leads to a longer estimated finish time in the next run in our approach. ReSS is used in the Swift as the resource selection service [188]. Ress requires a central information repository to gather information from different nodes or clusters. However, our approach is a decentralized approach that does not need any central information repository.

A system that can automatically scale its share of infrastructure resources is designed in [189]. The adaptation manager monitors and autonomically allocating resources to users in a dynamic way, which is similar to the manager server in our proposed mechanism. However, this centralized approach cannot fit in the future multi-provider cloud environment, since different providers may not want to be controlled by such a centralized manager. Another resource sharing system that can trade machines in different domains without infringing autonomy of them is developed in [190]. A machine broker of a data center is proposed to trade machines with other data centers, which is a distributed approach to share resource among multiple data centers. However, the optimization of resource allocation is not considered in this paper. Our proposed resource allocation mechanism is a distributed approach. A manager server of a cloud communicates with others, and shares workloads with our dynamic scheduling algorithm. Our approach can improve federated heterogeneous cloud systems. Moreover, it can be adapted in the future multi-provider cloud system.

7.3 Model and Background

Cloud system

In this chapter, we consider an infrastructure-as-a-service (IaaS) cloud system. In this kind of system, a number of data centers participate in a federated approach. These data centers deliver basic on-demand storage and compute capacities over Internet. The provision of these computational resources is in the form of virtual machines (VMs) deployed in the data center. These resources within a data center form a cloud. Virtual machine is an abstract unit of storage and compute capacities provided in a cloud. Without loss of generality, we assume that VMs from different clouds are offered in different types, each of which has different characteristics. For example, they may have different numbers of CPUs, amounts of memory and network bandwidths. As well, the computational characteristics of different CPU may not be the same.

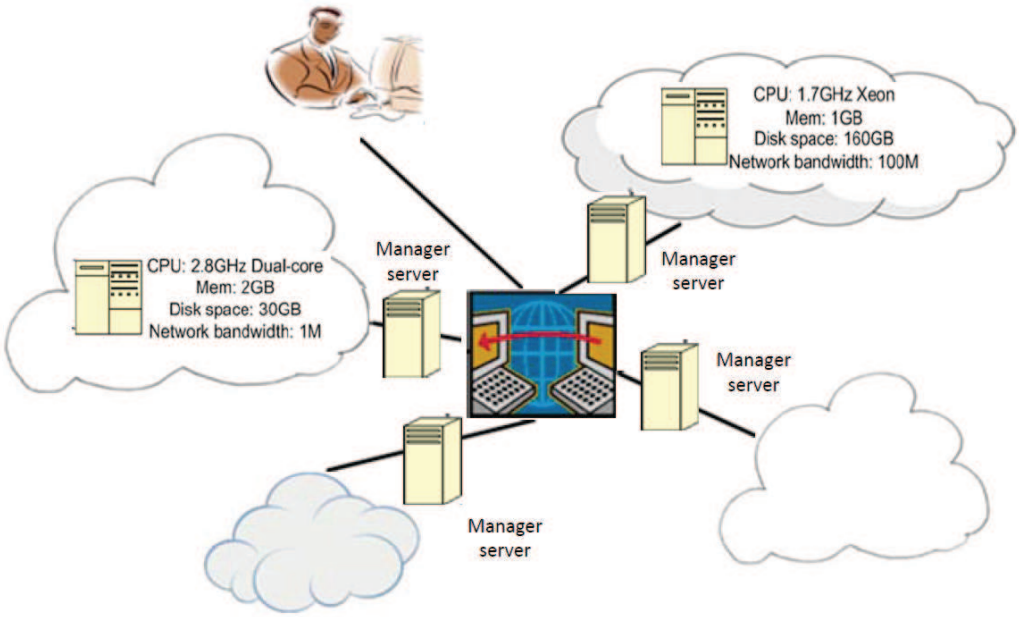


Figure 7.1: An example of our proposed cloud resource allocation mechanism. Heterogeneous VMs are provided by multiple clouds. And clouds are connected to the Internet via manager servers.

For a federated cloud system, a centralized management approach, in which a super

node schedule tasks among multiple clouds, may be a easy way to address the scheduling issues in such system. However, as authors in [155, 156] have indicated, the future cloud computing will consist of multiple cloud providers. In this case, the centralized management approach may be accepted by different cloud providers. Thus we propose a distributed resource allocation mechanism that can be used in both federated cloud system or the future cloud system with multiple providers.

As shown in Fig. 7.1, in our proposed cloud resource allocation mechanism, every data center has a manager server server that knows the current statuses of VMs in it own cloud. And manager servers communicate with each other. Clients submit their tasks to the cloud where the dataset is stored. Once a cloud receives tasks, its manager server can communicate with manager servers of other clouds, and distribute its tasks across the whole cloud system by assigning them to other clouds or executing them by itself.

When distributing tasks in the cloud system, manager servers should be aware of the resource availabilities in other clouds, since there is not a centralized super node in the system. Therefore, we need the resource monitoring infrastructure in our resource allocation mechanism. In cloud systems, resource monitoring infrastructure involves both producers and consumers. Producers generate status of monitored resources. And consumers make use of the status information [191]. Two basic messaging methods are used in the resource monitoring between consumers and producers: the pull mode and the push model [192]. Consumers pull information from producers to inquire the status in the pull mode. In the push mode, when producers update any resource status, they push the information to the consumers. The advantage of the push mode is that the accuracy is higher when the threshold of a status update, i.e., trigger condition, is defined properly. And the advantage of the pull mode is that the transmission cost is less when the inquire interval is proper [191].

In our proposed cloud system resource allocation mechanism, we combine both communication modes in the resource monitoring infrastructure. In our proposed mechanism, when the manager server of cloud A assigns an application to another cloud B, the man-

ager server of A is the consumer. And the manager server of B is the producer. manager server of A needs to know the resource status from the manager server of B in two scenarios: 1) when the manager server of A is considering assigning tasks to cloud B, the current resource status of cloud B should be taken into consideration. 2) When there is an task is assigned to cloud B by manager server of A, and this task is finished, manager server of A should be informed.

We combine the pull and the push mode as the following:

- A consumer will pull information about the resource status from other clouds, when it is making scheduling decisions.
- After an application is assigned to another cloud, the consumer will no longer pull information regarding to this application.
- When the application is finished by the producer, the producer will push its information to the consumer. The producer will not push any information to the consumer before the application is finished.

In a pull operation, the trigger manager server sends a task check inquire to manager servers of other clouds. Since different cloud providers may not be willing to share detailed information about their resource availability, we propose that the reply of a task check inquire should be as simple as possible. Therefore, in our proposed resource monitoring infrastructure, these target manager servers only responses with the earliest available time of required resources, based on its current status of resources. And no guarantee or reservation is made. Before target manager servers check their resource availability, they first check the required dataset locality. If the required dataset is not available in their data center, the estimated transferring time of the dataset from the trigger cloud will be included in the estimation of the earliest available time of required resources. Assuming the speed of transferring data between two data centers is S_c, and the size of the required dataset is M_S, then the preparation overhead is M_S/S_c. Therefore, when a target cloud already has

the required in its data center, it is more likely that it can be respond with a sooner earliest available time of required resources, which may lead to an assignment to this target cloud. In a push operation, when B is the producer and A is consumer, the manager server of B will inform the manager server of A the time when the application is finished.

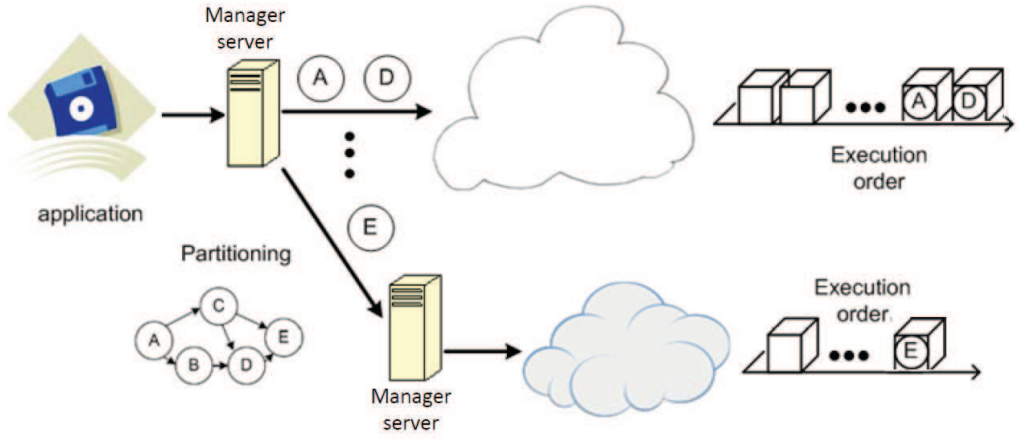


Figure 7.2: An application submitted in the cloud system. When an application is submitted to the cloud system, it is partitioned, assigned, scheduled, and executed in the cloud system

When a client submits his/her workload, typically an application, to a cloud, the manager server first partitions the application into several tasks, as shown in Fig. 7.2. Then for each task, the manager server decides which cloud will execute this task based on the information from all other manager servers and the data dependencies among tasks. If the manager server assigns a task to its own cloud, it will store the task in a queue. And when the resources and the data are ready, this task is executed. If the manager server of cloud A assigns a task to cloud B, the manager server of B first checks whether its resource availabilities can meet the requirement of this task. If so, the task will enter a queue waiting for execution. Otherwise, the manager server of B will reject the task.

Before a task in the queue of a manager server is about to be executed, the manager server transfers a disk image to all the computing nodes that provide enough VMs for task execution. We assume that all required disk images are stored in the data center and can be transferred to any clouds as needed. We use the multicasting to transfer the image to

all computing nodes within the data center. Assuming the size of this disk image is S_I, we model the transfer time as S_I/b, where b is the network bandwidth. When a VM finishes its part of the task, the disk image is discarded from computing nodes.

Resource allocation model

In cloud computing, there are two different modes of renting the computing capacities from a cloud provider.

- Advance Reservation (AR): Resources are reserved in advance. They should be available at a specific time;
- Best-effort: Resources are provisioned as soon as possible. Requests are placed in a queue.

A lease of resource is implemented as a set of VMs. And the allocated resources of a lease can be described by a tuple (n, m, d, b), where n is number of CPUs, m is memory in megabytes, d is disk space in megabytes, and b is the network bandwidth in megabytes per second. For the AR mode, the lease also includes the required start time and the required execution time. For the best-effort and the immediate modes, the lease has information about how long the execution lasts, but not the start time of execution. The best-effort mode is supported by most of the current cloud computing platform. The Haizea, which is a resource lease manager for OpenNebula, supports the AR mode [153]. The "map" function of "map/reduce" data-intensive applications are usually independent. Therefore, it naturally fits in the best-effort mode. However, some large scale "reduce" processes of data-intensive applications may needs multiple reducers. For example, a simple "word-count" application with tens of PBs of data may need a parallel "reduce" process, in which multiple reducers combine the results of multiple mappers in parallel. Assuming there are N reducers, in the first round of parallel "reduce", each of N reducers counts $1/N$ results from the mappers. Then $N/2$ reducers receive results from the other $N/2$ reducers, and

counts $2/N$ results from the last round of reducing. It repeats $log_2 N + 1$ rounds. Between two rounds, reducers need to communicate with others. Therefore, a AR mode is more suitable for these data-intensive applications.

When supporting the AR tasks, it may leads to a utilization problem, where the average task waiting time is long, and machine utilization rate is low. Combining AR and best-effort in a preemptable fashion can overcome this problems [186]. In this chapter, we assume that a few of applications submitted in the cloud system are in the AR mode, while the rest of the applications are in the best-effort mode. And the applications in AR mode have higher priorities, and are able to preempt the executions of the best-effort applications.

When an AR task A needs to preempt a best-effort task B, the VMs have to suspend task B and restore the current disk image of task B in a specific disk space before the manager server transfers the disk image of tasks A to the VMs. When the task A finishes, the VMs will resume the execution of task B. We assume that there is a specific disk space in every node for storing the disk image of suspended task.

There are two kinds of AR tasks: one requires a start time in future, which is referred to as "non-zero advance notice" AR task; and the other on requires to be executed as soon as possible with higher priority than the best-effort task, which is referred to as "zero advance notice" AR task. For a "zero advance notice" AR task, it will start right after the manager server makes the scheduling decision and assign it a cloud. Since our scheduling algorithms, mentioned in Section 7.5, are heuristic approaches, this waiting time is negligible, compared to the execution time of task running in the cloud system.

Local mapping and energy consumption

From the user's point of view, the resources in the cloud system are leased to them in the term of VMs. Meanwhile, from the cloud administrator's point of view, the resources in the cloud system is utilized in the term of servers. A server can provide the resources of multiple VMs, and can be utilized by several tasks at the same time. One important

function of the manager server of each cloud is to schedule its tasks to its server, according the numbers of required VMs. Assuming there are a set of tasks T to schedule on a server S, we define the remaining workload capacity of a server S is $C(S)$, and the number of required VM by task t_i is $wl(t_i)$. The server can execute all the tasks in T only if:

$$C(S) \geq \sum_{t_i \in T} (wl(t_i)) \tag{7.1}$$

We assume servers in the cloud system work in two different modes: the active mode and the idle mode. When the server is not executing any task, it is switched to the idle mode. When tasks arrive, the server is switched back to the active mode. The server consumes much less energy in the idle mode than that in the active mode.

Application model

In this chapter, we use the *Directed Acyclic Graphs* (DAG) to represent applications. A DAG $T = (V, E)$ consists of a set of vertices V, each of which represents a task in the application, and a set of edges E, showing the dependencies among tasks. The edge set E contains edges e_{ij} for each task $v_i \in V$ that task $v_j \in V$ depends on. The weight of a task represents the type of this task. Given an edge e_{ij}, v_i is the immediate predecessor of v_j, and v_j is called the immediate successor of v_i. A task only starts after all its immediate predecessors finish. Tasks with no immediate predecessor are entry-node, and tasks without immediate successors are exit-node.

Although the compute nodes from the same cloud may equip with different hardware, the manager server can treat its cloud as a homogeneous system by using the abstract compute capacity unit and the virtual machine. However, as we assumed, the VMs from different clouds may have different characteristics. So the whole cloud system is a heterogeneous system. In order to describe the difference between VMs' computational characteristics, we use an $M \times N$ execution time matrix (ETM) E to indicate the execution time of M types of tasks running on N types of VMs. For example, the entry e_{ij} in E indicate the

required execution time of task type i when running on VM type j. We also assume that a task requires the same lease (n, m, d, b) no matter on which type of VM the task is about to run.

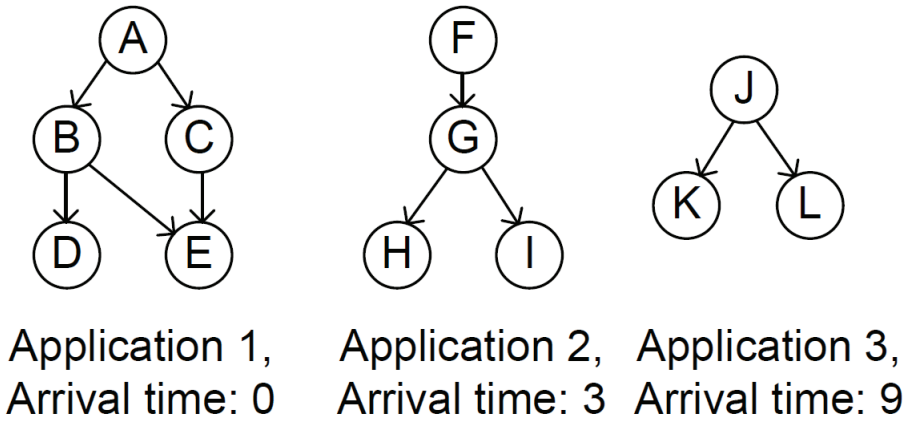
7.4 Motivational Example

An example of task scheduling in CMP

First we give an example of resource allocation in a cloud system. We schedule three applications in a three-cloud system. The DFGs representing these applications are shown in Fig. 7.3(a). Application 1 and 3 are best-effort applications, and Application 2 is AR applications. For simplicity, we assume that every cloud only execute one task at a time, and that the time to load an image of a task is negligible. We will relax these assumptions in the later part of this chapter. The execution times (t) of each task in these applications running on different cloud are shown in Fig. 7.3(b).

Round-robin vs. list scheduling

The round-robin algorithm is one of the load balancing algorithms used in cloud systems, such as the GoGrid [193]. As shown in the "RR" row of Fig. 7.3(c), the tasks are assigned to the clouds evenly, regardless of the heterogeneous performance across different clouds. The execution orders of three clouds are presented in Fig. 7.4(a). In this schedule, task G preempts task B at time 7, since task G is an AR task. And task J is scheduled as soon as possible, starting at time 9, pausing at time 15, and resuming right after previously assigned tasks, i.e., tasks I and D. The total execution time is 32. We assume the execution time of a given application starts from the time when the application is submitted to the time when the application is done. With this scheduling, the average of three application execution time is 22.67 time unit. By using our CLS algorithm, we generate a schedule with the consideration of the heterogeneous performance in the cloud system. The tasks assignment is shown in the "Sch" row of Fig. 7.3(c). And the execution order of three clouds are shown



(a)

	A	B	C	D	E	F	G	H	I	J	K	L
Cloud 1	2	6	5	7	5	4	8	2	4	8	9	2
Cloud 2	3	8	3	10	9	2	8	3	5	4	3	3
Cloud 3	5	4	8	5	2	3	4	6	7	6	7	4

(b)

	A	B	C	D	E	F	G	H	I	J	K	L
RR	1	2	3	1	2	1	2	3	1	1	2	3
Sch	1	3	2	3	3	2	3	1	2	2	2	1

(c)

Figure 7.3: An example of resource allocation in a cloud system. (a) The DFG of three applications, (b) the execution time table, and (c) two different task assignments, where “RR” is the round-robin approach, and ”Sch” is using the list scheduling

in Fig. 7.4(b). In this schedule, tasks are likely assigned to the cloud that can execute them in the shortest time. Task F and G preempt task C and B, respectively. The total execution time is only 21 time unit, which is 34% faster than the round-robin schedule. And the average execution time is 13.33, 41% faster than the round-robin schedule.

In this motivational example, we show the significant improvement by simply using CLS algorithm, even without considering the dynamic adapting scheduling. We will present the details of our algorithms in the following section.

time	0~2	3~7	9~15	15~19	19~26	26~28
Cloud 1	A	F	J	I	D	J
time		2~7	7~15	15~18	18~27	28~31
Cloud 2		B	G	B	E	K
time		2~10		15~21	28~32	
Cloud 3		C		H	L	

(a)

time	0~2				9~11			18~20
Cloud 1	A				H			L
time		2~3	3~5	5~7	9~14		14~18	18~21
Cloud 2		C	F	C	I		J	K
time		2~5		5~9	9~10	10~15	15~17	
Cloud 3		B		G	B	D	E	

(b)

Figure 7.4: Execution orders of three clouds, (a) with the round-robin schedule, and (b) with the list-schedule

7.5 Resource allocation and task scheduling algorithm

Since the manager servers neither know when applications arrive, nor whether other manager servers receive applications, it is a dynamic scheduling problem. We propose two algorithms for the task scheduling: *dynamic cloud list scheduling* (DCLS) and *dynamic cloud min-min scheduling* (AMMS).

Static resource allocation

When a manager server receives an application submission, it will first partition this application into tasks in the form of a DAG. Then a static resource allocation is generated offline. We proposed two greedy algorithms to generate the static allocation: the cloud list scheduling and the cloud min-min scheduling.

Cloud list scheduling (CLS)

Our proposed CLS is similar to CPNT [108]. Some definitions used in listing the task are provided as follow. The *earliest start time* (EST) and the *latest start time* (LST) of a task

are shown as in Equation (7.2) and (7.3). The entry-tasks have EST equals to 0. And The LST of exit-tasks equal to their EST.

$$EST(v_i) = \max_{v_m \in pred(v_i)} \{EST(v_m) + AT(v_m)\} \tag{7.2}$$

$$LST(v_i) = \min_{v_m \in succv_i} \{LST(v_m)\} - AT(v_i) \tag{7.3}$$

Because the cloud system concerned in this chapter is heterogeneous, the execution times of a task on VMs of different clouds are not the same. $AT(v_i)$ is the average execution time of task v_i. The critical node (CN) is a set of vertices in the DAG of which EST and LST are equal. Algorithm 7.1 shows a function forming a task list based on the priorities.

Algorithm 7.1 Forming a task list based on the priorities

Input: A DAG, Average execution time AT of every task in the DAG
Output: A list of tasks P based on priorities

```
1: The EST of every tasks is calculated
2: The LST of every tasks is calculated
3: Empty list P and stack S, and pull all tasks in the list of task U
4: Push the CN task into stack S in the decreasing order of their LST
5: while the stack S is not empty do
6:     if top(S) has un-stacked immediate predecessors then
7:         S ←the immediate predecessor with least LST
8:     else
9:         P ← top(S)
10:        pop top(S)
11:    end if
12: end while
```

Once the list of tasks is formed, we can allocate resources to tasks in the order of this list. The task on the top of this list will be assigned to the cloud that can finish it at the earliest time. Note that the task being assigned at this moment will start execution only when all its predecessor tasks are finished and the cloud resources allocated to it are available. After assigned, this task is removed from the list. The procedure repeats until the list is empty. An static resource allocation is obtained after this assigning procedure that is shown in Algorithm 7.2.

Algorithm 7.2 The assigning procedure of CLS

Input: A priority-based list of tasks P, m different clouds, ETM matrix
Output: A static resource allocation generated by CLS

while The list P is not empty **do**
  $T = top(P)$
  Pull resource status information from all other manager servers
  Get the earliest resource available time for T, with the consideration of the dataset transferring time, responsed from all other manager servers
  Find the cloud C_{min} giving the earliest estimated finish time of T, assuming no other task preempts T
  Assign task T to cloud C_{min}
  Remove T from P
end while

Cloud min-min scheduling (CMMS)

Min-min is another popular greedy algorithm [44]. The original min-min algorithm does not consider the dependencies among tasks. So in the dynamic min-min algorithm used in this chapter, we need to update the mappable task set in every scheduling step to maintain the task dependencies. Tasks in the mappable task set are the tasks whose predecessor tasks are all assigned. Algorithm 7.3 shows the pseudo codes of the CMMS algorithm.

Algorithm 7.3 Cloud min-min scheduling (CMMS)

Input: A set of tasks, m different clouds, ETM matrix
Output: A schedule generated by CMMS

Form a mappable task set P
while there are tasks not assigned **do**
  Update mappable task set P
  for i: task $v_i \in P$ **do**
    Pull resource status information from all other manager servers
    Get the earliest resource available time, with the consideration of the dataset transferring time, responsed from all other manager servers
    Find the cloud $C_{min}(v_i)$ giving the earliest finish time of v_i, assuming no other task preempts v_i
  end for
  Find the task-cloud pair$(v_k, C_{min}(v_k))$ with the earliest finish time in the pairs generated in for-loop
  Assign task v_k to cloud $D_{min}(v_k)$
  Remove v_k from P
  Update the mappable task set P
end while

Energy-aware local mapping

A manager server uses a slot table to record execution schedules of all resources, i.e., servers, in its cloud. When an AR task is assigned to a cloud, the manager server of this cloud will first check the resource availability in this cloud. Since AR tasks can preempt best-effort tasks, the only case where an AR task is rejected is that most of the resources are reserved by some other AR tasks at the required time, no enough resources left for this task. If the AR task is not rejected, which means there are enough resources for this task, a set of servers will be reserved by this task, using the algorithm shown in Alg. 7.4. The time slots for transferring the disk image of the AR task and the task execution are reserved in the slot tables of those servers. The time slots for storing and reloading the disk image of the preempted task are also reserved if preemption happens.

When a best-effort task arrives, the manager server will put it in the execution queue. Every time when there are enough VMs for the task on the top of the queue, a set of servers are selected by the algorithm shown in Alg. 7.5. And the manager server also updates the time slot table of those servers.

The objectives of Alg. 7.4 and 7.5 are to minimize the number of active servers as well as the total energy consumption of the cloud. When every active server is fully utilized, the required number of active servers is minimized. When task t_i is assigned to cloud j, we define the marginal workload of this task as:

$$wl_m(t_i) = wl(t_i) \mod C(S_j) \tag{7.4}$$

where S_j represents the kind server in cloud j, and $C(S_j)$ is the workload capacity of server S_j. To find the optimal local mapping, we group all the tasks that can be executed simultaneously, and sort them in the descending order of their marginal workloads. For each of the large marginal workload task, we try to find some small marginal workload tasks to fill the gap and schedule them on a server.

Algorithm 7.4 Energy-aware local mapping for AR tasks

Input: A set of AR tasks T, which require to start at the same time. A set of servers S
Output: A local mapping

for $t_i \in T$ **do**
  Calculate $wl_m(t_i)$
  if $wl(t_i) - wl_m(t_i) < \sum_{s_i \in idle}(C(s_i))$ **then**
    Schedule $wl(t_i) - wl_m(t_i)$ to the idle servers
  else
    First schedule a part of $wl(t_i) - wl_m(t_i)$ to the idle servers
    Schedule the rest of $wl(t_i) - wl_m(t_i)$ to the active servers, preempting the best-effort tasks
  end if
end for
Sort tasks in T in the descending order of marginal workload, form list L_d
Sort tasks in T in the ascending order of marginal workload, form list L_a
while T is not empty **do**
  t_a = top(L_d)
  if there exists a server j: $C(j) = wl_m(t_a)$ **then**
    Schedule the $wl_m(t_a)$ to server j
  end if
  $s_a = \max_{s_i \in S}(C(s_i))$
  Schedule t_a to s_a, delete t_a from T, L_d, and L_a
  for k: $t_k \in L_a$ **do**
    if $C(s_a) > 0$ and $C(s_a) \geq wl_m(t_k)$ **then**
      Schedule t_k to s_a, delete t_k from T, L_d, and L_a
    else
      Break
    end if
  end for
end while

Feedback information

In the two static scheduling algorithms presented above, the objective function when making decision about assigning a certain task is the earliest estimated finish time of this task. The estimated finish time of task i running on cloud j, $\tau_{i,j}$, is as below:

$$\tau_{i,j} = ERAT_{i,j} + S_I/b + ETM_{i,j} \tag{7.5}$$

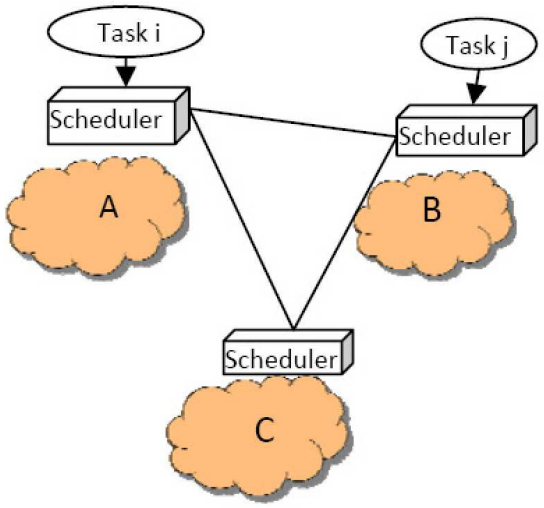
S_I is the size of this disk image, b is the network bandwidth. $ERAT_{i,j}$ is the earliest resource available time based the information from the pull operation. It is also based on

Algorithm 7.5 Energy-aware local mapping for best-effort task

Input: A set of best-effort tasks T, which can start at the same time. A set of servers S
Output: A local mapping

for $t_i \in T$ **do**
  Calculate $wl_m(t_i)$
  Schedule $wl(t_i) - wl_m(t_i)$ to the idle servers
end for
Form a set of active servers S_g that $C(s_i) > 0$, $\forall s_i \in S_g$
Sort tasks in T in the descending order of marginal workload, form list L_d
Sort tasks in T in the ascending order of marginal workload, form list L_a
while T is not empty **do**
  t_a = top(L_d)
  if there exists a server j in S_g: $C(j) = wl_m(t_a)$ **then**
    Schedule the $wl_m(t_a)$ to server j
  end if
  $s_a = \max_{s_i \in S_g}(C(s_i))$
  if $C(s_a) < wl_m(t_a)$ **then**
    $s_a = anyidleserver$
  end if
  Schedule t_a to s_a, delete t_a from T, L_d, and L_a
  for k: $t_k \in L_a$ **do**
    if $C(s_a) > 0$ and $C(s_a) \geq wl_m(t_k)$ **then**
      Schedule t_k to s_a, delete t_k from T, L_d, and L_a
    else
      Break
    end if
  end for
end while

the current task queue of cloud j and the schedule of execution order. But the estimated finish time from (7.5) may not be accurate. For example, as shown in Fig. 7.5(a), we assume there are three clouds in the system. The manager server of cloud A needs to assign a best-effort task i to a cloud. According to equation 7.5, cloud C has the smallest τ. So manager server A transfers task i to cloud C. Then manager server of cloud B needs to assign an AR task j to a cloud. Task j needs to reserve the resource at 8. Cloud C has the smallest τ again. manager server B transfers task j to cloud C. Since task j needs to start when i is not done, task j preempts task i at time 8, as shown in Fig. 7.6. In this case, the actual finish time of task i is not the same as expected.



(a)

Task i

cloud	ERAT	SI/b	ETMi
A	2	1	10
B	3	1	8
C	4	1	6

Task j

cloud	ERAT	SI/b	ETMj
A	8	1	4
B	8	1	3
C	8	1	2

(b)

Figure 7.5: An example of resource contention. (a) Two tasks are submitted to a heterogeneous clouds system. (b)The earliest resource available times (ERAT), the image transferring time (SI/b), and the execution time (EMT) of two tasks on different clouds

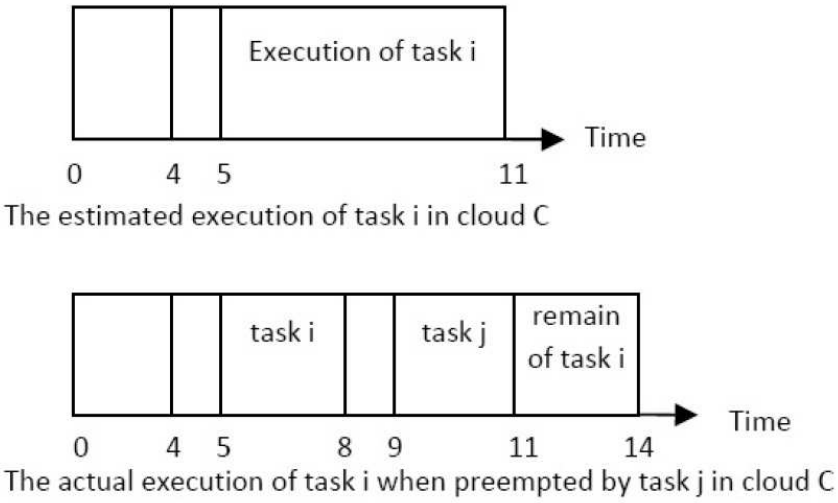


Figure 7.6: The estimated and the actual execution order of the cloud C

In order to reduce the impacts of this kind of delays, we use a feedback factor in computing the estimated finish time. As discussed previously in this chapter, we assume once a task is done, the cloud will push the resource status information to the original cloud. Again, using our example in Fig. 7.5, when task i is done at time T_{act_fin} (=14), manager server C informs manager server A that task i is done. With this information, the manager server A can compute the actual execution time $\Delta\tau_{i,j}$ of task i on cloud j:

$$\Delta\tau_{i,j} = T_{act\_fin} - ERAT_{i,j} \tag{7.6}$$

And the feedback factor fd_j of cloud j is :

$$fd_j = \alpha \times \frac{\Delta\tau_{i,j} - S_I/b - ETM_{i,j}}{S_I/b + ETM_{i,j}} \tag{7.7}$$

α is a constant between 0 and 1. So a feedback estimated earliest finish time $\tau_{fdi,j}$ of task i running on cloud j is as follows:

$$\tau_{fd_{i,j}} = ERAT_{i,j} + (1 + fd_j) \times (S_I/b + ETM_{i,j}) \tag{7.8}$$

In our proposed dynamic cloud list scheduling (DCLS) and dynamic cloud min-min scheduling (DCMMS), every manager server stores feedback factors of all clouds. Once a manager server is informed that a task originally from it is done, it will update the value of the feedback factor of the task-executing cloud. For instance, in the previous example, when cloud C finishes task i and informs that to the manager server of cloud A, this manager server will update its copy of feedback factor of cloud C. When the next task k is considered for assignment, the $\tau_{fd_{k,C}}$ is computed with the new feedback factor and used as objective function.

7.6 Experimental results

Experiment setup

We evaluate the performance of our dynamic algorithms through our own written simulation environment that acts like the IaaS cloud system. We simulate workloads with job

Table 7.1: The mapping of job traces to applications

parameter in our model	values in job traces
task id	job ID
application arrival time	Min(job start time)
task execution time	job end time - job start time
# of CPU required by a task	length(node list) * cpu per node

Table 7.2: Comparison of three data center. The job trace LLNL-uBGL was obtained from a small uBGL, which has the same single core performance as the one shown in this table

Data center	Peak performance (TFLOP/s)	Number of CPUs	normalized performance per core
Thunder	23	4096	1
Altas	44.2	9216	0.85
uBGL(big)	229.4	81920	0.50

traces from the Parallel Workloads Archive [194]. We select three different job traces: LLNL-Thunder, LLNL-Atlsa, and LLNL-uBGL. For each job tracer, we extract four values: the job ID, the job start time, the job end time, and the node list. However, job traces from the Parallel Workloads Archive do not include information about data dependencies. To simulate data dependencies, we first sort jobs by their start time. Then we group up to 64 adjacent jobs as one application, represented by a randomly generated DAG. Table 7.1 shows how we translate those values from job traces to the parameter we use in our application model. Note that we map the earliest job start time in an application as the arrival time of this application, since there is no record about job arrival time in these job traces.

There are three data center in our simulation: 1) 1024 node cluster, with 4 Inetl IA-64 1.4GHz Itanium processors, 8 GB memory, and 185 GB disk space per node; 2) 1152 node cluster, with 8 AMD Opteron 2.4GHz processors, 16 GB memory, and 185GB disk space per node; and 3) 2048 processors BlueGene/L system with 512 MB memory, 80 GB memory. We select these three data center configuration based on the clusters where LLNL-Thunder, LLNL-Atlsa, and LLNL-uBGL job traces were obtained.

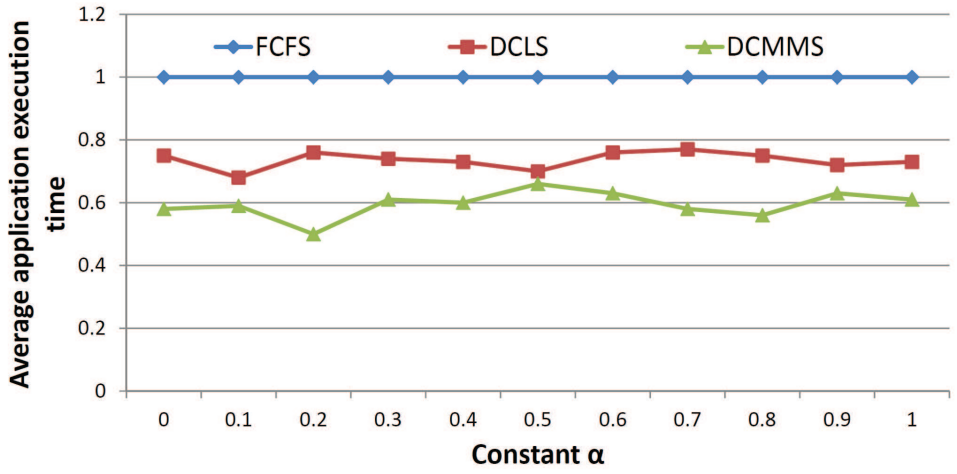


Figure 7.7: Average application execution time in the loose situation

Based on the information in [195], we compare the computational power of these three data center in Table 7.2. With the normalized performance per core, we can get the execution time of all tasks on three different data centers. Among these applications, 20% applications are in the AR modes, while the rest are in the best-effort modes. We assume the bandwidth between two data centers are 1Gbps [196], the bandwidth of nodes inside the data center are 4GBps [195], and the size of every dataset is 1TB [197]. We run these three jobs trace separately in our simulation.

We set the arrival of applications in two different ways. In the first way, we use the earliest start time of a application in the original job trace as the arrival time of this application. We also set the required start time of an AR application as a random start time no later than 30 minutes after it arrives. In most of the cases, applications do not need to contend resources in this setting. We call this a *loose situation*. In the other way, we set the arrival time of applications close to each other. In this setting, we reduce the arrival time gap between two adjacent application by 100 time. It means that applications usually need to wait for resources in clouds. We call this a *tight situation*. In both these two setting, we tunes the constant α to show how the dynamic procedure impacts the average application execution time. We define the execution time of an application as the time elapses from the application is submitted to the application is finished.

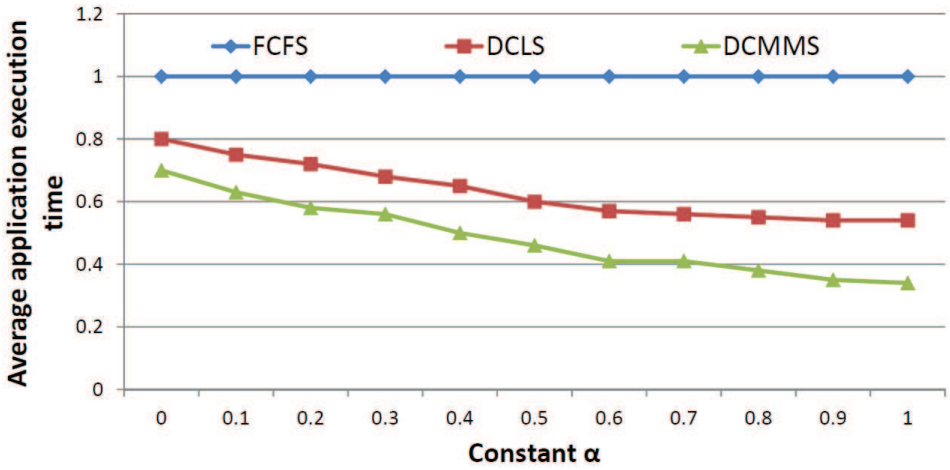


Figure 7.8: Average application execution time in the tight situation

Result

Fig. 7.7 shows the average application execution time in the loose situation. We compare our two dynamic algorithms with the First-Come-First-Serve (FCFS) algorithm [198]. We find out that the DCMMS algorithm has the shorter average execution time. And the dynamic procedure with updated information does not impact the application execution time significantly. The reason the dynamic procedure do not has a significant impact on the application execution time is that the resource contention is not significant in the loose situation. Most of the resource contentions occurs when a AR application preempts a best-effort application. So the estimated finish time of an application is usually close to the actual finish time, which limits the effect of the dynamic procedure. And the manager server does not call the dynamic procedure in most of the cases.

Figure 7.8 shows that DCMMS still outperforms DCLS and FCFS. And the dynamic procedure with updated information works more significantly in the tight situation than it does in the loose situation. Because the resource contentions are fiercer in tight situation, the actual finish time of a task is often later than estimated finish time. And the best-effort task is more likely preempted some AR tasks. The dynamic procedure can avoid tasks gathering in some fast clouds. We believe that the dynamic procedure works even better in a homogeneous cloud system, in which every task runs faster in some kinds of VMs than

Table 7.3: Feedback improvements in different cases

Arrical gap reduce times	DLS	FDLS (α = 1)	Feedback improv.	DMMS	FDMMS (α = 1)	Feedback improv.
1	237.82	253.59	-6.63%	206.31	223.47	-8.32%
20	309.35	286.55	7.37%	262.66	255.44	2.75%
40	445.74	397.15	10.9%	385.48	336.52	12.7%
60	525.32	420.83	19.89%	448.04	343.60	23.31%
80	729.56	537.28	26.36%	648.37	440.05	32.13%
100	981.41	680.22	30.69%	844.33	504.66	40.23%

Table 7.4: Average application execution time with various percentages of AR applications in the loose situation ($\alpha = 0.8$)

	0%	20%	50%	80%	100%
FCFS	1	1	1	1	1
DCLS	0.81	0.75	0.61	0.55	0.49
DCMMS	0.77	0.56	0.52	0.46	0.44

in some other kinds.

In order to find out the relationship between resource contention and feedback improvement, we increase the resource contention by reducing the arrival time gap between two adjacent applications. We reduce this arrival time gap by 20, 40, 60, 80, and 100 times, respectively. In the setting with original arrival time gap, an application usually come after the former application is done. Resource contention is light. And when arrival time gaps are reduced by 100 times, it means during the execution of an application, there may be multiple new applications arriving. Resource contention is heavy in this case. As shown in Table 7.3, the improvement caused by feedback procedure increases as the resource contention become heavier.

We also test our proposed algorithms in setups with various percentages of AR applications, as shown in Table 7.4 and 7.5. The values in the first row represent how many applications are set as the AR applications. The values in the second, the third, and the fourth row are the average application execution time, normalized by the corresponding

Table 7.5: Average application execution time with various percentages of AR applications in the tight situation ($\alpha = 0.8$)

	0%	20%	50%	80%	100%
FCFS	1	1	1	1	1
DCLS	0.63	0.55	0.49	0.43	0.38
DCMMS	0.51	0.38	0.32	0.30	0.27

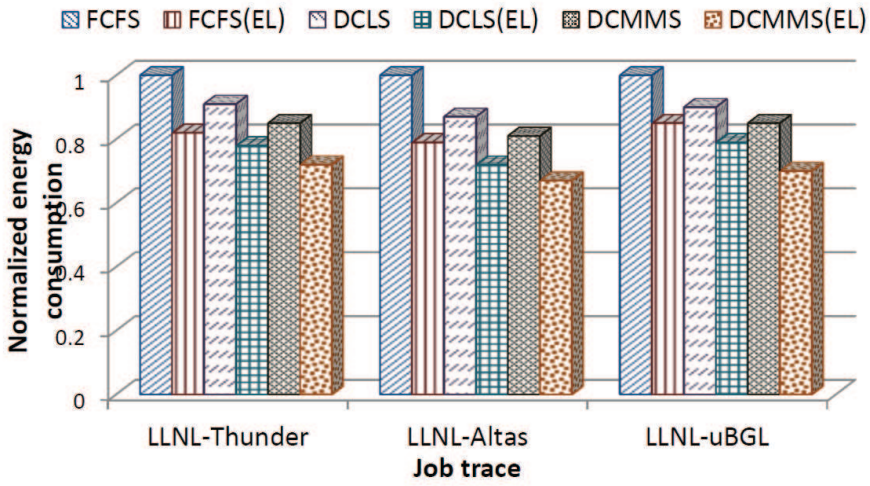


Figure 7.9: Energy consumption in the loose situation. Columns without "(EL)" are schedules without energy-aware local mapping. And columns with "(EL)" are schedules with energy-aware local mapping.

execution time with the FCFS algorithm. From these two tables, we can observe that higher percentage of AR applications leads to a better improvement of the DLS and the DCMMS algorithm, compared to the FCFS algorithm, in both the loose situation and the tight situation. The reason is that more AR applications cause longer delays of the best-effort applications. By using the feedback information, our DLS and DCMMS can reduce workload unbalance, which is the major drawback of the FCFS algorithm. Furthermore, we compare the energy consumption of three algorithms, shown in Fig. 7.9 and 7.10. Both DCLS and DCMMS can reduce energy consumption compared to the FCFS algorithm. In addition, our energy-aware local mapping further reduce the energy consumption significantly, in all three algorithms.

In the future work, we will evaluate our proposed mechanism in existing simulators, so

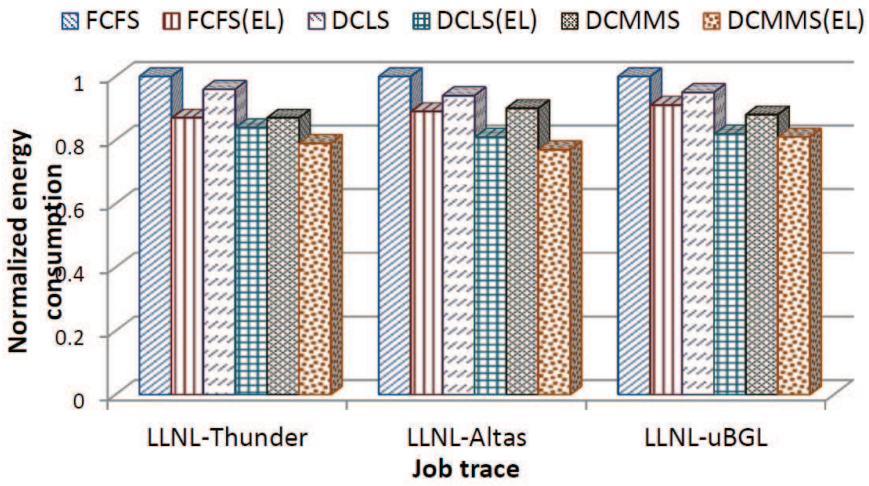


Figure 7.10: Energy consumption in the tight situation. Columns without "(EL)" are schedules without energy-aware local mapping. And columns with "(EL)" are schedules with energy-aware local mapping.

that results can be reproduced easier by other researchers. In addition, we will investigate the implementation of our design in the real-world cloud computing platform. A reasonable way to achieve this goal is to combine our design with the Hadoop platform [171]. The multi-cloud scheduling mechanism and algorithms in our design can be used on the top of the Hadoop platform, distributing applications in the federated multi-cloud platform. When a give task is assigned to a cloud, the Hadoop will be used to distribute tasks to multiple nodes. And our proposed energy-aware local mapping design can be implemented in the Hadoop Distributed File System, which enables the "rack awareness" feature for data locality inside the data center.

7.7 Conclusion

The cloud computing is emerging with rapidly growing customer demands. In case of significant client demands, it may be necessary to share workloads among multiple data centers, or even multiple cloud providers. The workload sharing is able to enlarge the resource pool and provide even more flexible and cheaper resources. In this chapter, we present a resource optimization mechanism for preemptable applications in federated heterogeneous

cloud systems. We also propose two novel online dynamic scheduling algorithms, DCLS and DCMMS, for this resource allocation mechanism. Experimental results show that the DCMMS outperforms DCLS and FCFS. And the dynamic procedure with updated information provides significant improvement in the fierce resource contention situation. The energy-aware local mapping in our dynamic scheduling algorithms can significantly reduce the energy consumptions in the federated cloud system.

Chapter 8 Conclusions

In this dissertation, we have discussed issues in the embedded system design, including thermal issues in the 3D CMP chip, the endurance issue in the PCM, the battery issue in the embedded system design, the impact of inaccurate information in embedded system, and the cloud computing to move the workload to remote cloud computing facilities. Furthermore, we have presented a comprehensive set of optimization techniques for energy-aware embedded systems.

We have presented an online 3D CMP temperature prediction model for multimedia embedded systems. We have also proposed our real-time constrained task scheduling algorithms, the TARS algorithms, to reduce peak temperature in a 3D CMP. By considering the the inter-iteration data dependencies and frequencies assignment collaboratively, our proposed TARS algorithms can significantly reduce the peak temperature on chip and avoid most of the temperature violations. Our simulation results showed that our TARS algorithms can reduce peak temperature by 8.1°C, and avoid up to 80% violations in the top layer and up to 100% violations in the bottom layer.

We have designed an ILP-based memory activities optimization algorithm for the PCM main memory. In order to increase the lifetime of the PCM memory, we schedule and share the data in SPMs, reducing the redundant writes to the PCM memory in this algorithm. Our experimental results show that our ILP algorithm can significantly reduce the number of write by 61% on average. In addition, the performance of the system is also improved due to less writes that are time-consuming.

We have proposed four optimization algorithms for embedded CMP systems equipped with the MLC/SLC PCM + DRAM hybrid memory. In our proposed algorithms, we not only schedule and assign tasks to cores in the CMP system, but also provide a memory configuration that balances the hybrid memory performance as well as the efficiency. Our

experiments show that our genetic-based algorithm generates the best solutions. It significantly reduces the maximum memory usage by 76.8%, compared to the DRAM+ uniform SLC configuration, and improves the efficiency of memory usage by 155.6%, compared to the DRAM + uniform 4 bits/cell MLC configuration. In addition, the performance of the system, in terms of total execution, is also improved by 101%, compared to the uniform 4 bits/cell MLC configuration.

For the battery issue in the embedded system design, we have presented a complete model for task scheduling in distributed mobile DSP system, which includes application model, network model as well as energy model. Using this model, we propose our battery-aware three-phase scheduling algorithms. We show that these algorithms can generate optimal schedules while satisfying lifetime constraint, especially the one based on Min-Min algorithm. These algorithms can also improve the complete ratio of the system.

We have propose a systematic method of measuring the robustness degradation with a stochastic approach. We evaluate impacts of inaccurate information on system robustness in two different scenarios. In our simulation, the makespan is the robustness metric. We find that the makespan with inaccurate information increases proportional to the increase of mean values of task execution time distribution caused by environment changes. Also, 20% to 70% penalty is caused by the inaccurate information used in making scheduling decisions. The impact of environment changes on the robustness is linear to the degree of how much inaccurate information (mainly the shift of means of PMFs) is generated by these environment changes. However, the improvement of re-scheduling with updated information mainly depends on how the task set consists of, not how inaccurate the information is. We also find that the impact of inaccurate means of PMFs is much larger than inaccurate standard deviations. Among these three greedy algorithms, MCT performs the best under inaccurate information. It generates schedules that are almost as optimal as ones from Min-min where accurate information is used. And inaccurate information has less impacts on schedules from MCT than it does on Min-min.

Finally, we have designed a resource optimization mechanism for preemptable applications in federated heterogeneous cloud systems. We also propose two novel online dynamic scheduling algorithms, DCLS and DCMMS, for this resource allocation mechanism. Experimental results show that the DCMMS outperforms DCLS and FCFS. And the dynamic procedure with updated information provides significant improvement in the fierce resource contention situation. The energy-aware local mapping in our dynamic scheduling algorithms can significantly reduce the energy consumptions in the federated cloud system.

Bibliography

[1] A. W. Topol, D. C. La Tulipe Jr., and L. Shi, “Three-dimensional integrated circuits,” *IBM Journal of Research and Development*, vol. 50, no. 4/5, pp. 491–506, 2006.

[2] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, and D. Pantuso, “Die stacking (3D) microarchitecture,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 469–479.

[3] JEDEC, “Failure mechanisms and models for semiconductor devices,” *http://www.jedec.org*, 2009.

[4] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano, “Improving read performance of phase change memories via write cancellation and write pausing,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2010, pp. 1–11.

[5] N. AbouGhazaleh, B. Childers, D. Mosse, and R. Melhem, “Power management in external memory using PA-CDRAM,” *The International Journal for Embedded Systems (IJES)*, vol. 3, no. 1, pp. 65–72, 2007.

[6] A. P. Ferreira, B. Childers, R. Melhem, D. Mosse, and M. Yousif, “Increasing PCM main memory lifetime,” in *Design, Automation & Test in Europe (DATE)*, 2010, pp. 914–919.

[7] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, “Energy management for commercial servers,” *IEEE Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[8] X. Dong and Y. Xie, "AdaMS: adaptive MLC/SLC phase-change memory design for file storage," in *The 16th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2011, pp. 31–36.

[9] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*, 2009, pp. 14–23.

[10] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009, pp. 24–33.

[11] Intel, "Intel microprocessor quick reference guide," http://www.intel.com/pressroom/kits/quickreffam.htm.

[12] B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, "Phase-change technology and the future of main memory," *IEEE Micro*, vol. 30, no. 1, pp. 131–143, 2010.

[13] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby *et al.*, "Phase-change random access memory: A scalable technology," *Journal of VLSI Signal Processing Systems (JSPS)*, vol. 52, no. 4.5, pp. 465–479, 2008.

[14] T. Nirschl, J. Phipp, T. Happ, G. Burr, B. Rajendran, M. Lee *et al.*, "Write strategies for 2 and 4-bit multi-level phase-change memory," in *IEEE International Electron Devices Meeting*, 2008, pp. 461–464.

[15] F. Bedeschi, R. Fackenthal, C. Resta, E. Donze, M. Jagasivamani, E. Buda *et al.*, "A bipolar-selected phase change memory featuring multi-level cell storage," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 1, pp. 217–227, 2009.

[16] M. K. Qureshi, M. M. Franceschini, and J. P. K. L. A. Lastras-Montano, "Morphable memory system: a robust architecture for exploiting multi-level phase change

memories," in *The 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010, pp. 153–162.

[17] L.-F. Chao, A. LaPaugh, and E. H.-M. Sha, "Rotation scheduling: A loop pipelining algorithm," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 3, pp. 229–239, Mar. 1997.

[18] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

[19] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced CPU energy," in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, 1995, pp. 374–382.

[20] D. Mosse, H. Aydin, B. Childers, and R. Melhem, "Compiler-assisted dynamic power-aware scheduling for real-time applications," in *Workshop on Compilers and Operating Systems for Low-Power (COLP00)*, 2000.

[21] D. Shin, J. Kim, and S. Lee, "Intra-task voltage scheduling for low-energy, hard real-time applications," *IEEE Design and Test of Computers*, vol. 18, no. 2, pp. 20–30, 2001.

[22] S. Shivle, R. Castain, H. J. Siegel, A. A. Maciejewski, T. Banka, K. Chindam *et al.*, "Static mapping of subtasks in a heterogeneous ad hoc grid environment," in *13th IEEE Heterogeneous Computing Workshop (HCW 2004)*, 2004.

[23] S. Shivle, H. J. Siegel, A. A. Maciejewski, P. Sugavanam, T. Banka, R. Castain, Chindam *et al.*, "Static allocation of resources to communicating subtasks in a heterogeneous ad hoc grid environment," *Journal of Parallel and Distributed Computing*, vol. 66, no. 4, pp. 600–611, 2006.

[24] M. Qiu and E. Sha, "Cost Minimization while Satisfying Hard/Soft Timing Constraints for Heterogeneous Embedded Systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 2, pp. 1–30, 2009.

[25] M. Qiu, L. Yang, Z. Shao, and E. H. M. Sha, "Rotation Scheduling and Voltage Assignment to Minimize Energy for SoC," in *Proceedings of the 2009 International Conference on Computational Science and Engineering*, 2009, pp. 48–55.

[26] M. Qiu, Z. Jia, C. Xue, Z. Shao, and E. H. M. Sha, "Voltage assignment with guaranteed probability satisfying timing constraint for real-time multiproceesor DSP," *The Journal of VLSI Signal Processing*, vol. 46, no. 1, pp. 55–73, 2007.

[27] V. Nookala, D. J. Lilja, and S. S. Sapatnekar, "Temperature-aware floorplanning of microarchitecture blocks with ipc-power dependence modeling and transient analysis," in *ACM/IEEE ISLPED*, 2006, pp. 298–303.

[28] K. Sankaranarayanan, S. Velusamy, M. Stan, and K. Skadron, "A case for thermal-aware floorplanning at the microarchitectural level," *IEEE TCAD*, vol. 7, pp. 1–16, July 2005.

[29] M. Pathak and S. Lim, "Thermal-aware steiner routing for 3D stacked ICs," in *ACM/IEEE ICCAD*, 2008, pp. 205–211.

[30] P. Zhou, Y. Ma, Z. Li, R. P. Dick, L. Shang, H. Zhou, X. Hong, and Q. Zhou, "3D-STAF: scalable temperature and leakage aware floorplanning for three-dimensional integrated circuits," in *ACM/IEEE ICCAD*, 2008, pp. 590–597.

[31] N. Allec, Z. Hassan, L. Shang, R. P. Dick, and R. Yang, "Thermalscope: Multi-scale thermal analysis for nanometer-scale integrated circuits," in *ACM/IEEE ICCAD*, 2008, pp. 75–82.

[32] Y. Han, I. Koren, and C. A. Moritz, "Temperature aware floorplanning," in *Workshop on Temperature-Aware Computer Systems*, 2005.

[33] P. Chaparro, J. González, Q. Cai, and G. Chrysler, “Dynamic thermal management using thin-film thermoelectric cooling,” in *Proc. ACM ISLPED*, San Fancisco, USA, 2009, pp. 111–116.

[34] K. Puttaswamy and G. Loh, “Thermal herding: microarchitecture techniques for controlling hotspots in high-performance 3d-integrated processors,” in *IEEE HPCA*, 2007, pp. 193–204.

[35] A. Coskun, T. Rosing, and K. Gross, “Proactive temperature balancing for low cost thermal management in MPSoCs,” in *ACM/IEEE ICCAD*, 2008.

[36] F. Mulas, M. Pittau, M. Buttu, S. Carta, A. Acquaviva, L. Benini, and D. Atienza, “Thermal balancing policy for streaming computing on multiprocessor architectures,” in *ACM/IEEE DATE*, 2008, pp. 734–739.

[37] A. K. Coskun, J. L. Ayala, D. Atienza, T. S. Rosing, and Y. Leblebici, “Dynamic thermal management in 3D multicore architectures,” in *ACM/IEEE DATE*, 2009, pp. 1410–1415.

[38] C. Lin, C. Yang, and K. King, “PPT: joint performance/power/thermal management of DRAM memory for multi-core systems,” in *Proc. ACM ISLPED*, San Fancisco, CA, USA 2009, pp. 93–98.

[39] R. Ayoub and T. S. Rosing, “Predict and act: dynamic thermal management for multi-core processors,” in *Proc. ACM ISLPED*, San Fancisco, CA, USA, 2009, pp. 99–104.

[40] C. Zhu, Z. Gu, L. Shang, R. P. Dick, and R. Joseph, “Three-dimensional chip-multiprocessor run-time thermal management,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 8, pp. 1479–1492, Aug. 2008.

[41] X. Zhou, J. Yang, Y. Xu, Y. Zhang, and J. Zhao, "Thermal-aware task scheduling for 3D multicore processors," *IEEE TPDS*, vol. 21, no. 1, pp. 60–70, Jan. 2010.

[42] S. Liu and M. Qiu, "Thermal-aware scheduling for peak temperature reduction with stochastic workloads," in *IEEE/ACM RTAS*, Stockholm, Sweden, Apr. 2010.

[43] Y. Tian and E. Ekici, "Cross-layer collaborative in-network processing in multihop wireless sensor networks," *IEEE Transactions on Mobile Computing*, vol. 6, no. 3, pp. 297–310, 2007.

[44] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *Journal of the ACM*, vol. 24, no. 2, pp. 280–289, 1977.

[45] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *IEEE ISCA*, 2000, pp. 83–94.

[46] K. Skadron, M. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, "Temperature-aware microarchitecture: Modeling and implementation," *ACM TACO*, vol. 1, no. 1, pp. 94–125, Mar. 2004.

[47] R. E. Kessler, "The Alpha 21264 microprocessor," *IEEE Micro*, vol. 19, no. 2, pp. 24–36, Mar./Apr. 1999.

[48] S. Liu, J. Zhang, Q. Wu, and Q. Qiu, "Thermal-aware job allocation and scheduling for three dimensional chip multiprocessor," in *IEEE International Symposium on Quality Electronic Design*, 2010, pp. 390–398.

[49] X. Dong, X. Wu, G. Sun, Y. Xie, H. Li, and Y. Chen, "Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement," in *DAC*, 2008, pp. 554–559.

[50] M. Corporation, "Mmc2001 reference manual," http://www.motorola.com/SPS/MCORE/info_documentation.htm.

[51] T. Instruments, "Tms370cx7x 8-bit microcontroller," http://www-s.ti.com/sc/psheets/spns034c/spns034c.pdf.

[52] M. Corporation, "CPU12 reference manual." http://e-www.motorola.com/brdata/PDFDB/MICROCONTROLLERS/16BIT/68HC12FAMILY/RE

[53] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel., "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *The 10th IEEE International Symposium on Hardware/Software Codesign*, 2002, pp. 73–78.

[54] M. Qiu, L. Zhang, and E. H.-M. Sha, "Ilp optimal scheduling for multi-module memory," in *CODES+ISSS*, 2009, pp. 277–286.

[55] O. Ozturk and M. Kandemir, "Ilp-based energy minimization techniques for banked memories," *ACM Transactions Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 50:1–50:40, 2008.

[56] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distribued Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[57] M. De Vuyst, R. Kumar, and D. M. Tullsen, "Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006, pp. 1–10.

[58] G. Dhiman, V. Kontorinis, D. Tullsen, T. Rosing, E. Saxe, and J. Chew, "Dynamic workload characterization for power efficient scheduling on CMP systems," in *ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, 2010, pp. 437–442.

[59] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas, “Compatible phase co-scheduling on a CMP of multi-threaded processors,” in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006, pp. 1–10.

[60] T. C. Xu, A. W. Yin, P. Liljeberg, and H. Tenhunen, “Operating system processor scheduler design for future chip multiprocessor,” in *International Conference on Architecture of Computing Systems*, 2010, pp. 1–7.

[61] R. Teodorescu and J. Torrellas, “Variation-aware application scheduling and power management for chip multiprocessors,” in *ISCA*, 2008, pp. 363–374.

[62] Micron, “DDR3 SDRAM system-power calculator,” http://www.micron.com/support/dram/power_calc.html.

[63] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger., “Architecting phase change memory as a scalable DRAM alternative,” in *International Symposium on Computer Architecture (ISCA)*, 2009, pp. 2–13.

[64] K. Lee and A. Orailoglu, “Application specific non-volatile primary memory for embedded systems,” in *CODES/ISSS*, 2008, pp. 31–36.

[65] C. Q. Xu, C. J. Xue, J. Hu, and E. H.-M. Sha, “Optimizing scheduling and inter-cluster connection for application-specific DSP processors,” in *IEEE TSP*, 2009, pp. 4538–4547.

[66] C. J. Xue, J. Hu, Z. Shao, and E. Sha, “Iterational retiming with partitioning: Loop scheduling with complete memory latency hiding,” *ACM TECS*, vol. 9, no. 3, pp. 1–26, 2008.

[67] M. Kandemir, G. Chen, F. Li, and I. Demirkiran, “Using data replication to reduce communication energy on chip multiprocessors,” in *ASP-DAC*, 2005, pp. 769–772.

[68] H. Koc, M. Kandemir, E. Ercanli, and O. Ozturk, “Reducing off-chip memory access costs using data recomputation in embedded chip multi-processors,” in *DAC*, 2007, pp. 224–229.

[69] N. Eisley, L.-S. Peh, and L. Shang, “Leveraging on-chip networks for data cache migration in chip multiprocessors,” in *The 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 197–207.

[70] J. Hu, C. J. Xue, W.-C. Tseng, Y. He, M. Qiu, and E. H.-M. Sha, “Reducing write activities on non-volatile memories in embedded CMPs via data migration and re-computation,” in *DAC*, 2010, pp. 350–355.

[71] V. Suhendra, C. Raghavan, and T. Mitra, “Integrated scratchpad memory optimization and task scheduling for MPSoC architecture,” in *CASES*, 2006, pp. 401–410.

[72] L. Zhang, M. Qiu, W.-C. Tseng, and E. H.-M. Sha, “Variable partitioning and scheduling for MPSoC with virtually shared scratch pad memory,” *Journal of VLSI Signal Processing Systems (JSPS)*, vol. 58, no. 2, pp. 247–265, 2010.

[73] J. Li, M. Qiu, J. Niu, and T. Chen, “Battery-aware task scheduling in distributed mobile systems with lifetime constraint,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2011, pp. 743–748.

[74] J. Li, M. Qiu, J. Niu, M. Liu, B. Wang, and J. Hu, “Impacts of inaccurate information on resource allocation for multi-core embedded systems,” in *IEEE 10th International Conference on Computer and Information Technology (CIT)*, 2010, pp. 2692 – 2697.

[75] V. Zivojnovic, H. Schraut, M. Willems, and R. Schoenen, “DSPs, GPPs, and multimedia applications - an evaluation using DSPstone,” in *The International Conference on Signal Processing Applications and Technology*, 1995, pp. 1–5.

[76] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE International Workshop on Workload Characterization*, 2001, pp. 3–14.

[77] N. Agarwal and N. Dimopoulos, "A DSPstone benchmark of CoDeL's automated clock gating platform," in *IEEE Computer Society Annual Symposium on VLSI*, 2007, pp. 508–509.

[78] Lindo Sys. Inc., "LINGO 13.0 - optimization modeling software for linear, nonlinear, and integer programming," http://www.lindo.com/index.php?option=com_content&view=article&id=2&Itemid=10, 2011.

[79] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems," in *IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 2–13.

[80] G. Dhimana, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *DAC*, 2009, pp. 664–669.

[81] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for NVM+DRAM hybrid main memory," in *Conference on Hot topics in operating systems*, 2009, pp. 1–5.

[82] T. Liu, Y. Zhao, C. J. Xue, and M. Li, "Power-aware variable partitioning for DSPs with hybrid PRAM and DRAM main memory," in *DAC*, 2011, pp. 1–6.

[83] T. Ishigaki, T. Kawahara, R. Takemura, K. Ono, and K. Ito, "A multi-level-cell spin-transfer torque memory with series-stacked magnetotunnel junctions," in *Symposium on VLSI Technology (VLSIT)*, 2010, pp. 47–48.

[84] Y. Chen, X. Wang, W. Zhu, H. Li, Z. Sun, G. Sun, and Y. Xie, "Access scheme of multi-level cell spin-transfer torque random access memory and its optimization," in

IEEE International Midwest Symposium on Circuits and Systems, 2010, pp. 1109–1112.

[85] J. Hsieh, C. Wu, and G. Chiu, "Design and implementation for multi-level cell flash memory storage systems," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2010, pp. 247–252.

[86] Y. Chang and T.-W. Kuo, "A management strategy for the reliability and performance improvement of MLC-based flash-memory storage systems," *IEEE Transactions on Computers*, vol. 60, no. 3, pp. 305–320, 2011.

[87] ——, "A reliable MTD design for MLC flash-memory storage systems," in *ACM international conference on Embedded software*, 2010, pp. 179–188.

[88] S. Li and T. Zhang, "Improving multi-level NAND flash memory storage reliability using concatenated TCM-BCH coding," in *ACM Great Lakes symposium on VLSI (GLSVLSI)*, 2009, pp. 499–504.

[89] S. Jung, J. Kim, and Y. Song, "Hierarchical architecture of flash-based storage systems for high performance and durability," in *DAC*, 2009, pp. 907–910.

[90] N. Papandreou, H. Pozidis, T. Mittelholzer, G. Close, M. Breitwisch, C. Lam, and E. Eleftheriou, "Drift-tolerant multilevel phase-change memory," in *IEEE International Memory Workshop*, 2011, pp. 1–4.

[91] A. Jagmohan, L. A. Lastras-Montano, M. M. Franceschini, M. Sharma, and R. Cheek, "Coding for multilevel heterogeneous memories," in *IEEE International Conference on Communications*, 2010, pp. 1–6.

[92] J. Lin, Y. Liao, M. Chiang, I. Chiu, C. Lin, W. Hsu *et al.*, "Design optimization in write speed of multi-level cell application for phase change memory," in *IEEE International Conference of Electron Devices and Solid-State Circuits*, 2009, pp. 525–528.

[93] M. Joshi, W. Zhang, and T. Li, "Mercury: A fast and energy-efficient multi-level cell based phase change memory system," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 345–356.

[94] W. Zhang and T. Li, "Helmet: A resistance drift resilient architecture for multi-level cell phase change memory system," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2011, pp. 197–208.

[95] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms (3rd edition)*. The MIT Press, 2009.

[96] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *Computer*, vol. 27, no. 6, pp. 17–26, 1994.

[97] H. Chen, N. S. Flann, and D. W. Watson, "Parallel genetic simulated annealing: A massively parallel simd approach," *IEEE Trans. Parallel Distributed. Computing*, vol. 9, no. 2, pp. 126–136, 1998.

[98] I. DeFalco, R. DelBalio, E. Tarantino, and R. Vaccaro, "Improving search by incorporating evolution principles in parallel tabu search," in *IEEE Conference on Evolutionary Computation*, 1994, pp. 823–828.

[99] T. D. Braun, H. J. Siegel, N. Beck, L. B. Lasislau *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[100] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO*, 1997.

[101] "Wind River Simics," www.windriver.com/products/simics.

[102] H. Chung, B. H. Jeong, B. Min, Y. Choi *et al.*, "A 58nm 1.8V 1Gb PRAM with 6.4MB/s Program BW," in *International Solid-State Circuits Conference*, 2011, pp. 500–502.

[103] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND flash based disk caches," in *IEEE International Symposium on Circuits and Systems*, 2008, pp. 327–338.

[104] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free," in *International workshop on Hardware/software codesign*, 1998.

[105] T. D. Braun, H. J. Siegel, A. A. Maciejewski, and Y. Hong, "Static resource allocation for heterogeneous computing environments with tasks having dependencies, priorities, deadlines, and multiple versions," *Journal of Parallel and Distributed Computing*, vol. 68, no. 11, pp. 1504–1516, 2008.

[106] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 8–22, 1997.

[107] A. Dogan and F. Ozguner, "Matching and scheduling algorithms for minimizing execution time and failure probability of applications in heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, pp. 308–323, 2002.

[108] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environments," in *Parallel Processing Workshops, 2003. Proceedings. 2003 International Conference on*, 2003, pp. 149–155.

[109] Y. Tian, E. Ekici, and F. Ozguner, "Energy-constrained task mapping and scheduling in wireless sensor networks," in *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, 2005, pp. 211–218.

[110] P. Kumar and M. Srivastava, "Predictive strategies for low-power rtos scheduling," in *Proceedings of the 2000 IEEE International Conference on Computer Design*, 2000, pp. 343–348.

[111] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos, "Data driven signal processing: an approach for energy efficient computing," in *Proceedings of the 1996 international symposium on Low power electronics and design*, 1996, pp. 347–352.

[112] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 89–102.

[113] L. Yan, J. Luo, and N. K. Jha, "Joint dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 1030–1041, 2005.

[114] R. Mishra, N. Rastogi, D. Zhu, D. Mossé, and R. Melhem, "Energy aware scheduling for distributed real-time systems," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003, pp. 22–26.

[115] Y. Yu and V. K. Prasanna, "Power-aware resource allocation for independent tasks in heterogeneous real-time systems," in *Proceedings of the 9th International Conference on Parallel and Distributed Systems*, 2002, pp. 341–348.

[116] Y. Zhang, X. S. Hu, and D. Z. Chen, "Task scheduling and voltage selection for energy minimization," in *Proceedings of the 39th conference on Design automation*, 2002, pp. 183–188.

[117] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proceedings of the 1998 international symposium on Low power electronics and design*, 1998, pp. 197–202.

[118] J. Luo, N. K. Jha, and L. S. Peh, "Simultaneous dynamic voltage scaling of processors and communication links in real-time distributed embedded systems," *IEEE Transactions on Very Large Scale Integrated System*, vol. 15, no. 4, pp. 427–437, 2007.

[119] D. Rakhmatov and S. Vrudhula, "Energy management for battery-powered embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 2, no. 3, pp. 277–324, 2003.

[120] D. Panigrahi, C. Chiasserini, S. Dey, R. Rao, A. Raghunathan, and K. Lahiri, "Battery life estimation of mobile embedded systems," in *Proceedings of the 14th International Conference on VLSI Design (VLSID'01)*, 2001, pp. 55–63.

[121] C. Ma, Z. Zhang, and Y. Yang, "Battery-aware router scheduling in wireless mesh networks," in *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, 2006, pp. 228–241.

[122] J. Paradiso and T. Starner, "Energy scavenging for mobile and wireless electronics," *IEEE Pervasive Computing*, vol. 4, no. 1, pp. 18–27, 2005.

[123] E. M. Yeatman, "Advances in power sources for wireless sensor nodes," in *Proceedings of 1st International Workshop on Body Sensor Networks*, 2004, pp. 6–7.

[124] J. Stevens, "Optimized Thermal Design of Small ΔT Thermoelectric Generators," in *Proceedings of 34th Intersociety Energy Conversion Engineering Conference*, 1999.

[125] P. Mitcheson, T. Green, E. Yeatman, and A. Holmes, "Architectures for vibration-driven micropower generators," *Journal of microelectromechanical systems*, vol. 13, no. 3, pp. 429–440, 2004.

[126] S. C. Kim, S. Lee, and J. Hahm, "Push-Pull: Deterministic Search-Based DAG Scheduling for Heterogeneous Cluster Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 11, pp. 1489–1502, 2007.

[127] M. Levy and T. M. Conte, "Embedded Multicore Processors and Systems," *IEEE Micro*, vol. 29, no. 3, pp. 7–9, 2009.

[128] S. Ali, A. A. Maciejewski, H. J. Siegel, and J. K. Kim, "Measuring the robustness of a resource allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 7, pp. 630–641, 2004.

[129] E. G. Coffman and J. L. Bruno, *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[130] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, and D. Hensgen, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[131] T. D. Braun, H. J. Siegel, A. A. Maciejewski, and S. D. Noemix, "Static mapping heuristics for tasks with dependencies, priorities, deadlines, and multiple versions in heterogeneous environments," in *Parallel and Distributed Processing Symposium, Proceedings International, IPDPS 2002*, 2002, pp. 78–85.

[132] M. M. Eshaghian, *Heterogeneous computing*. Artech House Publishers, 1996.

[133] D. Fernandez-Baca, "Allocating modules to processors in a distributed system," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1427–1436, 1989.

[134] C. Leangsuksun, J. Potter, and S. Scott, "Dynamic task mapping algorithms for a distributed heterogeneous computing environment," in *4th IEEE Heterogeneous Computing Workshop (HCW95)*, 1995, pp. 30–34.

[135] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "A comparison of dynamic strategies for mapping a class of independent tasks onto heterogeneous

computing systems," in *Proc. of the Heterogeneous Computing Workshop. Orlando, USA: IEEE Computer Society Press*, 1998, pp. 57–69.

[136] B. Li and K. Nahrstedt, "Qualprobes: Middleware QoS profiling services for configuring adaptive applications," *Lecture Notes in Computer Science*, vol. 1795/2000, pp. 256–272, 2002.

[137] L. David and I. Puaut, "Static determination of probabilistic execution times," in *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, 2004, pp. 223–230.

[138] Y. A. Li, J. Antonio, H. J. Siegel, M. Tan, and D. W. Watson, "Determining the execution time distribution for a data parallel program in a heterogeneous computing environment," *Journal of Parallel and Distributed Computing*, vol. 44, no. 1, pp. 35–52, 1997.

[139] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *IEEE Real-Time Systems Symposium*, 2002, pp. 279–288.

[140] V. Shestak., J. Smith., H. J. Siegel, and A. A. Maciejewski, "A stochastic approach to measuring the robustness of resource allocations in distributed systems," in *International Conference on Parallel Processing (ICPP)*, 2006, pp. 459–470.

[141] J. Smith, E. K. P. Chong, A. A. Maciejewski, and H. J. Siegel, "Stochastic-based robust dynamic resource allocation in a heterogeneous computing system," in *International Conference on Parallel Processing (ICPP)*, 2009, pp. 188–195.

[142] M. A. Iverson, F. Özgüner, and L. Potter, "Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment," *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1374–1379, 1999.

[143] V. Shestak, S. J., A. A. Maciejewski, and H. J. Siegel, "Stochastic robustness metric and its use for static resource allocations," *Journal of Parallel and Distributed Computing*, vol. 68, no. 8, pp. 1157–1173, 2008.

[144] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System support for automatic profiling and optimization," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[145] C. Krintz, "Coupling on-line and off-line profile information to improve program performance," in *International Symposium on Code Generation and Optimization*, 2003, pp. 69–78.

[146] R. Armstrong, D. Hensgen, and T. Kidd, "The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions," in *7th IEEE Heterogeneous Computing Workshop (HCW98)*, vol. 5, 1998.

[147] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima *et al.*, "Scheduling resources in multi-user, heterogeneous, computingenvironments with SmartNet," in *7th IEEE Heterogeneous Computing Workshop (HCW98)*, 1998, pp. 184–199.

[148] S. Ali, H. J. Siegel, M. Maheswaran, D. Hensgen, and S. Ali, "Representing task and machine heterogeneities for heterogeneous computing systems," *Tamkang Journal of Science and Engineering*, vol. 3, no. 3, pp. 195–208, 2000.

[149] "The R project for statistical computing," 2010. [Online]. Available: http://www.r-project.org/

[150] A. Burns, G. Bernat, and I. Broster, "A probabilistic framework for schedulability analysis," *Lecture Notes in Computer Science*, vol. 2855/2003, pp. 1–15, 2003.

[151] Y. Li, J. K. Antonio, H. J. Siegel, M. Tan, and D. W. Watson, "Determining the execution time distribution for a data parallel program in a heterogeneous computing

environment," *Journal of Parallel and Distributed Computing*, vol. 44, no. 1, pp. 35–52, 1997.

[152] C. Germain-Renaud and O. Rana, "The Convergence of Clouds, Grids, and Autonomics," *IEEE Internet Computing*, vol. 13, no. 6, p. 9, 2009.

[153] B. Sotomayor, R. Montero, I. Llorente, and I. Foster, "Virtual infrastructure management in private and hybrid clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, 2009.

[154] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph *et al.*, "Above the clouds: A Berkeley view of cloud computing," http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[155] T. Forell, D. Milojicic, and V. Talwar, "Cloud management challenges and opportunities," in *IEEE International Symposium on Parallel and Distributed*, 2011, pp. 881–889.

[156] A. I. Avetisyan, R. Campbell, M. T. Gupta, I. Heath, S. Y. Ko, G. R. Ganger *et al.*, "Open Cirrus a global cloud computing testbed," *IEEE Computer*, vol. 43, no. 4, pp. 35–43, 2010.

[157] M. A. Kozuch, M. P. Ryan, R. Gass, S. W. Schlosser, D. O'Hallaron *et al.*, "Cloud management challenges and opportunities," in *Workshop on Automated control for datacenters and cloud*, 2009, pp. 43–48.

[158] D. Cearley and G. Phifer, "Case studies in cloud computing," http://www.gartner.com/it/content/1286700/1286717/.

[159] M. Qiu, M. Guo, M. Liu, C. J. Xue, L. T. Yang, and E. H.-M. Sha, "Loop scheduling and bank type assignment for heterogeneous multi-bank memory," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 69, no. 6, pp. 546–558, 2009.

[160] “Amazon AWS,” http://aws.amazon.com/.

[161] “GoGrid,” http://www.gogrid.com/.

[162] “RackSpace,” http://www.rackspacecloud.com/.

[163] “Microsoft cloud,” http://www.microsoft.com/en-us/cloud/.

[164] “IBM Cloud,” http://www.ibm.com/ibm/cloud/.

[165] “Google Apps,” http://www.google.com/apps/intl/en/business/index.html.

[166] “HP Cloud,” http://www8.hp.com/us/en/solutions/solutionsdetail.html?compURI=tcm:245-300983

[167] “Eucalyptus,” http://www.eucalyptus.com/.

[168] R. Moreno-Vozmediano, R. S. Montero, and I. M., “Elastic management of cluster-based services in the cloud,” in *Workshop on Automated control for datacenters and cloud*, 2009, pp. 19–24.

[169] “RESERVOIR,” www.reservoir-fp7.eu.

[170] J. Dean and S. Ghemawat, “MapReduce: simplified data processing on large clusters,” *Communications of the ACM*, no. 1, pp. 107–113, 2008.

[171] “Apache Hadoop,” http://wiki.apache.org/hadoop/.

[172] C. Moretti, J. Bulosan, and P. J. Thain, D.and Flynn, “All-pairs: An abstraction for data-intensive cloud computing,” in *IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–11.

[173] H. Liu and D. Orban, “GridBatch: Cloud computing for large-scale data-intensive batch applications,” in *IEEE International Symposium on Cluster Computing and the Grid*, 2008, pp. 295–305.

[174] X. Wang, J. Zhang, H. Liao, and L. Zha, “Dynamic split model of resource utilization in mapreduce,” in *International Workshop on Data Intensive Computing in the Clouds*, 2011, pp. 1–10.

[175] H. J. M. Y. Chi and H. Hacigumus, “Performance evaluation of scheduling algorithms for database services with soft and hard SLAs,” in *International Workshop on Data Intensive Computing in the Clouds*, 2011, pp. 1–10.

[176] W. Emeneker and D. Stanzione, “Efficient Virtual Machine Caching in Dynamic Virtual Clusters.” in *SRMPDS Workshop of International Conference on Parallel and Distributed Systems, Hsinchu, Taiwan*, 2007.

[177] N. Fallenbeck, H. J. Picht, M. Smith, and B. Freisleben, “Xen and the art of cluster scheduling,” in *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing, Tampa, Florida, USA*, 2006, p. 4.

[178] B. Sotomayor, R. Llorente, and I. Foster, “Resource leasing and the art of suspending virtual machines,” in *11th IEEE International Conference on High Performance Computing and Communications, Seoul, Korea*, 2009, pp. 59–68.

[179] T. Wood, A. Gerber, K. Ramakrishnan, and J. van der Merwe, “The case for enterprise-ready virtual private clouds,” in *Workshop on Hot Topics in Cloud Computing, San Diego, Califoria, USA*, 2009.

[180] M. Aron, P. Druschel, and W. Zwaenepoel, “Cluster reserves: A mechanism for resource management in cluster-based network servers,” in *Proceedings of the ACM Sigmetrics, Santa Clara, California, USA*, 2000.

[181] I. T. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, “A distributed resource management architecture that supports advance reservations and co-allocation,” in *International Workshop on Quality of Service (IWQoS), London, UK*, 1999, pp. 27–36.

[182] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. Sprenkle, “Dynamic virtual clusters in a grid site manager,” in *International Symposium on High-Performance Distributed Computing (HPDC), Seattle, Washington, USA*, 2003, pp. 90–103.

[183] E. Walker, J. Gardner, V. Litvin, and E. Turner, “Dynamic virtual clusters in a grid site manager,” in *proceedings of Challenges of Large Applications in Distributed Environments, Paris, France*, 2006, pp. 95–103.

[184] K. Keahey and T. Freeman, “Contextualization: Providing one-click virtual clusters,” in *IEEE International Conference on eScience, Indianapolis, Indiana, USA*, 2008.

[185] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum, “Sharing networked resources with brokered leases,” in *USENIX Annual Technical Conference, Boston, Massachusetts, USA*, 2006.

[186] B. Sotomayor, K. Keahey, and I. Foster, “Combining batch execution and leasing using virtual machines,” in *Proceedings of the 17th international symposium on High performance distributed computing, Boston, Massachussets, USA*, 2008, pp. 87–96.

[187] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, “Swift: A language for distributed parallel scripting,” *Parallel Computing*, vol. In Press, no. May, pp. 633–652, 2011.

[188] G. Garzoglio, T. Levshina, P. Mhashilkar, and S. Timm, “ReSS: a resource selection service for the open science grid,” in *International Symposium on Grid Computing*, 2007, pp. 1–10.

[189] P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen, “Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure,” in *IEEE International Conference on Autonomic Computing*, 2006, pp. 5–14.

[190] P. Ruth, P. McGachey, and D. Xu, "Viocluster: virtualization for dynamic computational domains," in *Proceedings of the IEEE International Conference on Cluster Computing, Boston, Massachusetts, USA*, 2005, pp. 1–10.

[191] H. Huang and L. Wang, "P&p: a combined push-pull model for resource monitoring in cloud computing environment," in *IEEE International Conference on Cloud Computing, Miami, Florida, USA*, 2010, pp. 260–266.

[192] S. Zanikolas and R. Sakellariou, "A taxonomy of grid monitoring systems," *Future Generation Computer systems*, no. 1, pp. 163–188, 2005.

[193] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *International Joint Conference on INC, IMS and IDC*, 2009, pp. 44–51.

[194] "Parallel workloads archive," www.cs.huji.ac.il/labs/parallel/workload/.

[195] "Open computing facility - ocf," https://computing.llnl.gov/?set=resources&page=OCF-resource.

[196] W. Jiang, Y. Turner, J. Tourrilhes, and M. Schlansker, "Controlling traffic ensembles in open cirrus," https://www.hpl.hp.com/techreports/2011/HPL-2011-129.pdf.

[197] A. Pavlo, E. Paulson, A. Rasin, D. J. Ababi, D. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD*, 2009, pp. 1–14.

[198] W. Smith, I. Foster, and V. Taylor, "Scheduling with advanced reservations," in *IEEE International Parallel and Distributed Processing Symposium, Cancun, Mexico*, 2000, pp. 127–132.

Vita

Jiayin Li

- **Date and place of birth**
  - 10/02/1983, Guangzhou, China
- **Eduation**
  - **Ph.D. Candidate in Electrical and Computer Engineering**

    University of Kentucky (UKY), Lexington, KY (2009 - present)

    Dissertation title: Energy-aware Optimization for Embedded System with Chip Multiprocessor and Phase-Change Memory
  - **M.S. in Electrical and Computer Engineering**

    Huazhong University of Sci. and Tech. (HUST), Wuhan, China (2006 - 2008)

    Thesis title: Multi-DSP system design (consisting of one ARM, two FPGAs, and four DSPs)
  - **B.S. in Automation** Huazhong University of Sci. and Tech. (HUST), Wuhan, China (2002 - 2006)
- **Professional Position**
  - **Research Assistant**, Department of Electrical and Computer Engineering, University of Kentucky, 2009 - 2011. Advisor: Prof. Meikang Qiu.
  - **Research Assistant**, Institute for Pattern Recognition and Artificial Intelligence, Huazhong University of Sci. and Tech., 2006 - 2008. Advisor: Prof. Jianguo Liu.
- **Honors**

– Kentucky Opportunity Fellowship Award 2011, UKY

- **Publications**

  – **Journals published/accepted**

    1. J. Li, M. Qiu, J. Niu, and Z. Gu, “Online Algorithms for Scheduling Preemptable tasks on IaaS Cloud systems,” accepted in *Journal of Parallel and Distributed Computing*, 2012.
    2. J. Li, M. Qiu, J. Niu, L. Yang, and S. Yeo, “Thermal-Aware Task Scheduling in 3D Chip Multiprocessor with Real-Time Constrained Workloads,” accepted in *ACM Trans. on Embedded Computing Systems (TECS)*, 2011.
    3. J. Li, M. Qiu, J. Niu, Y. Zhu, M. Liu, and T. Chen, “Three-Phase Algorithms for Task scheduling in Distributed Mobile DSP System with Lifetime constraints,” accepted in *Journal of Signal Processing Systems*, 2010.
    4. J. Li, M. Qiu, J. Niu, and T. Chen, “Resource Allocation Robustness in Multi-Core Embedded Systems with Inaccurate Information,” accepted in *Journal of System Architecture*, 2010.
    5. J. Niu, M. Qiu, X. Wang, J. Li, G. Wu, and T. Chen, “Cost Minimization with HPDFG and Data Mining for Heterogeneous DSP,” accepted in *Journal of Signal Processing Systems*, 2010.
    6. M. Qiu, M. Chen, M. Liu, S. Liu, J. Li, X. Liu, and Y. Zhu, “Online Energy-Saving Algorithm for Sensor Networks in Dynamic Changing Environments,” *Journal of Embedded Computing (JEC)*, Vol 3, No. 3, pp. 289-298, Apr. 2010.
    7. F. Hu, M. Qiu, J. Li, T. Grant, D. Tylor, S. McCaleb, L. Butler and R. Hamner, “A Review on Cloud Computing: Design Challenges in Architecture and Security,” *Journal of Computing and Information Technology*, Vol 19, No. 1, pp. 25-56, Mar. 2011

– **Conference published/accepted**

1. J. Li, M. Qiu, J. Niu, and T. Chen, “Battery-Aware Task Scheduling in Distributed Mobile Systems with Lifetime Constraint Systems,” in *IEEE ASP-DAC*, Yokohama, Japan, Jan. 2011, pp. 743-748.
2. J. Li, M. Qiu, J. Niu, and T. Chen, “Security Protection on FPGA against Differential Power Analysis Attacks,” in *7th Annual Cyber Security and Information Intelligence Research Workshop*, Oak Ridge, TN, Oct. 2011.
3. J. Li, M. Qiu, J. Niu, M. Liu, B. Wang, J. Hu, “Impacts of Inaccurate Information on Resource Allocation for Multi-Core Embedded System,” in *IEEE ScalCom*, Bradford, UK, Jun. 2010, pp. 2692- 2697
4. J. Li, M. Qiu, J. Hu, and E. H.-M. Sha, “Thermal-Aware Rotation Scheduling for 3D Multi-Core with Timing Constraint,” in *IEEE SiPS 2010*, San Francisco Bay Area, CA, Oct. 2010
5. J. Li, M. Qiu, J. Niu, W. Gao, Z. Zong, and X. Qin, “Feedback Dynamic Algorithms for Preemptable Job Scheduling in Cloud Systems,” in *Proc. 2010 IEEE/WIC/ACM International Conference on Web Intelligence*, Toronto, Canada, Sep. 2010, pp. 561-564.
6. J. Li, M. Qiu, J. Niu, Y. Zhu, and T. Chen, “Real-Time Constrained Task Scheduling in 3D Chip Multiprocessor to Reduce Peak Temperature,” in *IEEE EUC*, Hong Kong, Dec. 2010, pp. 170-176.
7. J. Li, M. Qiu, J. Niu, Y. Chen, and Z. Ming, “Adaptive Resource Allocation for Preemptable Jobs in Cloud Systems,” in *IEEE ISDA*, Cairo, Egypt, Nov. 2010, pp. 31-36.
8. M. Qiu, J. Liu, J. Li, Z. Fei, Z. Ming, E. H.-M. Sha, “A Novel Energy-Aware Fault Tolerance Mechanism for Wireless Sensor Networks,” in *IEEE/ACM GreenCom*, Chengdu, China, Aug. 2011, pp. 56-61.

9. L. D. Briceno, J. Smith, H. J. Siegel, A. A. Maciejewski, P. Maxwell, R. Wakefield, A. Al-Qawasmeh, R. C. Chiang, and J. Li, “Robust Resource Allocation of DAGs in a Heterogeneous Multicore System,” in *IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, 2010, PP. 1-11