TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Echtzeitsysteme und Robotik

Resource Management in Real-time Multicore Embedded Systems: Performance and Energy Perspectives

Gang Chen

Vollständiger Abdruck der von der Fakultät der Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Tobias Nipkow, Ph.D.
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. habil. Alois Knoll
	2. Prof. Dr. Zhihua Wang, Tsinghua University/China

Die Dissertation wurde am 13.07.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.11.2015 angenommen.

Abstract

Multi-core architectures are believed to be one of the major solutions for future embedded systems, due to their advantages in the high average performance and procurement cost. However, the use of the architecture of multi-core processors in real-time systems poses important challenges on timing analysis, which stems from the interferences among cores when accessing shared hardware resources. To exploit the potential of a multi-core platform in real-time systems, multi-core platform does not only need to meet the requirements of high performance and power efficiency, but also should provide analyzable timing behavior.

In this thesis, we present novel reconfiguration techniques and scheduling algorithms for resource management in real-time multi-core embedded systems. We provide approaches by considering effects of computation and cache memory together to optimize overall performance and energy of real-time multi-core embedded systems. To address the impacts of cache interference on predictability and performance of multi-core embedded systems, we developed a dynamic partitioned cache memory to provide strict cache resource isolation among real-time tasks. The proposed cache is physically implemented and prototyped on FPGA. Based on the proposed dynamic partitioned cache memory, an integrated cache management framework is presented to study and verify the interactions between the task scheduling and the shared cache interference. We also explored system-level power managements to reduce power consumption of the multi-core system under real-time constraints. An energy-aware scheduling technique based on period power management scheme is presented to reduce the static energy consumption for multi-core system with non-deterministic workload.

Zusammenfassung

Multikern-Architekturen stellen aufgrund ihrer Performance- und Kostenvorteile eine bedeutende Lösung im Bereich zukünftiger Embedded Systeme dar. Jedoch bedeutet die Verwendung von Multikernprozessor-Architekturen in Echtzeit große Herausforderungen hinsichtlich Zeitanalysen, welche von Interferenzeffekten zwischen den Kernen herrühren, sobald auf gemeinsame Hardware-Ressourcen zugegriffen wird. Um das Potential von Multikern-Plattformen in Echtzeit-Systemen auszuschöpfen, muss diese Multikern-Plattform nicht nur Anforderungen in puncto hoher Performance und Leistungseffizienz erfüllen, sondern sie muss auch analysierbares Zeitverhalten aufweisen.

In dieser Dissertation werden neuartige Rekonfigurations-Techniken und Scheduling-Algorithmen im Rahmen des Ressourcen-Managements innerhalb von Echtzeit-Multikern Embedded Systems präsentiert. Es werden Lösungen entwickelt, welche sowohl Berechnungseffekte als auch Cache-Speicher berücksichtigen, um Performance und Energie des Gesamtsystems des Echtzeit-Multikern Systems zu optimieren. Um die Auswirkungen von Cache-Interferenzen auf Vorhersehbarkeit und Performance des Multikern Embedded Systems zu adressieren, wurde ein dynamisch partitionierter Cache-Speicher entwickelt, welcher eine strikte Trennung von Cache-Ressourchen zwischen Echtzeit-Aufgaben gewährleistet. Der vorgeschlagene Cache wird physikalisch implementiert und innerhalb eines Prototyps auf einem FPGA umgesetzt. Basierend auf dem vorgeschlagenen dynamisch partitionierten Cache-Speicher wird ein integriertes Cache-Management Framework präsentiert und die Wechselwirkungen zwischen dem Aufgaben-Scheduling und den Shared Cache-Interferenzen diskutiert und verifiziert. Weiterhin werden Leistungs-Management auf System-Level untersucht, um den Leistungsverbrauch des Multikern-Systems unter Echtzeitbedingungen zu reduzieren. Hierbei wird eine energieeffiziente Scheduling-Technik basierend auf periodischem Leistungs-Management-Schema präsentiert, welche den statischen

Energieverbrauch für Multikern-Systeme mit nicht-deterministischer Arbeitsauslastung senkt.

Acknowledgements

PhD study is a very special journey in one's life. I'm not alone on my way to Ph.D. but surrounded by many outstanding people who gives me a lot of help and support. I am sincerely grateful for their generous help. Without them, this work would not have been possible to be finished.

The foremost thanks go to my supervisor, Prof. Dr. Alois Knoll, for his guidance, support, and encouragement over these years. I am very grateful that Prof. Knoll provides me the opportunity to prepare this thesis and give me a lot of freedom to pursue my research interest. I also would like to thank my second advisor Prof. Dr. Zhihua Wang for reviewing my thesis and giving in-depth comments.

I owe many thanks to Dr. Kai Huang for all his valuable suggestions and enduring support. You guided me to overcome challenging problems in the research field and helped me to know how to explore new directions.

I also would like to thank: Biao Hu for his helps on testing the proposed cache and recording the experimental datasets; Long Cheng for taking his time of proofreading my thesis and for nice research cooperation, and Hardik Shah for his valuable suggestions on my research work. Also thanks to the members of our previous lunch-meeting group, M. Ali Nasseri, Martin Eder, and Dr. Dongkun Han. I broaden my research views from the interesting chats while enjoying nice lunch food. I also want to thank the master student, Li Feng, from Sun Yat-Sen Universiy for his helps on conducting chip verification for our cache module. I would like to thank all the other members at the chair, especially Amy Buecherl, Gertrud Eberl, Ute Lomp, and Marie-Luise Neitz, for their very kind help and support.

Finally, I would like to thank my family for their everlasting love and support throughout all these years of my PhD study.

To my wife, Zhe, and
to my son, Yi-fan.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems are autonomous microprocessor-based information processing systems designed for performing certain routines of specific functions repeatedly. Safety-critical real-time systems is one important domain of embedded systems, which require unique design considerations, since timing constraints are imposed by the critical applications. Many of today's existing legacy real-time systems such as automotive systems are still developed on single-core processors [5]. An increasing need for safety, comfort, services, and lower emissions from current and future real-time embedded systems requires higher performance, which single-core embedded processors cannot deliver. To meet such high performance demands, multi-core computing platforms are believed to be one of the major solutions for future real-time embedded systems. However, applying multi-core architectures in the domain of real-time embedded systems still faces important challenges on timing analysis, which are stemmed from the contentions on the access of shared hardware resource. This thesis presents a set of novel resource management techniques to optimize overall performance and energy consumption for real-time multi-core embedded systems. We focus on dealing with part of these challenges and contribute by providing solutions to shared cache management and energy-aware scheduling aspects. The rest of the chapter is organized as follows. The relevant background of our work is reviewed in Section 1.1 and Section 1.2. The potential improvement opportunities as well as major research challenges are studied in Section 1.3. Finally, Section 1.4 summarizes the contributions and draws the outline of this thesis.

1.1 Real-time Embedded Systems

Embedded systems are of great importance in our modern society. Following the success of information and communication technology (ICT), embedded systems are considered to be one of the most important application areas of ICT in recent years [6]. These small and smart electronic systems are driving a new round of information revolution and are widely used in our daily life, ranging from commercial electronics such as cell phones, intelligent wearable devices, cameras, printers to critical infrastructures such as nuclear power plants, communication networks, and factory production lines. Compared to the general computing systems, embedded systems perform a limited set of dedicated functions with limited computing capability and limited power source. Despite of these limitations, the number of processors in embedded systems has still exceeded the number of the processors in general computing systems, and this trend is expected to continue [6]. According to the survey data in [7], there may be two or three general computers in one modern family home, while more than 300 embedded systems co-exist.

Most of current embedded systems are required to work in dynamic environments, where the characteristics of the computational workload cannot always be predicted in advance [8]. The correctness of an embedded system usually depends not only upon its logical correctness, but also upon timely responses to the event within the precise timing constraints. Embedded systems that are subject to such timing constraints are named **real-time embedded systems**. Examples of real-time embedded systems are power-train controller for vehicles, embedded controllers for aircrafts, health monitoring systems, and industrial plant controllers.

Distinguished from the general computing system, real-time embedded systems interact continuously with the environment and have demanding quality specifications which require the system to timely react to external events and execute computational activities within precise timing constraints. For example, adaptive cruise controller (ACC) in automotive system should continuously monitor the speed and brake sensors and react to the value changes of these sensors. The acceleration and deacceleration information should be computed within a limited delay to perform the correct control of a vehicle. Fig. 1.1 illustrates the system structure and application domains of real-time embedded systems. The real-time operating system (RTOS) executed on one specific hardware is responsible for ensuring a predictable execution behavior of the application to allow an off-line guarantee of the required performance [8]. In general, we can classify the real-time embedded systems into two categories as shown in Fig. 1.1.

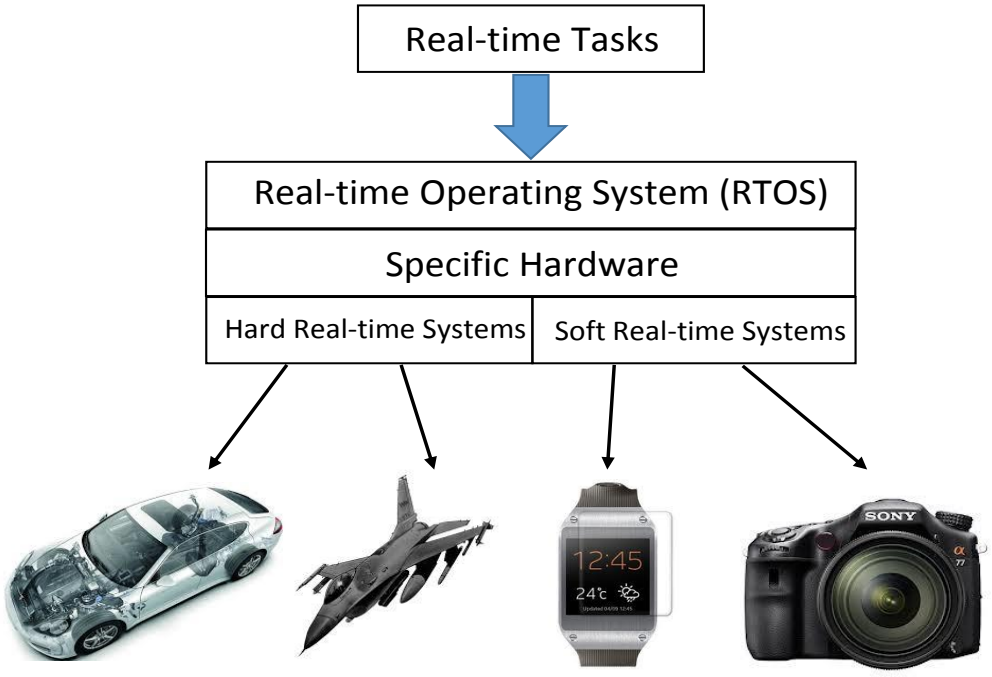


Figure 1.1: Typical structure of real-time embedded systems.

- Hard real-time (HRT) embedded systems: A hard real-time embedded system requires to be high-confident to the predefined timing constraints. The tasks executed on such a system should be constrained to complete the execution and produce the result within a certain deadline. If any of the tasks cannot complete the execution before the deadline, the system is considered to be malfunctioned. The tasks executed on a hard real-time embedded system are usually safety-critical. Failing to meet the deadline constraint will result in severe consequences.

- Soft real-time (SRT) embedded systems: In contrast to the hard real-time embedded systems which require the absolute satisfaction on the timing constraints, the soft real-time embedded systems impose the timing constraints on tasks in terms of the average response time. Minor deadline violations will not be considered as system failure and merely result in temporary system performance degradation. For example, we can tolerate the display delay of the requested page for a certain time in a web browser, while we do not consider the web browser is failed.

Timing constraints imposed on the workload (i.e., real-time tasks) in the real-time embedded systems introduce difficulties which make the design of embedded system becomes par-

ticularly challenging and requires unique design considerations. Real-time embedded systems usually consist of a number of heterogeneous tasks with different features in terms of timing constraints (e.g., worst-case execution time, deadlines, arrival times, etc.) and workload characteristics (e.g., periodic/sporadic, preemptive/non-preemptive, etc.). To ensure the correct system behavior, the design methodology should guarantee the tasks produce the responses to events before their deadlines. Especially for safety-critical applications in the hard-real time system with tight timing constraints, failing to meet the deadline constraints may lead to disaster consequences. For example, to guarantee the safety of the drivers, a brake in the automotive system needs to react within $50\,ms$ after the braking pedal is hit by the driver [9]. Due to these safety concerns, the timing requirement should be guaranteed to be met under any assumption made in the design process. This design requirement is called as the system predictability [10]. In case of hard real-time systems, the classical worst-case design approaches are widely adopted to guarantee the timing constraints in all possible scenarios. Based on the pre-computed worst-case execution time (WCET) of tasks, the system designers can conduct the task schedulability analysis and design a feasible scheduler to guarantee the timing correctness of the complete system. Therefore, the timing correctness of real-time tasks highly depends on the correctness of worst-case execution time (WCET) analysis. One should offer the guarantees on the basic assumption of the worst-case execution time at both run-time and design-time to achieve the timing predictability.

However, the timing predictability of real-time embedded systems highly depends not only upon the hardware infrastructure such as the processor, cache subsystem, and bus subsystem, but also upon the software infrastructure such as operating systems. These dependences threaten the system predictability. At the same time, due to the low-cost design concerns, the standard hardware components, which are aimed at improving the average case performance for the general computing systems, are adopted in the design of real-time embedded systems [11]. Such hardware components are usually disastrous to the timing predictability of the system. One example is the unpredictable performance behavior of the complex cache subsystem which is designed to bridge the performance gap between the off-chip memory and processor. Besides, due to the continuous increase of the functional complexity of real-time systems and stringent timing requirements they have to satisfy, the tremendously increasing design gap existing between requirements and realizations has been a pertinacious problem in the modern real-time embedded systems [11]. This trend makes the system design that supports the timing predictability even harder.

1.2 Multi-core Embedded Systems

As the complexity of the applications is increasing, an embedded system requires an efficient computing platform with the massive computational power to efficiently execute these computationally intensive embedded applications. The trend of the demand for the high performance is expected to continue. According to the predictions from the ITRS Roadmap [12], a need for 300x more performance is predicted by 2022. To achieve high performance, the traditional complex single core architectures usually increase operating frequency. However, the performance of a system is not simplify related to the operating frequency. According to [13], the performance can be computed as a result of frequency and instructions per clock cycle (IPC). This means both of two factors (i.e., frequency and IPC) needs to be taken into account to achieve the high performance. According to the tendencies of today's processor design in industries, many chip makers are focusing on a parallelism oriented design, rather than looking for a possibility to increase clock speed. Besides, the performance gain cannot be continuously achieved by increasing the operating frequency under the constraints of power consumption and thermal dissipation, i.e., hitting the power wall [14]. Therefore, due to these technology limitations and power/thermal limitations, the traditional complex single core architectures can hardly overtake the increasing performance demand of the emergent applications.

Although the pure performance is important, increasing emphasis on energy efficiency design becomes another new trend in the embedded system design. The embedded systems are typically operated with very limited power sources such as battery. Especially in the mobile environment where battery life and size are critical, all the computation must be executed in very low power consumption. However, the increase of the energy density of the battery cannot keep up with the pace of the increasing power consumption demand of modern embedded systems. This gap is expected to continue. According to the recent research in [15], the annual increase in the required power for mobile devices is predicted to about 20%, while the annual advancement in the energy density of batteries is expected to be only 10%. Therefore, it is important to use energy-efficient design techniques to extend the lifetime for battery-operated systems. Besides, using appropriate energy-efficient design techniques in the embedded system can bring several other benefits such as decreasing the heat dissipation and lowering the requirement for expensive packaging and cooling technologies.

At present, the state-of-the-art computing platforms of embedded systems are increasingly moving towards multi-core platforms for the next computing performance leap. In contrast

to single core architectures, multi-core architectures take advantage of Moore's Law, which promises a the constant increase trend on the transistor count in the chip area, to integrate more computing cores into one die. One key advantage that multi-core architecture can leverage is parallel processing. Many computational intensive applications can be divided to several parallel tasks or phases, which can be executed on the multi-core computing platforms simultaneously to achieve parallel processing. Therefore, the raw performance gain in the multi-core computing platforms is achieved by increasing the number of computing cores for the parallel processing execution, rather than increasing the operation frequency. Implementing the multi-core processors with moderate frequencies can bring the performance gains without the growth in power consumption. According to [16], a dual-core solution clocked at 20% less would bring, in theory, 73% more performance than a single core under the same power consumption.

The emergence of multicore computer architectures will have a profound effect on the general computing system due to the advantages in the scalable performance. The multicore processors are already used in real-time systems only with low criticality (also called soft real-time systems), but they are not yet typically employed in the safety-critical real-time computing domain [17]. The main reason is that many of today's multi-core systems are primarily designed for increasing the average-case performance. However, system predictability is the first-class design concern in the safety-critical real-time systems. To achieve system predictability, temporal and spatial isolation of computing components should ideally be provided by the hardware itself [17]. Spatial isolation ensures that an application in one partition cannot change private data of another. Temporal isolation guarantees that the timing characteristics of an application, such as the worst-case execution time (WCET), are not affected by the execution of an application in another partition. Unfortunately, the presence of parallel processing and shared hardware resource in multicore computing platforms destroys temporal and spatial isolation among the safety-critical applications. Therefore, applying multi-core architecture into safety-critical real-time computing domain still faces several challenges. In the next section, we will study major research challenges as well as the potential improvement opportunities in real-time multi-core systems.

1.3 Challenges and Opportunities

Migrating real-time systems to multi-cores computing platforms is a significant challenging task. To apply the multicore computing platforms in the real-time system domains, multi-cores

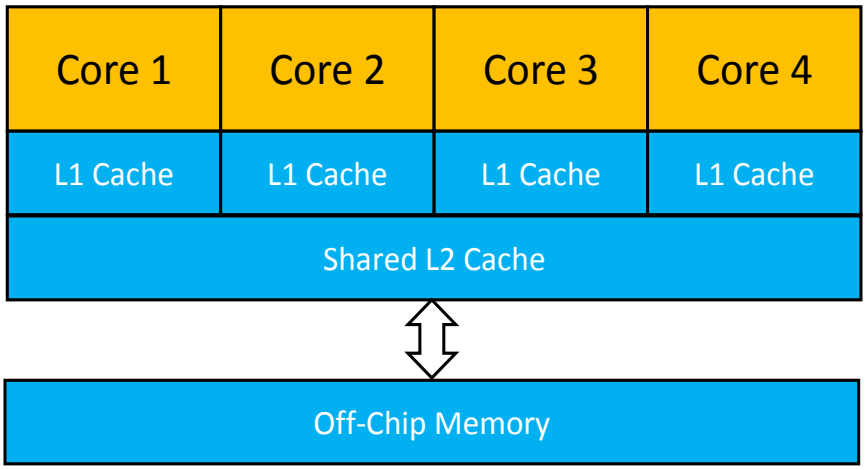


Figure 1.2: A typical multi-core architecture with shared cache memory.

computing platforms are usually used in a conservative manner to resolve the predictability problem. For example, avionics manufacturers usually turn off all cores but one for their highly safety-critical subsystems to guarantee the correct behavior of safety-critical subsystems [18]. Therefore, there are tremendous optimization opportunities based on the different objects when migrating real-time systems to multi-cores. In this section, we discuss the potential improvement opportunities as well as major research challenges in real-time multi-core systems.

1.3.1 Opportunities

Over the past few decades, the continual improvements in processor cycle speed occurs at a much faster rate than improvements in off-chip memory access speed. This ever-widening gap is referred as 'Hitting the Memory Wall', where further increases in processor speed yield little or no performance benefits due to memory access time acting as the primary bottleneck [19]. To alleviate the high latency of the off-chip memory, the cache component is used to keep frequently accessed data of the cores. The performance gap between the processor and off-chip memory is thus alleviated by serving the request directly from caches which are faster than off-chip memory. Currently, multi-core architectures are typically equipped with small L1 caches for every core and a relatively large L2 cache shared among all cores [17, 20]. ARM Cortex-A15 series [21] and openSPARC series [22] are examples of this class of architectures. Fig. 1.2 depicts the typical multi-core architecture with the shared cache memory.

It is extremely challenging to derive tight timing estimates for shared caches because the

behavior of shared cache is hard to predict and analyze statically [23, 24]. A task running on one core may evict useful cache space that is used by another task in another core. These inter-core cache interferences will cause an increase in the miss rate [25], leading to a corresponding decrease in performance. In addition, inter-core cache interferences are extremely difficult to analyze accurately [24], thus resulting in difficulty of estimating the worst-case execution time (WCET) of the application program.

Safety-critical systems must be certified before being deployed. The certification requirements imposed by the industry ensure that the safety standards of critical real-time systems are met. During the certification process, all the components including the software and hardware are scrutinized to ensure conformance to safety standards. The tasks are categorized into different Safety Integrity Levels (SILs). Tasks with different SILs can co-exist and share the same physical hardware resources in a multi-core system. In order to limit the risk of failure of tasks with high SILs, systems must be designed to isolate the execution of the tasks, both in the spatial and temporal domains. To meet such design requirements for the shared cache in the multi-core system, there are two techniques called *cache reservation* and *cache arbitration* [17]. *Cache reservation* technique statically reserves a part of the shared cache for each core, which makes that the shared cache works like private cache for each core, while *Cache arbitration* technique assigns the shared cache only to one core or turns the shared cache off for all cores.

Unfortunately, both of above techniques use the shared cache in a conservative manner because different applications have different performance behaviors under the allocated cache resource. Due to this varying cache requirement of applications, some applications require a large cache to fit the working set for performance efficiency, while the working sets of some applications can fit in a small cache. Therefore, we should partition the shared cache in a more flexible manner to fit such different demands of cache resources among the applications. The difference of performance behaviors among the applications will result in different timing properties such as WCET estimation, which will further have impacts on the scheduler. Thus, there are interesting trade-offs between cache resource allocation and scheduler design that can be explored for optimizations.

Energy conservation is another primary optimization objective in almost every system design. As chip manufacturing technologies are shifting toward sub-micron domains (e.g. Intel has shift its manufacturing technologies into $22\,nm$ in 2011 [26]), the static power increases exponentially and becomes comparable or even greater than dynamic power (as shown in Fig. 1.3). According to [1, 13], the static power accounts for as much as 50% percentage of the total power

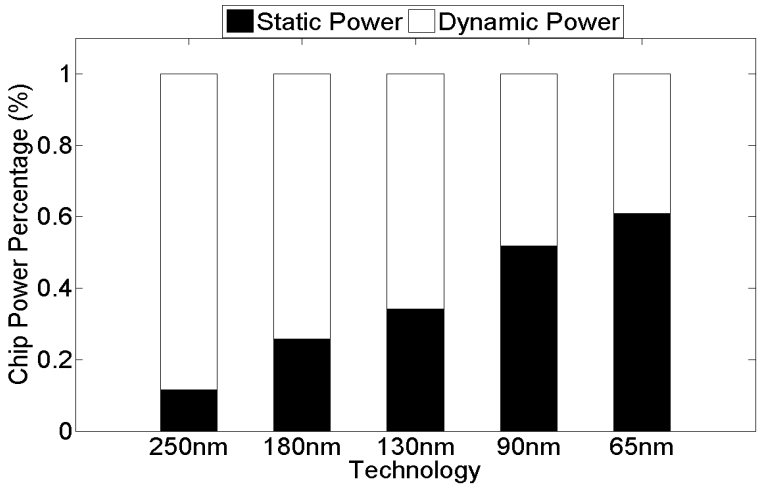


Figure 1.3: Trends in power across process technologies [1].

dissipation for high-end processors in 90 nm technologies. Such sharp increases in power densities hamper powering-on all the cores simultaneously at the nominal voltage while keeping the chip temperature in the safe operating range, because power converts to heat and too much heat can destroy a chip beyond use. Therefore, this effect will in turn limit the performance scale of the multi-core, because not all cores can be powered on due to the temperature constraints. This phenomenon is termed as dark silicon [27]. Thus, efficient power management technology which aimed at reducing static power consumption becomes more and more important for real-time multi-core design.

1.3.2 Challengs

Although state-of-art multi-core architectures provide enormous potential for real-time systems, applying state-of-art multi-core architectures in the domain of real-time embedded systems still faces several challenges. We identify a few new challenges as follows:

- To safely employ multi-core processors for today's and future real-time applications, designers must be able to accurately predict the timing behavior of the real-time applications. Unfortunately, the multi-core architectures, designed for improving average-case performance (i.e., throughput), significantly complicate the timing behavior analysis due to the complex hardware architecture. In the multi-core architectures, different cores typically share hardware resources such as low-level cache (as shown in Fig. 1.2). Shared cache interference in a multi-core system has been recognized as one of major factors that

degrade the average performance [28, 29], as well as predictability of a system [18, 24]. Inter-core interference among these shared caches is difficult to predict and analyze statically. This difficulty actually prohibits an efficient use of the multi-core architecture for real-time systems. For instance, to resolve the predictability problem for multi-core systems, avionics manufacturers usually turn off all cores but one for their highly safety-critical subsystems [18, 30]. The questions are:

How to tackle the shared cache in the context of real-time systems? Can we use such the shared cache in efficient manner while still guaranteeing the real-time constraints?

- Currently, most of the state-of-the-art techniques [31–35] manage the cache resource in core level. However, reserving a region with a constant size to individual cores is often ineffective with respect to the performance, since the tasks assigned on the same core might have different requirements to the amount of cache allocated. Unfortunately, there is no such implementable reconfigurable cache architectures which can guarantee the strict cache isolation among the real-time applications in multi-core system. Most of research work are devoted to analyze theoretical proposals and the simulation of reconfigurable caches. The questions here are:

  How to design a highly flexible cache architecture which allows us to manage the cache resource in an efficient manner? How to realize and prototype it?

- Energy efficiency has become one of the major goals in embedded system design. Using appropriate power management techniques, the lifetime for battery-operated systems could be extended and the heat dissipation could be decreased, lowering the requirement for expensive packaging and cooling technology. According to the International Technology Roadmap for Semiconductors [12], leakage power increases its dominance of total power consumption as semiconductors progress toward 32nm. Designing the power management scheme for the multi-core systems under the requirements of both energy efficiency and timing guarantee is however non-trivial. In general, energy efficiency and timing guarantee are conflict objectives, i.e., techniques that reduce the energy consumption of the system will usually pay the price of longer execution time, and vice versa. The question hereby is:

  How to effectively reduce the static power consumption of multi-core architectures under real-time constraints?

This thesis aims to give partial answers to these new challenges imposed in real-time multi-core architectures. The results of this thesis can be categorized as three parts: (1) A dynamic partitioned cache memory for real-time multi-core systems and its prototype on FPGA (Chapter 2); (2) A integrated framework for automatic cache management and scheduling on the real-time multi-core systems (Chapter 3); (3) Energy-aware scheduling for real-time multi-core embedded system design (Chapter 4).

1.4 Thesis Outline and Research Contributions

In this thesis, we propose a set of novel techniques to address design challenges mentioned in Section 1.3.2. The objective of my research is to develop reconfigurable hardware, scheduling algorithms, and efficient tools for real-time embedded multi-core system optimizations. The proposed research focus on major system components (i.e., processors and cache memory) with various optimization objectives (i.e., energy and performance) by using dynamic resource management techniques (i.e., dynamic cache partitioning and dynamic power management) for the real-time multi-core system. The major contributions of the thesis are summarized as follows:

In Chapter 2, we present a parameterized dynamic partitioned cache memory for real-time multi-core systems. In this cache architecture, the cache resources are strictly isolated to prevent the cache interference among cores. Therefore, the proposed cache can provide predictable cache performance for real-time applications. To efficiently use cache resource and maximize the performance of applications, the proposed cache allows cores to dynamically allocate cache resource with minimal timing overhead according to the demand of applications. The dynamic partitioned cache memory can be interfaced to CPUs for embedded systems such as Altera NIOS II processor and can be physically implemented on an FPGA.

- A parameterized dynamic partitioned cache memory is developed for the real-time multicore systems. The cache size, line size, and associativity of the cache memory can be parameterized during compile time while the partition of the cache can be reconfigured in a flexible manner during runtime. We also design a complete set of APIs with atomic operation, such that the application tasks can reconfigure their cache sizes during runtime.

- In contrast to most existing work [35–40] in the literature, which is devoted to analyze theoretical proposals and the simulation of reconfigurable caches, the cache proposed in

this thesis is physically implemented and prototyped on FPGA. This prototype will bridge the gap between simulation and real systems, and will serve us a real (not simulation) reconfigurable cache for studying and validating cache management strategies on the real-time multi-core system under different cache settings.

- The dynamic partitioned cache memory is interfaced to Altera NIOS II based multi-core system and a functional verification is implemented to verify the correctness of the reconfigurable cache prototype implementation.

- We investigate the chip design process for the proposed cache and find the implementation of the proposed cache is practical in terms of the chip area and power consumption.

In Chapter 3, we tackle schedule-aware cache management scheme for real-time multi-core systems. We present an integrated framework to study and verify the interactions between the task scheduling and the shared cache interference. For a given set of tasks and a mapping of the tasks on a multi-core system, our approach can generate a fully deterministic time-triggered non-preemptive schedule and a set of cache configurations during the compilation time. During runtime, the cache is reconfigured by the scheduler according to offline computed configurations. The generated schedule and the cache configurations together minimize the cache miss of the cache subsystem while preventing deadline miss and cache overflow. Specifically, the detailed contributions are listed below:

- We proposed an integrated cache management framework that improves the execution predictability for real-time multi-core systems. The proposed framework can automatically generate fully deterministic time-triggered non-preemptive schedule and cache configurations for system performance optimization with real-time constraints.

- A synthesis approach for scheduling and cache management is presented to improve the performance of the cache subsystem. In this synthesis approach, the co-design problem of cache partitioning and task scheduling is formulated as integer linear programming (ILP) to minimize the cache miss of the system. With this formulation, the cache size allocation and time-triggered scheduling for each task can be generated automatically, which could avoid deadline miss and cache overflow.

- A share-clock multi-port timer component is developed for the implementation of time-triggered schedule on the multi-core system. Based on this customized hardware compo-

nent, we develop a time-triggered scheduler which can implement cache configuration and execute on the predefined multi-core system.

Chapter 4 explores system-level power managements to reduce power consumption under real-time constraints. We present an energy-aware scheduling technique based on period power management scheme [41, 42] to reduce the static energy consumption for non-deterministic workload. To guarantee real-time requirements, we apply real-time calculus [43] to model the irregular event arrivals and use Real-Time Interface theory [44] for the schedulability analysis. By an inverse use of the well-known pay-burst-only-once principle [45], we develop a new approach to solve the energy-minimization problem for pipelined multi-core embedded systems while guaranteeing the worst-case end-to-end delay. The contributions of this chapter are as follows:

- A new method is developed to solve the energy-minimization problem for pipelined multi-processor embedded systems by inversely using the pay-burst-only-once principle.
- A minimization problem is formulated based on the needed resource of individual stages of the pipeline architecture and a transformation of the formulation to a standard quadratic programming problem with box constraints. The formulated problem is proved to be NP-Hard.
- A quadratic programming heuristic is developed to solve the formulated problem and a formal proof is provided to show the correctness of our approach, i.e., guarantee on the end-to-end deadline requirement.
- A fast heuristic is developed to solve the formulated problem, running with the complexity $O(mn)$.

1.5 Summary

In this chapter, we provide an overview of research problems in the real-time multi-core systems. It has provided an outline of the thesis, specifically dedicating three chapters to target the problems associated with shared caches and energy optimization in the real-time multi-core systems.

Chapter 2

Dynamic Partitioned Shared Cache Memory

This chapter presents design details of our dynamic partitioned shared cache memory, which aims at providing performance predictability of the cache subsystem in a multi-core system. In contrast to most existing work [36–39] in the literature which is devoted to analyze theoretical proposals and simulations of reconfigurable caches, the proposed cache is physically implemented and prototyped on FPGA. In this chapter, we firstly identify the key challenges of designing such dynamic partitioned shared cache memory for multi-core systems. Afterwards, we provide the details about how to design and prototype the proposed cache. Finally, experimental results are presented to evaluate the effectiveness.

2.1 Introduction

Over the past few decades, both the speed and the number of transistors in a dense integrated circuit of processors doubled approximately every two years. This trend is commonly known as the Moore's Law. However, the access speed of the off-chip memory did not follow the same trend. To bridge the performance gap between the off-chip memory and processor speed, the cache component is included in nearly all processors to transparently store frequently accessed instructions and data. Since the access speed of the cache component is much faster than the off-chip memory, the cache component can effectively alleviate the performance gap between the processor and off-chip memory by exploiting the temporal and spatial locality properties of programs.

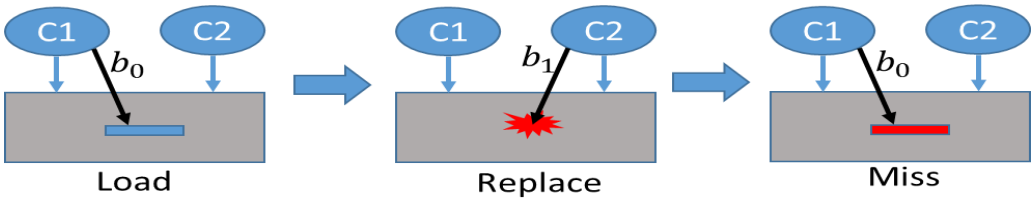


Figure 2.1: A graphic example of cache interference.

Nowadays, the computing systems are increasingly moving towards multi-core platforms for the next computing performance leap. Increasing the number of cores increases the demanded memory access speed. The performance gap between memory and processor is further increased in multi-core platforms. To alleviate the increasing high latency of the off-chip memory, multi-processor system-on-chip (MPSoC) architectures are typically equipped with hierarchical cache subsystems. For instance, ARM Cortex-A15 series [21] and openSPARC series [22] all use small L1 caches for individual cores and a relatively large L2 cache shared among different cores. In such hierarchical cache subsystems, the shared cache can be accessed by all cores so that several important advantages can be achieved, such as increased cache space utilization and data-sharing opportunities.

At the same time, the shared caches also bring several drawbacks. The main disadvantage of shared caches is that uncontrolled cache interference can occur among cores, because all cores are allowed to freely access the entire shared caches. A graphic example of uncontrolled cache interference is illustrated in Fig. 2.1. In this example, the data element b_0 is loaded into shared cache when core 1 needs to access the data element b_0. One cache line is occupied by core 1 for future usage. Later, when core 2 needs to access another data element b_1 which is mapped in the same place of b_0, the cache line occupied by b_0 is replaced by b_1. This will result in a cache miss for the later access of b_0 on core 1. As a result, scenarios may occur where one core may constantly evict useful cache lines belonging to another core, while such cache evictions cannot bring a significant improvement for itself. Such cache interferences will cause the increase in the miss rate [25], leading to a corresponding decrease in the performance. In addition, uncontrolled cache interferences also result in unfairness [46] and the lack of Quality-of-Service (QoS) [47]. For example, a low priority application running on one core may rapidly occupy the entire shared cache and evict most of the cache lines of higher priority applications co-executed on another core.

Multi-core platforms have been used to realize a wealth of new products and services across

many domains due to the average high performance. However, safety-critical real-time embedded systems are failed to be benefited by this trend. In safety-critical real-time embedded systems including avionic and automotive systems, failures may lead to disastrous consequences, such as losses of lives. Therefore, the safety-critical systems must be certified to ensure their reliability before being applied. System predictability is one of the most important principles for the development of the certifiable computing platforms [18]. In addition, system predictability is also one of the fundamental requirements for the real-time correctness. The timing correctness of real-time systems usually depends on worst-case execution time (WCET) analysis of programs. In the modern real-time computing theory, worst case execution time (WCET) of individual tasks can be calculated as a prior to compute the schedulability of the complete system. Unfortunately, this assumption is not even true in a modern multi-core platform equipped with a shared cache. The main problem is that the behavior of shared cache is hard to predict and analyze statically [23, 24] in multi-core systems. Cache interferences as shown in Fig. 2.1 are extremely difficult to accurately analyze[24], thus resulting in difficulties of estimating the worst-case execution time (WCET) of the application program. How to tackle the shared cache in the context of real-time systems is still an open issue [23] and the difficulty actually prohibits an efficient use of multi-core computing platforms for real-time systems. For instance, to resolve the predictability problem for multi-core computing platforms, avionics manufacturers usually turn off all cores but one for their highly safety-critical subsystems [18, 30]. The work in [31] also reports that inter-core cache interferences on a state-of-the-art quad-core processor increased the task completion time by up to 40%, compared to when it runs alone in the system. Therefore, it is crucial to design an interference-free shared cache memory component to improve the performance and predictability of multi-core systems.

Cache partitioning is a promising technique to tackle the aforementioned problem [25,48,49], which partitions the shared cache into separate regions and designates one or a few regions to individual cores. Cache partitioning also has the advantage that it can provide spatial isolation of the cache, which is required by safety standards such as ARINC 653 in the avionic domain. According to [25], cache partitioning technique can be classified as software-based and hardware-based approach. The software-based approach, which is also known as *page coloring*, assigns different cache sets to different partitions by exploiting the translation from virtual to physical memory addresses. Although the software-based approach has been extensively studied in the community and can derive some promising results to improve the system performance for general purpose computing systems [29, 50–53] and guarantee system prediction for safety

real-time computing systems [18,31,54,55], it has three important limitations: first, it requires significant modifications of the virtual memory system, a complex component of the OS. Second, one main problem for page-coloring based techniques is the significantly large timing overhead when performing recoloring. This timing overhead on the one hand prohibits a frequent change of the colors of pages [28,29], on the other hand makes color changes of tasks whose execution time is less than the page-change overhead not worthy. Thus, software cache partitioning approach can only work well when recoloring is performed infrequently [25]. Third, the page-coloring techniques [18,31,55] partition the cache by sets at OS-level, cooperating OS timing overhead also needs to be carefully considered in real-time systems. Besides, the state-of-the-art studies [18,31,54,55] implement and evaluate the proposed approaches in a general-purpose operating system Linux (OS) patched with real-time extensions. Due to the complexity of the Linux kernel, the impacts of kernel activities, which have a considerable effect on real-time tasks, are hard to be predicted and evaluated. In contrast, hardware-based approach usually assigns cache ways within each cache set to different partitions with minimal timing overhead. However, most of the hardware-based cache partitioning approaches in the literature can only be used in uni-processor systems [56–58] or cannot strictly guarantee the cache space isolation among real-time applications [2].

To tackle these problems, we present a dynamic partitioned cache memory for multi-core systems and implement dynamic cache partitioning in our customized reconfigurable cache hardware component with minimal timing overhead. In this cache architecture, the cache resources are strictly isolated to prevent the cache interference among cores. Therefore, the proposed cache can provide predictable cache performance for real-time applications. To efficiently use cache resources, the proposed cache allows cores to dynamically allocate cache resource according to the demand of applications. The proposed cache is physically implemented and prototyped on FPGA, enabling us to evaluate dynamic cache management scheme within a real embedded system. Besides, we also investigate the chip design process for the proposed cache and find that the implementation of the proposed cache is practical in terms of the chip area and power consumption. Finally, we also implement a functional test to verify the correctness of the reconfigurable cache prototype implementation. The functional test can prove that the correctness of our cache design.

The rest of the chapter is organized as follows: Section 2.2 summarizes the existing research work on reconfigurable cache architecture. Section 2.3 presents the hardware design and implementation of the dynamic partitioned cache memory. Section 2.4 explains the process of

hardware generation of the dynamic partitioned cache memory. Section 2.5 describes the software interfaces. Experimental evaluation is presented in Section 2.6 and Section 2.7 concludes this chapter.

2.2 Related Work

Numbers of general or application specific reconfigurable cache architectures have been proposed in the literature. Albonesi et al. [36] proposed a selective ways cache architecture for uni-processor system, which can disable a subset of the ways in a set associative cache during periods of modest cache activity and enable the full cache to remain operational for more cache-intensive periods. By collecting cache performance of applications on runtime, Suh at al. [37] proposed a general dynamic partitioning scheme for the set associative cache. The simulation based evaluation shows the potentials for performance improvement. Benitez et al. [38] proposed amorphous cache aimed at improving performance as well as reducing energy consumption. As opposed to the traditional cache architectures, the proposed cache architecture uses homogenous sub-caches which can be selectively turn-off according to the workload of the application and reduce both its access latency and power consumption. Based on the cache architecture in [2], Sundararajan et al. [39] presented a set and way management cache architecture for efficient run-time reconfiguration.

Most of above work [36–39] is devoted to analyze theoretical proposals and the simulation of reconfigurable caches. Thus, their systems are only tailored at simulation. Only few research work [2, 56–58] is devoted to the physical implementation of the proposed cache models. Zhang et al. [2] proposed a reconfigurable cache architecture where the cache ways configuration could be tuned via the combination of configuration register and physical address bits. Fig. 2.2 illustrates a four-way set-associative reconfigurable cache architecture proposed in [2]. In this architecture, the cache ways selection during the reconfiguration is related to the address bits of the application, which cannot guarantee the strict cache isolation among real-time applications. As shown in Fig. 2.2, one way is selected when Reg0= 0 and Reg1= 0. However, which exact one way is selected is also determined by two physical address bits $A18$ and $A19$. The overlapped address mapping of the real-time applications on these two physical address bits $A18$ and $A19$ will result in cache interference. In addition, the number of the allocated cache ways can *only* be configured to be a power of two, which prevents the efficient usage of the limited cache ways. Gil et al. [56, 57] presented one general-purpose reconfigurable cache design *only* for uni-processor

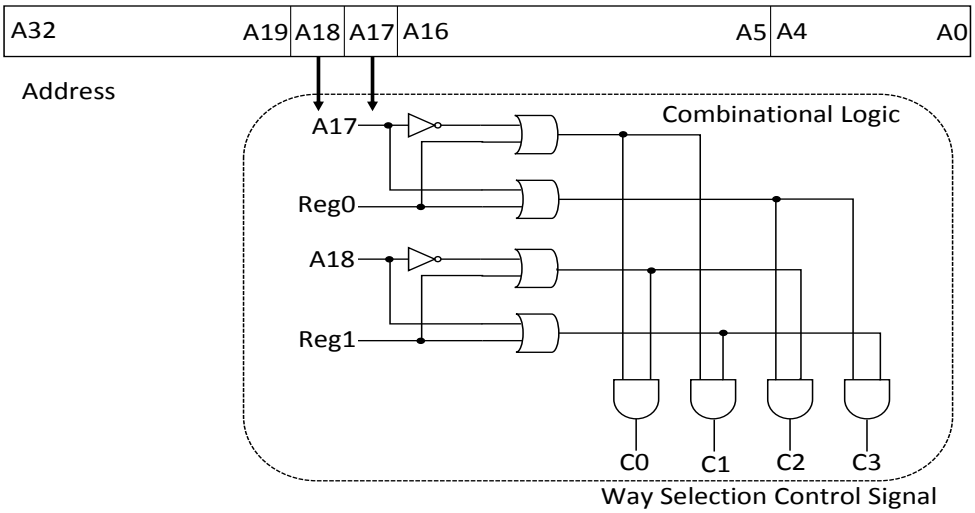


Figure 2.2: Illustration of four-way set-associative reconfigurable cache architecture in [2].

systems to be implemented on FPGA. Besides, the proposed reconfigurable cache design [56,57] can only work as direct mapped cache or 2-way set associative cache. Thus, this cache design is quite limited for usage. Motorola M*CORE processor [58] supports a configurable unified set-associative cache whose four ways could be individually shutdown to reduce dynamic power during cache accesses. Besides, the cache in M*CORE processor [58] can be configured as different functional cache (instruction cache, data cache, or unified cache). However, M*CORE processor is developed for uni-processor systems. It is not easy to extend such reconfigurable cache into multi-core systems due to synchronization and atomic operation issue [59].

In this chapter, we propose a parameterized reconfigurable cache architecture for real-time multi-core system and physically implement it on FPGA. In this architecture, cache ways can be tuned without constraints and can be efficiently and dynamically partitioned and allocated to applications, which can guarantee the cache resource is strictly isolated among real-time applications to prevent the cache interference. Besides, our reconfigurable cache memory supports parameterized design. The cache size, line size, and associativity of the cache memory can be parameterized during compile time. The reconfigurable cache memory can be automatically generated by setting the parameters, e.g., cache size, line size, and associativity. Thus, the proposed reconfigurable cache memory supports hardware generation. The dynamic partitioned cache memory can be interfaced and executed with CPUs for embedded systems such as Altera NIOS II processor. We provide one physical prototype on FPGA and this prototype will serve us a real (not simulation) reconfigurable cache for studying and validating cache management

strategies on the real-time multi-core system under different cache configurations.

2.3 Dynamic Partitioned Cache Design and Implementation

In this section, we present the development and implementation of the reconfigurable cache, which can be interfaced to NIOS-based multi-core systems. The developed cache supports dynamic way-based cache partitioning. As shown in Fig. 2.3, the shared cache is partitioned in ways. Each core can dynamically tune the number of selective-ways during runtime. For example, core 2 can select the 3rd and 6th way by calling the cache reconfiguration APIs. We firstly present the challenges of designing such a reconfigurable cache for multi-core systems. Then, we provide the details about how to design and implement the reconfigurable cache.

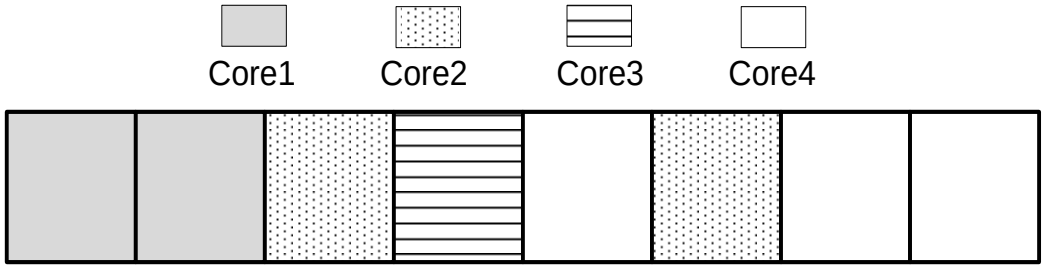


Figure 2.3: Way-based cache partitioning.

2.3.1 Design Consideration and Challenge

Cache coherency problem is one of critical design considerations for the dynamic way-based cache partition infrastructure. According to the Altera NIOS II datasheet [60], the current NIOS architecture does not provide hardware cache coherency. When creating multiprocessor systems, software for each processor is required to locate in its own unique region of off-chip memory to avoid to implement cache coherency [60]. NIOS II SBT provides a simple scheme of memory partitioning that allows multiple processors to run their software from different regions of the same off-chip memory [60]. Besides, according to the state-of-the-art research work in [61], current cache coherence strategies are not suitable for the real-time system. In this chapter, we mainly focus on studying the cache interference among the cores, and follow this official design presented in [60] from Altera to create our multi-core system. Actually, this kind of memory architecture known as Partitioned Global Address Space (PGAS) has been widely

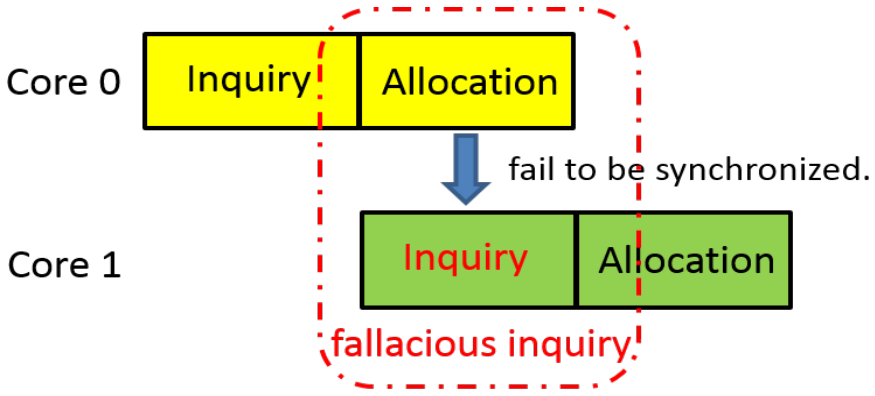


Figure 2.4: Atomic operations.

accepted in the embedded community for efficiency reasons and real-life examples come from Adapteva Parallella multi-core chip E16G301 and E64G401 [62]. Note that inter-core cache interference still exists although software on each core runs in different regions of the same off-chip memory[1]. Besides, the proposed shared cache architecture is multi-port cache, which allows NIOS cores to access the cache concurrently.

Another important part that should be carefully considered is atomic operations. In general, to adaptively change the cache size, one core needs a two-phase operation, i.e., inquiry and allocation (as shown in Fig. 2.4). In the inquiry phase, the core needs to check which ways are available at the current moment. Then, based on the inquiry results, the core can acquire cache resource in the allocation phase. Normally, this procedure works well in a uni-processor system due to no core interference. However, in a multi-core system, when one core is checking the cache resource state, the cache management logic might be conducting cache allocation for other cores. This may lead to the fallacious cache resource state inquiry, because the results of the on-going cache allocation fail to be synchronized to the current cache resource state. Therefore, in a multi-core system, the APIs for adjusting the cache size should be guaranteed to be atomic for implementing synchronization primitives. Hence, we develop a component, called *cache ways management unit (CWMU)* to execute cache ways allocation and release, which grantees the offered APIs atomicity.

The implementation of the replacement policy for the way-based partitioning cache is another design challenge. To efficiently use the limited cache resource, the proposed cache architecture allows each core to dynamically tune its cache ways without any constraints. This will result in that the cache ways occupied by one core might not be adjacent to each other. As

[1]Unique region of each processor on off-chip memory is larger than the total cache size

shown in Fig. 2.3, the 3rd and 6th ways are occupied by core 2. Therefore, standard replacement policies cannot be applied. In this chapter, we develop block reference field logic (BRFL) to maintain this discontinuous cache ways distribution.

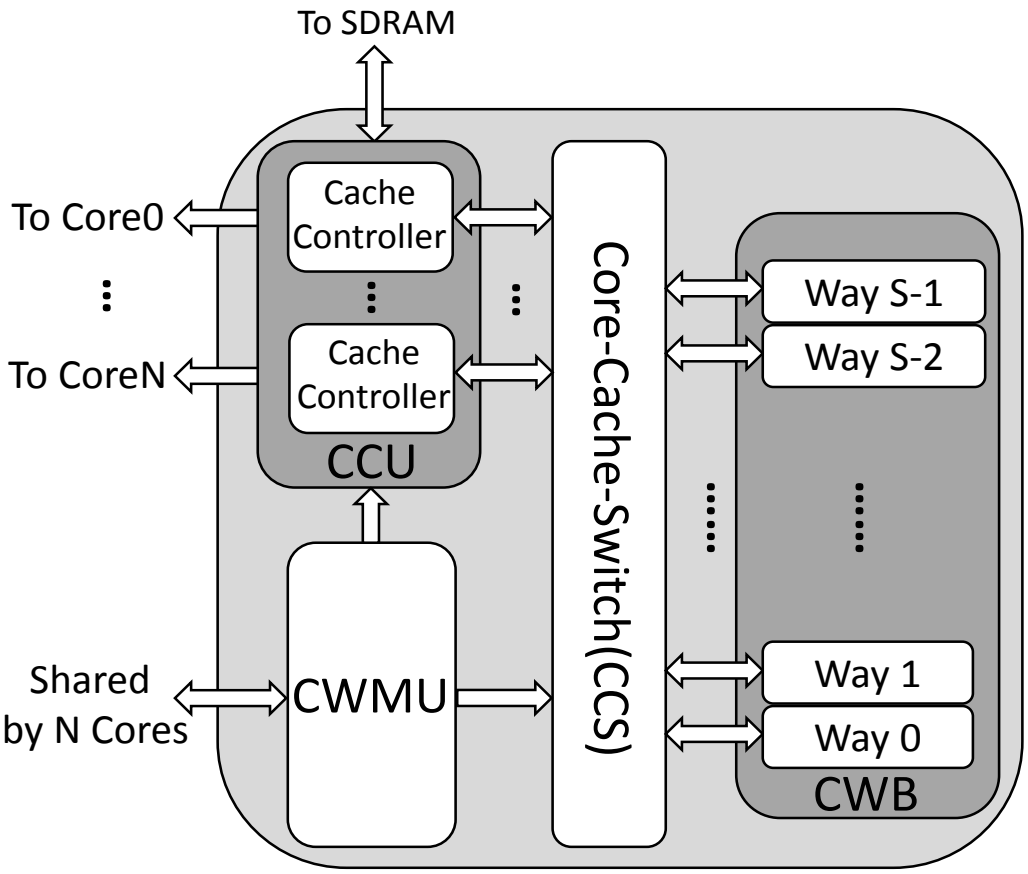


Figure 2.5: Reconfigurable cache architecture.

2.3.2 Reconfigurable Cache Architecture

This section presents an overview of the proposed reconfigurable shared cache architecture. The reconfigurable shared cache component allows cores to dynamically change the number of owned cache ways. As depicted in Fig. 2.5, the proposed reconfigurable shared cache consists of *cache ways management unit (CWMU)*, *cache control unit (CCU)*, *core to cache switch (CCS)*, and *cache ways block (CWB)*. In the proposed architecture, *cache ways management unit (CWMU)* controls the cache ways allocation according to the reconfiguration requests of the cores. The reconfiguration port of *CWMU* is shared by all cores. *Cache control unit (CCU)* manages the

cache memory accesses by instantiating N *cache controllers* for a N-core system. *Core to cache switch (CCS)* can dynamically connect cores to cache ways blocks according to ways mask register of each core, which is maintained by *CWMU* according to the private cache ways pool of the cores. *Cache ways blocks (CWB)* are memory blocks used for tag and data storage.

2.3.3 Cache Ways Management Unit (CWMU)

The *cache ways management unit (CWMU)* is used to manage cache ways in a centralized manner, by which each core can send reconfiguration command to dynamically regulate its cache ways. *CWMU* is connected to N NIOS cores by *avalon slave interface (ASI)* and a round-robin arbiter is automatically created between N NIOS cores and CWMU by Altera SOPC builder. As shown in Fig. 2.6, when *CWMU* receives one command from one NIOS core, the *CMD decoder* component can distinguish the core ID (i.e., identity which core sends this command) and its command type (i.e., identity command types in Tab. 2.2). If it is allocation ways command, ways IDs will be fetched from the *global ways pool*. Then, the fetched ways IDs are put into the cache ways pool of the distinguished core. Then, *core to cache switch (CCS)* is controlled to connect cache ways to the distinguished core according to the cache ways pool. Before fetching ways IDs from *global ways pool*, the logic will check whether there are enough ways in the pool. If no enough ways exist in the pool, *cache overflow* error will be returned to the distinguished core. Note that the approach in [48] can be applied to calculate one safe cache configuration for real-time applications, which can guarantee that *cache overflow* error will never occur. In contrast to the procedure of allocation ways command, release ways command will fetch ways IDs from the cache ways pool of the distinguished core to the *global ways pool*. Ways occupied by the distinguished core and replacement information are correspondingly updated at this point. Note that due to this centralized management scheme, cores do not need to inquiry the cache state any more before the allocation operation. Therefore, the APIs for cache reconfigurations are atomic.

2.3.4 Cache Control Unit (CCU)

Cache control unit (CCU) instantiates N *cache controllers* for an N-core system, where each core owns one *cache controller*. *Cache controller* is used to maintain the access for its corresponding NIOS core. Thus, this shared cache allows NIOS cores to access the cache concurrently. For *cache controller*, we employ the write-through policy for each write operation. Cache write-through policy is inherently tolerant to soft errors due to its immediate update feature [63].

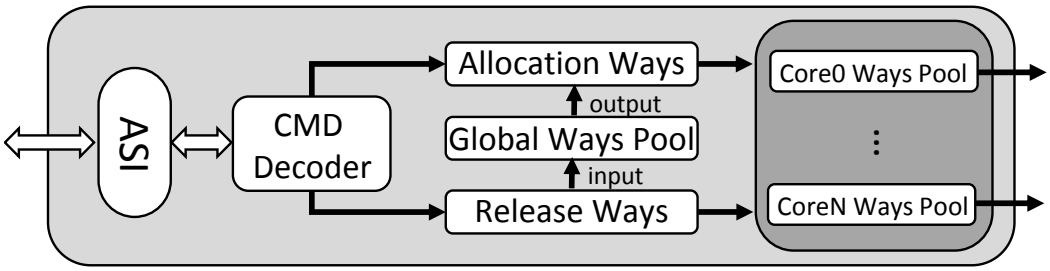


Figure 2.6: Cache ways managment unit (CWMU).

The cache architecture with write-through policy has been adopted in many real-life high-performance processors such as Niagara processor [64], IBM POWER5 processor [65], and Itanium processor [66].

Fig. 2.7 depicts the block diagram of *cache controller*. Transactions from NIOS cores are injected through the cache ports, which is instantiated as *avalon slave interface (ASI)*. Evictions, refills and write-through are asserted from off-chip memory port, which is instantiated as *avalon master interface (AMI)*. The data-width of both *ASI* and *AMI* in our case is 32 bit. The supported maximum burst of both ports depends on the cache line size. Thus, muxs and demuxs in *ASI* and *AMI* are used to packet and de-packet bytes in the corresponding cache line size. The control logic performs hit/miss check, returns the read data, and asserts evictions and refills. The victim cache line is selected by the block reference field logic (BRFL) during the refill phase. The implementation of the partitioned replacement policy is presented in Section 2.3.5.

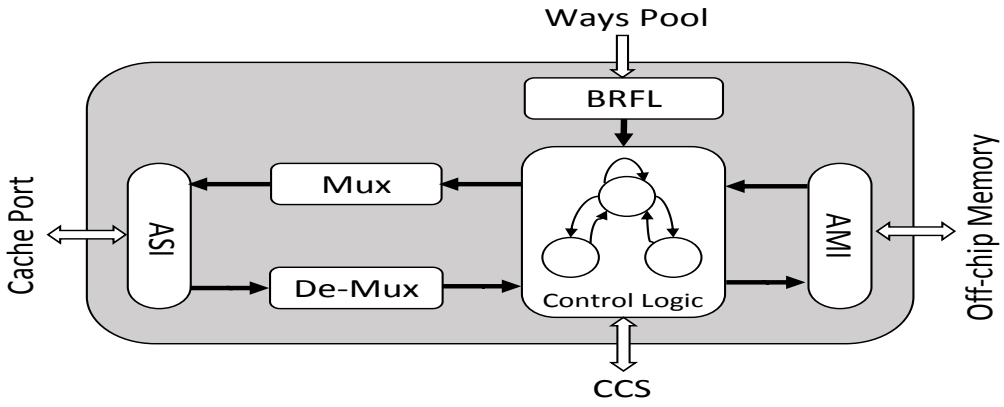


Figure 2.7: Cache controller (CC).

2.3.5 Implementation of Partitioned FIFO Replacement Policy

When a new data must be stored in a cache memory and all cache ways have been occupied, one of the existing cache line must be selected for replacement. Standard replacement policies include LRU, FIFO, etc. As the cache with the FIFO replacement policy could support accurate quantitative WCET estimations [67] and prevent timing anomalies [68] for the real-time applications, we consider FIFO cache replacement policy in our design. In addition, the FIFO replacement policy has been widely used in the state-of-the-art processors such as ARM 11 processor and Intel X86 processor [67].

As mentioned in Section 2.3.1, dynamic cache partitioning may result in that cache ways occupied by one core might not be adjacent to each other. To maintain the discontinuous cache ways distribution, the block reference field logic (BRFL), as shown in Fig. 2.8, is proposed to perform victim selection for cache write operations. The reference field contains selection reference memory (SRM) and valid bits memory (VBM). The references of the next selection of victim cache lines are stored in the selection reference memory (SRM). SRM can be instantiated by one FPGA dual port memory block with the depth Q and width $Log_2(u)$, where Q and u denote cache depth and cache associativity, respectively. When the core release ways, all the contents of SRM should be cleared to initial reference in one clock. Unfortunately, no FPGA can support this feature. In this chapter, we propose one solution to reset SRM by using VBM, which can be instantiated as Q-bit register and be cleared in one clock. By using this similar approach, the cache ways can be flushed in one clock when the core release the ways. We use one bit valid register to associate with each reference in SRM. When we read a reference from one location of SRM, the valid bit register acts as a toggle to determine the output. Based on the current reference, the write control logic (WCL) updates the write data for reference field on each cache write operation and write the next selection to reference field of SRM and VBM, making that ways are selected in FIFO replacement manner. Note that write control logic (WCL) can also be easily extended for other replacement policies, e.g., LRU. BRFL outputs a valid reference and the victim can be referred from the ways pool.

2.4 Cache Generation

The multi-core system executing different applications has different demand requirements for the cache memory. Choosing one best cache for the system is, therefore, quite application-specific. However, this selection process is usually manually and subjectively done by user,

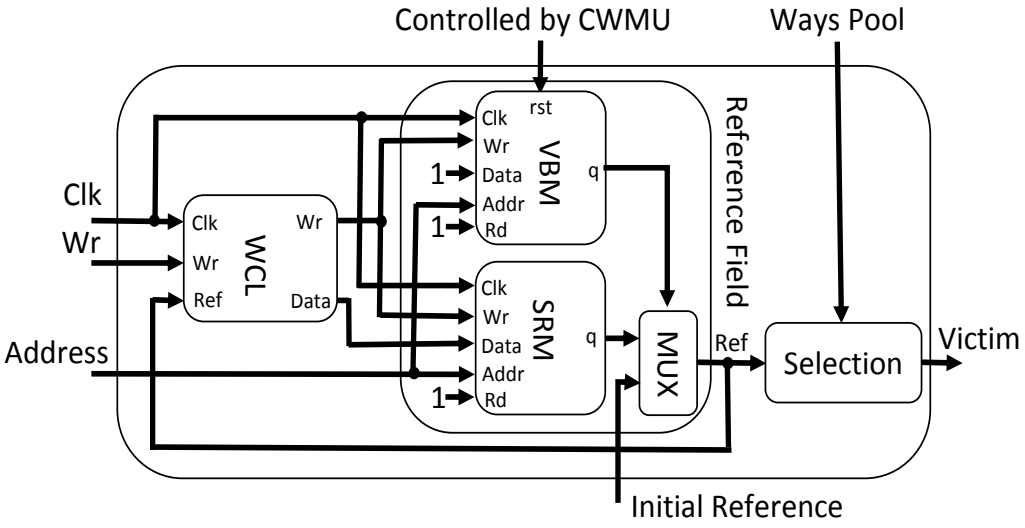


Figure 2.8: Block reference field logic (BRFL).

which will lead to pessimistic cache settings for the system. In order to find one better cache settings for a system, one requires more precise information of the cache performance under different cache settings, such as chip area, power consumption, implementation frequency and etc. Therefore, the feature of automatic hardware generation for our proposed dynamic partitioned cache is required to facilitate collecting such information. The cache generator feature allows us to automatically generate the dynamic partitioned cache memory by inputting the specific cache parameters, such as the number of cache port, the depth of cache bank, line size, and associativity of the cache memory. Besides, the cache generator feature also allows us to evaluate the efficiency of the cache resource management schemes in different cache partitioning scales.

To enable such cache generator features, we have a macro definition file to represent the configurable cache parameters. According to the settings in the macro definition file, the Verilog HDL code can be automatically organized in various code structures by using *generate* statement and *for* loop statement. The bit width of the signals and registers in all hardware module are all defined by using macro definition variable. The register maps are also re-organized according to macro definition variable. Each module in the dynamic partitioned cache memory is designed in a parameterized manner. Therefore, the whole dynamic partitioned cache memory can be automatically generated by setting the parameters in the macro definition file.

2.5 Software Programming Interface

This section discusses the application programming interfaces (APIs) for dynamic partitioned cache memory. As shown in Fig. 2.5, cache ways are managed in a centralized manner to guarantee atomic operation via the *cache ways management unit (CWMU). CWMU* exposes several registers, which can be accessed by cores, to control the cache ways configuration. Each core is connected to *CWMU* by a shared bus, by which cores can access these registers to dynamically regulate its cache ways. Each core controls the dynamic partitioned cache memory by following registers:

- Setting the number of cache ways need to be allocated or released by writing the **CONTROL** registers.
- Getting the mask bits of cache ways occupied by the core by reading the **MASK** registers.
- Getting the value of cache hit counter for the core by reading the **HITC** registers.
- Getting the value of cache miss counter for the core by reading the **MISSC** registers.
- Clearing cache performance (hit and miss) counter for the core by writing the clear bit in the **CONTROL** registers.

Currently the dynamic partitioned cache memory works with 32-bit systems. Therefore, all embedded processor-accessible registers are 32 bits wide. The i-th core can only access its own 4 registers: $\mathbf{CONTROL}_i$, $\mathbf{MAST}_i$, $\mathbf{HITC}_i$, $\mathbf{MISSC}_i$. Thus, the address space of total registers in dynamic partitioned cache memory is $4 \cdot m$, where m is the number of the cores in the multi-core system. The registers address space of each core can be distinguished by the following address mapping scheme, as shown in Fig. 2.9. The last 2 bits of the physical address is used as word offset for 32-bit wide register. The 2nd and 3rd bits the physical address are used to index the four 32-bit registers. The upper $log_2(m)$ bits of the physical address are used to distinguish the address space of the cores, which can be considered as core ID. For example, for 4 core system, the 4th and 5th bits is used as core ID. By using these two address bits, the *cache ways management unit (CWMU)* can distinguish which core sends the configuration command and conduct the cache configuration for the distinguished core.

Tab. 2.1 presents the inner register map for the case of 16-ways associative dynamic partitioned cache memory. The **CONTROL** register contains individual bits, set by the core, which controls the cache management operation and the performance counter operation. If Rel

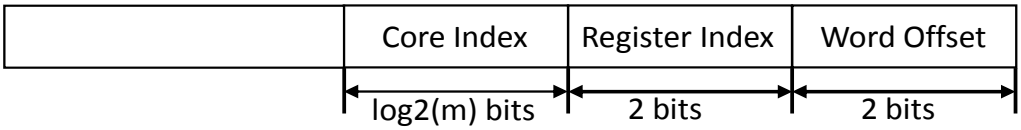


Figure 2.9: Address mapping.

bit (or Alloc bit) is set as 1, *CWMU* releases (or allocates) cache ways, while the number of cache ways is represented by Way# bits (bit 3:0). When Clc bit is set as 1, cache performance counters **HITC** and **MISSC** is cleared. Besides, the **CONTROL** register contains 12 bits (bits 31:20) to return the state of runtime cache management. The ErrorID bits (bits 31:28), AvaWays# (bits 27:24), OccuWays# (bits 23:20) in the **CONTROL** register represent error code for cache management, the available cache ways number of global ways pool in *CWMU*, and the occupied cache ways number of the current core, respectively. When the core read the **CONTROL** register, the status of dynamic partitioned cache memory can be obtained for debugging. The **MASK** register is used to indicate which exact ways are be occupied by cores. Besides, we also develop one customized performance counter to evaluate the performance behavior of the cache. The customized performance counter is integrated into the proposed cache. The users can obtain the performance of the cache by reading the performance counter registers **HITC** and **MISSC**.

Table 2.1: Inner register map for the case of 16-ways associative cache.

A3 to A2	Register Name	31:28	27:24	23:20	·····	6	5	4	3:0
0	CONTROL	ErrorID	AvaWays#	OccuWays#		Clc	Rel	Alloc	Way#
1	MASK	The Maks of the occupied cache ways							
2	HITC	Cache hit performance counter							
3	MISSC	Cache Miss performance counter							

We develop a set of APIs to facilitate control of the dynamic partitioned cache memory at a reasonable level of abstraction. Tab. 2.2 lists all the atomic APIs currently supported by reconfigurable cache IP. By calling the following APIs, cache ways can be dynamically tuned and performance statistic data can be obtained from the customized performance counter.

Table 2.2: APIs supported by reconfigurable cache

allo_ways(way_num)	Allocate cache ways to cores
rel_ways(way_num)	Release cache ways from cores
clc_perf_cnt()	Clear the performance counter
get_hit_cnt()	Get the value of cache hit counter
get_miss_cnt()	Get the value of cache miss counter
get_state()	Return occupied ways, ways# in pool, error state

2.6 Hardware Prototype and Verfication

In this section, we present the experimental results obtained with an implementation of the prototype of the proposed dynamic partitioned cache memory. At first, we summarize the hardware characterization to evaluate the effectiveness of our cache hardware prototype. To obtain the hardware characterization on FPGA platforms, different types of caches are synthesized to Altera Stratix V FPGA with Quartus II (version 13.0). Besides, we also investigate the chip design process for the proposed dynamic partitioned cache memory by using Synopsys design compilers [69]. The design flow is based on the SMIC 130nm standard technology library [70]. Finally, we report the experimental results to verify the functionality of the proposed dynamic partitioned cache memory. The results are obtained by implementing the constructed NIOS-based multi-core system together with the proposed reconfigurable cache on the Altera DE5 development board equipped with Stratix V FPGA.

2.6.1 FPGA Synthesis Results

First of all, we compare the different types of caches with respect to their maximum operating frequency and resource consumption in terms of logic and memory usage. Different types of caches are synthesized to Altera Stratix V FPGA with Quartus II (version 13.0) to obtain area and critical path delay (maximum operating frequency F_{max}) numbers. The effect of increased cache depth, associativity, line size, and port number will be examined for all cache types. Tab. 2.3 summarizes the results for different types of caches. The 'cache settings' column is organized as form of *associativity/depth/line size*. For example, 4/128/256 indicates 4-ways cache architecture with 128 cache depth and 256-bit line size. F_{max} indicates the maximum frequency that the constructed multi-core system can run on.

For increase in depth address and ways number, the number of combinational ALUTs and registers also increases. As explained in Section 2.3.5, to flush cache ways and reset the re-

Table 2.3: Speed and resource consumption on Stratix V FPGA

Port Number	Cache Settings	Combinational ALUTs	Total Registers	$\mathbf{F_{max}}$ (MHz)
Two Core	4/256/256	11510	8899	168.41
	4/512/256	14453	11461	159.41
	8/256/256	17619	10506	151.10
	8/512/256	21609	14604	152.14
Four Core	8/256/256	29809	18683	140.29
	8/512/256	36074	24831	134.34
	16/256/256	39821	22014	126.90
	16/512/256	49225	31234	125.83

placement reference in one cycle, we separate the valid bit of each line from memory block and implement it in customized memory block which supports clearing contents globally. Thus, the increment of address depth will result in the increment of the number of valid bit, which leads to more logic resource in combinational ALUTs and registers. Regarding the ways number, the contributing factors are the core-cache-switch circuitry, FIFO replacement policy circuitry, and wide logical OR, all of which grow with the increased ways number. Regarding the maximum operating frequency F_{max}, we notice that 2-core cache is faster than 4-core cache and the cache architecture with less associativity is faster than the one with more associativity.

2.6.2 Physical Chip Synthesis Results

In this section, we report physical chip synthesis results for the proposed dynamic partitioned cache memory. The proposed cache memory is implemented in synthesizable Verilog HDL code and synthesized by using Synopsys design compilers [69] with the SMIC 130nm standard technology library [70]. We use ARM Artisan 130nm memory IPs [71] to generate RAM blocks for our cache. Considering that the proposed cache memory supports way-based dynamic cache partitioning, we mainly focus on studying how the cache way numbers impact chip design process in terms of chip area and power consumption. We conduct the experiments to report the chip area and power consumption of the proposed cache memory under different configurations. In the experiment, we implemented 4 different configurations for the dual-core caches memory, where the cache way numbers are varied from 4 to 32. The cache depths and cache lines are fixed as 1024 and 128, respectively. For comparison, a standard shared cache without dynamic partitioning functionality is also developed and verified by using the same experiment setups.

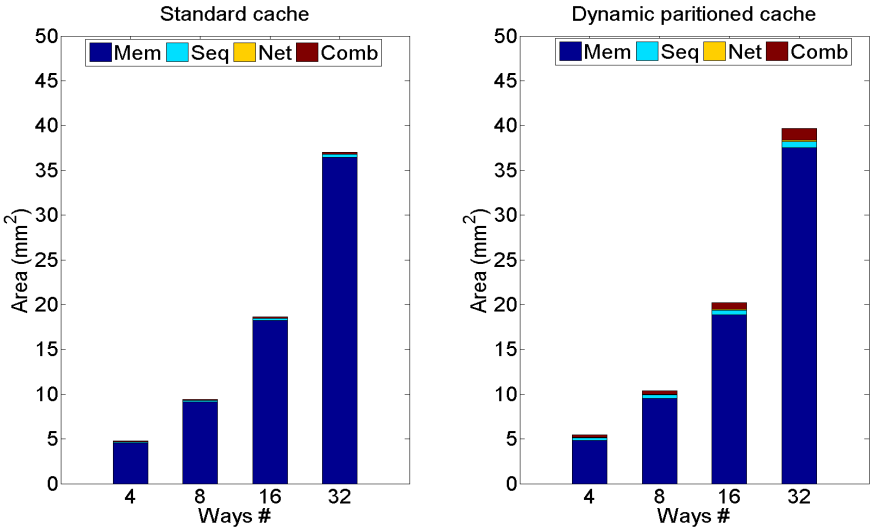


Figure 2.10: Chip area for dual-core caches with the varying cache way numbers.

Considering chip manufacturing technology we used (i.e., 130nm technology), we restrict the frequency of all cache designs at 400MHz and report the chip area and power consumption under this speed level.

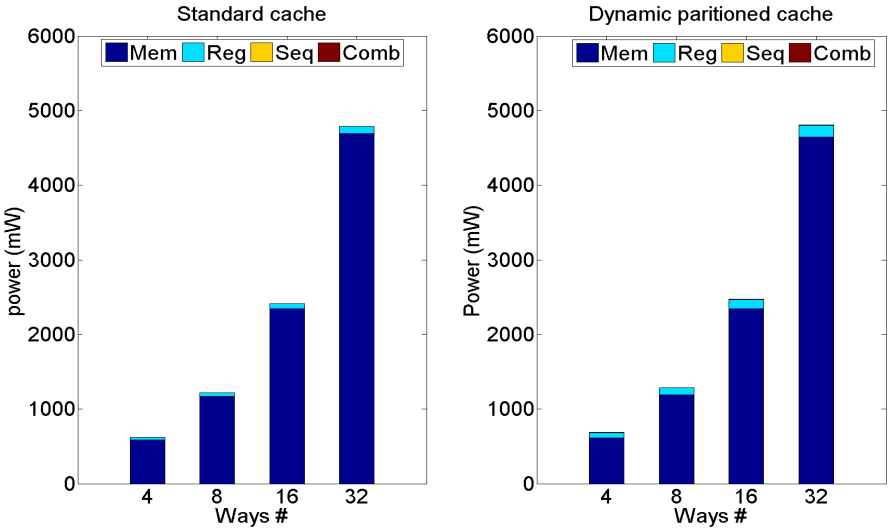


Figure 2.11: Power consumption for dual-core caches with the varying cache way numbers.

Fig. 2.10 and Fig. 2.11 illustrate the chip area and power consumption for different types of caches, respectively. As shown in Fig. 2.10, the chip density is mainly contributed by the

memory blocks in both cache architectures because the cache is mainly composed by the memory blocks. Comparing to pure cache without dynamic cache partitioning, the total density overhead of our cache implementation ranges from 7% to 13% and maily comes from memory and combinational blocks. This density overhead is introduced with the addition of selection reference memory (SRM) in FIFO replacement policy circuitry and the routing logic in core-cache-switch circuitry. Another important observation is that the chip area is nearly increased linearly with the ways configurations. The cache with 32-ways configuration occupies 7X chip area than the cache with 4-ways configuration. Fig. 2.11 depicts the power consumption for both cache architectures under the different cache ways configurations. The main power overhead is caused by the increase of registers for cache controller. The power overhead of our cache design ranges from 0.3% to 10%. Thus, our cache design has a close power consumptions with respect to the standard cache design. Besides, the more cache ways we configure, the more power the cache memory will consume. From the results, we can see that reducing one more cache ways can on average reduce 148 mW power consumption. This means turning off cache ways can significantly reduce the power consumption of the system. This brings another potential research direction about how to dynamically manage the cache ways resource to achieve energy efficiency for the cache subsystem.

2.6.3 Functionality Verification

We implemented a functional test to verify the correctness of the reconfigurable cache prototype implementation. This verification is based on memory reuse code, as shown in Fig. 2.12, which can mimic the behavior of cache access behavior. According to the test presented in Fig. 2.12, the program firstly access the array $b[Cache\_Depth * Ways\_Num][Line\_Size]$, whose size equals the predefined cache, in the first *for* loop. The parameter $Cache\_Depth$, $Ways\_Num$, and $Line\_Size$ are denoted as the cache depth, cache way number, and the word number of cache line, respectively. After the first loop, the assigned N-ways cache ($N < Ways\_Num$) will remain the last visited $N \times Cache\_Depth \times Line\_Size$ array data elements. For example, if we assign one cache way to this functional test program, this one-way assigned cache will be occupied by the array data elements from $b[Cache\_Depth * (Ways\_Num - 1)][0]$ to $b[Cache\_Depth * Ways\_Num - 1][Line\_Size - 1]$. In the second *while* loop, the array $b[Cache\_Depth * Ways\_Num][Line\_Size]$ is revisited in the reverse order for the sake of cache reuse. The more the cache is assigned, the more cache reuse can be achieved which in turn can lead to less cache miss.

```
1  unsigned int b[Cache_Depth*Ways_Num][Line_Size];
2  unsigned int i,temp;
3  // Load data into cache
4  for(i=0;i<Cache_Depth*Ways_Num;i++){
5   temp=b[i][0];
6  }
7  //start to reuse cache
8  while(i>0){
9   temp=b[i][0];
10  i--;
11 }
```

Figure 2.12: The code for functionality verification.

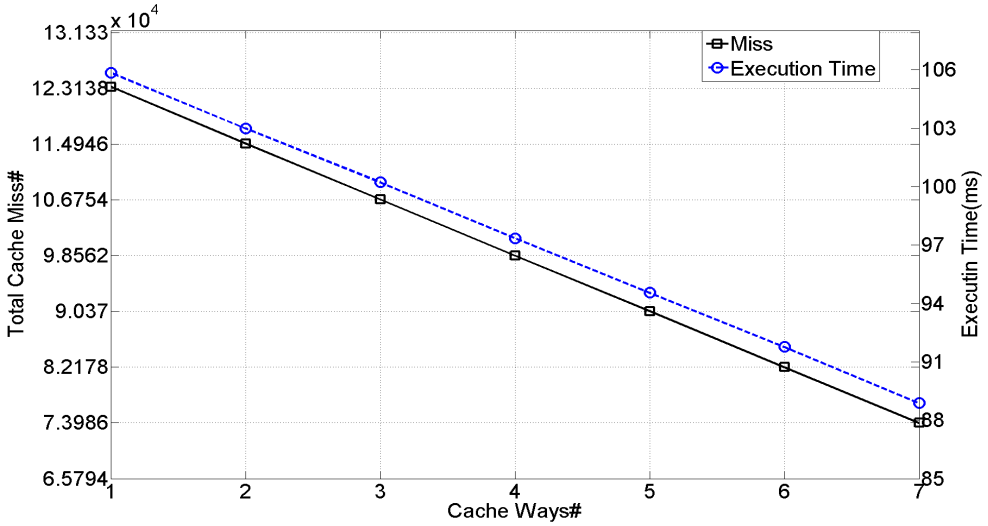


Figure 2.13: # Cache miss and execution time for memory reuse code.

This functional test is conducted on the two-core system with 2MB reconfigurable shared L2 cache (8 ways, 8192 cache depth, 256 bit line size), which is implemented on the Altera DE5 development board equipped with Statrix V FPGA. By calling cache reconfiguration listed in Tab. 2.2, we implement memory reuse code under different cache ways. Fig. 2.13 shows cache miss numbers and execution times under different cache ways. We can see that both cache miss numbers and execution times predictably decrease linearly with reconfigured cache ways. By increasing one way, cache miss numbers decrease linearly with step 8192 (i.e., cache depth). This is expected since 8192 more cache lines are buffered for memory reuse when increasing one way.

Lets give a quantitative analysis to this result. According to the test in Fig. 2.12, each cache access in the first *for* loop always result in cache miss. Thus, there should be roughly 8192×8 cache misses to happen during the data load phase (i.e., the first *for* loop in Line 4-6). According to the analysis we state above, only $N \times 8192$ cache lines can be reused during the cache reuse phase (i.e., the second *while* loop in Line 8-11). Thus, we will roughly get another $(8 - N) \times 8192$ cache misses during the cache reuse phase. Totally, we are expected to roughly get $(16 - N) \times 8192$ cache misses if we assign N cache ways to this test program. From this analysis, we can see the cache miss number should decrease linearly with reconfigured cache ways. It is worthy noting that our cache works as a unified shared cache in the experiment setup. Instruction access will also result in additional cache miss numbers. Thus, the above cache miss number is a rough number which do not take instruction access into account. To eliminate the impact of the cache miss caused by instruction access, we use the array with the large size (2M byte) in the test program and set our cache with the large size in this experiment to relieve the impact of instruction access and make our verification more accurate. By these settings, the cache miss caused by instruction access can be ignored comparing to the cache miss caused by data access. From the result as shown in Fig. 2.13, we can see that cache miss numbers are expected to decrease linearly with reconfigured cache ways. Fig. 2.14 shows the cache miss difference rate between the observed cache miss and the expected cache miss. From Fig. 2.14, we can see that the cache miss differences between the expectation and measurement, which is caused by instruction access, are quite small and the maximum cache miss difference normalized with respect to the expected cache miss is up to 0.39%. This means our cache works as the expected manner and the reconfiguration functionality of the designed cache is correct.

2.6.4 Reconfiguration Overhead Measurement

Finally, we conduct experiments to measure the timing overhead for cache reconfiguration operations. According to Section 2.3.3, the port of cache ways management unit (CWMU) is shared by all cores. To inject traffic on the shared bus, we implement allocation and release cache configuration instructions in Infinite loop concurrently on the interference core. To measure the timing overhead, allocation and release cache configuration instructions are implemented for 10000 times on the target core. In each iteration, we implement allocation-release cache configuration instruction pair to avoid the cache overflow. And we directly read the time stamp counter and report the average latency as the timing overhead of allocation-release instruction pair. According to our experiment, the average timing overhead of one allocation-release cache

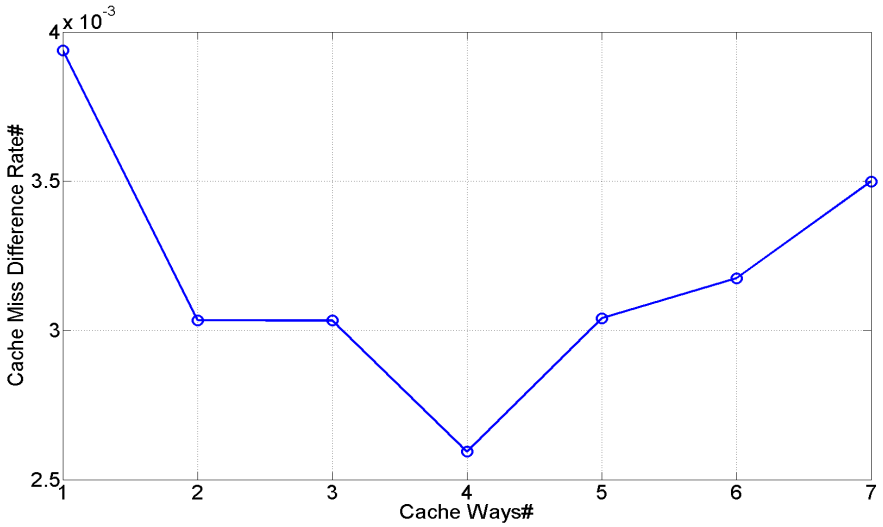


Figure 2.14: The difference rate between the observed cache miss and the expected cache miss.

configuration instruction pair is 16 cycles, which is ignorable when comparing to OS-based cache partitioning.

2.7 Summary

In this chapter, we present the dynamic partitioned shared cache memory to achieve the time-predicable execution of shared cache subsystem in multi-core systems. By using the proposed cache, the cache resource can be strictly isolated to prevent the cache interference among cores. Furthermore, the proposed cache supports dynamic cache partitioning and allows cores to dynamically allocate cache resource according to the demand of applications, which will enable us to efficiently use cache resource and maximize the performance of applications. The proposed cache can be interfaced to multi-core embedded systems such as Altera NIOS-based multi-core systems, and this has been tested in Altera FPGAs. The FPGA implementation shows a reasonable implications for the clock frequency and hardware resource consumption. Besides, we also investigate the chip design process for the proposed cache memory. The synthesis result shows the power consumption and chip area of the proposed cache is scalable with respect to the cache way numbers. The usage of the proposed cache within a real C code has been examined by a functionality test. By using a memory reuse code which can mimic the behavior of cache access, the functional test can verify the correctness of the reconfigurable cache

prototype implementation. Finally, we also conduct experiments to show our cache memory can be reconfigured with small overhead (scaling to cycles).

The existence of our FPGA prototype of the proposed cache provides one practical way to implement dynamic cache partitioning on the multi-core system. In contrast to most previous work [35–40] in the literature, which are devoted to analyze theoretical proposals and the simulation of reconfigurable caches, we provide one physically implementation and prototype on FPGA. This prototype will bridge the gap between simulation and real systems, and will serve us a real (not simulation) reconfigurable cache for studying and validating cache management strategies on the real-time multi-core system.

In the next chapter, we will present an integrated cache management framework, which combines real-time task scheduling and task-level cache partitioning to improve the execution predictability and performance for real-time multi-core systems. In this framework, the dynamic partitioned shared cache memory is used to implement task-level cache partitioning.

Chapter 3

Shared Cache Management Framework for Real-time Multicore Systems

In the previous chapter, we have introduced the details of the hardware design of the dynamic partitioned shared cache memory and its prototype on the FPGA platform. This shard cache can guarantee the cache resource is strictly isolated among real-time applications to prevent the cache interference. This chapter mainly focuses on studying how to manage the dynamic partitioned shared cache memory in an efficient manner while still guaranteeing the real-time constraints. Several challenging tasks are involved in this phase in particular for the multi-core implementation of the predictable scheduler and the dynamic configuration of the cache subsystem. To resolve these issues, we proposed an integrated cache management framework to improve the execution predictability for real-time multi-core systems. The proposed framework can automatically generate fully deterministic schedule and cache configurations for system performance optimization with real-time constraints. This chapter discusses the design of this automatic cache management framework in details.

3.1 Introduction

Multi-core systems have become one of the preferable choices in modern embedded systems to achieve more powerful computing ability while reducing the cost of the system at the same time. Safe-critical real-time embedded systems, such as electronic vehicles [72], are also one of the promising domains which use the multi-core systems as their computing platforms, because

there are growing number of applications which require more powerful computing platform to provide more computing power. For example, the driver assistant system in the modern automotive system needs to process high-resolution video in real-time to track objects, which requires significant computing power [73].

The multi-core architecture, however, poses a significant challenge in designing a safe-critical real-time embedded system due to timing unpredictability caused by shared resource interference. The shared cache has been recognized as one of the most important sources of unpredictability in multi-core systems [17, 49, 59]. To resolve such unpredictability stemmed from the shared cache, real-time systems usually are developed by either disabling or statically partitioning shared cache [33–35]. Another major challenge is that no real-time scheduling policy taking cache space demands into account is established [23]. How to choose cache partition size integrated with task scheduling design to optimize system performance while guaranteeing the system predictability is still an open problem [23]. In this chapter, we study the problem of how to use the shared cache in a predictable and efficient manner under real-time requirements with the existence of cache interference.

Most of the state-of-art techniques [33–35] on this topic consider statically partitioning cache at core level. The shared cache is statically partitioned to individual cores and all tasks mapped on the same core should use this pre-partitioned cache. However, designating a region with a constant size to individual cores is often ineffective with respect to the system performance, since the tasks assigned on the same core might have different requirements and sensitivities to the amount of cache allocated. We argue that by carefully designing a task schedule and reconfiguring the cache partitioning for each task according to the schedule at runtime, the performance of the system can be improved, compared to the core-level strategies. A motivation example will further elaborate this issue in Section 3.4. In contrast to core-based cache partitioning, we consider task-level cache partitioning, which enables us to allocate cache resource in an efficient manner according to different features of tasks.

Combining real-time task scheduling and task-level cache partitioning allocation is however more involved. On the one hand, the WCET of a task depends on the allocated cache size. On the other hand, the maximal cache budget that can be assigned to a task depends on the cache sizes occupied by other tasks that are currently running on other cores, i.e., depending on the scheduler. Furthermore, the performance (e.g., cache miss, energy consumption, and execution time) of running tasks may have different requirements to the amount of used cache because memory access pasterns of tasks vary greatly from task to task. In principle, the task

scheduling and the cache size allocation interrelate to each other with respect to the system performance, such as cache misses [28] and energy consumption [35]. Therefore, a sophisticated framework is needed to find the best trade-off between them in order to improve the system performance [35].

In this chapter, we tackle schedule-aware cache management scheme for real-time multi-core systems. We present an integrated framework to study and verify the interactions between the task scheduling and the shared cache interference. For a given set of tasks and a mapping of the tasks on a multi-core system, our approach can generate a fully deterministic time-triggered non-preemptive schedule and a set of cache configurations during the compilation time. During runtime, the cache is reconfigured by the scheduler according to the offline computed configurations. The generated schedule and the cache configurations together minimize the cache miss of the cache subsystem while preventing deadline misses and cache overflow. With a customized reconfigurable cache component and share-clock multi-port timer component, our framework can generate multi-core systems with different cache modules (different cache configurations with respect to cache lines, size, and associativity) and prototype on Altera FPGA. Finally, we analyze and discuss the experiment results under different hardware environments with respect to the number of cores and cache settings. A case study is presented to demonstrate the completeness of our framework.

The rest of the chapter is organized as follows: Section 3.2 reviews related work in the literature. Section 3.3 presents some background principles. Section 3.4 presents the motivation example for schedule-aware cache management scheme. Section 3.5 overviews the proposed framework and Section 3.6 describes the proposed synthesis approach for scheduling and cache management. Section 3.7 illustrates the time-triggered scheduling implementation on multi-core system. Section 3.7 describes the detailed process of the automatic generation. Experimental evaluation is presented in Section 3.9. The case study aiming at demonstrating the completeness of our framework is presented in Section 3.10. The usability for our cache management framework is discussed in Section 3.11. Section 3.12 summarizes this chapter.

3.2 Related Work

3.2.1 Cache Partitioning

Shared cache interference in a multi-core system has been recognized as one of major factors that degrade the average performance [28, 29], as well as predictability of a system [18, 24].

3. SHARED CACHE MANAGEMENT FRAMEWORK FOR REAL-TIME MULTICORE SYSTEMS

Many work has been done in general-purpose computing to optimize different performance objectives by cleverly partitioning shared cache, including cache performance [74, 75] and energy consumption [40]. Sundararajan et al. [76] proposed an energy-saving technique for the last-level-cache (LLC) multi-core processors, where the cache are partitioned into shared and private regions. The dynamic energy savings can be achieved by privately access to the ways which contains useful data. By using curve fitting to dynamically partition the storage cache, Patrick et al. [77] presented an effective shared storage cache management scheme to provide differentiated services to applications. However, most existing studies are evaluated by simulation. The simulation-based study may only simulate a few billion instructions for a program. Besides, evaluations on simulators are prone to inaccuracy [29]. Instead of using simulation, Lin et al. [29] evaluated a dynamic cache partitioning scheme on an Intel 5160 processor based on page coloring technique. Above work [28, 29, 40, 74, 75] is mainly focused on improving system performance and do not consider real-time requirements.

Cache partitioning techniques have also been actively studied to improve the performance of real-time embedded systems. Bui et al. [78] exploited cache partitioning to minimize the task real-time utilization while taking into account the tasks' criticality. The results in [78] show that cache partitioning can be used to improve system schedulability because cache partitioning can help to reduce the interference. By leveraging configurable cache architectures, the authors in [79] proposed a technique to eliminate inter-task cache interference and reduce cache energy consumption. Wang et al. [32] proposed a profile-based scheduling-aware dynamic cache reconfiguration technique to reduce the cache energy consumption for soft real-time systems. Unfortunately, above simulation-based studies do not consider the multi-core platforms.

Few work in the literature has been done in the context of real-time multi-core system. Given a task-level cache partitioning, the authors in [24] developed a sufficient schedulability test for non-preemptive fixed priority scheduling for multi-core systems. However, the work does not consider how to partition the cache size to individual tasks. How to choose cache partition size to optimize system performance while guaranteeing the system predictability is still an open problem [23]. Liu et al. [34] proposed a joint task assignment and cache partitioning technique to minimize the overall WCET, where the shared cache is partitioned on core level and cache locking is applied to guarantee a precise WCET. The authors in [33] proposed a two-level utilization control solution for energy optimization in real-time multi-core systems, in which the cache assigned to a task is upper-bounded by the cache quota of the core. Wang et al. [35] proposed an approach to optimize the energy of the cache subsystem for multicore

systems. In this work, they dynamically reconfigure the private L1-cache on task-level and statically partition the shared L2-cache to cores. However, Most of the work [33–35] considers static cache partitioning on core level. This kind of static cache partitioning scheme cannot fully exploit the different cache demand features of the tasks and, therefore, will result in inefficient usage of the cache resource. Besides, all above research work is evaluated by simulation.

In the context of practical real-time systems, cache partitioning techniques have been explored mostly by using software-based solution [18,31,55,80,81]. In [80,81], the off-chip memory mapping of the tasks is altered to guarantee the spatial isolation in the cache by using compiler technology. However, altering tasks' mapping in the off-chip memory is far from trivial, which requires significant modifications of the compilation tool chain. In addition, the partitioning of the tasks can only be statically suppressed in fixed cache set regions due to the pre-decided memory mapping, which also prevents the efficient usage of the limited cache resource. Recently, the techniques [18,31,55] on the multi-core cache management in the context of real-time systems have been proposed by using page-coloring, which partitions the cache by sets at OS-level. However, page-coloring based techniques usually suffer from a significant timing overhead inherent to changing the color of a page, which results in that making decision of changing the color of a page cannot be frequent. The authors in [29] report that the observed overhead of page-coloring based dynamic cache partitioning reaches 7% of the total execution time even after conducting the optimization to reduce the recoloring times. Besides, the page-coloring techniques [18, 31, 55] partition the cache by sets at OS-level. Cooperating OS overhead also needs to be carefully considered in real-time systems. Distinct to above cache management schemes, we present one cache management framework to improve the system predictability and performance for the proposed FPGA-based time-triggered multi-core system, where a reconfigurable cache architecture can execute dynamic way-based cache partitioning in hardware level. Our approach can dynamically change the cache size with minimal overhead (scaling to cycles). Besides, compared to set-based cache partitioning techniques, our way-based reconfigurable cache can turn off the whole unused ways to save static energy [2, 36]. Therefore, our way-based reconfigurable cache can also bring benefits for low-power design. Based on the dynamic partitioned cache memory presented in Chapter 2, the proposed cache management framework can dynamically partition the cache resource on task-level to improve cache usage.

3.2.2 Time-triggered Scheduling

Time-triggered execution models can offer a fully deterministic real-time behavior for safety-critical systems. Current practice in many safety-critical system domains, such as electric vehicles [72] and avionics systems [82], favors a time-triggered approach [83]. Sagstetter et al. [84] presented a schedule integration framework for time-triggered distributed systems tailored to the automotive domain. The proposed framework uses two-step approach, where a local schedule is computed first for each cluster and the local schedules are then merged to the global schedule, to compute the schedule for the entire FlexRay network and task schedule on ECUs. To optimize the control performance of distributed time-triggered automotive systems, Goswami et al. [85] presented an automatic schedule synthesis framework, which generates time-triggered scheduling for tasks on processor and messages on bus. Nghiem et al. [86,87] presented an implementation of PID controller using time-triggered scheduling paradigm and showed the effectiveness of such time-triggered implementation. Based on time-triggered scheduling, Jia et al. in [88] presented an approach to compute message scheduling based on Satisfiability Modulo Theories (SMT) for Time-Triggered Network-on-Chip. All above techniques are evaluated by simulation. In [89], Ayman et al. describe a two-stage search technique which is intended to support the configuration of time-triggered schedulers for single-processor embedded systems. However, none of them apply time-triggered scheduling and cache management jointly on real-time multi-core platform in order to achieve timing predictability and system performance.

3.3 Background

3.3.1 Way-based Cache Partitioning

Our cache management scheme implements way-based cache partitioning, where each cache set is partitioned in the granularity of ways. In this work, we consider dynamic way-based cache partitioning. Each core is dynamically allocated a group of ways and will only access that portion in all cache sets. Compared to static cache partitioning scheme in [35] which partitions cache ways on core-level, dynamic way-based cache partitioning allows each core can dynamically tune the numbers of ways according to the cache demand of the executed applications. Considering that real-time systems normally should have fully deterministic characteristics, we partition cache ways on task-level. In other words, a fixed number of cache ways is allocated to the application during the application execution. By this way, we can achieve not only deterministic characteristics of the system but also the efficient usage of the limited cache ways. In

this work, we implement cache partitioning on our customized reconfigurable cache component presented in Chapter 2 and dynamically assign cache ways to tasks.

3.3.2 Hardware Platform

In this section, we present one FPGA-based multi-core system which supports dynamic cache partitioning and time-triggered scheduling. A major benefit of choosing FPGA for prototyping our multi-core system is the high configurability of the processor. This allows us to evaluate the proposed integrated scheduling and cache management framework under various hardware configurations with different cache sizes and varied arithmetic units. Fig. 3.1 illustrates the proposed multi-core system on FPGA, where the cache is shared among cores. We adopt the NIOS II core in the system. Modules highlighted with white color in Fig. 3.1 indicate the hardware components specifically designed and implemented for our framework. The system consists of several NIOS II cores along with reconfigurable cache IP which supports dynamic cache partitioning and share-tick timer IP for time-triggered scheduling.

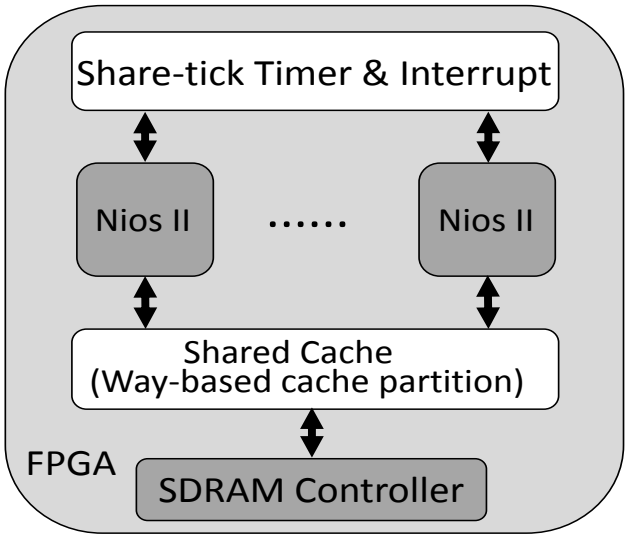


Figure 3.1: System architecture.

3.3.3 Task Model

We consider the functionality of the entire system as a task set $\tau = \{T_1, ..., T_n\}$, which consists of a set of independent periodic tasks. We use w_{ij} to denote the worst case execution time (WCET) of task $T_i \in \tau$ with j ways shared cache allocated and $W_i = \{w_{i1}, w_{i2}, ..., w_{is}\}$ to

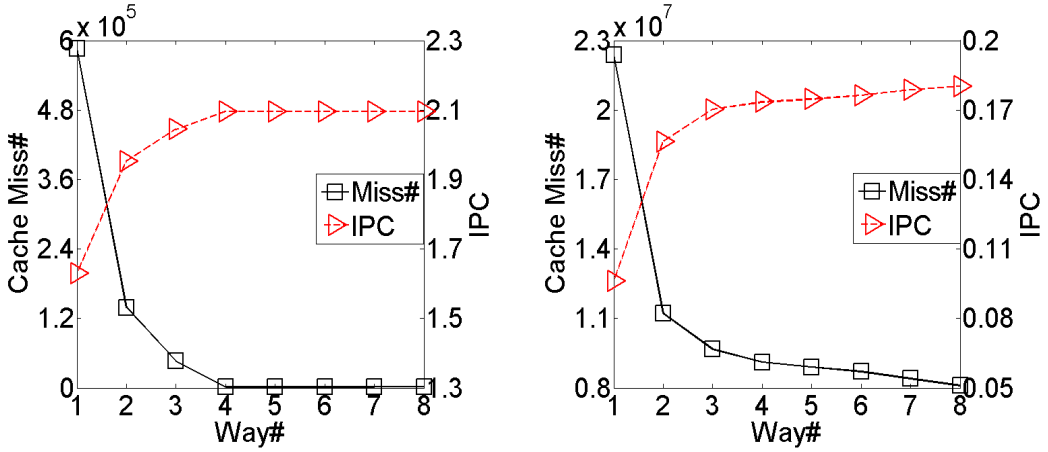
denote the WCET profile of task T_i, where s is the total number of ways in the shared cache (cache capacity). In this chapter, a measurement-based WCET estimate technique is used to determine the worst case execution time.

Timing predictability is highly desirable for safety-related applications. We consider a periodic time-triggered non-preemptive scheduling policy, which can offer a fully deterministic real-time behavior for safety-critical systems. Note that we consider non-preemptive scheduling as it is widely used in industry practice, especially in the case of hard real-time systems [90]. Furthermore, non-preemptive scheduling eliminates the cache-related preemption delays (CPRDs), and thus alleviates the need for complex and pessimistic CRPD estimation methods. We use R to denote the set of the profiles for all tasks in task set τ. A task profile $r_i \in R$ is defined as a tuple $r_i =< W_i, s_i, h_i, d_i >$, where s_i, h_i, d_i are respectively the start time, period, and deadline of the task T_i. The deadline d_i of the task T_i is equal to its period h_i.

3.4 Motivation

Fig. 3.2 shows the number of L2 cache misses and instruction per cycle (IPC) for two benchmark applications (CRC and qsort from MiBench [91] executed in SimpleScalar [92]) under different L2 cache partition ways c. Unallocated L2 cache ways remain idle and cannot be accessed in run-time. We can see that changing cache ways number will result in different numbers of cache performance (i.e., cache miss) and system performance (i.e., IPC). It is expected that the number of L2 cache misses decreases while IPC increases by increasing allocated L2 cache ways. Besides, we can also observe that different applications have different performance behaviors (L2 cache misses and IPC) under the allocated cache ways. For example, the number of L2 cache misses starts to converge for CRC application after $c = 4$ while it becomes almost identical at $c = 8$ for qsort application. It is sufficient to assign 4 cache ways to CRC application for performance improvement. However, keep increasing the cache ways for qsort application can bring benefits in improving the system performance.

Based on above observations, we can see that the difference of performance behaviors among the applications will result in different timing properties. For instance, different cache misses and IPC will lead to different worst case execution time estimations. Therefore, designating a region with a constant size to individual cores is often ineffective with respect to the system performance. In addition, the varied timing properties will further have impacts on the scheduler. There are interesting trade-offs between cache resource allocation and scheduler design that can



(a) CRC (b) qsort

Figure 3.2: Cache impact on the performance of the application.

be explored for optimizations. Therefore, we should explore both dynamic cache configuration and schedule simultaneously to optimize the system performance, while still utilizing the full capacity of the cache.

3.5 Framework Overview

In this section, we give an overview of our system design framework depicted in Fig. 3.3, which takes both real-time scheduling and cache partitioning into consideration to study and verify the interactions between the multi-core real-time scheduling and shared cache management. As shown in Fig. 3.3, the input specifications of the proposed framework consist of the following three parts.

1. *Platform Specification* describes the settings of a multiprocessor platform, such as the number of cores, the settings of the cache memory with respect to cache size, line size and associativity.

2. *Mapping Specification* describes the relation between all tasks in the *task specification* and all cores in the *platform specification*. The mapping specifications can be written by hand or automatically generated by design space exploration tools.

3. *Task Specification* describes task timing requirements such as period and deadline, and task profile information such as the WCETs and cache miss number under different cache

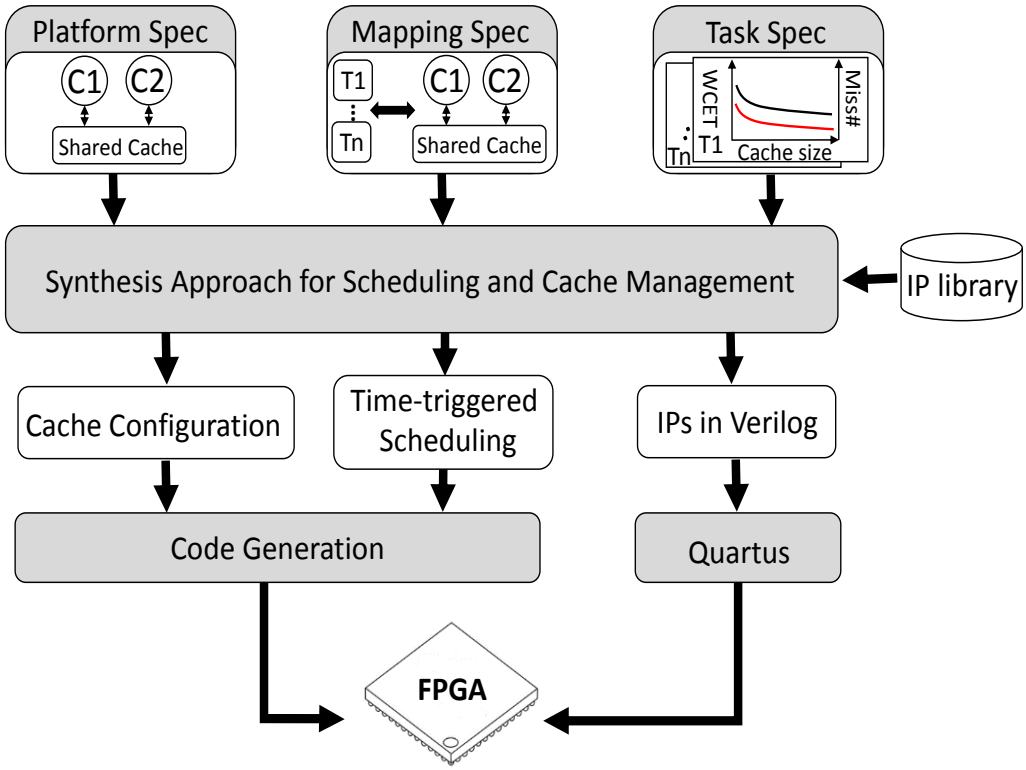


Figure 3.3: System design framework.

sizes.

As output, the synthesis approach can generate cache size allocation and time-triggered scheduling for each task according to the *input specification*, by which the total cache miss number is minimized. Based on this optimal schedule and cache allocation, tasks can be scheduled with insertion of cache size allocation instructions. Task code can be generated by integrating this optimal approach into real-time scheduler. At the same time, parameterized reconfigurable cache IP and share-clock multi-port timer IP can be generated according to the settings in *platform specification*.

3.6 Synthesis Approach for Scheduling and Cache Management

This section presents one synthesis approach for timing schedule and cache management. In this synthesis approach, we formulate the co-design problem of cache partitioning and task

scheduling as integer linear programming (ILP) to minimize the cache miss of the system. With this formulation, the cache size allocation and time-triggered scheduling for each task can be generated automatically, which could avoid deadline miss and cache overflow. We start with an MILP formulation that focuses only on the scheduling problem. Then, the constraints of cache capacity are integrated. Based on the observation that the MILP formulation may suffer from the state explosion, we develop a refinement, the so-called unified resource demand function (URDF) that captures the cache demand of every task and effectively models the interference between tasks, to reduce the exploration space of the formulation.

3.6.1 Time-Triggered Task Scheduling

In the framework, we consider time-triggered non-preemptive schedule. For each task T_i with the profile $< W_i, s_i, h_i, d_i >$, the k-th instance of task T_i starts at $s_i + k \cdot h_i$. W_i contains the WCETs of the task with different cache configurations. We use a set of binary variables c_{ij} to describe the amount of cache allocated to the task T_i: $c_{ij} = 1$ if exactly j cache ways are allocated to T_i and $c_{ij} = 0$ otherwise. In this case, the actual WCET of T_i can be obtained as $\sum_{j=1}^{s} c_{ij} w_{ij}$, where s is the total number of ways of the shared cache. To formulate the scheduling problem by means of MILP, we have to cope with the design constraints of deadlines and non-preemption. We present our formulation as follows.

For deadline constraint, task T_i has to finish no later than its deadline:

$$s_i + \sum_{k=1}^{s} c_{ik} w_{ik} \leq d_i \tag{3.1}$$

The non-preemptive constraint requires that any two tasks mapped to the same core must not overlap in time. Let binary variable denote the execution order of task T_i and T_j: $z_{p\widetilde{p}}^{ij} = 1$ if the i-th instance of task T_p finishes before the start of j-th instance of $T_{\widetilde{p}}$, and 0 otherwise. H_r and $H_{p\widetilde{p}}$ denote the hyper-period of all tasks and the hyper-period of only task T_p and $T_{\widetilde{p}}$ (i.e., LCM of periods of T_p and $T_{\widetilde{p}}$), respectively. $TS(T_p)$ denotes the set of tasks that are mapped to the same core as T_p does. Let ξ denote the overheads for dynamic frequency scaling and task switch. The non-preemption constraint can thereby be expressed as follows.
$\forall T_p, T_{\widetilde{p}} \in TS(T_p), i = 0, ..., (\frac{H_{p\widetilde{p}}}{h_p} - 1), j = 0, ..., (\frac{H_{p\widetilde{p}}}{h_{\widetilde{p}}} - 1)$:

$$i \cdot h_p + s_p + \sum_{k=1}^{s} c_{pk} w_{pk} - (1 - z_{p\widetilde{p}}^{ij}) H_r + \xi \leq j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} \tag{3.2}$$

$$j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} + \sum_{k=1}^{s} c_{\widetilde{p}k} w_{\widetilde{p}k} - z_{p\widetilde{p}}^{ij} H_r + \xi \leq i \cdot h_p + s_p \tag{3.3}$$

The constraints (3.2) and (3.3) ensure that either the instance of T_p runs strictly before the instance of $T_{\widetilde{p}}$, or vice versa.

3.6.2 Cache Partitioning Constraints

The constraints described above guarantee a valid time-triggered schedule. The next step is to add the cache partitioning constraints. The goal here is to guarantee the feasibility of cache partitioning, i.e., at any point in time, the sum of cache ways allocated to the tasks currently being executed does not exceed the cache capacity. In other words, we must avoid *cache overflow*. Before presenting the formulation, we state the following lemma to guarantee the cache never overflows.

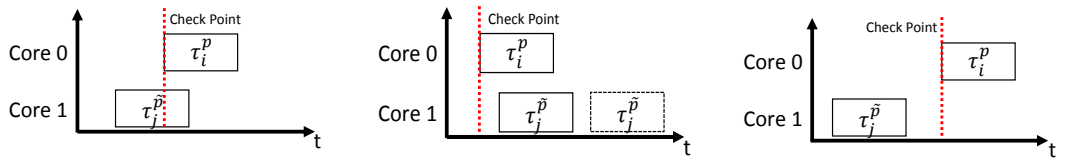
Lem. 1. *If the cache does not overflow at start instant of any task within one hyper-period, the cache never overflows.*

Proof. Note that the amount of cache allocated to a task is constant during its execution interval. It acquires the resources at the start instant and releases the resources at the finish instant. Hence, cache overflow will not occur if the available resources fulfill the requirement of tasks at its beginning. □

From Lem. 1, we know only a finite number of time instants, i.e., at the start of any task, need to be checked for cache overflow. For a specific task T_p, we have to gather all tasks that overlap with it and inspect the total cache demands. Let $T_{\widetilde{p}} \notin TS(T_p)$ be a task running on a different core as T_p. The timing relationship between T_p and $T_{\widetilde{p}}$ can have three possibilities.

- T_p starts during the execution of $T_{\widetilde{p}}$, i.e., the two tasks overlap in time (or in other words $T_{\widetilde{p}}$ *interferes* with T_p), as shown in Fig. 3.4(a).
- Task $T_{\widetilde{p}}$ starts after task T_p starts (Fig. 3.4(b)). In this case, the interference occurs if the start time of $T_{\widetilde{p}}$ is earlier than the finish time of T_p.
- Task $T_{\widetilde{p}}$ ends before task T_p starts (Fig. 3.4(c)), i.e., no overlap in time.

Two binary variables are used to describe the three scenarios above. The variable $x^{ij}_{p\widetilde{p}}$ is 1 if the start time of i-th instance of T_p is later than the start time of the j-th instance of $T_{\widetilde{p}}$, and 0 otherwise. The variable $y^{ij}_{p\widetilde{p}} = 1$ if the start time of i-th instance of T_p is earlier than the finish time of j-th instance of $T_{\widetilde{p}}$ ends. Based the above definitions, we define the following constraints.



(a) scenario 1 (b) scenario 2 (c) scenario 3

Figure 3.4: Timing relationship between two tasks.

$\forall T_p, T_{\widetilde{p}} \notin TS(T_p), i = 0, ..., (\frac{H_r}{h_p} - 1), j = 0, ..., (\frac{H_r}{h_{\widetilde{p}}} - 1)$:

$$j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} - (1 - x_{p\widetilde{p}}^{ij}) H_r \leq i \cdot h_p + s_p \tag{3.4}$$

$$i \cdot h_p + s_p - x_{p\widetilde{p}}^{ij} H_r < j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} \tag{3.5}$$

$$i \cdot h_p + s_p - (1 - y_{p\widetilde{p}}^{ij}) H_r \leq j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} + \sum_{k=1}^{s} c_{\widetilde{p}k} w_{\widetilde{p}k} \tag{3.6}$$

$$j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} + \sum_{k=1}^{s} c_{\widetilde{p}k} w_{\widetilde{p}k} - y_{p\widetilde{p}}^{ij} H_r < i \cdot h_p + s_p \tag{3.7}$$

The constraints (3.4-3.7), cover the interference scenarios in Fig. 3.4, depending on the combination of $x_{p\widetilde{p}}^{ij}$ and $y_{p\widetilde{p}}^{ij}$.

- $x_{p\widetilde{p}}^{ij} = 1$ and $y_{p\widetilde{p}}^{ij} = 1$: (3.5) and (3.7) are trivially satisfied while (3.4) and (3.6) restrict T_p and $T_{\widetilde{p}}$ to be overlapped. This corresponds to the scenario in Fig. 3.4(a).

- $x_{p\widetilde{p}}^{ij} = 0$ and $y_{p\widetilde{p}}^{ij} = 1$: (3.4) and (3.7) are trivially satisfied while (3.5) and (3.6) constraint the execution order of the two tasks to be the scenario in Fig. 3.4(b). Still, as mentioned before, two possibilities could occur, i.e., T_p interferes $T_{\widetilde{p}}$ [1] and T_p does not interfere with $T_{\widetilde{p}}$. The two interference scenarios, that T_p interferes with $T_{\widetilde{p}}$ and $T_{\widetilde{p}}$ interferes with T_p, are distinguished by the symmetrical binary variable $x_{\widetilde{p}p}^{ji}$ and $y_{\widetilde{p}p}^{ji}$.

- $x_{p\widetilde{p}}^{ij} = 1$ and $y_{p\widetilde{p}}^{ij} = 0$: (3.5) and (3.6) are trivially satisfied while (3.4) and (3.7) restrict the execution order to be the scenario shown in Fig. 3.4(c).

Note that $x_{p\widetilde{p}}^{ij}$ and $y_{p\widetilde{p}}^{ij}$ cannot be 0 at the same time, due to contradiction between (3.5) and (3.7). Based on above analysis, we know $x_{p\widetilde{p}}^{ij} + y_{p\widetilde{p}}^{ij} - 1 \in \{0, 1\}$ and use the term $x_{p\widetilde{p}}^{ij} + y_{p\widetilde{p}}^{ij} - 1$ in the MILP formulation to indicate whether $T_{\widetilde{p}}$ interferes with T_p, i.e, whether the scenario in Fig. 3.4(a) occurs.

[1] Note that T_p interferes with $T_{\widetilde{p}}$ and $T_{\widetilde{p}}$ interferes with T_p are two different scenarios.

Another related constraint is that each task must have exactly one cache configuration.

$$\sum_{k=1}^{s} c_{ik} = 1 \tag{3.8}$$

Based on Lem. 1, we can now formulate the constraint to guarantee feasibility of cache partitioning. At the start time of a task T_p, the total cache demand consists of the parts from T_p itself and the tasks that interfere with T_p, computed by $\sum_{k=1}^{s} c_{pk} \cdot k$ (for T_p) and $(x_{p\widetilde{p}}^{ij}+y_{p\widetilde{p}}^{ij}-1)\sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$(for interference task), respectively. The term $(x_{p\widetilde{p}}^{ij}+y_{p\widetilde{p}}^{ij}-1)$ indicates whether $T_{\widetilde{p}}$ interferes with T_p. Thus, we have the following constraint.

$\forall T_p, i = 0, ..., (\frac{H_r}{h_p} - 1)$:

$$\sum_{k=1}^{s} c_{pk} \cdot k + \sum_{T_{\widetilde{p}} \notin TS(T_p), j=0}^{j=\frac{H_r}{h_{\widetilde{p}}}-1} (x_{p\widetilde{p}}^{ij} + y_{p\widetilde{p}}^{ij} - 1) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k \leq s \tag{3.9}$$

One may notice that there are quadratic items in (3.9), i.e., $(x_{p\widetilde{p}}^{ij}+y_{p\widetilde{p}}^{ij}-1)\sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$. However, we can transform this quadratic term into a set of linear constraints using Lem. 2. Here, we define an intermediate variable $t_{p\widetilde{p}}^{ij} = (x_{p\widetilde{p}}^{ij} + y_{p\widetilde{p}}^{ij} - 1)\sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$ and can obtain two linear conditions $0 \leq \sum_{k=1}^{s} c_{\widetilde{p}k} k \leq s$ from (3.8) and $(x_{p\widetilde{p}}^{ij} + y_{p\widetilde{p}}^{ij} - 1) \in \{0, 1\}$.

Lem. 2. *Given a constant $s > 0$ and two constraint spaces $P_1 = \{[t, b, x] | t = b \cdot x,\ 0 \leq x \leq s,\ b \in \{0, 1\}\}$ and $P_2 = \{[t, b, x] | 0 \leq t \leq b \cdot s,\ t \leq x,\ t - b \cdot s - x + s \geq 0,\ b \in \{0, 1\}\}$, then $P_1 \rightleftharpoons P_2$*

Proof. $\mathbf{P_1} \Rightarrow \mathbf{P_2}$: We obtain $0 \leq t \leq b \cdot s$ according to $t = bx$ and $0 \leq x \leq s$. From $b \in \{0, 1\}$ and $t = bx$, we have $t \leq x$. Based on $0 \leq x \leq s$ and $b \in \{0, 1\}$, we can obtain $(b-1)(x-s) \geq 0$. Hence, $t - b \cdot s - x + s \geq 0$ holds. $\mathbf{P_2} \Rightarrow \mathbf{P_1}$: If $b = 0$ holds, we can prove that $t = 0$ and $0 \leq x \leq s$ according to the definition of P_2. If $b = 1$ holds, we can obtain $0 \leq t = x \leq s$ from P_2. Thus, $P_2 \rightleftharpoons P_1$. □

Up to now, we have presented the formulation for the task scheduling and cache partitioning. To minimize the cache miss number in one hyper-period, the following objective function is used:

$$CM = \sum_{\forall T_i} \frac{H_r}{h_i} \sum_{j=1}^{s} c_{ij} CM^{ij} \tag{3.10}$$

where s and CM_{cache}^{ij} represent the cache capacity (in the number of ways) and the cache miss of task T_i under j-way cache configuration, respectively.

3.6.3 MILP Formulation Refinement

The formulation described in Section 3.6.2 utilizes $3Q_0$ variables to model the interference between tasks at each checking instant (i.e., variable $x^{ij}_{p\widetilde{p}}$, $y^{ij}_{p\widetilde{p}}$, and $t^{ij}_{p\widetilde{p}}$). Here, Q_0 is the number of task instances that may interfere with the task under consideration, i.e., $Q_0 = \sum_{T_{\widetilde{p}} \notin TS(T_p)} (\frac{H_r}{h_{\widetilde{p}}})$. Since the task T_p has $\frac{H_r}{h_p}$ instances, the total amount of variables can be calculated as:

$$V_0 = 3 \sum_{\forall T_p} \sum_{T_{\widetilde{p}} \notin TS(T_p)} \frac{H_r}{h_p} \frac{H_r}{h_{\widetilde{p}}} \tag{3.11}$$

As it can be seen, the total number of used variables increases in a quadratic manner with the number of tasks in the application, resulting in dramatically increased exploration space for the MILP. To maintain the scalability of the approach, it is important to develop techniques that can reduce the exploration space. Here, we propose a novel approach based on the concept of Unified Resource Demand Function (URDF). URDF models the resource demand of a task in the time domain. For task T_p with start time s_p and execution time e_p, the cache demand at instant t can be defined as:

$$URDF(t, T_p) = \left\lfloor \frac{t - s_p}{h_p} \right\rfloor + 1 - \left\lceil \frac{t - s_p - e_p}{h_p} \right\rceil \tag{3.12}$$

The URDF above indicates that T_p requires the cache only in interval $[s_p + i \cdot h_p,\ s_p + e_p + i \cdot h_p]$. The URDF has several mathematic properties that are beneficial to model the interference between tasks.

Prop. 1. *$URDF(t, T_p) \in \{0, 1\}$. The case $URDF = 1$ represents that the task T_p requires the cache at time instant t and 0 otherwise.*

Prop. 2. $URDF(t, T_p) = URDF(mod(t, h_p), T_p)$.

Prop. 3. *Define intermediate variables $X_{t,T_p} = \left\lfloor \frac{t-s_p}{h_p} \right\rfloor$ and $Y_{t,T_p} = \left\lceil \frac{t-s_p-e_p}{h_p} \right\rceil$, then UDRF could be linearized as $UDRF(t, T_p) = X_{t,T_p} + 1 - Y_{t,T_p}$ with two extra constraints $\frac{t-s_p}{h_p} - 1 < X_{t,T_p} \leq \frac{t-s_p}{h_p}$ and $\frac{t-s_p-e_p}{h_p} \leq Y_{t,T_p} < \frac{t-s_p-e_p}{h_p} + 1$.*

Using URDF to model the cache demand, we can reformulate constraint (3.9) as following. $\forall T_p, i = 0, ..., (\frac{H_r}{h_p} - 1)$:

$$\sum_{k=1}^{s} c_{pk} \cdot k + \sum_{T_{\widetilde{p}} \notin TS(T_p)} URDF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k \leq s \tag{3.13}$$

Similar to the linear procedure of (3.9), we define intermediate variable $a^i_{p\widetilde{p}} = UDRF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$. According to Prop. 1 and Prop. 3, $UDRF(s_p + i \cdot h_p, T_{\widetilde{p}})$ could be linearized as $X_{s_p+i \cdot h_p, T_{\widetilde{p}}} + 1 - Y_{s_p+i \cdot h_p, T_{\widetilde{p}}}$ with $(X_{s_p+i \cdot h_p, T_{\widetilde{p}}} + 1 - Y_{s_p+i \cdot h_p, T_{\widetilde{p}}}) \in \{0, 1\}$. Based on Lem. 2, the non-linear item $UDRF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$ in (3.13) can also be linearized.

The advantage of this reformulation is that we do not need to have separate variables and constraints for every instance of task $T_{\widetilde{p}} \notin TS(T_p)$, resulting in significant reduction of the number of variables and the exploration space. After the refinement, we just need $3Q_1$ variables to model the inter-core interference at a specific checking instant, where Q_1 is the cardinality of the set $T_{\widetilde{p}} \notin TS(T_p)$, i.e. $Q_1 = |\{T_{\widetilde{p}} | T_{\widetilde{p}} \notin TS(T_p)\}|$. Moreover, by applying Prop. 2, we can further reduce the exploration space. We calculate $i_{p\widetilde{p}} = \frac{H_{p\widetilde{p}}}{h_{\widetilde{p}}}$ for each task $T_{\widetilde{p}} \notin TS(T_p)$. Based on Prop. 2, we have $UDRF(s_p + i \cdot h_p, T_{\widetilde{p}}) = UDRF(s_p + mod(i, i_{p\widetilde{p}}) \cdot h_p, T_{\widetilde{p}})$. It indicates that $UDRF$ at checking instant $s_p + mod(i, i_{p\widetilde{p}}) \cdot h_p$ can be reused at checking instant $s_p + i \cdot h_p$. In this case, we only need $i_{p\widetilde{p}}$ URDFs to model the interference between T_p and $T_{\widetilde{p}}$. Since each URDF needs 3 variables in the MILP formulation, the total amount of variables is computed as:

$$V_1 = 3 \sum_{\forall T_p} \sum_{T_{\widetilde{p}} \notin TS(T_p)} \frac{H_{p\widetilde{p}}}{h_{\widetilde{p}}} \tag{3.14}$$

Compared to (3.11), we can see the number of variables is significantly reduced after the refinement.

3.7 Time-triggered Scheduling Implementation on Multi-core System

In this section, we discuss the software implementation of the scheduler and cache configuration generated in Section 3.6. Regarding the scheduler, we consider time-triggered scheduler because it can be expected to have highly-predictable patterns of behavior. For the computing system which has a time-triggered architecture, it can be determined in advance (before the system begins executing) exactly what the system will do at every moment of time while the system is operating. This time-triggered execution model have been found to be useful in many practical systems [86]. Unfortunately, most of the existing time-triggered schedulers [93] on the practical system can only be implemented on the uni-processor system. In a multi-core system, we should guarantee time-triggered scheduler executed on each core should be triggered by the common clock. Otherwise, the tasks executed on each core cannot be synchronized on the time due to

no common time reference. To solve this problem, we firstly develop one special timer, where each core has one private decrementer to implement time-triggered task activation and these private decrementers share one global clock to maintain the common time reference. Based on this share-clock timer, we extend a time-triggered scheduler, which is implemented at the proposed multi-core system presented in Section 3.3.2, to be implemented time-triggered task activation.

3.7.1 Share-clock Multi-port Timer IP

To support the dynamic timekeeping functionality in the time-triggered scheduling, a free-running counter and timer per processor are required. For the single processor system, this role is adequately served by the normal timer peripheral. While this is sufficient for a single core system, it does not work well with multiple processors due to a synchronization problem. In a multi-core system, we should guarantee that all the cores in the system are triggered in one global timer. Only in that way, the tasks on different cores can be precisely triggered and well synchronized.

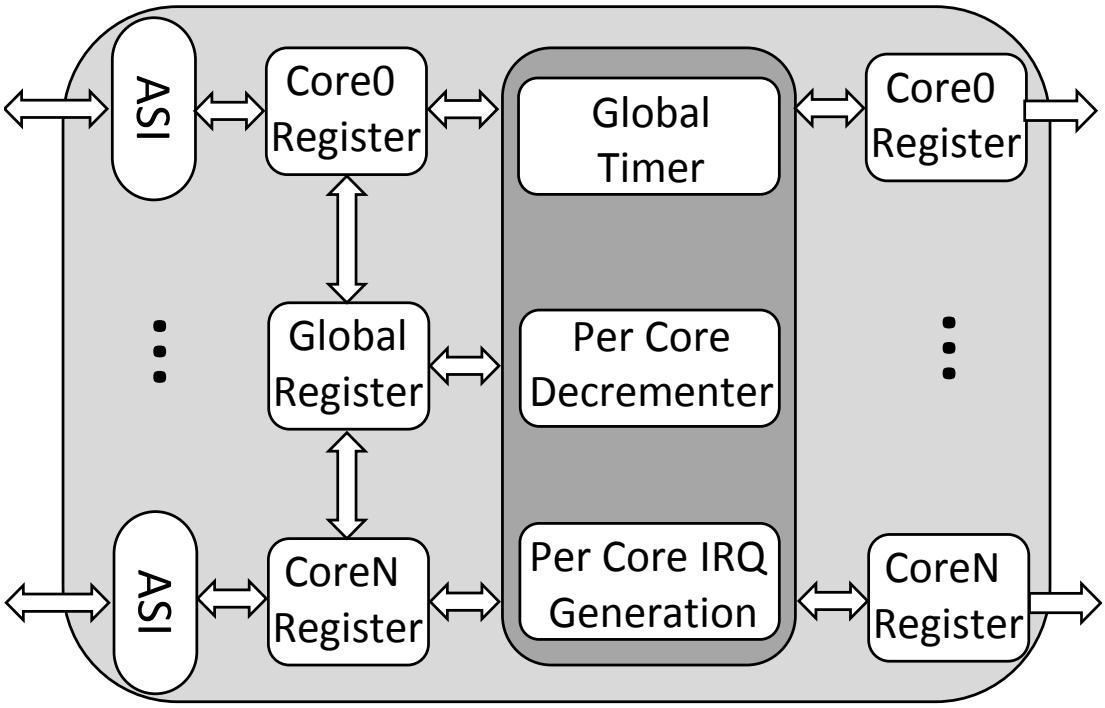


Figure 3.5: Share-clock timer IP.

Fig. 3.5 shows the block diagram of the share-clock multi-port timer, in which each port is connected to one NIOS core by *avalon slave interface (ASI)*. The share-clock multi-port timer provides each core with a dedicated 32-bit decrementer, which decrements based on the shared global timer. Here, the shared global timer expires every constant time (e.g., 1ms), which triggers each decrementer to decrement once. When one decrementer expires, an interrupt is generated to the corresponding core. Each core can dynamically control the period of each decrementer by setting its register via *ASI*, which triggers the task to execute in different points. The global register is used to synchronize the cores to be launched at the same point. Only when all cores call the APIs to start timer, the global register is set to 1. Otherwise, each core will keep waiting until this global register is active.

3.7.2 Implementation of Time-triggered Scheduling

In this section, we present the implementation of the time-triggered scheduling on the proposed multi-core system. One individual scheduler is implemented on each core. Each scheduler can be well synchronized by the customized multi-port shared-clock timer. The time-triggered scheduler on uni-processor system in [89] is extended to be executed on each core by making use of the customized multi-port shared-clock timer. Compared to the implementation of the time-triggered scheduler in [89], we make a well-organized functional division between the scheduler code and the task code and the scheduler can be executed in a more flexible manner. This functional division allows us to avoid a significant amount of hand coding to control the task activation timing. More importantly, it brings several benefits for automatic code generation (see Section 3.8 for details). Besides, our implementation of the time-triggered scheduling provides the interface to integrate the cache configuration features, which enables us to implement the proposed co-design framework of time-triggered scheduling and cache resource managements (as shown in Section 3.5) for real-time multi-core system.

Table 3.1: APIs in time-triggered scheduler

CreateTask(TaskHandler,Period Offset,CacheNum)	Create a new task and add it to the list of tasks that are ready to run.
SchedInit()	Initialize the scheduler
SchedLauch()	Lauch the scheduler
StartTimer()	Start the timer to enable the global register
WaitStartSig()	Get the value of cache hit counter
TimerInit(Ms)	Initialize the timer

To achieve the benefits mentioned above, we developed well-defined scheduler functions. Tab. 3.1 lists all the supported APIs in the scheduler. Fig. 3.6 illustrates the whole software execution process in the developed time-triggered scheduling. The whole process can be divided to three phases: initialization, synchronization, and scheduling. The software starts from the initialization phase, where the software creates new tasks and configure timer for the scheduler. The software employs the **CreateTask** function to help the scheduler to add new tasks. When the scheduler add one new tasks, two parameters related to the activation timing of tasks need to be defined by the user: period and offset. **Period** parameter denotes the interval among the repeated execution of tasks, while **offset** indicates the time after the task is released. Note that the time in the scheduler is triggered by our customized multi-port shared-clock timer. To prevent cache interference and achieve the predicable execution of the tasks, a certain number of cache partitions need to be assigned and isolated during the execution of tasks. Therefore, when one new task is added into the scheduler, we also need to add one parameter **CacheNum** to indicate how many cache ways should be allocated for this new created task. Similar the other real-time operating system [94, 95], we also define one task handler array **TaskHandler TaskArray**[**MAXTASKNUM**] to maintain the newly created tasks, where the data type **TaskHandler** and the macro constant **MAXTASKNUM** represent the task handler with above configuration information and the maximum number of the tasks allowed in the scheduler, respectively. We also use one link list to dynamically maintain the trigger sequence of task invocations. The node sequence in the link list indicates the trigger sequence of task invocations. In the initialization phase, **SchedInit()** is called to sort the link nodes based on the trigger points in an ascending manner. The header node of the link list indicates the earliest task invocation that need to execute in future. As mentioned in Section 3.7.1, the proposed multi-port timer can provide one private decrementer for each core. The software can use **TimerInit(Ms)** to set up the dedicated interrupt service routine (ISR) and set the first trigger point at **Ms** millisecond for the first task node at link list.

In the synchronization phase, the software will call the **StartTimer()** function to enable one dedicated bit in one local register. Until all cores call the **StartTimer()** function and all dedicated bits in the local register are set as 1, the global register is automatically set as 1 by internal logic in the customized multi-port shared-clock timer. Then, the scheduler on each core can start to schedule the tasks. Otherwise, the core will keep waiting until the global register is set. This synchronization scheme can guarantee that all private decrementers can

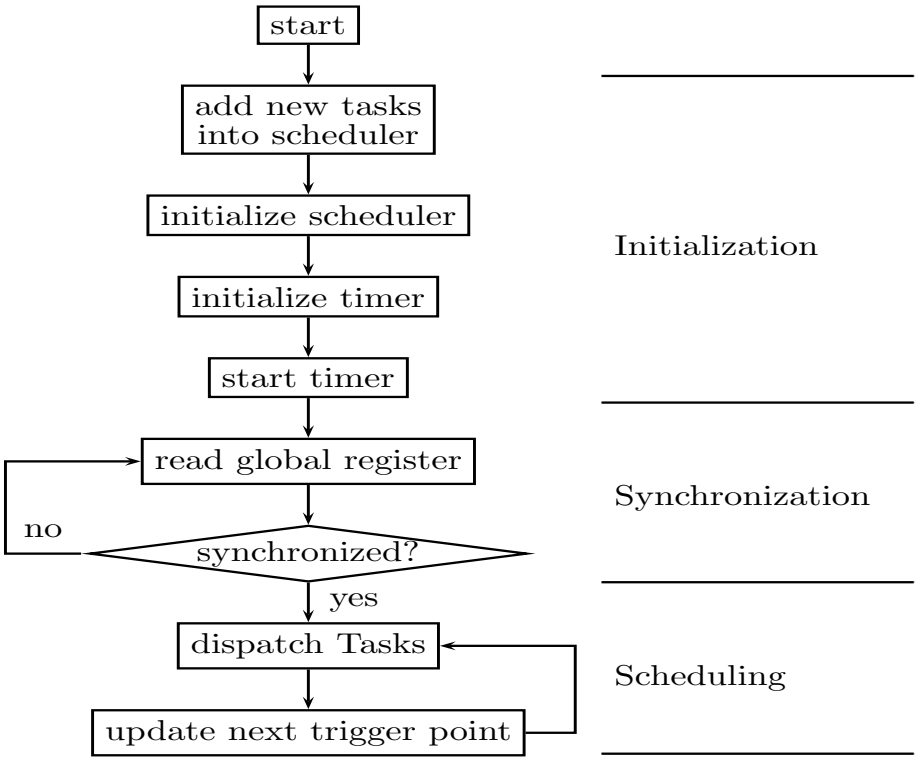


Figure 3.6: The flowchart of time-triggered scheduling.

start to count the shared clock at the same point and the scheduler on each core can be precisely synchronized.

Once all the cores are synchronized, the scheduler on each core is launched by calling **SchedLauch()** function, which is an infinite loop with calling **TaskDispatch()** function and **TriggerPointUpdate()** function. The trigger point of each task invocation is driven by the interrupt generated from the corresponding decrementer. When the interrupt occurs, interrupt service routine (ISR) activates the task invocation which should execute at current point and set up next trigger point for another task invocation. **TaskDispatch()** function is called to execute the corresponding task invocation. Then, **TriggerPointUpdate()** function inserts next trigger point of current task invocation into task link list and updates the header of the task link list.

3.8 Automatic Generation

Our cache management framework not only provides the synthesis approach for the improvement on system predictability and performance, but also support automatic generation of implementation artifacts including application scheduling code, cache configuration code, and a part of hardware RTL code such as the shared cache and timer. In the section, we present in details our automatic generation strategy for the hardware component and cache-aware programming. For the sake of clarity, we explain the automatic generation strategy by going through an illustrative example. At first, we give one example of the input specifications to describe the settings of the multi-core system, task mapping, and task timing features. Then, based on this example, we present in details how the back-end of our framework automatically generates implementation artifacts including the software code for each core and RTL code for the hardware components.

3.8.1 Input Specifications

As we stated in Section 3.5, the input specifications of our framework include platform , mapping , and task specifications, which are used to model the system design requirements. These input specifications are all written in XML format.

An example of the system-level specification of the platform is depicted in Fig. 3.7. In the example as shown in Fig. 3.7, the platform specification consist of four definition parts which specify the core components, cache subsystem component, timer subsystem component, and FPGA platform. The core components (line 2 and line 3) describe that the computing platform has 2 cores. The cache subsystem component (line 4-6) defines the hardware settings of the shared cache presented in Chapter 2 such as cache ways number, cache depths, and cache lines. According to the described configurations, cache ways number, cache depth, cache line size are defined as 8, 512, and 128, respectively. The timer subsystem component (line 7-9) defines the hardware settings for the share-clock multi-port timer presented in Section 3.7.1 such as the frequency of input clock and the period of global timer. The FPGA platform (line 10) is designated as Alter DE-115 development board. This setting decides which quartus template projects we select.

An example of task specification is shown in Fig. 3.8. For the sake of clarity, we only show the description of one task in the XML code. The task specification consists of three definition parts which specifies the function code, the timing requirement, and the profile information. In

```
1  <Platform name = "TwoCoreSystem">
2          <Core name = "Core1"> </Core>
3          <Core name = "Core2"> </Core>
4          <SharedCache name = "DynamicPartitionedCache">
5                  <Parameter waynumber= "8" depth = "512" linesize = "128"/>
6          </SharedCache>
7          <SharedTimer name = "SharedClockMultiportTimer">
8                  <Parameter clock= "50" tick = "11"/>
9          </SharedTimer>
10         <FPGABoard type= "DE115"> </FPGABoard>
11 </Platform>
```

.

Figure 3.7: The example of platform specification.

the example depicted in Fig. 3.7, the task executes a function code called sobel (line 3). This function code is provided by users. The period and deadline of task are both 170 *ms* (line 4 and line 5). The WCET and cache miss number under different cache allocations are listed in line 6 and line 7, respectively. The number of task definition blocks in XML code indicates the number of the tasks executing on the pre-defined multi-core system.

```
1  <Taskset name = "Taskset0">
2          <Task ID = "1">
3          <Task_code value = "Sobel"/>
4          <Period value = "170"/>
5          <Deadline value = "170"/>
6          <WCET value = "[209.0,116.0,102.0,99.0,96.0,85.0,79.0]"/>
7          <Miss value = "[21770,6175,3811,3401,3054,1597,725]"/>
8          </Task>
9          <!--other tasks omitted-->
10 </Taskset>
```

Figure 3.8: The example of task specification.

An example of mapping specification is given in Fig. 3.8. 4 tasks are mapped in 2-core system. Task1 and task2 are mapped on core1, while task3 and task4 are mapped on core 2.

```
1  <Mapping name = "Mapping0">
2          <Core name = "Core1"> <Task ID = "1"/> <Task ID = "2"/>  </Core>
3          <Core name = "Core2"> <Task ID = "3"/> <Task ID = "4"/>  </Core>
4  </Mapping>
```

Figure 3.9: The example of mapping specification.

3.8.2 Software Code Generation

In this framework, we use template-based approach to transform abstract data (input specifications) into executable software code. Template-based code generation is a classical technology that transforms abstract data into structured text such as software code via the use of templates. The templates provide the structure of the target code used by the generator in the process of code production. The template is a mixture of static text blocks and dynamic control blocks, in which the static text blocks are used to describe static code patterns and dynamic control blocks are used to dynamically substitute and expand the data or code into place holders in the text file according to user inputs.

```
1  <main_template file = "main.c">
2  <Static_Content>
3  #include "io.h"
4  #include "system.h"
5  #include "cache.h"
6  #include "uart0.h"
7  #include "timer.h"
8  #include "tts.h"
9  #include "TaskHandler.h"
10 int main()
11 {
12         unsigned int arg[NUM_TASKS];
13         initial_ways();
14         //add the tasks here
15 </Static_Content>
16        <Input_ADD_TASKS></Input_ADD_TASKS>
17 <Static_Content>
18         shed_init();
19         Timer_init(header_task->delta_time);
20         Start_timer();
21         wait_syn_signal_timer();
22         TTS_lauch();
23         return 0;
24 }
25 </Static_Content>
26 </main_template>
```

Figure 3.10: The example of code template for main function.

The schedule-aware cache management scheme presented in Section 3.6 can determine the scheduling and cache resource allocations for each task according to the input specifications. According to these generated designs, tasks can be scheduled with inserting cache configuration instructions in each task invocation. We can use control blocks in the template to control the

insertions of these cache configuration instructions for code generations. The scheduling and cache configurations generated by the cache management scheme presented in Section 3.6 can also be integrated into the corresponding task structures when new tasks are created. Besides, user-defined C code such as the function code of tasks can also be integrated to the generated code during the creation of new tasks.

Our framework uses XML language to develop the code templates including both static text blocks and dynamic control blocks. A snippet of XML code shown in Fig. 3.10 is used to describe the code template developed for the code generation of main function. In this example, we use a tag pairs <**Static_Content**> and </**Static_Content**> to indicate the static code blocks (line 2-line 16 and line 18-line 24). The parts among two static code blocks are dynamic control blocks which need us to generate specific code depending on input specifications (line 17). In this example, new tasks are added according to the number of tasks indicated in task specification by calling the function **CreateTask(TaskHandler,Period,Offset,CacheNum)** listed in Tab. 3.1. Besides, the parameters in this function such as **Offset** and **CacheNum** are determined by the cache management scheme presented in Section 3.6. These specific code for task creation can be generated by our code generation back-end. We will generate a C source file **main.c** for the main function.

Except for **main** function, we also need to generate task handler code, cache driver code, and timer driver code. Since the generation for these code has a similar process, we only take the process of task handler code generation as an example to explain the main steps for the sake of clarity. The code template developed for the code generation of task handler code is depicted in Fig. 3.11. In this example, we also use tag pairs <**Static_Content**> and </**Static_Content**> to separate the static code blocks and dynamic control blocks. In the template code, each task routine is wrapped with cache configuration instructions in Tab. 2.1 and the name of each wrapped function begins with **TaskHander**. The first dynamic control block uses tag pairs <**Input_TaskID**> and </**Input_TaskID**> to add task ID (see the task specification) into the name of task handler function. By this way, the executed tasks can be recolonized by the scheduler (line 5). The second dynamic control block uses tag pairs <**Input_userTask**> and </**Input_userTask**> to input the user-defined function code of tasks (line 21). In the task wrapper, the task firstly requires cache ways by calling **allo_ways(cache_allo)** for predictable execution (line 14-line 17). After the task is terminated, the task release the occupied ways by calling **rel_ways(cache_allo)** and the released cache ways can be dynamically used by other cores for high performance (line 23-line 26). This kind of task-level cache resource management

```
<TaskHander_template file = "taskhandler.h">
<Static_Content>
void TaskHander
</Static_Content>
<Input_TaskID></Input_TaskID>
<Static_Content>
(void *arg, unsigned int arg_num)
{
        unsigned int *cache_size;
        unsigned int cache_allo;
        cache_size=(unsigned int *)arg;
        cache_allo=*cache_size;
        //the first argument is cache size
        if(cache_allo>0)
        {
                allo_ways(cache_allo);
        }

        // User Task
</Static_Content>
       <Input_userTask> </Input_userTask>
<Static_Content>
        if(cache_allo>0)
        {
                rel_ways(cache_allo);
        }
}
</Static_Content>
</TaskHander_template>
```

Figure 3.11: The example of code template for task handler.

scheme cannot only guarantee the cache isolation and prevent cache interference among tasks, but also can efficiently use the cache resource according to different cache resource demand of real-time tasks. Using the similar approach, we can extract the hardware configurations from the platform specification to substitute the data in a certain place and generate the code for cache driver code and timer driver code.

3.8.3 Hardware Component Generation

The scope of this thesis focuses on developing a shared cache architecture to provide predictability for real-time multi-core systems. Regarding the processors, we use the commercial processor NIOS from Altera to construct the multi-core systems. The multi-core systems are required to be constructed by using Altera's system integration tool QSYS [96]. Currently, QSYS does

not provide interfaces to connect NIOS processors by using RTL code. Due to this limitation, our framework currently only supports generate the customized hardware component such as dynamic partitioned cache memory and share-clock multi-port timer. As mentioned above, our dynamic partitioned cache memory and share-clock multi-port timer are developed in a parameterized manner. According to platform specifications presented in Section 3.8.1, the shared cache memory and the customized timer can also be automatically generated during compile time. Thus, our design framework also allows us to conduct hardware generation.

3.9 Performance Evaluations

In this section, we present the results obtained with an implementation of the proposed framework, as well as the performance of the proposed hardware platform. In our framework, the CPLEX [97] solver is used to solve the ILP problems for our synthesis approach.

3.9.1 Experiment Setup

We implement the proposed time-triggered cache reconfigurable multi-core system on the Altera DE5 board equipped with Statrix V FPGA, which is based on the NIOS II multi-core architecture. All cores are shared with the unified cache, which is an instance of the proposed reconfigurable cache IP. By cooperating with the proposed share-clock multi-port timer, we implement the partitioned time-triggered scheduling on each core. The global tick of the shared clock timer is 1ms. To guarantee the predictability of the implementation of the scheduler, we reserve 1 fixed way for each core for the scheduler implementation (e.g., task switch).

Table 3.2: Benchmark sets for two-core system

	Core 1	**Core 2**
Set 1	Sobel, Fir	Histogram, Lms
Set 2	Fir2dim, Pbmsrch	Blackscholes, Fir
Set 3	Lms, FFT	Nsichneu, Sobel
Set 4	Lms, Histogram, FFT	Fir, Aes, Sobel
Set 5	Lms, Histogram Corner_turn,Pbmsrch	FFT, Sobel Nsichneu, Fir

To evaluate the effectiveness of our framework and hardware platform, we use 27 benchmark programs selected from MiBench [91] (Qsort, Dijkstra, Pbmsrch, FFT), CHStone [98] (Adpcm, Aes, Gsm, Sha, Mpeg2), DSPstone [99] (Dot_product, Fir2dim, Fir, Biquad, Lms, Matrix,

Table 3.3: Benchmark sets for four-core system

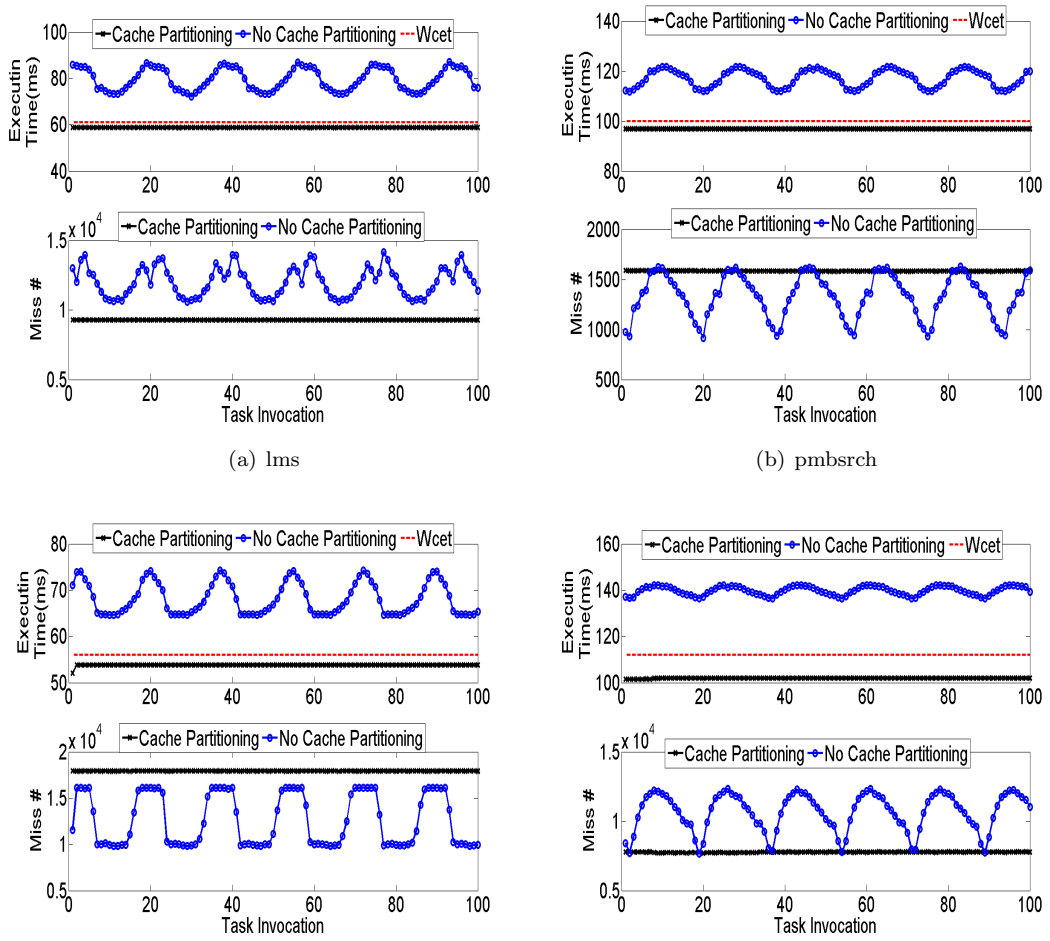
	Core 1	Core 2	Core 3	Core 4
Set 1	Lms,FFT	Fir2dim,Pbmsrch	Matrix1,N_complex	Fir,Biquad
Set 2	Fir,Mpeg2 Histogram	Biquad Qurt	Lms,Gsm Qsort	Fdct,Sobel Dijkstra,Aes
Set 3	Matrix,FFT Spectral_estimation	Fir2dim Sobel	Biquad Decode	Beamformer Histogram
Set 4	Corner_turn Dotproduct	Fir Sha	Histogram Nsichneu	Nsichneu Lms
Set 5	Fdct, Lpc Fir2dim	Histogram,Sha Sobel,decode	FFT,Adpcm Corner_turn	Blackscholes Fir

N_complex_update), PARSEC [100] (Blackscholes), UTDSP [101] (Histogram, Spectral, Lpc, Decode), Verabench [102] (Beamformer, Corner_turn), and some other research works [103, 104] (Sobel,Nsichneu,Qurt,Fdct). To avoid the selected task to saturate fast, we made some adaptations to the input scales of some benchmarks, such that they comply with the specified cache size. Tab. 3.2 and Tab. 3.3 respectively list the task sets used in our experiments for two-cores system and four-core system, which are combinations of the selected benchmarks. According to [35], we specify the task mappings based on the rule that the total execution time of each core is comparable.

3.9.2 Timing Predictability

The purpose of this experiment is to verify how effective the proposed framework is in avoiding cache interference. In this experiment, we evaluate the system timing predictability on the two-core platform with 256KB cache (8 ways with 32KB size for each way, 256 bit line size). We run 4 tasks on different cores simultaneously (Pmbsrch and Lms are on core 1, while Sobel and Ncomplex are on core 2). For comparison, we also developed one standard single port shared cache without cache partitioning, which is shared by all core. For this cache architecture, the entire cache space is competitively used by all tasks. For our reconfigurable cache, the schedule and cache configuration are automatically generated by our synthesis approach to minimize the cache miss: 1 way for Pmbsrch, 7 ways for Lms, 7 ways for Sobel, 1 way for Ncomplex.

Fig. 3.12 shows the observed execution time and cache miss of each task invocations for the four tasks on two cache architectures. From the results, we can make the following observations: (1) All tasks on our proposed cache run in a stable manner and the execution time of all task



(a) lms

(b) pmbsrch

(c) ncomplex

(d) sobel

Figure 3.12: Cache partition and no cache partition.

does not exceed their WCETs that are estimated with cache space isolation. The execution time and cache miss of all tasks on our proposed cache are steady. It means that the timing of tasks on our proposed cache can be well predicted. As a comparison, we can see the execution time and cache miss of all tasks on normal shared cache vary significantly. Without cache isolation, tasks compete for the shared cache and useful cache lines for one task on one core may be evicted by one task on another core. This cache interference will result in poor timing predictability. (2) Because only one way is assigned to Pmbsrch and Ncomplex, we get a direct-mapped cache during the execution of Pmbsrch and Ncomplex. On normal shared cache, Pmbsrch and Ncomplex can still use the whole cache size although inter-core cache interferences exist,

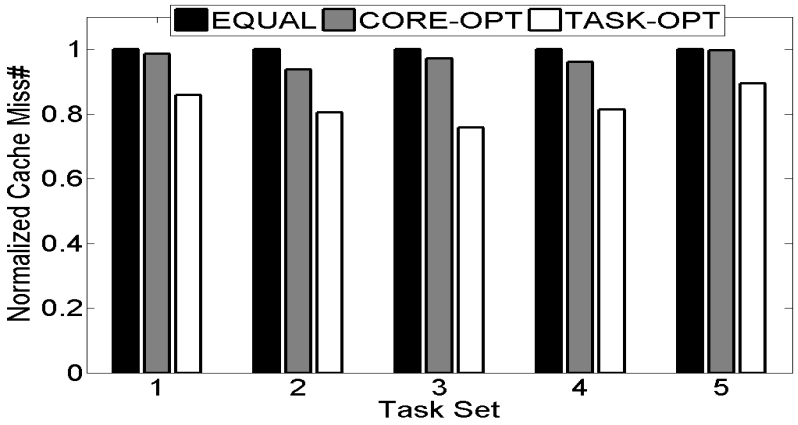
which may lead to less cache miss compared to direct-mapped cache. Note that, the system predictability is the prerequisite in real-time systems. Only when the system predictability is guaranteed, we can then consider how to improve performance. In this experiment, we aim to evaluate the system timing predictability. One interesting observation is that, even with smaller cache miss, the execution time of pmbsrch and ncomplex on normal shared cache is still greater than the one on our proposed cache. This may be caused by the fact that, all cores share normal cache via only one port, which will degrade the performance. In contrast, our proposed cache is a multi-port cache, which allows cores to access cache concurrently.

3.9.3 Runtime Performance

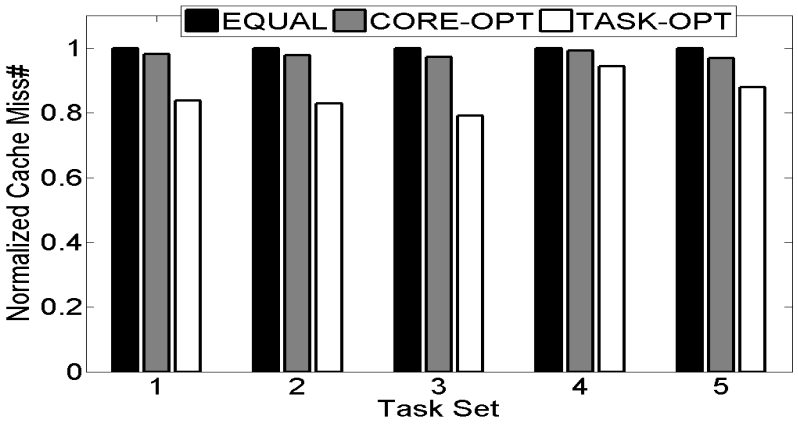
Finally, we evaluate the effectiveness of the proposed automatic cache management framework under timing predictability requirement. In this experiment, we implement the cache management scheme and scheduling on two hardware platforms: two-core system with 256KB shared unified cache (8 ways with 32KB size for each way, 256 bit line size) and four-core system with 256KB shared unified cache (16 ways with 16KB size for each way, 256 bit line size). In the two hardware platforms, each NIOS core runs at 125Mhz. Tab. 3.2 and Tab. 3.3 list the task sets used in our experiments and the task mapping information for the two-core system and the four-core system, respectively. We compared the cache miss numbers with the following technique:

- `EQUAL`: Equal partitioning cache on cores.
- `CORE-OPT`: According to the cache reservation technique in the state-of-the-art work [31], a portion of cache partitions is statically reserved for each core to prevent inter-core cache interference. For fairness comparison, we integrate this cache reservation technique [31] into our framework to generate optimal cache reservations for each core.
- `TASK-OPT`: Our synthesis approach.

Fig. 3.13 shows the total cache miss number in one hyper-period of the approaches normalized w.r.t `EQUAL`. All results are collected by implementing the cache management scheme and scheduling on the proposed multi-core system. From the result measured by real hardware, we can see cache reservation technique (`CORE-OPT`) fails to improve system performance of most benchmark sets. This is because tasks assigned on the same core might have different requirements and sensitivities to the allocated cache amount, and a designed region with a constant



(a) # Cache Miss on Two-core System



(b) # Cache Miss on Four-core System

Figure 3.13: # Cache miss reduction on different hardware platform.

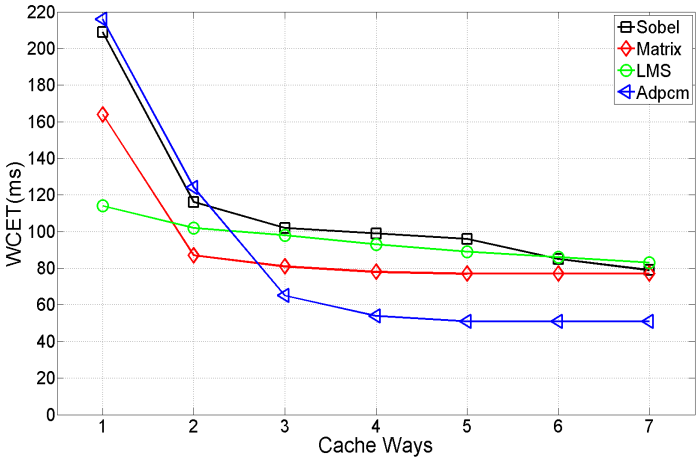
size to individual cores cannot fully meet the features of the tasks. In contrast to the cache reservation technique (`CORE-OPT`), our synthesis approach (`TASK-OPT`) partitions the cache in task level and integrates cache partitioning globally with scheduling. We can observe that our synthesis approach (`TASK-OPT`) outperforms the cache reservation technique (`CORE-OPT`). Our approach (`TASK-OPT`) can on average reduce 14.93% (up to 22.03%) and 12.56% (up to 18.6%) cache miss with respect to `CORE-OPT` on 2-core and 4-core architectures, respectively.

3.10 Case Study

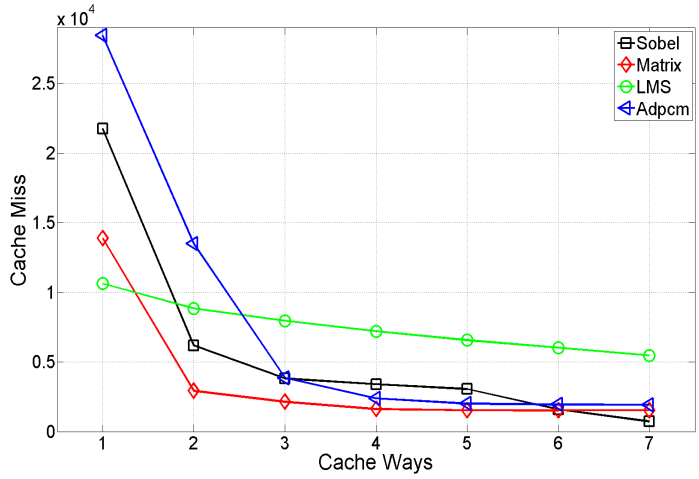
The effectiveness of the proposed cache management scheme under timing predictability requirement has been evaluated by using experiments presented in above sections. In this section, we present one case study to demonstrate the completeness of our design framework. This case study aims at demonstrating the practical usability of our framework. In this case study, the target platform is two-core system running on an Altera DE2-115 development board equipped with Cyclone IV FPGA. We construct two-core system based on NIOS II soft-cores from Altera. In the multi-core architecture, all cores are shared with one common cache instanced by the proposed dynamic partitioned cache IP. The shared cache is configured as 8 cache ways with 2KB size for each way. Share-clock multi-port timer presented in Section 3.7.1 is equipped to provide the common time base. Each core runs the time-triggered scheduler presented in Section 3.7.2 based on this equipped share-clock multi-port timer.

To represent the execution timing of the applications at runtime, each application is associated with one GPIO pin. At the start point of the execution of the application, the associated GPIO pin is pulled up. At the termination point of the execution of the application, the associated GPIO pin is pulled down. Then, we can use an oscilloscope to monitor the pull-up and pull-down timing of GPIO pins, which represents the execution timing of the applications. By this way, we can use the oscilloscope to visualize the run-time execution timing of the applications and demonstrate the execution predictability of the applications. Since the oscilloscope has four signal input channels, we can monitor up to four applications. In this case study, we consider four applications Sobel, Matrix, LMS and Adpcm to execute on two-core system. According to the mapping rules presented in [35], Sobel and Matrix are mapped on core 1 while LMS and Adpcm are on core 2. The periods of four applications are set as $180\,ms$.

We use the measurement-based approach in [93] to estimate the WCET of a task and obtain the cache miss numbers from the customized performance counter by calling the related APIs in Tab. 2.2. Fig. 3.14 shows the WCETs and cache miss numbers under different cache way configurations. As shown in Fig. 3.14, we can see that the applications we adopt in this case study have different cache sensitivities. Adpcm, Matrix, and Sobel are cache-sensitive applications. Both cache miss and WCETs of these three applications drastically decrease as the number of cache ways increases. However, the cache miss and WCETs of these three applications saturate at different amount of cache ways allocated. For example, when the number of cache ways is allocated to Adpcm and Matrix applications exceeds 4, the benefit



(a) WCETs



(b) Cache Miss

Figure 3.14: Profile data of tasks.

of assigning even more cache is very limited. However, keep increasing the cache ways for Sobel application can bring benefits to improve the system performance. On the other hand, LMS is quite insensitive to the allocated cache resource. Comparing to three cache-sensitive applications, cache miss and WCETs of LMS decrease very slowly as the number of cache ways increases.

All above system requirements are written in XML format as input specifications presented in Section 3.8.1. The task IDs of four applications Sobel, Matrix, LMS and Adpcm are set as

1, 2, 3, and 4, respectively. The applications are represented as T_i in our framework, where i is task ID. Our framework also provides a GUI to input these XML specifications, compute feasible cache configurations and scheduling, visualize the results, and automatically generate the implementation artifacts(as shown in Fig. 3.15). The GUI is developed by using Matlab GUIDE tools. The input specifications written in XML language can be parsed by using XML Matlab interface such as XMLread function. The specification data within XML files can be extracted into Matlab environment. The extracted data can be used to formulated ILP problems. Then, the ILP formulated problem presented in Section 3.6 can be solved with CPLEX solver through provided Matlab interface and generate the performance optimized scheduling and cache configurations. Then, the code generation back-end can integrate these generated results into our templates for implementation generation.

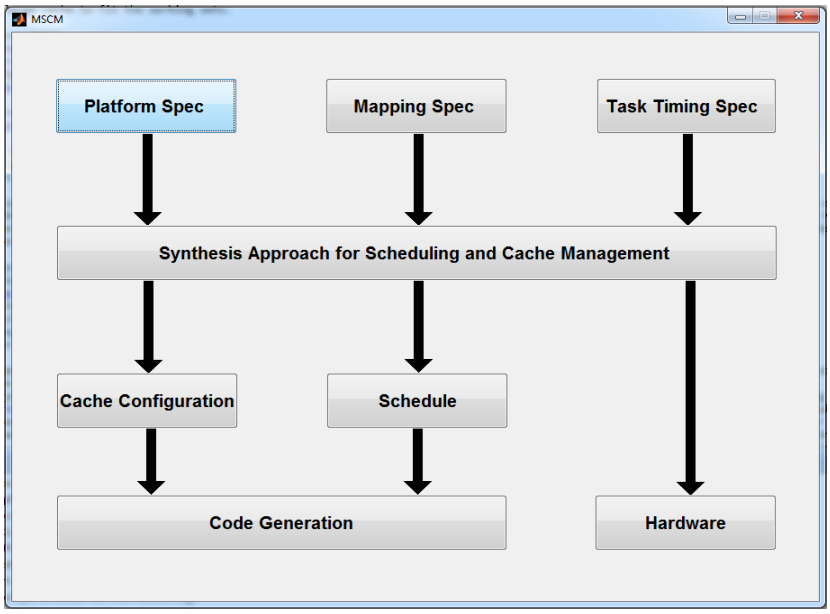


Figure 3.15: Graphic user interface of our framework.

As the output results, our approach can generate a time-triggered schedule and a set of cache configurations for each task. The generated schedule and the cache configurations together minimize the cache miss of the cache subsystem while preventing deadline misses and cache overflow. Our framework can also visualize the computed time-triggered scheduling (as shown in Fig. 3.16(a)) and cache configurations (as shown in Tab. 3.4) for each task. Our framework can sense the varying cache requirement of the tasks and balance this different cache requirements of the tasks to generate results for performance optimization while guaranteeing the deadline

constraints are satisfied and the required cache resource does not exceeds the cache capacity. In particular, Adpcm and Matrix application are allocated only 4 cache ways because assigning more cache resource will not achieve the decrease of WCET and cache miss. Sobel application is assigned with 6 cache ways to continuously achieve the performance. In contrast, insensitive application LMS is assigned with 2 cache ways.

Table 3.4: Cache configuration.

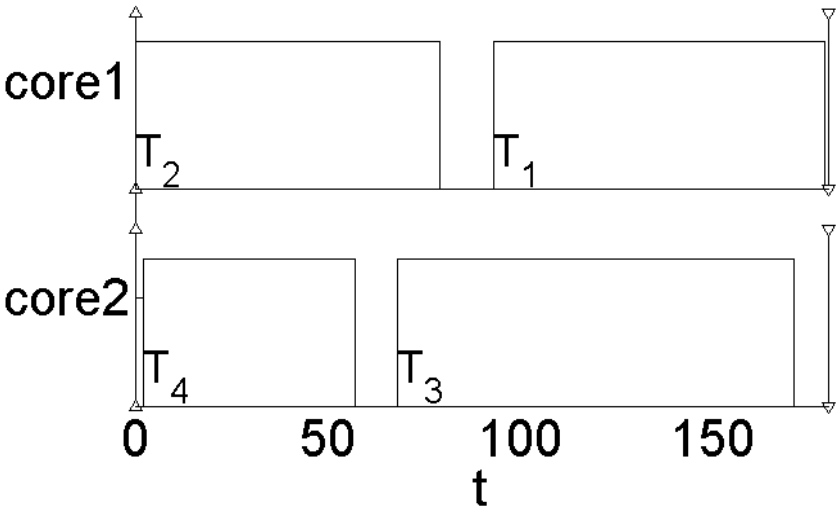
Task	T_1	T_2	T_3	T_4
Cache Allocation	6	4	2	4

Besides, we also use an oscilloscope to monitor the run-time execution timing of the task, which is represented by the timing of the associated GPIO pin. The screenshot of an oscilloscope is shown in Fig. 3.16(b). In this case study, Channel 1 (yellow line), channel 2 (blue line), channel 3 (purple line), and channel 4 (green line) are respectively connected to the GPIOs associated with task T_1, T_2, T_3, and T_4. By comparing the run-time execution timing (as shown in Fig. 3.16(b)) and the pre-computed execution timing (as shown in Fig. 3.16(a)), we can see that the tasks are executed and scheduled according to the scheduling graph generated by out framework. This means the implementation generated by our framework can be executed according to the timing we precomputed in our framework.

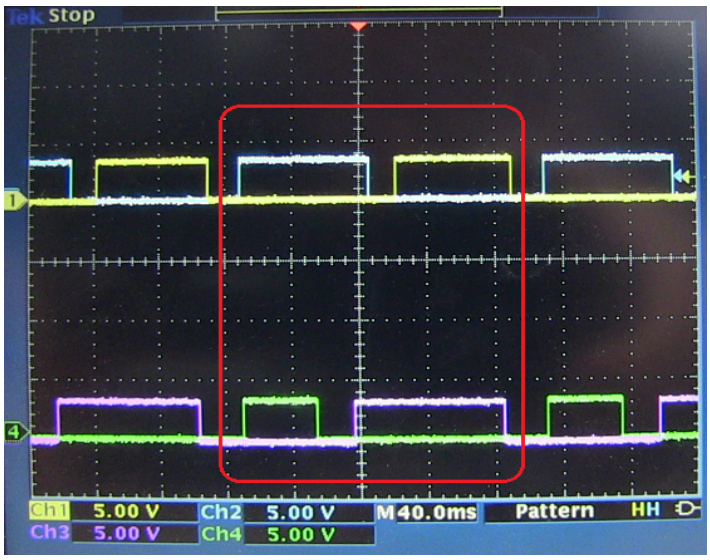
3.11 Discussion

According to the state-of-the-art survey in [23, 25], how to manage the shared cache in a predictable and efficient manner under real-time requirements is still an open issue. As one of the uniqueness of our approaches, we provide not only a reconfigurable cache architecture, which enables us to use the shared cache in a predictable and efficient manner, but also one schedule-aware cache management scheme. Besides, we also provide a physical implementation on both of hardware and software to evaluate the usability of our approaches. In this section, we summarizes the features that is currently supported and also discuss the next steps for our approaches.

Dynamic partitioned cache and its implementation. We propose a parameterized reconfigurable cache architecture, so called dynamic partitioned cache memory, for real-time multi-core system and physically implement it on FPGA. The dynamic partitioned cache memory is interfaced to Altera NIOS II based multi-core system. In principle, our cache can be



(a) Scheduling graph generated by framework.



(b) Scheduling graph measured on run-time.

Figure 3.16: Scheduling graph.

implemented at any level of caches (L1 or L2) in the cache hierarchy. Due to the technology limitations stemmed from Altera NIOS II soft-core processor, we currently do not implement cache coherency protocol on the proposed cache. Besides, according to the state-of-the-art research work in [61], current cache coherence strategies are not suitable for the real-time system.

Another aspect for improvement is to enable write-back policy on the proposed dynamic

partitioned cache memory. Currently, the proposed shared cache architecture is multi-port cache with using write-through policy, which allows NIOS cores to access the cache concurrently. By using write-through policy, the data in cache is always consistent to the off-chip memory. Thus, the cache ways can be released immediately and we can conduct the cache reconfiguration with the minimal timing overhead.

However, if write-back policy is adopted, all dirty data in the released cache ways need to write back to off-chip memory during the cache reconfiguration phase. This will result in a significant timing overhead for the cache reconfiguration. Nevertheless, we proposed one solution to efficiently use the write-back policy in the dynamic partitioned cache memory. In this solution, we need to extend our cache to single port shared cache. Each write operation from one core keeps updating the content of dirty data in the entire cache associativity to make it updated only when cache hit occurs, even though the dirty data is not located in the cache partitions belongs to the core. Note that, for the case of cache data which does not belongs to the cache partitions of the core, the core can only update the content of data during cache hit and however cannot evict the data out of cache. By this way, we can keep the cache data are consistent among the ways during the write-back policy.

Schedule-aware cache management scheme. We provide one integrated cache management framework which combines the time-triggered scheduling and dynamic cache partitioning together to decide how to allocate the cache to tasks to achieve system predictability as well as system performance. Currently, our framework only supports time-triggered scheduling policy. It will be a great advantage to add the support for other scheduling policy such as fixed-priority scheduling.

3.12 Summary

In this chapter, we present an integrated cache management framework that improves the execution predictability for real-time multi-core systems. In contrast to the state-of-art techniques [33–35] which statically partition cache at core level, our framework manage the shared cache at task-level based on the dynamic partitioned cache memory present in Chapter 2. Thus, our cache management can sense the varying cache requirements of tasks and allocate cache resource in a more efficient manner. For a given set of tasks and a mapping of the tasks on the predefined multi-core system, the proposed framework can automatically generate

fully deterministic time-triggered non-preemptive schedule and cache configurations for system performance optimization. The generated schedule and the cache configurations together minimize the cache miss of the cache subsystem while preventing deadline misses and cache overflow. Experiment results based on FPGA implementation demonstrate the effectiveness of the proposed cache management framework over the state-of-the-art cache management strategies when tested 27 benchmark programs on the constructed multi-core systems.

Besides, our cache management framework supports automatic generation of implementation artifacts including application scheduling code and a part of hardware RTL code. For the implementation of time-triggered schedule on the multi-core system, a share-clock multi-port timer component and time-triggered scheduler are developed in this chapter. Based on these hardware and software components including dynamic partitioned cache and time-triggered scheduler, a template-based code generator is provided to produce executable application software directly from the input specifications. Finally, we provide a case study to demonstrate the completeness of the design flow and the correctness of the run-time execution timing of the implementation generated by the proposed framework.

Chapter 4

Power Management for Real-time Multi-core Systems

In the previous chapters, we developed a highly flexiable reconfiguable cache for the real-time multi-core embedded systems and presented a shared cache managment framework to utilize the shared cache resource in a predictable and efficient manner under real-time requirements. This chapter studies another important design metric in the multi-core embedded system, i. e., power dissipation. We focus on streaming embedded systems at system level and study efficient power management under certain performance constraints. Specifically, we make use of the worst-case analysis method, i.e., real-time calculus and explore how to apply dynamic power management to reduce static power consumption of pipelined multi-core embedded systems while satisfying real-time constraints.

4.1 Introduction

Energy conservation has been a primary optimization objective in almost every system design. To achieve ever-increasing demand for computing power within limit energy budget, multi-core architectures are considered as a promising solution towards energy-efficient computing. However, as chip manufacturing technologies are currently entering post-silicon era, the number of cores on die continues to increase along with transistor count increases. Such the increase of chip density has led to sharp rising on power density that forces not all cores can be powered on due to the power and temperature constraints. This phenomenon is termed as dark silicon [27]. Thus, efficient power management technology becomes more and more important for efficient processing in real-time multi-core design.

Pipelined computing is a promising paradigm for embedded system design, which can, in principle, provide high throughput and low energy consumption [105]. In pipelined computing systems, we can split a streaming application into a sequence of functional blocks that are computed by a pipeline of processors where power-gating techniques can be applied to achieve energy efficiency. Performance constraints of a streaming application are usually imposed on two principle metrics, i.e., throughput and latency. The latency is the main concern for applications such as video/telephone conferencing and automatic pattern recognition applications, where the latency beyond a certain boundary is not tolerated. In the case of pipelined real-time systems, the latency of a streaming application can be expressed as the end-to-end deadline requirement that the application is processed through the pipeline.

Designing the scheduling policy for the pipeline stages under the requirements of both energy efficiency and timing guarantee is however non-trivial. In general, energy efficiency and timing guarantee are conflict objectives, i.e., techniques that reduce the energy consumption of the system will usually pay the price of longer execution time, and vice verse. Previous work on this topic either requires precise timing information of the system [106, 107] or tackles only soft real-time requirements [105, 108]. However, this precise timing of task arrivals might not be guaranteed in practice. Thus, the previous approaches can not guarantee the worst-case deadline and can not be applied to the embedded systems where violating deadlines could be disastrous. Compared to above work, our work tackles a pipelined event stream with non-deterministic workloads in hard real-time system by an inverted use of the pay-burst-only-once principle for energy efficiency.

This chapter studies the energy-minimization problem of coarse-grained pipelined systems under hard real-time requirements. We consider a streaming application that is split into a sequence of coarse-grained functional blocks which are mapped to a pipeline architecture for processing. The workload of the streaming application is abstracted as an event stream and the event arrivals of the stream are modeled as the arrival curves in interval domain [45]. The event stream has an end-to-end deadline requirement, i.e., the time by which any event in the stream travels through the pipeline should be no longer than this required deadline. The objective is thereby to find the optimal scheduling policies for individual stages of the pipeline with minimal energy consumption while the deadline requirement of the event stream is guaranteed.

Intuitively, the problem can be solved by partitioning the end-to-end deadline into sub-deadlines for individual pipeline stages and optimizing the energy consumption based on the partitioned sub-deadlines. However, any partition strategy based on the end-to-end deadline

and the follow-up optimization method will suffer from counting multiple times of the burst of the event stream, which will inevitably over-estimate the needed resource for every pipeline stage and lead to poor energy saving. A motivation example in Section 4.4 will demonstrate this drawback in details. Therefore, more sophisticated method is needed to tackle this problem.

In this chapter, we develop a new approach to solve the energy-minimization problem for pipelined multi-processor embedded systems while guarantee the worst-case end-to-end delay. Our idea to solve this problem lies in an inverse use of the well-known pay-burst-only-once principle [45]. Rather than directly partitioning the end-to-end deadline, we compute for the entire pipeline one service curve which serves as a constraint for the minimal resource demand. The energy minimization problem is then formulated with respect to the individual resource demands of pipeline stages. To solve this problem, we propose two heuristics, i.e., a quadratic programming heuristic and a fast heuristic. In quadratic programming heuristic, the minimization problem is transformed to a standard quadratic programming problem with box constraint and then solved by a standard solver. Observing that the formulated problem is NP-Hard, we present a fast heuristic to find a sub-optimal solution by analyzing the properties of the optimal solution, running with the complexity $O(mn)$ (m and n are the stage number and sample step number, respectively). For simplicity, we consider power-gating energy minimization and use periodic dynamic power management in [41, 42] to reduce the leakage power, i.e., to periodically turn on and off the processors of the pipeline. In this chapter, we compute period power management schemes off-line and the fixed T_{on}/T_{off} for processors of every pipeline stages are applied during runtime. With this approach, we can not only guarantee the overall end-to-end deadline requirement but also retrieve the pay-burst-only-once phenomena, achieving a significant reduction of the energy consumption. In addition, our methods are scalable with respect to the number of the pipeline stages.

The rest of the chapter is organized as follows: Section 4.2 reviews related work in the literature. Section 4.3 presents basic models and the definition of the studied problem. Section 4.4 presents the motivation example and Section 4.5 describes the proposed approach. Experimental evaluation is presented in Section 4.6 and Section 4.7 concludes the chapter.

4.2 Related Work

Pipelined computing is a promising paradigm for the embedded system design, which can in principle provide high performance and low energy consumption. Pipelined multiprocessor sys-

tems are wildly applied as a viable platform for high performance implementation of multimedia applications [109–112]. Energy optimization for pipelined multiprocessor systems is an interesting topic where numbers of techniques have been proposed in the literature. Carta et al. [105] and Alimonda et al. [113] proposed a feedback-control technique for dynamic voltage/frequency scaling (DVFS) in a pipelined MPSoC architecture with soft real-time constraints, aimed at minimizing energy consumption with throughput guarantees. Each pipelined processor is associated with a dedicated controller which monitors the occupancy level of the queues to determine when to increase or decrease the voltage-frequency levels of the processor. Javaid et al. [108] proposed a adaptive pipelined MPSoC architecture and a run-time balancing approach based on workload prediction to achieve energy efficiency. The authors in [114] proposed a dynamic power management scheme for adaptive pipelined MPSoCs. In this work, the duration of idle periods is determined based on future workload prediction and is used to select an appropriate power state for the idle processor. However, above approaches are under soft real-time constraints. When coming to hard real-time systems, these approaches can not be applicable.

There are also methods [106, 115–119] for hard real-time systems. To guarantee the end-to-end delay, the anthers in [119] studied the problem of minimizing the number of processors required for scheduling the end-to-end deadline constrained streaming applications modeled as CSDF graphs, where the actors of a CSDF are executed as strictly periodic tasks. In [115], Davare et al. optimized periods for dependent tasks on hard real-time distributed automotive systems in order to meet the end-to-end constraints. In [116], Hong et al. proposed a distributed approach to assign local deadlines for periodical tasks on distributed systems to meet the end-to-end deadline constraints. To reduce the energy consumption, Yu et al. [106] presented an integer linear programming (ILP) formulation for the problem of frequency assignment of a set of periodic independent tasks on heterogeneous multi-processor system. The authors in [117, 118] proposed leakage-aware scheduling heuristics to reduce the energy consumption by translating the real-time applications with periodic tasks to DAGs using the frame-based scheduling paradigm and considering the trade-offs among DVFS, DPM and the number of the processors. But these methods require precise timing information, such as periodical real-time events. However, in practice, this precise timing information of task arrivals might not be determined in advance. The non-determinism in the timing of event arrivals comes from two main reasons: (a) An event may be triggered by the physical environment, which, in general, is not able to be accurately predicted. (b) When a distributed system is considered, an event might be triggered by other events on different processing components, in which variable execution

workloads would make the prediction of precise information on event arrivals extremely complicated. In above researches, there is no guarantee that a event will arrive in time. Therefore, these approaches can not be applied to guarantee the worst-case deadline in embedded systems where violating deadlines could be disastrous. Unlike previous works, we focus on improving energy-efficiency in hard real-time embedded systems while guarantee the system satisfy under the worst-case deadline constraint.

To model irregular event arrivals, Real-Time Caculus(RTC) [43], which is based on Network Calculus [45], can be applied. Specifically, the arrival curve in RTC model an upper bound and a lower bound of the number of event arrivals or the demand of computation under specified time interval domain. Considering the DVFS system, Maxiaguine et al. [120] computed a safe frequency at periodical interval to prevent buffer overflow of a system. By adopting RTC models, Chen et al. [121] explored the schedulability for online DVFS scheduling algorithms proposed in [122]. Combining optimistic and pessimistic DVFS Scheduling, Perathoner et al.[123] presented an adaptive scheme for the scheduling of arbitrary event streams. When only consider dynamical power management(DPM), Huang et al. [41, 42] presented a algorithm to find periodic time-driven patterns to turn on/off processor for energy saving. On-line algorithms are proposed in [124–126] to adaptively control the power mode of a system, procrastinating the processing of arrived events as late as possible. In one algorithm in [124,125], a tight bound of event arrivals is computed based on historical information of event arrivals in the recent past. Instead of using historical information, dynamic counter technique [126] is used to predicted the future workload. Compared to above work, the distinct difference of our work is that we can tackle the correlation of a pipelined event stream by an inverted use of the pay-burst-only-once principle. With this new method, retrieving this correlation of the same event stream between different pipeline stages, we can compute longer deadlines for each pipeline stage and reduce the overall power consumption of the system.

4.3 Models and Problem Definition

4.3.1 Hardware Model

The hardware architecture that we have chosen is a simplified architecture which has no shared cache and shared bus among different processing cores. The processing cores are connected in a pipelined fashion via dedicated FIFOs. We consider the system with pipeline architecture showed in Fig.1(a). Sub-tasks of a partitioned application are mapped and executed in different

processors. The processors communicate data only through distributed memory units. Each memory unit can be organized as one or several FIFOs. The data communication and synchronization among processors are realized by blocking read and write SW primitives. This kind of hardware architecture has been realized in [127]. As the service curve of each stage can be computed for energy efficiency by our proposed approaches off-line, the worst-case FIFO size of each stage can be determined by applying the analysis approach in [128].

Each processor in the pipelined system has three power consumption modes, namely active, standby, and sleep modes, as shown in Fig. 4.1(b). To serve events, the processor must be in the active mode with power consumption P_a. When there is no event to process, the processor can switch to sleep mode with lower power consumption P_σ. However, mode-switching from sleep mode to active mode will cause additional energy and latency penalty, respectively denoted as $E_{sw,on}$ and $t_{sw,on}$. To prevent the processor from frequent mode switches, the processor can stay at standby mode with power consumption P_s, which is less than P_a but more than P_σ, i.e. $P_a > P_s > P_\sigma$. Moreover, the mode-switch from active (standby) mode to sleep mode will cause energy and time overhead, respectively denoted by $E_{sw,sleep}$ and $t_{sw,sleep}$.

Consider the overhead of switching system from active mode to sleep mode, the system break-even time T_{BET} denotes the minimum time length that system stays at sleep mode. If the interval that system can stay at sleep mode is smaller than T_{BET}, the mode-switch mode overheads are larger than the energy saving. Therefore, switching mode is not worthwhile. And break-even time T_{BET} can be defined as follows:

$$T_{BET} = \max(t_{sw}, \frac{E_{sw}}{P_s - P_\sigma}) \tag{4.1}$$

where $t_{sw} = t_{sw,on} + t_{sw,sleep}$ and $E_{sw} = E_{sw,on} + E_{sw,sleep}$.

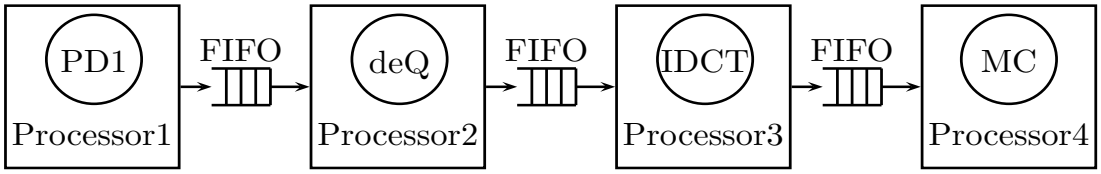
4.3.2 Energy Model

The analytical processor energy model in [3, 4, 118, 129] is adopted in this chapter, whose accuracy has been verified with SPICE simulation [3, 4, 118]. The dynamic power consumption of the core on one voltage/frequency level (V_{dd}, f) can be given by:

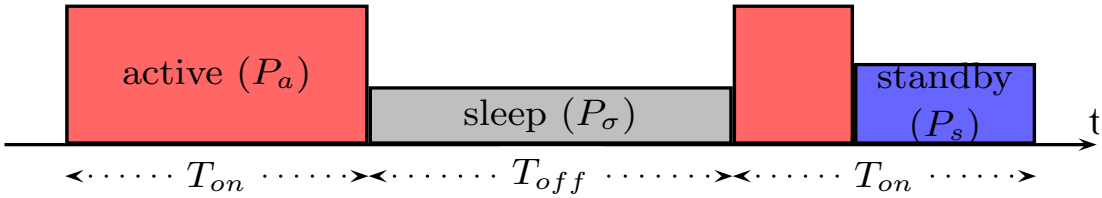
$$P_{dyn} = C_{eff} \cdot V_{dd}^2 \cdot f \tag{4.2}$$

where V_{dd} is the supply voltage, f is the operating frequency and C_{eff} the effective switching capacitance. The cycle length t_{cycle} is given by a modified alpha power model.

$$t_{cycle} = \frac{L_d \cdot K6}{(V_{dd} - V_{th})^\alpha} \tag{4.3}$$



(a) H.263 decoder on pipeline hardware architecture



(b) Power model of a processor

Figure 4.1: System model

where K_6 is technology constant and L_d is estimated by the average logic depth of all instructions critical path in the processor. The threshold voltage V_{th} is given below.

$$V_{th} = V_{th1} - K_1 \cdot V_{dd} - K_2 \cdot V_{bs} \tag{4.4}$$

where V_{th1}, K_1, K_2 are technology constants and V_{bs} is the body bias voltage.

The static power is mainly contributed by the subthreshold leakage current I_{subn}, the reverse bias junction current I_j and the number of devices in the circuit L_g. It can be presented by:

$$P_{sta} = L_g \cdot (V_{dd} \cdot I_{subn} + |V_{bs}| \cdot I_j) \tag{4.5}$$

where the reverse bias junction current I_j is approximated as a constant and the subthreshold leakage current I_{subn} can be determined as:

$$I_{subn} = K_3 \cdot e^{K_4 V_{dd}} \cdot e^{K_5 V_{bs}} \tag{4.6}$$

where K_3, K_4 and K_5 are technology constants. To avoid junction leakage power overriding the gain in lowering I_{subn}, V_{bs} should be constrained between 0 and -1V. Thus, the power consumption at active mode and at stand-by mode, i.e., P_a and P_s, under one voltage/frequency (V_{dd}, f) can be respectively computed as:

$$P_a = P_{dyn} + P_{sta} + P_{on} \tag{4.7}$$

$$P_s = P_{sta} + P_{on} \tag{4.8}$$

where P_{on} is an inherent power needed for keeping the processor on.

4.3.3 Task Model

This chapter considers streaming applications that can be split into a sequence of tasks. As shown in Fig. 4.1(a), a H.263 decoder is represented as four tasks (i.e., PD1, deQ, IDCT, MC) implemented in a pipeline fashion [130]. To model the workload of the application, the concept of arrival curve $\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)]$, originated from Network Calculus [45], is adopted. $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ provides the upper and lower bounds on the number of arrival events for the stream S in any time interval Δ. Many other traditional timing models of event streams can be unified in the concept of arrival curves. For example, a periodic event stream can be modeled by a set of step functions where $\bar{\alpha}^u(\Delta) = \lfloor \frac{\Delta}{p} \rfloor + 1$ and $\bar{\alpha}^l(\Delta) = \lfloor \frac{\Delta}{p} \rfloor$. For a sporadic event stream with minimal inter arrival distance p and maximal inter arrival distance p', the upper and lower arrival curve is $\bar{\alpha}^u(\Delta) = \lfloor \frac{\Delta}{p} \rfloor + 1$, $\bar{\alpha}^l(\Delta) = \lfloor \frac{\Delta}{p'} \rfloor$, respectively. Moreover, a widely used model to specify an arrival curve is the PJD model, where the arrival curve is characterized by with period p, jitter j, and minimal inter arrival distance d. In PJD model, the upper arrival curve can be determined as $\bar{\alpha}^u(\Delta) = \min\{\lceil \frac{\Delta+j}{p} \rceil, \lceil \frac{\Delta}{d} \rceil\}$. Fig. 4.2 depicts arrival curves for the above cases.

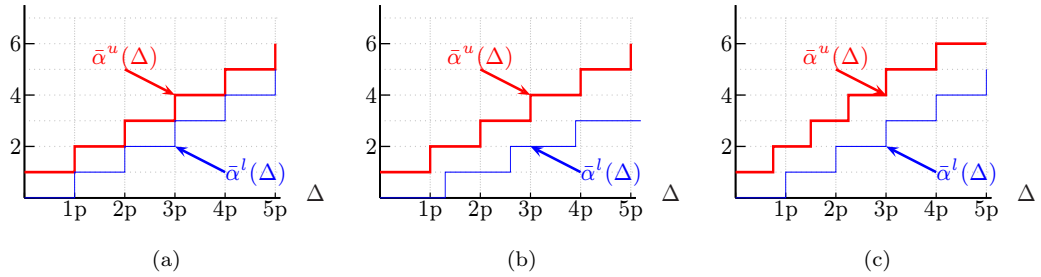


Figure 4.2: Examples for arrival curves, where (a) periodic events with period p, (b) events with minimal inter-arrival distance p and maximal inter-arrival distance $p' = 1.3p$, and (c) events with period p, jitter $j = p$, and minimal inter-arrival distance $d = 0.75p$.

Analogous to arrival curves that provide an abstract event stream model, a tuple $\beta(\Delta) = [\beta^u(\Delta), \beta^l(\Delta)]$ defines an abstract resource model which provides an upper and lower bounds on the available resources in any time interval Δ. Further details are referred to [43]. Note that arrival curves are event-based which specifies the number of event of the steam in one interval time, while service curves are based on amount of computation time. Therefore, service curve β has to be transformed to $\bar{\beta}$ to indicate the number of the event of the stream that processor can processed in specified interval time. Suppose that the execution time of an event is c, the

transformation of the service curves can be done by $\bar{\beta}^l = \lfloor \frac{\beta^l}{c} \rfloor$ and $\bar{\beta}^u = \lfloor \frac{\beta^u}{c} \rfloor$. With these definitions, a processor with lower service curve $\bar{\beta}^{Gl}(\Delta)$ is said to satisfy the deadline D for the event stream specified by $\alpha^u(\Delta)$, if the following condition holds.

$$\bar{\beta}^{Gl}(\Delta) \geq \alpha^u(\Delta - D), \forall \Delta \geq 0 \tag{4.9}$$

Note that we adopt the same assumption as [41,120,121,124,126] and assume that the worst case execution time (WCET) of each task can be is predefined and considered as system input in the chapter. As mentioned in the previous section, the hardware architecture that we have chosen is a simplified architecture which has no shared cache and shared bus among different processing cores. In this sense, we can safely assume the WCET of the running tasks as system inputs.

4.3.4 Problem Statement

This chapter considers periodic power management [41] that periodically turns on and off a processor. In each period $T = T_{on} + T_{off}$, switch the processor to active (standby) mode for T_{on} time units, following by T_{off} time units in sleep mode, as shown in Fig. 4.1(b). Given a time interval L, where $L \gg T$ and $\frac{L}{T}$ is an integer. Suppose that $\gamma(L)$ is the number of events of event stream S served in L. If all the served events finish within L, the energy consumption $E(L, T_{on}, T_{off})$ by applying this periodic scheme is

$$\begin{aligned} E(L, T_{on}, T_{off}) &= \frac{L}{T_{on} + T_{off}}(E_{sw,on} + E_{sw,sleep}) \\ &\quad + \frac{L \cdot T_{on}}{T_{on} + T_{off}} P_s + \frac{L \cdot T_{off}}{T_{on} + T_{off}} P_\sigma \\ &\quad + c \cdot \gamma(L)(P_a - P_s) \\ &= \frac{L \cdot E_{sw}}{T_{on} + T_{off}} + \frac{L \cdot T_{on}(P_s - P_\sigma)}{T_{on} + T_{off}} \\ &\quad + L \cdot P_\sigma + c \cdot \gamma(L)(P_a - P_s) \end{aligned}$$

where E_{sw} is $E_{sw,on} + E_{sw,sleep}$ for brevity. Given a sufficiently large L, without changing the scheduling policy, the minimization of energy consumption $E(L, T_{on}, T_{off})$ of a single processor is to find T_{off} and T_{on} such that the average idle power consumption $P(T_{on}, T_{off})$ is minimized.

$$\begin{aligned} P(T_{on}, T_{off}) &\stackrel{\text{def}}{=} \frac{\frac{L \cdot E_{sw}}{T_{on} + T_{off}} + \frac{L \cdot T_{on} \cdot (P_s - P_\sigma)}{T_{on} + T_{off}}}{L} \\ &= \frac{E_{sw} + T_{on} \cdot (P_s - P_\sigma)}{T_{on} + T_{off}} \end{aligned} \tag{4.10}$$

By defining $K = \frac{T_{on}}{T_{on}+T_{off}}$, the average idle power consumption P in(4.10) can be defined by T_{off} and $K(0 \leq K \leq 1)$ as follows:

$$P(K, T_{off}) \stackrel{\text{def}}{=} \frac{E_{sw}}{T_{off}} + ((P_s - P_\sigma) - \frac{E_{sw}}{T_{off}}) \cdot K \tag{4.11}$$

By analyzing (4.11), it is obvious that the following properties hold.

Prop. 4. *$\forall T_{off}, T_{off} \geq \frac{E_{sw}}{P_s - P_\sigma}$, $P(K, T_{off})$ gets its minimum when K gets its minimum.*

Prop. 5. *$\forall T_{off}, T_{off} < \frac{E_{sw}}{P_s - P_\sigma}$, $P(K, T_{off})$ gets its minimum as $P_s - P_\sigma$ when $K = 1$.*

According to Prop. 4 and Prop. 5, when $T_{off} > \frac{E_{sw}}{P_s - P_\sigma}$ holds, the processing unit should turn on as short as possible in one period. when $T_{off} \leq \frac{E_{sw}}{P_s - P_\sigma}$ holds, the processing unit should turn on all the time with $T_{off} = 0$. In this context, $\frac{E_{sw}}{P_s - P_\sigma}$ can be seen as the break-even time of the processing unit.

Based on (4.10), the energy minimization problem of a m-stage pipeline can be formulated as minimizing following function:

$$P(\vec{T}_{on}, \vec{T}_{off}) = \sum_{i}^{m} \frac{E_{sw}^i + T_{on}^i \cdot (P_s^i - P_\sigma^i)}{T_{on}^i + T_{off}^i} \tag{4.12}$$

where $\vec{T}_{on} = [T_{on}^1\ T_{on}^2\ \ldots\ T_{on}^m]$ and $\vec{T}_{off} = [T_{off}^1\ T_{off}^2\ \ldots\ T_{off}^m]$. Now we can define the problem that we studied as follows:

> *Given pipelined platform with m stages, an event stream $\mathcal{S}$ processed by this pipeline, and an end-to-end deadline requirement D, we are to find a set of periodic power managements characterized by $\vec{T}_{on}$ and $\vec{T}_{off}$ that minimize the average idle power consumption P defined in (4.12), while guaranteeing that the worst-case end-to-end delay does not exceed D.*

4.4 Motivation Example

A phenomenon called pay-bursts-only-once is well known that can give a closer upper estimate on the delay, when an end-to-end service curve is derived prior to delay computations [131]. When a workload flow with a burst traverses a number of stages in sequence, the effect of the burst of the flow on the end-to-end delay bound is the same as if the flow traversed only one node. The end-to-end delay bound computed with this property can be tighter than the sum of delay bounds of each node.

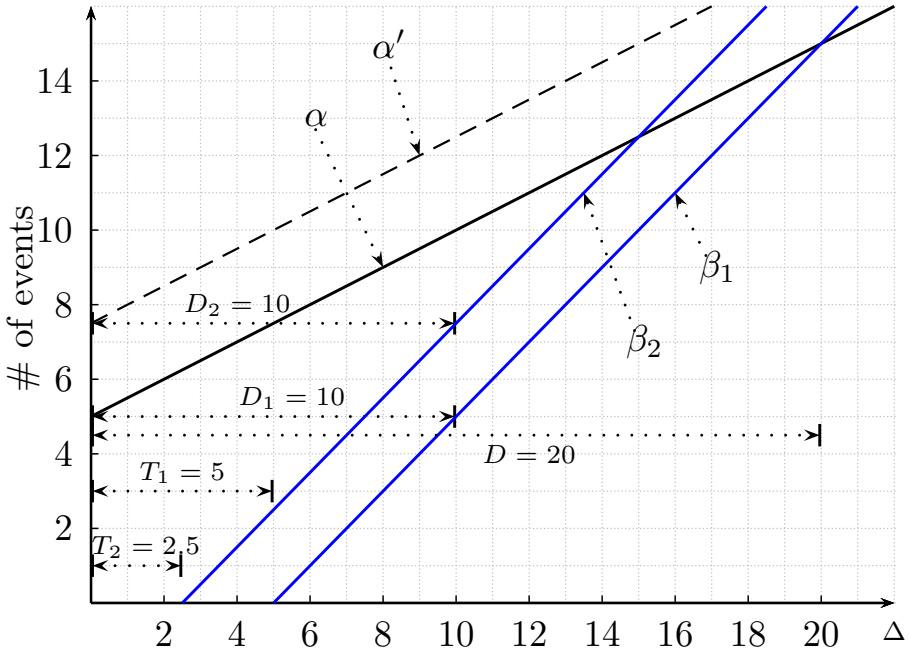
This section presents a motivation example, where an event stream passes through a 2-stage pipeline with a deadline requirement D. For simplicity, arrival curves in the leaky-bucket form and service curves in rate-latency form [45] are used. In this representation, an arrival curve is modeled as $\alpha(\Delta) = b + r \cdot \Delta$, where b is the burst and r is the leaky rate. Correspondingly, a service curve is modeled as $\beta(\Delta) = R \cdot (\Delta - T)$, where R is service rate and T is the delay. A graphical illustration of the example is shown in Fig. 4.3, where $D = 20$, $b = 5$, $r = 0.5$, and $R_1 = R_2 = 1$.

We first inspect the strategy of partitioning the end-to-end deadline and using the partitioned sub-deadlines for the two pipeline stages. For simplicity, we split the D equally, i.e., $D/2$ for each stage. As shown in Fig. 4.3(a), given $D/2$ deadline requirement for the first pipeline stage, we obtain the maximal $T_1 = \frac{D}{2} - \frac{b}{R_1} = 5$, corresponding to the minimal service demand $\beta_1 = \Delta - 5$. To derive the minimal β_2 for the second stage of the pipeline is more involved. We need the output arrival curve α' from the first stage. According to [45], $\alpha'(\Delta) = b + r \cdot T_1 + r \cdot \Delta$. Now again with a deadline requirement $D/2$ for α', we have $T_2 = \frac{D}{2} - \frac{b + r \cdot T_1}{R_1} = 2.5$.

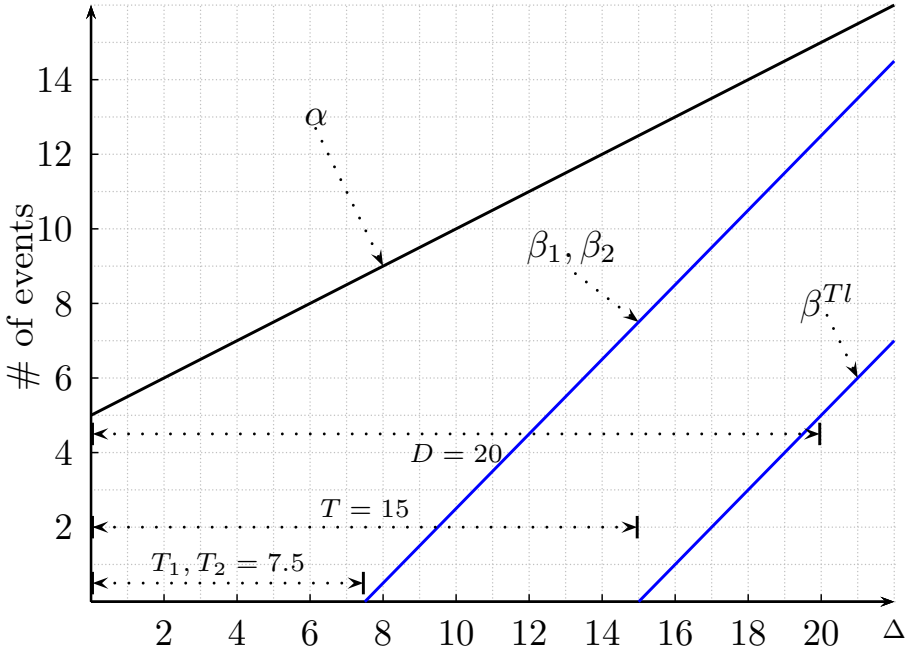
Lets take a close look at this solution. According to the concatenation theorem $\beta_{R_1,T_1} \otimes \beta_{R_2,T_2} = \beta_{\min(R_1,R_2),T_1+T_2}$, we get a concatenated service curve $\beta = \Delta - (T_1 + T_2) = \Delta - 7.5$. With this concatenated service curve, the maximal overall end-to-end deadline for β_1 and β_2 is 12.5 which is far too stricter than D. This example indicates that the obtained β_1 and β_2 based on partitioning the end-to-end deadline is too pessimistic.

The reason for the pessimism comes from paying the burst b/R_1 for the second stage of the pipeline as well as the additional delay $\frac{r \cdot T_1}{R_2}$ from the first stage, as the pay-burst-only-once principle points out. These effects will be accumulated for every stage of the pipeline, leading to even more pessimistic results, as the number of the pipeline stages increases. In addition, computing the resource demand of each stage requires the lower bound of the output arrival curve from the previous stage. Computing this output curve requires numerical min-plus convolution which will incur considerable computational and memory overheads. In conclusion, the strategy based on partitioning the end-to-end deadline is not a viable approach, in particular for the cases of pipelined systems with many stages.

On the other hand, one can first derive the total concatenated server demand β^{Tl}, in this case $T = 15$ as shown in Fig. 4.3(b). Any partition based on this T will result in smaller but valid service curves for each pipeline stage, as we can always retrieve the original end-to-end deadline by means of the pay-burst-only-once principle. For example, by an equal partition of



(a) Partition deadline



(b) Pay-burst-only-once principle

Figure 4.3: Motivation example.

T, both T_1 and T_2 are 7.5 and D is still preserved. This brings the basic idea of our approach that will be presented in the next section.

4.5 Proposed Approach

Our approach lies in an inverse use of the pay-burst-only-once principle, as mentioned in the previous section. Rather than directly partitioning the end-to-end deadline, we compute one service curve for the entire pipeline which serves as a constraint for the minimal resource demand. The energy minimization problem is then formulated with respect to the resource demands for individual pipeline stages. To solve this minimization problem, the formulation is transformed into a quadratic programming form and solved by a 2-phase heuristic.

Without loss of generality, a pipelined system with m heterogeneous stages ($m \geq 2$) is considered. The processor of the i stage can provide minimal β_i^{Gl} service. Since periodic power management is considered, the minimal service β_i^{Gl} can be modeled as an T_{on}^i and T_{off}^i pair:

$$\beta_i^{Gl}(\Delta) = (T_{on}^i \left\lceil \frac{\Delta - T_{off}^i}{T_{on}^i + T_{off}^i} \right\rceil) \otimes \Delta \tag{4.13}$$

The derivation of Eqn. (4.13) is presented in Lem. 3. In addition, to obtain a tight lower bound of service curve of the entire pipeline, we restrict T_{on}^i as a multiple of the worst case execution time c_i, i.e., $T_{on}^i = n_i\, c_i, n_i \in N^+$.

Lem. 3. *The service curve of period power management specified by T_{on} and T_{off} can be represented as follows.*

$$\beta_i^{Gl}(\Delta) = (T_{on}^i \left\lceil \frac{\Delta - T_{off}^i}{T_{on}^i + T_{off}^i} \right\rceil) \otimes \Delta \tag{4.14}$$

Proof. According to [41], the service curve of period power management specified by T_{on} and T_{off} can be represented as Eqn. (4.15).

$$\beta^{Gl}(\Delta) = \max\left(\left\lfloor \frac{\Delta}{T_{on} + T_{off}} \right\rfloor \cdot T_{on}, \Delta - \left\lceil \frac{\Delta}{T_{on} + T_{off}} \right\rceil \cdot T_{off}\right) \tag{4.15}$$

This proof presents the derivation of Eqn. (4.14), which is used to represent the service curve of period power management, to indicate that Eqn. (4.15) and Eqn. (4.14) are equivalent.

According to the definition of the min-plus convolution,

$$\begin{aligned} \beta^{Gl}(\Delta) &= (T_{on} \left\lceil \frac{\Delta - T_{off}}{T_{on} + T_{off}} \right\rceil) \otimes \Delta \\ &= \inf_{0 \leq s \leq \Delta} (\Delta - s + T_{on} \cdot \lceil \frac{s - T_{off}}{T_{on} + T_{off}} \rceil) \end{aligned}$$

We make some transformations as follows.

$$\begin{aligned} T &= T_{on} + T_{off} \\ \Delta &= k_\Delta \cdot T + r_\Delta, \;\; k_\Delta \in N^+, 0 \le r_\Delta < T \\ s &= k_s \cdot T + r_s, \;\; k_s \in N^+, 0 \le r_s < T \end{aligned}$$

Then, we have:

$$\beta^{Gl}(\Delta) = \inf_{0 \le s \le \Delta} ((k_\Delta - k_s) \cdot T + (r_\Delta - r_s) + T_{on} \cdot k_s + T_{on} \cdot \lceil \frac{r_s - T_{off}}{T} \rceil) \tag{4.16}$$

As $s \le \Delta$, there are two possibilities between the parameters r_Δ and r_s: (1) when $k_s = k_\Delta$, $r_s \le r_\Delta$ should be held for $s \le \Delta$. (2) when $k_s \le k_\Delta - 1$, there is no constraint between r_Δ and r_s because $k_s \le k_\Delta - 1$ is sufficient to guarantee $s \le \Delta$.

- Case 1: $k_s \le k_\Delta - 1$

  For this case, there is no constraints between r_Δ and r_s. Thus, we can have Eqn. (4.17) by calling Eqn. (4.16).

$$\begin{aligned} \beta^{Gl}(\Delta) &= \inf_{0 \le s \le \Delta} ((k_\Delta - k_s) \cdot T + (r_\Delta - r_s) + k_s \cdot T_{on} + T_{on} \cdot \lceil \frac{r_s - T_{off}}{T} \rceil) \\ &= \inf_{0 \le s \le \Delta} (k_\Delta \cdot T + r_\Delta - k_s \cdot (T - T_{on}) - r_s + T_{on} \cdot \lceil \frac{r_s - T_{off}}{T} \rceil) \\ &= \inf_{0 \le s \le \Delta} (k_\Delta \cdot T + r_\Delta - k_s \cdot T_{off} - r_s + T_{on} \cdot \lceil \frac{r_s - T_{off}}{T} \rceil) \end{aligned} \tag{4.17}$$

  - When $T_{off} < r_s < T$ holds, we have Eqn. (4.18) by calling Eqn. (4.17).

$$\begin{aligned} \beta^{Gl}(\Delta) &= \inf_{0 \le s \le \Delta} (k_\Delta \cdot T + r_\Delta - k_s \cdot T_{off} - r_s + T_{on}) \\ &> (k_\Delta \cdot T + r_\Delta - k_s \cdot T_{off} - T + T_{on}) \\ &= (k_\Delta \cdot T + r_\Delta - k_s \cdot T_{off} - T_{off}) \\ &\ge (k_\Delta \cdot T + r_\Delta - k_\Delta \cdot T_{off}) \end{aligned} \tag{4.18}$$

  - When $0 \le r_s \le T_{off}$ holds, we have Eqn. (4.19) by calling Eqn. (4.17).

$$\beta^{Gl}(\Delta) = \inf_{0 \le s \le \Delta} (k_\Delta \cdot T + r_\Delta - k_s \cdot T_{off} - r_s) \xrightarrow[r_s = T_{off}, k_s = k_\Delta - 1]{inf} k_\Delta \cdot T + r_\Delta - k_\Delta \cdot T_{off} \tag{4.19}$$

  For above two cases, the infimum of $\beta^{Gl}_{k_s \le k_\Delta - 1}(\Delta)$ for the case $k_s \le k_\Delta - 1$ can be obtained as Eqn. (4.20) by calling Eqn. (4.18) and Eqn. (4.19).

$$\beta^{Gl}_{k_s \le k_\Delta - 1}(\Delta) = k_\Delta \cdot T + r_\Delta - k_\Delta \cdot T_{off} = \Delta - k_\Delta \cdot T_{off} \tag{4.20}$$

- Case 2: $k_s = k_\Delta$

For this case, $r_s \leq r_\Delta$ should be held for $s \leq \Delta$. Thus we have Eqn. (4.21) by calling Eqn. (4.16).

$$\beta^{Gl}(\Delta) = \inf_{0 \leq s \leq \Delta} ((r_\Delta - r_s) + k_\Delta \cdot T_{on} + T_{on} \cdot \lceil \frac{r_s - T_{off}}{T} \rceil) \tag{4.21}$$

As r_s should be constrained by r_Δ, there are two cases for r_Δ.

- $r_\Delta \leq T_{off}$: For this case, we have $0 \leq r_s \leq r_\Delta \leq T_{off}$. Thus, we have Eqn. (4.22) by calling Eqn. (4.21).

$$\beta^{Gl}_{k_s = k_\Delta, r_\Delta \leq T_{off}}(\Delta) = \inf_{0 \leq s \leq \Delta} ((r_\Delta - r_s) + k_\Delta \cdot T_{on}) \xRightarrow[r_s = r_\Delta]{inf} k_\Delta \cdot T_{on} \tag{4.22}$$

  By integrating the cases of $k_s = k_\Delta$ and $k_s \leq k_\Delta - 1$, we have Eqn. (4.23) by calling Eqn. (4.20) and Eqn. (4.22)

$$\beta^{Gl}_{r_\Delta \leq T_{off}}(\Delta) = \min(\Delta - k_\Delta \cdot T_{off}, k_\Delta \cdot T_{on}) = k_\Delta \cdot T_{on} \tag{4.23}$$

- $r_\Delta > T_{off}$: For this case, there are two sub-cases for r_s, i.e., $0 \leq r_s \leq T_{off} < r_\Delta < T$ and $T_{off} < r_s \leq r_\Delta < T$.

  ♢ $0 \leq r_s \leq T_{off} < r_\Delta < T$: For this case, we have Eqn. (4.24) by calling Eqn. (4.21).

$$\beta^{Gl}(\Delta) = \inf_{0 \leq s \leq \Delta} ((r_\Delta - r_s) + k_\Delta \cdot T_{on}) \xRightarrow[r_s = T_{off}]{inf} r_\Delta - T_{off} + k_\Delta \cdot T_{on} < T_{on} + k_\Delta \cdot T_{on} \tag{4.24}$$

  ♢ $T_{off} < r_s \leq r_\Delta < T$: For this case, we have have Eqn. (4.25) by calling Eqn. (4.21).

$$\beta^{Gl}(\Delta) = \inf_{0 \leq s \leq \Delta} ((r_\Delta - r_s) + k_\Delta \cdot T_{on} + T_{on}) \xRightarrow[r_s = r_\Delta]{inf} T_{on} + k_\Delta \cdot T_{on} \tag{4.25}$$

  For above two cases, the infimum of $\beta^{Gl}_{k_s = k_\Delta, r_\Delta > T_{off}}(\Delta)$ can be obtained as Eqn. (4.26) by calling Eqn. (4.24) and Eqn. (4.25).

$$\beta^{Gl}_{k_s = k_\Delta, r_\Delta > T_{off}}(\Delta) = r_\Delta - T_{off} + k_\Delta \cdot T_{on} = \Delta - (k_\Delta + 1) \cdot T_{off} \tag{4.26}$$

  By integrating the cases of $k_s = k_\Delta$ and $k_s \leq k_\Delta - 1$, we have Eqn. (4.27) by calling Eqn. (4.26) and Eqn. (4.22)

$$\beta^{Gl}_{r_\Delta > T_{off}}(\Delta) = \min(\Delta - k_\Delta \cdot T_{off}, \Delta - (k_\Delta + 1) \cdot T_{off}) = \Delta - (k_\Delta + 1) \cdot T_{off} \tag{4.27}$$

With Eqn. (4.23) and Eqn. (4.27), we can obtain the service curve as Eqn. (4.28).

$$\beta^{Gl} = \begin{cases} k_\Delta \cdot T_{on} & r_\Delta \leq T_{off} \\ \Delta - (k_\Delta + 1) \cdot T_{off} & r_\Delta > T_{off} \end{cases} \tag{4.28}$$

When $0 \leq r_\Delta \leq T_{off}$ holds, we have $k_\Delta \cdot T_{on} \geq \Delta - (k_\Delta + 1) \cdot T_{off}$ and $k_\Delta = \left\lfloor \frac{\Delta}{T_{on}+T_{off}} \right\rfloor$. When $r_\Delta > T_{off}$ holds, we have $k_\Delta \cdot T_{on} < \Delta - (k_\Delta + 1) \cdot T_{off}$ and $k_\Delta + 1 = \left\lceil \frac{\Delta}{T_{on}+T_{off}} \right\rceil$.

Then, we can obtain the service curve as Eqn. (4.29).

$$\beta^{Gl} = \max\left(k_\Delta \cdot T_{on}, \Delta - (k_\Delta + 1) \cdot T_{off}\right) \tag{4.29}$$

By transforming Eqn. (4.29), we can obtain Eqn. (4.15). Thus, the service curve of period power management can be represented as Eqn. (4.14).

□

4.5.1 Problem Formulation

Regarding the problem formulation, we firstly present an approximation approach (See Lemma 5.1) to derive a lower bound of the PPM service curve. By using this approximated curve, we derive the concatenated service curve directly (See Lemma 5.2), which can be used to guarantee the real time properties (See Theorem 5.3). Then, the energy minimization problem is then formulated with respect to the resource demands for individual pipeline stages. Before presenting the formulation, we first state a few bases. By defining $K_i = \frac{T^i_{on}}{T^i_{on}+T^i_{off}}$, we have the following two lemmas.

Lem. 4. $\bar{\beta}^{Gl}_i(\Delta) \geq \frac{K_i}{c_i}(\Delta - T^i_{off} - c_i)$

Proof. According to the definition of min-plus convolution operation $\bigotimes$, the inequality $\lfloor a + b \rfloor \geq \lfloor a \rfloor + \lfloor b \rfloor$, and the inequality Eqn. (4.13), we have:

$$\bar{\beta}^{Gl}_i(\Delta) \geq \left\lfloor \frac{T^i_{on} \left\lceil \frac{\Delta - T^i_{off}}{T^i_{on}+T^i_{off}} \right\rceil}{c_i} \right\rfloor \otimes \left\lfloor \frac{\Delta}{c_i} \right\rfloor$$

With the restriction $T^i_{on} = n_i\, c_i, n_i \in N^+$ and $\lceil a \rceil \geq a$, we have:

$$\begin{aligned} \left\lfloor \frac{T^i_{on} \left\lceil \frac{\Delta - T^i_{off}}{T^i_{on}+T^i_{off}} \right\rceil}{c_i} \right\rfloor &= n_i \cdot \left\lceil \frac{\Delta - T^i_{off}}{T^i_{on} + T^i_{off}} \right\rceil \\ &\geq \frac{K_i}{c_i}(\Delta - T^i_{off}) \end{aligned}$$

According to $\lfloor a \rfloor \geq a - 1$, we have $\left\lfloor \frac{\Delta}{c_i} \right\rfloor \geq \frac{1}{c_i}(\Delta - c_i)$.

According to the rule of min-plus convolution of rate-latency service curve $\beta_{R_1,T_1} \otimes \beta_{R_2,T_2} = \beta_{min(R_1,R2),T_1+T_2}$ in [45] and $K_i \leq 1$, we have:

$$\frac{K_i}{c_i}(\Delta - T^i_{off}) \otimes \frac{1}{c_i}(\Delta - c_i) = \min(\frac{K_i}{c_i}, \frac{1}{c_i})(\Delta - T^i_{off} - c_i) = \frac{K_i}{c_i}(\Delta - T^i_{off} - c_i)$$

Then, we get the right side of the inequality. □

Lem. 5. $\bigotimes_{i=1}^{m} \bar{\beta}_i^{Gl} \geq \min_{i=1}^{m}(\frac{K_i}{c_i})\big(\Delta - \sum_{i=1}^{m}(T^i_{off} + c_i)\big)$

Proof. According to the rule of min-plus convolution of rate-latency service curve $\beta_{R_1,T_1} \otimes \beta_{R_2,T_2} = \beta_{min(R_1,R2),T_1+T_2}$ in [45] and Lem. 4, we have:

$$\bigotimes_{i=1}^{m} \bar{\beta}_i^{Gl} \geq \bigotimes_{i=1}^{m} \frac{K_i}{c_i}(\Delta - T^i_{off} - c_i) = \min_{i=1}^{m}(\frac{K_i}{c_i})\big(\Delta - \sum_{i=1}^{m}(T^i_{off} + c_i)\big)$$

□

With Lem. 5, we state below theorem.

Thm. 1. *Assuming an event stream modeled with arrival curve α is processed by an m-stage pipeline and the lower service curve of each pipeline stage is defined by a T^i_{on} and T^i_{off} pair, the pipelined system satisfies an end-to-end deadline D, if the following condition holds:*

$$\min_{i=1}^{m}(\frac{K_i}{c_i})\Big(\Delta - \sum_{i=1}^{m}(T^i_{off} + c_i)\Big) \geq \alpha^u(\Delta - D) \tag{4.30}$$

Proof. In Lem. 5, the right hand side of inequality is a lower bound of $\bigotimes_{i=1}^{m} \bar{\beta}_i^{Gl}$ which is the concatenated service curve of the pipeline. With $\bigotimes_{i=1}^{m} \bar{\beta}_i^{Gl} \geq \alpha^u(\Delta - D)$, the end-to-end delay of the pipeline is no more than D, according to the pay-burst-only-once principle. Therefore, the theorem holds. □

The left hand side of the inequality Eqn. (4.30) can be considered as a bounded-delay function $bdf(\Delta, \rho_0, b_0) = max(0, \rho_0(\Delta - b_0))$ with slope $\rho_0 = \min_{i=1}^{m}(\frac{K_i}{c_i})$ and bounded-delay $b_0 = \sum_{i=1}^{m}(T^i_{off} + c_i)$. For the stream S with deadline D, a set of minimum bounded-delay functions $bdf_{min}(\Delta, \rho, b)$ can be derived by varying b (See Section 4.5.2). Therefore, we should find a solution of $[\vec{K}, \vec{T}_{off}]$ such that the resulting bounded-delay function $bdf(\Delta, \rho_0, b_0)$ is no less than minimum bounded-delay functions $bdf_{min}(\Delta, \rho, b)$. Therefore, we can formulate our optimization problem as following:

$$\begin{aligned} \underset{\vec{K},\vec{T}_{off}}{\text{minimize}} \quad & P(\vec{K},\vec{T}_{off}) \\ \text{subject to} \quad & \min_{i=1}^{m}(\frac{K_i}{c_i}) \geq \rho \\ & \sum_{i=1}^{m}(T^i_{off}+c_i) \leq b \\ & 0 \leq K_i \leq 1,\ i=1,\ldots,m \\ & T^i_{off} \geq 0,\ i=1,\ldots,m \end{aligned} \tag{4.31}$$

where $\vec{K} = [K_1, \ldots, K_n]$. $P(\vec{K}, \vec{T}_{off})$ is obtained as follows by conducting a transformation $K_i = \frac{T^i_{on}}{T^i_{on}+T^i_{off}}$ to the average power consumption (4.10) of each stages.

$$P(\vec{K},\vec{T}_{off}) = \sum_{i}^{m}(\frac{E^i_{sw}\,(1-K_i)}{T^i_{off}} + (P^i_s - P^i_\sigma)\,K_i)$$

The advantage of the formulation (4.31) is two-fold. First of all, the service curves of individual pipeline stages are the variables of the optimization problem, which on the one hand overcomes the problem of paying burst multiple times, on the other hand avoids the costly $\bigotimes$ computation during the optimization. Second, this formulation allows us to use more efficient method to analyze the problem, which will be present in the following sections.

4.5.2 Quadratic Programming Transformation

How to solve the minimization problem (4.31) is not obvious. The constraints b and ρ indeed are not fixed values. In addition, these two constraints are correlated. For a fixed b, the minimum bounded-delay function $bdf_{min}(\Delta, \rho, b)$ can be determined by computing ρ:

$$\rho = \inf\{\rho : bdf(\Delta,\rho,b) \geq \alpha^u(\Delta - D), \forall\Delta \geq 0\} \tag{4.32}$$

In this chapter, we conduct the optimization by varying b and computing ρ for every possible b. For a fixed b, we can transform (4.31) into a quadratic programming problem with box constraints(QPB), as stated in the following lemma.

Lem. 6. *The minimization problem in (4.31) can be transformed as the following quadratic programming problem with box constraints:*

$$\begin{aligned} \underset{\vec{x}=[x_1\ \ldots\ x_m]}{\text{minimize}} \quad & \vec{x}^T Q \vec{x} \\ \text{subject to} \quad & 0 \leq x_i \leq \sqrt{E^i_{sw}(1-\rho\,c_i)},\ i=1,\ldots,m. \end{aligned} \tag{4.33}$$

where $Q = A - B$, A is $m \times m$ matrix of ones and B is $m \times m$ diagonal matrix with i^{th} diagonal element $\frac{(b-\sum_{j=1}^{m} c_j)(P^i_s - P^i_\sigma)}{E^i_{sw}}$.

Denote $\vec{x}^*$ as the optimal solution for the QPB *problem in (4.33), then the optimization solution for (4.31) can be obtained with $K_i = 1 - \frac{(x_i^*)^2}{E_{sw}^i}$ and $T_{off}^i = \frac{x_i^*}{\sum_{j=1}^m x_j^*}(b - \sum_{j=1}^m c_j)$.*

Proof. With Cauchy-Buniakowski-Schwartz's inequality, we can get that:

$$\sum_{i=1}^{m} T_{off}^i \cdot \sum_{i=1}^{m} \frac{E_{sw}^i(1-K_i)}{T_{off}^i} \geq (\sum_{i=1}^{m} \sqrt{E_{sw}^i(1-k_i)})^2$$

The minimum value of $\sum_{i=1}^m \frac{E_{sw}^i(1-K_i)}{T_{off}^i}$ can be obtained at $\frac{(\sum_{i=1}^m \sqrt{E_{sw}^i(1-k_i)})^2}{b-\sum_{j=1}^m c_j}$ when the following equation holds.

$$T_{off}^i = \frac{\sqrt{E_{sw}^i(1-K_i)}}{\sum_{j=1}^m \sqrt{E_{sw}^j(1-K_j)}}(b - \sum_{j=1}^{m} c_j)$$

Then optimization formulation in (4.31) can be formulated as:

$$\begin{aligned} \underset{K_1,K_2,\ldots,K_m}{\text{minimize}} \quad & \frac{(\sum_{i=1}^m \sqrt{E_{sw}^i(1-K_i)})^2}{b-\sum_{j=1}^m c_j} + \sum_{i=1}^{m}(P_s^i - P_\sigma^i)K_i \\ \text{subject to} \quad & \rho\, c_i \leq K_i \leq 1,\ i = 1, \ldots, m \end{aligned}$$

By defining $x_i = \sqrt{E_{sw}^i(1-K_i)}$, formulation (4.31) can be transformed as the QPB problem in (4.33). □

Note that there is a feasible region for b. To guarantee all the resulting $T_{off}^i \geq 0$, the bound-delay b should not be less than $\sum_{i=1}^m c_i$. According to (4.30), the maximum slope ρ of bound-delay function will not exceed $\frac{1}{\max_{i=1}^m c_i}$. Correspondingly, we derive the minimum bound-delay function $bdf_{min}(\Delta, \frac{1}{\max_{i=1}^m c_i}, b)$. By inverting (4.32), we can derive the maximum delay b^u by (4.34), which can guarantee that all the resulting K_i will not exceed 1. In summary, the feasible region of $b \in [b^l, b^u]$ can be bounded as follows:

$$\begin{aligned} b^u &= \sup\{d : bdf(\Delta, \frac{1}{\max_{i=1}^m c_i}, d) \geq \alpha^u(\Delta - D), \forall \Delta \geq 0\} \\ b^l &= \sum_{i=1}^{m} c_i \end{aligned} \tag{4.34}$$

4.5.3 Quadratic Programming Heuristic

With above information, we can now present the overall algorithm to the energy minimization problem defined in Section 4.3.4. Basically, bounded-delay b is scanned by step ϵ within the range $[b^l, b^h]$. For each b, we first solve the sub-problem (4.33) with a QPB solver. Then, the obtained solution is repaired to fulfill further constraints (will explain later on). The pseudo code of the algorithm is depicted in Algo. 1.

Algorithm 1 Quadratic Programming Heuristic

Input: α^u, b^l, b^h, ϵ, and $P_{min} = \infty$
Output: $\vec{K}_{opt}$, $\vec{T}_{off,\,opt}$

1: **for** $b = b^l$ to b^h with step ϵ **do**
2: compute ρ by Eqn. (4.32);
3: obtain $\vec{K}$ and $\vec{T}_{off}$ by solving (4.33);
4: repair $\vec{K}$ and $\vec{T}_{off}$;
5: **if** $P(\vec{K}, \vec{T}_{off}) < P_{min}$ **then**
6: $\vec{K}_{opt} \leftarrow \vec{K}\,; \vec{T}_{off,\,opt} \leftarrow \vec{T}_{off}$;
7: $P_{min} \leftarrow P(\vec{K}_{opt}, \vec{T}_{off,\,opt})$;
8: **end if**
9: **end for**

Thm. 2. *$\exists i \in \{1, 2, \ldots, m\}$ and $\frac{E^i_{sw}}{P^i_s - P^i_\sigma} < b - \sum_{j=1}^{m} c_j$, then the problem is NP-hard.*

Proof. If it exists an stage p_i that the condition $\frac{E^i_{sw}}{P^i_s - P^i_\sigma} < b - \sum_{j=1}^{m} c_j$ holds, the matrix Q in Lem. 6 is not positive semi-definite. Thus, QPB is the non-convex quadratic programming problem which is NP-Hard [132]. □

To solve the sub-problem (Line 3 in Algo. 1), we apply existing QPB solver. According to Thm. 2, QPB is NP-Hard when the scanned bounded-delay b is bigger enough (i.e., $\frac{E^i_{sw}}{P^i_s - P^i_\sigma} < b - \sum_{j=1}^{m} c_j$). It is, in general, difficult to solve the problem optimally. Nevertheless, there are approximation schemes [133] that can efficiently solve the non-convex QPB and there are many excellent off-the-shelf software packages [134] available. In this chapter, state-of-the-art finite B&B algorithm [134] is applied to solve our QPB problem.

After obtaining a pair of $\vec{K}$ and $\vec{T}_{off}$, the repair phase (Line 4 in Algo. 1) is conducted to fulfill further constraints. This repair scheme is represented in Algo. 2. First of all, the resulting T^i_{off} of pipeline stage i may be smaller than t^i_{sw}. In the case that $T^i_{off} < t^i_{sw}$, turning off the processor of stage i is not possible. Therefore, the solution for stage i is repaired by $[K^{'}_i, T^{i'}_{off}] = [1, 0]$, stage i is on all the time (Line 2 in Algo. 2). However, this repair step will lead to the loss of sleep time Q_i for each stage(Line 21 in Algo. 2). We record this loss and try to reassign the loss to each stage at the end of algorithm (Line 21–Line 32 in Algo. 2) to minimize the power consumption further. Second, the resulting T^i_{on} may not be a multiple of c_i, which is one of our basic requirement. The repair steps are conducted to make T^i_{on} to be a multiple of c_i (Line 6–Line 20 in Algo. 2). To guarantee the resulting $K^{'}_i$ is constant with respect to K_i, $T^{i'}_{off}$ should be adjusted to $\frac{T^{i'}_{on}}{K_i} - T^{i'}_{on}$. ΔT_i indicates how much sleep time of the stage i should adjust comparing to the original T^i_{off} (Line 14 in Algo. 2). If $\Delta T_i > 0$ holds, it means that

Algorithm 2 Repair Scheme

Input: solution of QPB problem:$[\vec{K}, \vec{T}_{off}]$
Output: $[\vec{K}^{'}, \vec{T}^{'}_{off}]$

1: compute the stage set: $S_1 = \{p_i | T^i_{off} < t^i_{sw}\}$;
2: repair $[K^{'}, T^{'}_{off}]$ of the stage $p \in S_1$ as $[1, 0]$;
3: update budget $\Delta T_i \leftarrow T^i_{off}$ and power increase $\Delta E_i \leftarrow 0$ for stages $p \in S_1$;
4: compute T_{on} and the stage set: $S_2 = \{p_i | T^i_{off} \geq t^i_{sw}\}$;
5: **for** each stage $p_i \in S_2$ **do**
6: **if** $T^i_{on} < c_i$ **then**
7: $T^{i'}_{on} \leftarrow c_i$;
8: **else**
9: $T^{i'}_{on} \leftarrow \lfloor \frac{T^i_{on}}{c_i} \rfloor c_i$;
10: **if** $\frac{T^{i'}_{on}}{K_i} - T^{i'}_{on} < t^i_{sw}$ **then**
11: $T^{i'}_{on} \leftarrow \lceil \frac{T^i_{on}}{c_i} \rceil c_i$;
12: **end if**
13: **end if**
14: compute budget $\Delta T_i = T^i_{off} - (\frac{T^{i'}_{on}}{K_i} - T^{i'}_{on})$;
15: **if** $\Delta T_i >= 0$ **then**
16: $T^{i'}_{off} \leftarrow \frac{T^{i'}_{on}}{K_i} - T^{i'}_{on}$; $\Delta E_i \leftarrow 0$;
17: **else**
18: $T^{i'}_{off} \leftarrow T^i_{off}$; $\Delta E_i \leftarrow P(T^i_{on}, T^i_{off}) - P(T^{i'}_{on}, T^{i'}_{off})$;
19: **end if**
20: **end for**
21: compute total budget $Q = \sum_{\Delta T_i > 0} \Delta T_i$;
22: **while** $Q > 0$ **do**
23: find stage p_i with maximum power increase ΔE_i;
24: **if** $\Delta T_i < 0$ **then**
25: compute available allocation $allo = \min(Q, |\Delta T_i|)$;
26: $T^{i'}_{off} \leftarrow T^{i'}_{off} + allo$; $\Delta T_i \leftarrow \Delta T_i + allo$;
27: $\Delta E_i \leftarrow P(T^{i'}_{on}, T^{i'}_{off}) - P(T^i_{on}, T^i_{off})$;
28: $Q \leftarrow Q - allo$;
29: **else**
30: break ;
31: **end if**
32: **end while**
33: update $[\vec{K}^{'}, \vec{T}^{'}_{off}]$;

$T^{i'}_{on}$ decreases and the stage i should decrease sleep time T^i_{off} to make $K^{'}_i$ constant (Line 16 in Algo. 2), which will result the loss Q_i and this part can be reassigned to prolong the sleep

time of other stages. $\Delta T_i \leq 0$ indicates $T_{on}^{i'}$ increases and the stage should increase sleep time T_{off}^{i} to make $K_i^{'}$ constant. For this case, we make $T_{off}^{i'}$ constant with respect to T_{off}^{i}, which results $K_i^{'}$ increase and power consumption increase ΔE_i (Line 18 in Algo. 2). In the end, the total loss Q should be reassigned to the stage with $\Delta T_i < 0$ to reduce the power consumption further (Line 21–Line 32 in Algo. 2). The reassignment heuristic uses power increase ΔE_i as a metric to decide which stage should be assigned first. Specifically, the heuristic iterates through all stages which need to compensate. In each iteration, it picks the stage with maximum power increase ΔE_i and increase T_{off}^{i} without causing $K_i^{'} < K_i$. The reassignment heuristic terminates when there is no loss to reassign or no stage need to compensate. It is worth noting that the repair phase we conduct can still guarantee the repaired solution to satisfy the constraints, as stated in Lem. 7.

Lem. 7. *The solution repaired by Algo. 2 satisfies the constrains in (4.31).*

Proof. The operation in line 2-line 20 will not increase the term $\sum_{i=1}^{m} T_{off}^{i}$ without causing $K_i^{'} < K_i$, which satisfy the constrains in (4.31). Reassignment heuristic in (line 21–line 32) reassign the total loss Q to each stage which need to compensate and increase its sleep time T_{off}^{i} without increasing the total sleep time $\sum_{i=1}^{m} T_{off}^{i}$ and causing $K_i^{'} < K_i$. Thus, the solution repaired by Algo. 2 satisfies the constrains in (4.31). □

4.5.4 Fast Heuristic

In Section 4.5.3, we present an quadratic programming heuristic with QPB transformation. According to Thm. 2, QPB is NP-Hard when the scanned bounded-delay b is bigger enough. Assume that bounded-delay b is scanned by n steps, then the heuristic in Section 4.5.3 needs to solve this NP-Hard problem for several times, which is time-consuming. Besides, in the first optimization step, quadratic programming heuristic do not consider the break-even time constraint (i.e., T_{off}^{i} of pipeline stage i is not smaller than T_{BET}^{i}), which could also makes the result be pessimistic. To overcome above drawbacks, we present a fast heuristic to find a sub-optimal solution, running with $O(mn)$ time complexity (m is the stage number). Different with the heuristic in Section 4.5.3, we consider the break-even time constraint in the optimization phase and partition stage set P into two stage sets according to this constraint, rather than decoupling the break-even time constraint and optimization. Based on this stage set partition, we can derive a sub-optimal solution, as stated in Lem. 8.

Lem. 8. *Give a fixed bounded-delay b and denote $[\vec{K}, \vec{T}_{off}]$ as the optimal solution for the problem. Partition the stage set P into two subsets S_1 and S_2, where $S_1 = \{p_i | T_{off}^{i} < T_{BET}^{i}\}$ and $S_2 = \{p_i | T_{off}^{i} \geq T_{BET}^{i}\}$. Then, the optimal solution $[\vec{K}, \vec{T}_{off}]$ can be determined as:*

(1) For the stage $p_i \in S1$, $[K_i, T^i_{off}] = [1, 0]$.

(2) For the stage $p_i \in S2$, $[K_i, T^i_{off}] = [\rho c_i, x_i]$, *where* $x_i = \frac{w_i}{\sum_{p_i \in S2} w_i}(b - \sum_{i=1}^{m} c_i)$ *and* $w_i = \sqrt{E^i_{sw}(1 - \rho \cdot c_i)}$.

Proof. For the stage subset S_2, $T^i_{off} \geq T^i_{BET} \geq \frac{E^i_{sw}}{P^i_s - P^i_\sigma}$ holds. The average power consumption $P(K_i, T^i_{off})$ gets its minimum at $K_i = \rho c_i$ according to Prop. 4. Thus, the average power consumption of the stage subset S_2 can be transformed as $\sum_{p_i \in S2} \frac{w_i^2}{T^i_{off}} + \sum_{p_i \in S2} \rho c_i (P^i_s - P^i_\sigma)$ with constraint $\sum_{p_i \in S2} T^i_{off} \leq b - \sum_{i=1}^{m} c_i - \sum_{p_i \in S1} T^i_{off}$. According to Cauchy-Buniakowski-Schwartz's inequality, the optimal average power consumption of the stage subset S_2 can be determined as (4.35) when $[K_i, T^i_{off}] = [\rho c_i, \frac{w_i}{\sum_{p_i \in S2} w_i}(b - \sum_{i=1}^{m} c_i - \sum_{p_i \in S1} T^i_{off})]$ holds.

$$\sum_{p_i \in S2} P(K_i, T^i_{off}) = \frac{(\sum_{p_i \in S2} w_i)^2}{b - \sum_{i=1}^{m} c_i - \sum_{p_i \in S1} T^i_{off}} + \sum_{p_i \in S2} \rho c_i (P^i_s - P^i_\sigma) \tag{4.35}$$

According to (4.35),the average power consumption of the stage subset S_2 gets the minimum when $\sum_{p_i \in S1} T^i_{off}$ gets the minimum.

For the stage set S_1, there are two cases: (a) $T^i_{BET} = t^i_{sw}$: For this case of $T^i_{off} < t^i_{sw}$, turning off the processor of stage i is not possible as we stated in repair scheme, due to hardware requirement that the sleep time T^i_{off} of the processor should not be smaller the overhead t^i_{sw}. Thus, the solution for stage i is forced as $[K_i, T^i_{off}]_{p_i \in S1} = [1, 0]$. (b) $T^i_{BET} = \frac{E^i_{sw}}{P^i_s - P^i_\sigma}$: With Prop. 5, the optimal average power consumption of the stage subset S_1 gets its minimum at $[K_i, T^i_{off}]_{p_i \in S1} = [1, 0]$.

At this point, $\sum_{p_i \in S1} T^i_{off}$ gets the minimum as 0. Thus, the average power consumption of the stage subset S_2 gets it minimum. □

According to Lem. 8, the optimal solution can be derived directly if the stage partition $P = \{S1, S2\}$ is determined. Thus, optimal solution can be derived by exhaustive exploring all possible stage partition with the complexity $\emptyset(2^n)$. When the stage number is increasing, the complexity will increase exponentially. To reduce its complexity, fast stage partition scheme is proposed in this chapter. In this scheme, we first greedily put all stage into the stage set $S_2 = \{p_i | T^i_{off} \geq T^i_{BET}\}$ (i.e, we assume all the stage can enter sleep mode). Under this greedy partition, we compute the optimal T_{off} according to Lem. 8, as descried in Line (1) and Line (2) in Algo. 3. Then, we can assign the stages by checking whether the resulting optimal T_{off} under greedy partition is greater than T^i_{BET} (see Line (3)-Line (9) in Algo. 3). The feasibility of this partition scheme can be guaranteed by Lem. 9.

Lem. 9. *Stage partition* $P = \{S1, S2\}$ *generated by Algo. 3 is feasible.*

Proof. In Algo. 3, $\frac{w_i}{\sum_{p_i \in P} w_i}(b - \sum_{i=1}^{m} c_i) \geq T^i_{BET}$ holds for $S2$. According to Lem. 8, T^i_{off} in $S2$ can be determined as $\frac{w_i}{\sum_{p_i \in S2} w_i}(b - \sum_{i=1}^{m} c_i)$. As $S2 \subseteq P$, we can gets $T^i_{off} \geq \frac{w_i}{\sum_{p_i \in P} w_i}(b - \sum_{i=1}^{m} c_i) \geq T^i_{BET}$ holds for $S2$. Thus, Stage partition generated by Algo. 3 is feasible. □

Algorithm 3 Greedy Partition Scheme

Input: ρ, b, P
Output: $S1$,$S2$
1: Compute $w_i = \sqrt{E^i_{sw}(1-\rho \cdot c_i)}$ for each stage p_i;
2: Compute $x_i = \frac{w_i}{\sum_{p_i \in P} w_i}(b - \sum_{i=1}^{m} c_i)$ for each stage p_i;
3: **for** $p_i \in P$ **do**
4: **if** $x_i < T^i_{BET}$ **then**
5: Insert stage p_i into set $S1$;
6: **else**
7: Insert stage p_i into set $S2$;
8: **end if**
9: **end for**

For each b, we can first obtain a sub-optimal partition by greedy partition scheme, depicted in Algo. 3. Then, the optimal solution under the obtained partition can be determined. The pseudo code of the algorithm is depicted in Algo. 4.

Algorithm 4 Fast Heuristic

Input: α^u, b^l, b^h ϵ, $P_{min} = \infty$
Output: $[K_i, T^i_{off}]^m_{i=1}$
1: **for** $b = b^l$ to b^h with step ϵ **do**
2: compute ρ by Eqn. (4.32);
3: generate the feasible partition $S1$ and $S2$ by Algo. 3;
4: obtain $\vec{K}$ and $\vec{T}_{off}$ according to Lem. 8;
5: repair $\vec{K}$ and $\vec{T}_{off}$ by Algo. 2;
6: **if** $P(\vec{K}, \vec{T}_{off}) < P_{min}$ **then**
7: $\vec{K}_{opt} \leftarrow \vec{K}$; $\vec{T}_{off,\,opt} \leftarrow \vec{T}_{off}$;
8: $P_{min} \leftarrow P(\vec{K}_{opt}, \vec{T}_{off,\,opt})$;
9: **end if**
10: **end for**

4.6 Performance Evaluations

In this section, we demonstrate the effectiveness of our approach. We compare three approaches in this section: (1) Pay-burst-only-once Algorithm based on Quadratic Programming (PBOOA-QP) presented in Section 4.5.3; (2) Pay-burst-only-once Algorithm based on Fast Heuristic (PBOOA-FH) presented in Section 4.5.4; (3) Deadline Partition Algorithm (DPA): DPA partitions the end-to-end deadline into sub-deadlines for individual pipeline stages and explore

Table 4.1: Constants for 70 nm technology [3, 4].

Const	Value	Const	Value	Const	Value
K_1	0.063	K_6	5.26×10^{-12}	V_{th1}	0.244
K_2	0.153	K_7	-0.144	I_j	4.8×10^{-10}
K_3	5.38×10^{-7}	V_{dd}	[0.5,1]	C_{eff}	0.43×10^{-9}
K_4	1.83	V_{bs}	[-1,0]	L_d	37
K_5	4.19	α	1.5	L_g	4×10^6

Table 4.2: Power parameters

V_{dd}	P_a	P_s	P_σ	E_{sw}	t_{sw}
$0.7V$	$656mW$	$390mW$	$50\mu W$	$483\mu J$	$10ms$

all the possible deadline partition combinations to find deadline partition with the minimum energy consumption. For each deadline partition combination, DPA use the scheme in [41] to minimize the energy consumption of individual pipeline stages to optimizes the overall energy consumption. To show the effects of our scheme, we report the average idle power that is computed as Eqn. (4.10) as well as the computation time of all the schemes. The simulation is implemented in Matlab using RTC-toolbox [135] and the finite B&B algorithm [134] is used to solve QPB. All results are obtained from a 2.83GHz processor with 4GB memory.

4.6.1 Simulation Setup

The experiments are conducted based on classical energy model of 70nm technology processor in [3, 4, 129], whose accuracy has been verified with SPICE simulation. Tab. 4.1 lists the energy parameter under 70nm technology [3, 4, 129]. According to [129], executing at $V_{dd} = 0.7V$ is more energy efficient than executing at lower voltages levels. To achieve the minimize the overall energy consumption of the system, we assume that the processor runs at this critical frequency level when the processor is in the active state. From [4, 129], body bias voltage V_{bs} is obtained as $-0.7V$. From [129], P_{on} related to idle power can be obtained as $100mW$ and the power consumption in sleep mode P_σ is set as $50\mu W$. In [129], we can obtain energy overhead E_{sw} of state transition as $483\mu J$. We set time overhead t_{sw} of state transition as $10ms$. According to the energy parameter in Tab. 4.1 and the energy model in Section 4.3.2, we can calculate the corresponding active power P_a and stand-by power P_s under voltage level $V_{dd} = 0.7V$. Tab. 4.2 lists all the power parameters used in the experiment.

Table 4.3: Average power savings with respect to DPA

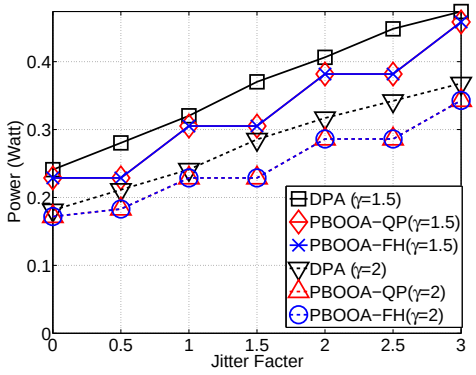
	H.263 2-stages	MP3 2-stages	TDE 2-stages	H.263 3-stages	MP3 3-stages	TDE 3-stages
PBOOA-QP	10.46%	11.57%	39.62%	23.59%	26.37%	30.60%
PBOOA-FH	10.46%	11.57%	39.65%	23.31%	25.69%	34.09%

An event stream is specified by the PJD model with period p, jitter j, and minimal inter-arrival distance d. It is worthy noting that a worst-case execution time c is associated with the service curve of different stage, as stated in Section 4.3.3. The jitter j and the relative deadline D of the stream are respectively defined as $j = \varphi \cdot p$ and $D = \gamma \cdot p$ and varies according to the corresponding factors.

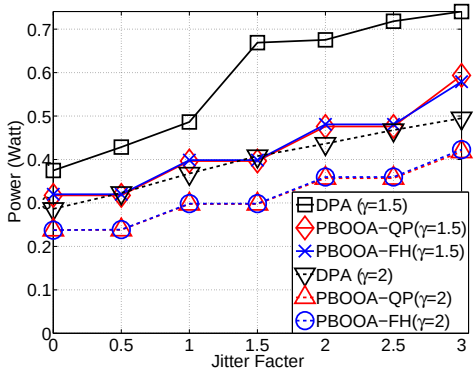
To evaluate the effectiveness of our approach, we conduct the experiments with three applications. We collected results for these three applications with deadline and jitter varied with the corresponding factor γ and φ. In following, we give a brief overview of the three applications. The H.263 decoder application [130] was modeled by four tasks consisting of: packet decoding(PD1), inverse-quantization operation(deQ),inverse DCT operation(IDCT) and motion compensation(MC). The execution time of each subtask in H.263 decoder application can be found in [130]. The activation period of the H.263 decoder application is $100ms$ with varying the jitter and the end-to-end deadline. MP3 decoder application is implemented in a pipeline fashion [130], which can be split into five tasks including packet decoding(PD2), Huffman decoding(HD), inverse-quantization operation(deQ), inverse DCT operation(IDCT), antialiasing(FB). The execution time of each subtask in H.263 decoder application can be found in [130]. The activation period of the MP3 decoder application is $100ms$ with varying the jitter and the end-to-end deadline. Time Delay Equalization (TDE) comes from the GMTI (Ground Moving Target Indiciator) application, which is obtained from StreamIt Benchmarks [136]. Time Delay Equalization (TDE) application contains 4 tasks including tasks like FFT reorder, combined DFT, FFT reorder, and combined IDFT. We set activation period of the consumer application as $30ms$.

4.6.2 Simulation Result

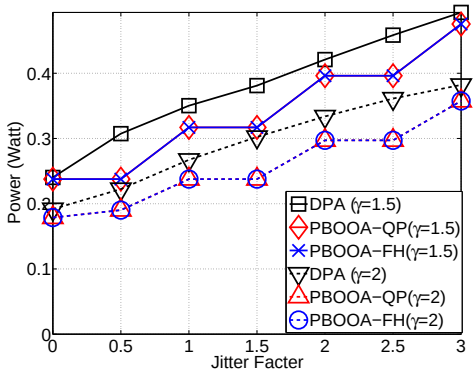
We first evaluate how the power consumptions of the compared approaches change as the jitter and deadline vary. Cases of 2-stage and 3-stage pipeline architectures with homogeneous 70-nm processors are evaluated. We vary the jitter factor φ from 0 to 3 with step 0.5 and the



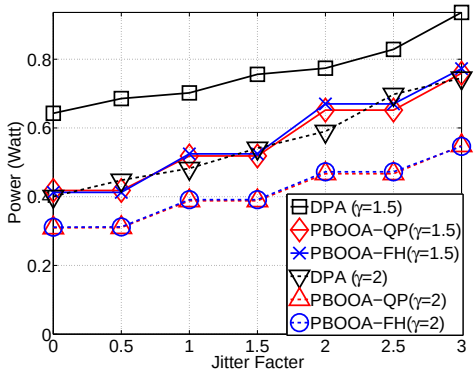
(a) H.263 on 2-stages pipeline (PD1, deQ → Core 1, IDCT, MC → Core 2)



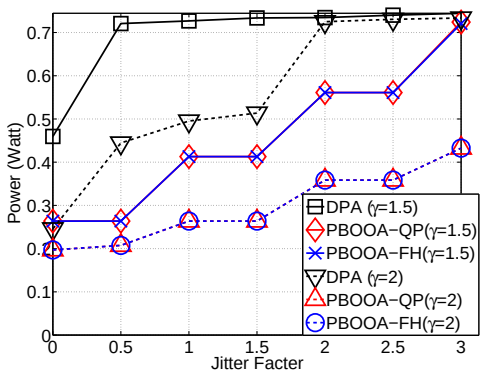
(b) H.263 on 3-stages pipeline (PD1, deQ → Core 1, IDCT → Core 2, MC → Core 3)



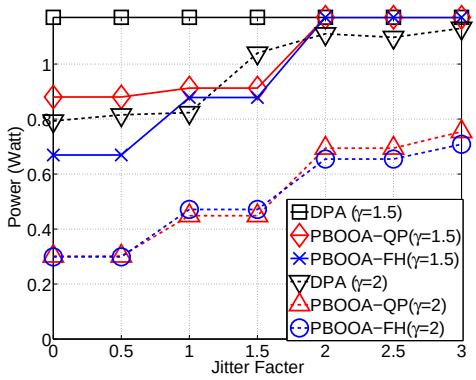
(c) MP3 on 2-stages pipeline (PD2, HD, deQ → Core 1, IDCT, FB → Core 2)



(d) MP3 on 3-stages pipeline (PD2, HD, deQ → Core 1, IDCT → Core 2, FB → Core 3)



(e) TDE on 2-stages pipeline (FFT, DFT → Core 1, FFT, IDFT → Core 2, FB → Core 3)



(f) TDE on 3-stages pipeline (FFT → Core 1, DFT → Core 2, FFT, IDFT → Core 3)

Figure 4.4: Average idle power consumption for three applications on 2-stage and 3-stage pipeline architectures

deadline factor γ from 1.5 to 2 with step 0.5. The simulation results of three approaches are shown in Fig. 4.4. In Fig. 4.4, each line represents the average energy consumption under the varied jitter factor settings with the fixed deadline factor and task mapping. From figures, we can make the following observations: (1) Pay-Bust-Only-Once based approaches always outperform deadline partition approach for all settings on both pipeline architectures. We list average normailized power savings of PBOOA-QP and PBOOA-FH with respect to DPA in Tab. 4.3. (2) The average idle power consumptions of three approaches increase as jitter increases, since the bigger jitter requires the longer T_{on} to gurantee the worst-case end-to-end deadline. (3) The average idle power consumptions of three approaches decrease as end-to-end deadline increases. This is expected becasue the loose end-to-end deadline requirement could result smaller execution time T_{on} and longer sleep time T_{off}. (4) One interesting obervation is that Pay-Bust-Only-Once based approaches could achieve more power savings on 3-stage pipeline than 2-stage pipeline for different jitter and deadline settings. This is caused by the fact that DPA on 3-stage pipeline pay burst for more times than 2-stage pipeline, which leads PBOOA-QP and PBOOA-FH can achieve more power savings on 3-satge pipeline.

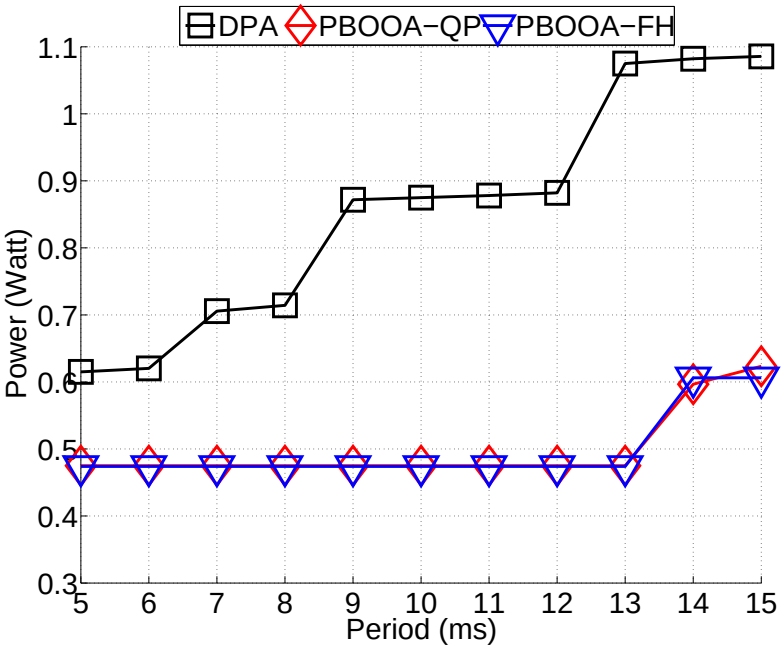


Figure 4.5: Average power consumption with t_{sw} varying.

Next, we conduct the experiment to show the impact of time overhead of state transition t_{sw} to the effectiveness of our approaches. H.263 application with jitter factor $\varphi = 0.5$ and deadline

factor $\gamma = 1$ runs in 3-stage pipeline architectures with homogeneous 70-nm processors. We vary the time overhead of state transition t_{sw} from 5ms to 15ms with fixed step size 1ms. Fig. 4.5 illustrates the average power consumptions for the three compared approaches. In Fig. 4.5, we can observe that our approaches can find efficient solutions and outperform DPA at all of t_{sw} settings. Besides, when t_{sw} increases, the average power consumptions of DPA increases faster, compared to pay-burst-only-once based approaches. This is because, DPA generates the less idle time due to suffering from paying burst for many times, compared to pay-burst-only-once based approaches, as we show in Section 4.4. The increase of t_{sw} will reduce the opportunities of turnning off the processor, which results entering sleep modes should be more difficult for DPA.

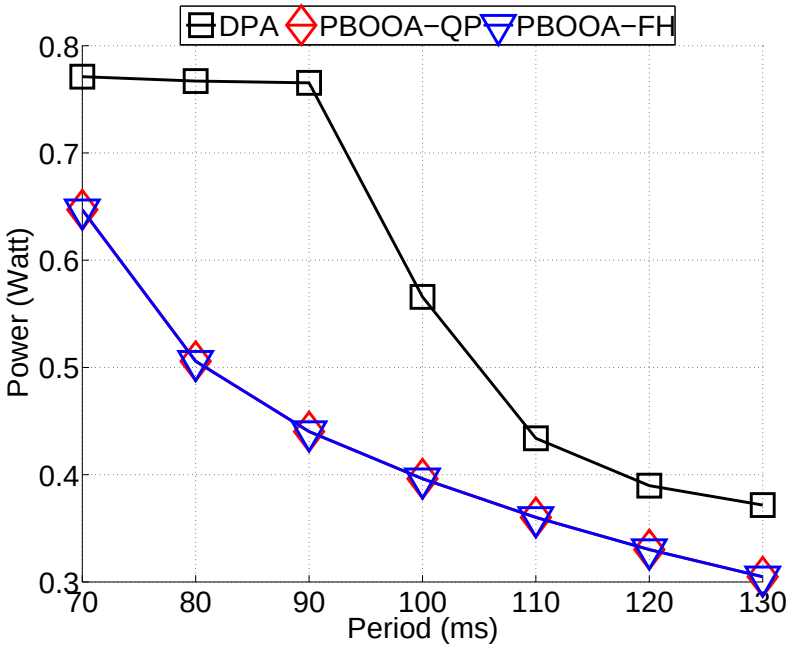
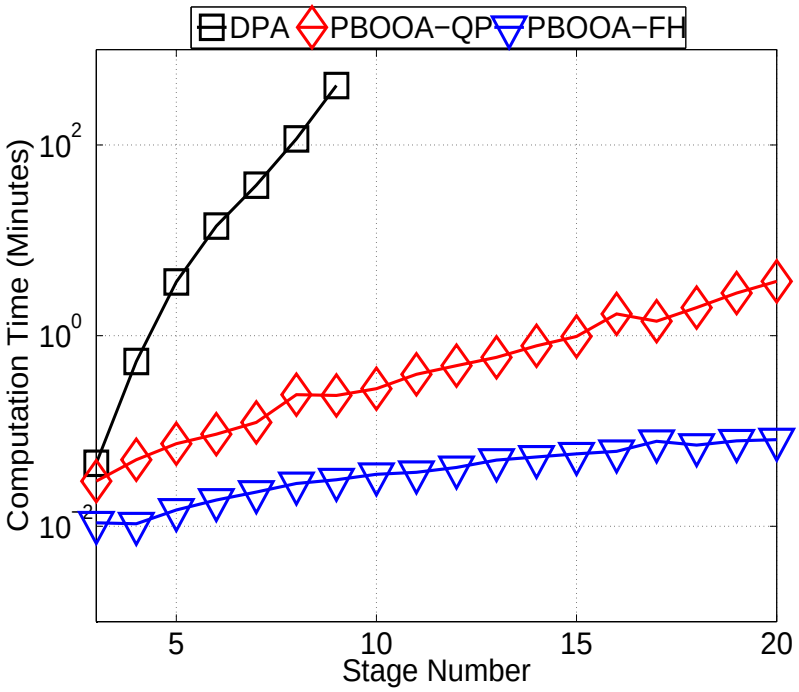


Figure 4.6: Average power consumption with period varying.

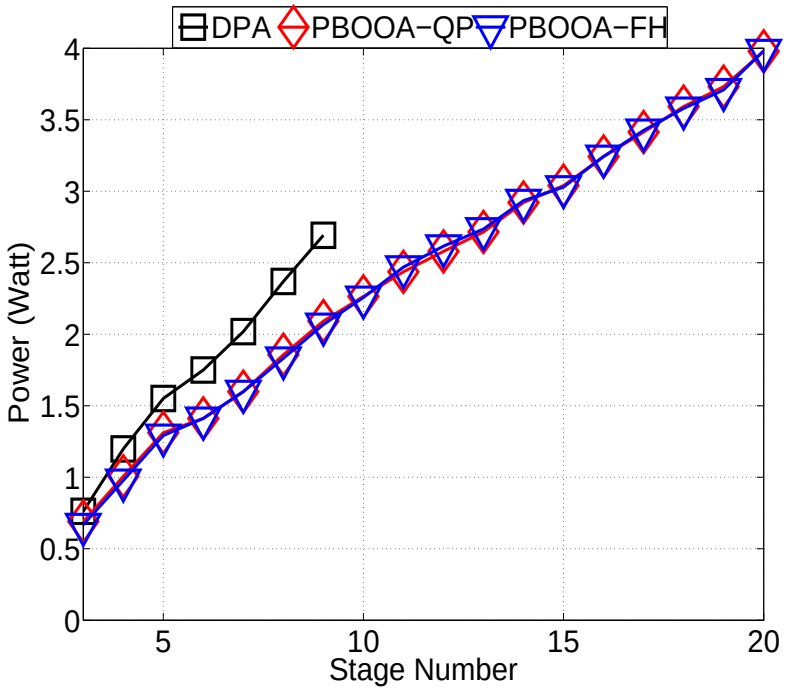
Then, we discuss the impact of the period setting to the effectiveness of the approaches. MP3 application with jitter factor $\varphi = 1$ and deadline factor $\gamma = 1.5$ runs in 2-stage pipeline architectures with homogeneous 70-nm processors. We vary period settings from 70ms to 130ms with fixed step size 10ms. Fig. 4.6 illustrates the average power consumptions for the three compared approaches under different period settings. From Fig. 4.6, we can see that the pay-burst-only-once based approaches outperforms DPA at all the period settings. Furthermore, the average power consumption of all approaches decreases when the period increases. This is

expected because the bigger period of the application can prolong the idle intervals.

In the end, we demonstrate the scalability of our approaches. We test our approaches by up to 20-stage heterogeneous pipeline. The execution time of subtasks mapped on each stage are randomly generated between $5ms$ and $15ms$. According to power model presented in Section 4.3.2, power profile of each stage can be generated by randomly select voltage V_{dd} between $0.5V$ and $0.8V$. The activation period of the event stream is $40ms$ with jitter factor $\varphi = 1$. The end-to-end deadline for the test case with different stage number is determined by $n \cdot 20$, where n is the stage number. The overhead value of state transition t_{sw} and E_{sw} of different stages are randomly selected between [$1ms$, $5ms$] and [$400uJ$, $800uJ$], respectively. Based on the observation that deadline partition algorithm may suffer from the deadline combination explosion and the costly $\bigotimes$ computation, we set search step as 5 for three compared approaches. Fig. 4.7 shows the power consumption and computation overhead on different pipelines architecture. From this figure, we can have below observations: (1) As shown in Fig. 4.7(a), the computation overhead of deadline partition algorithm increases exponentially. When stage number exceeds 10, deadline partition algorithm (DPA) fail to generate the results due to expiration of time budget of 8 hours. For the case of 9-stage pipeline, DPA takes almost 420 minutes, which is 9182 times longer than the the 3-stage pipeline case. This is expected because the deadline combinations will increase exponentially as stage number increases. In addition, as the stage number is increasing, the times of computing the resource demand of each following stage, which requires the lower bound of the output arrival curve from the previous stage, are increasing. Computing this output curve requires numerical min-plus convolution which will incur considerable computational and memory overheads. (2) Compared to deadline partition algorithm, Pay-Burst-Only-Once based approaches are fast and the computation time increases slowly with respect to the stage number, especially for PBOOA-FH. With the case of 20-stage pipeline, PBOOA-QP approach takes 3.7 minutes, 124 times more computing time than the 3-stage case. PBOOA-FH takes only 0.08 minutes to generate the result, only 7.5 times than the 3-stage case. (3) In the context of average idle power consumption, Pay-Burst-Only-Once based approaches are more energy-efficient than deadline partition algorithm. In Fig. 4.7(b), we can see PBOOA-QP and PBOOA-FH approaches always outperform DPA for all pipeline architectures. It indicate that our approaches are not only faster but also more energy-efficient than DPA approach. Besides, as observed in above experiments, the gap of power consumption between deadline partition algorithm and Pay-Burst-Only-Once based algorithm are increasing as the stage number is increasing. This is expected because, as the stage number increases,



(a) Time consumption



(b) Power consumption

Figure 4.7: Computation time and power computation and for heterogeneous pipelined system

the times that DPA should pay burst also increases. In contrast, the proposed approaches only need to pay burst only once, which leads the tighter end-to-end delay bound and prolong the idle intervals of the stages for energy efficiency. (4) PBOOA-FH approach can achieve almost identical average idle power consumption with respect to PBOOA-QP approach with almost 10x speedup. In some cases, PBOOA-FH approach can even achieve more energy savings than PBOOA-QP. This is because that, in contrast to PBOOA-QP approach, PBOOA-FH approach integrates break-even time constraints into the optimization phase, which leads PBOOA-FH approach can find the better solutions than PBOOA-QP approach.

4.7 Summary

In this chapter, we study the problem of energy minimization for coarse-grained pipelined systems under hard real-time constraints and propose new approaches based on an inverse use of the pay-burst-onlyonce principle. We formulate the problem by means of the resource demands of individual pipeline stages and propose two new approaches, a quadratic programming-based approach and fast heuristic, to solve the problem. In the quadratic programming approach, the problem is transformed into a standard quadratic programming with box constraint and then solved by a standard quadratic programming solver. Observing the problem is NP-hard, the fast heuristic is designed to solve the problem more efficiently. Moreover, our approaches are scalable with respect to the numbers of pipelined stages. Proof-of-concept simulation results demonstrate the effectiveness of our approaches.

In the future, we intent to extend our approaches to dynamic voltage frequency scaling (DVFS) to reduce dynamic power for pipelined system. Another interesting future work would be to target the multi-dimensional issues such as energy and thermal constraints simultaneously. In addition, how to combine our approaches with the consideration of the mapping of the application is also deemed for our future work.

Chapter 5

Conclusion and Future Work

5.1 Main Results

In this thesis, we present a set of novel techniques to efficiently manage the resource for real-time multi-core systems. We categorize the challenges into a few major topics and provide corresponding solutions. Both hardware implementations and algorithms are presented to overcome part of these challenges in real-time multi-core system design. The main results of this work are summarized in the following:

- We present a dynamic partitioned cache architecture for real-time multi-core systems and provide a implementation prototype on FPGA platform. The proposed cache architecture allows us to dynamically allocate the cache resource with minimal timing overhead. In this cache architecture, the cache resources are strictly isolated to prevent the cache interference among cores. Therefore, the proposed cache can provide predictable cache performance for real-time applications. To efficiently use cache resource and maximize the performance of applications, the proposed cache allows cores to dynamically allocate cache resource according to the demand of applications. Comparing to most existing research work [35–40] in the literature, which are devoted to analyze theoretical proposals and the simulation of reconfigurable caches, the proposed cache is physically implemented and prototyped on FPGA. The usage of the proposed cache within a real C code has been examined by a functionality test, which validates the correctness of the proposed cache prototype implementation. Besides, we also investigates the chip design process for the proposed cache memory and find the proposed cache has reasonable implementation for the chip area and power consumption under SMIC 130nm standard cell library.

- We present an integrated cache management framework that improves the execution predictability for real-time multi-core systems. Our designed framework tackles schedule-aware cache management scheme for real-time multi-core systems. The interactions between the task scheduling and the shared cache interference are studied and verified in this framework. Comparing to most of the state-of-art work [33–35] which statically partitioning cache at core level, our framework partition the shared cache in task-level based on the developed dynamic partition cache memory and manage the shared cache resource in an efficient manner. By given design requirement, the proposed framework can automatically generate fully deterministic time-triggered non-preemptive schedule and cache configurations for system performance optimization while preventing deadline misses and cache overflow. Besides, we discuss the design of the back-end for automatic generation of implementation artifacts on target hardware platforms.

- We also investigate system-level power-efficient design for the pipelined real-time multi-core system. We target the streaming application with non-deterministic workload arrivals under hard real-time constraints. By an inverse use of the well-known pay-burst-only-once principle [45], we develop new approaches to solve the energy-minimization problem for pipelined multi-core embedded systems while guarantee the worst-case end-to-end delay. Rather than directly partitioning the end-to-end deadline, we compute for the entire pipeline one service curve which serves as a constraint for the minimal resource demand and formulate an energy minimization problem with respect to the individual resource demands of pipeline stages. Two heuristics, i.e., a quadratic programming heuristic and a fast heuristic are porposed to solve this energy minimization problem. With this approach, we can not only guarantee the overall end-to-end deadline requirement but also retrieve the pay-burst-only-once phenomena, achieving resulting in a significant reduction in both the energy consumption and computing overhead. Moreover, our approaches are scalable with respect to the numbers of pipelined stages.

5.2 Future Work

This thesis presented partial solutions to various resource management problems in real-time multi-core embedded systems. As future work, a few interesting issues and extensions can be studied further:

- In general, not all tasks in embedded system are equally critical for the system [137]. Co-hosting non-safety and safety critical tasks on a common powerful multicore processor is of paramount importance in the embedded system market. In one platform which offer support for multiple applications, the typical case is that some of these tasks will be more critical to the overall welfare of the platform than others. In the mixed-critical real-time systems, co-hosting tasks with different criticalities on the same platform, cache interferences of non-critical tasks will degrade the predictability of critical tasks. Therefore, the cache component in mixed-critical real-time systems should be able to protect critical tasks from cache interferences of non-critical tasks and guarantee the predicable cache performance for critical tasks. Apart from enforcing predictability for critical tasks, achieving the highest possible performance for the non-critical workload is another important optimization object in the cache resource management scheme for mixed-critical real-time systems. As future work, it would be interesting to extent and apply the current dynamic partitioned cache memory for mixed-critical real-time multi-core systems.

- Cache memories are a key target for energy reduction due to its large on-chip area and high access frequency. Several studies [2, 35] show that the energy consumption of the cache subsystem accounts for over 50% of the overall chip. The results presented in Section 2.6.2 also shows that reducing one more cache ways can on average reducing 127 mW power consumption under the specific cache settings. This means turning off cache ways can significantly reduce the power consumption of the system. This brings another potential research direction about how to dynamic manage the cache ways resource to achieve energy efficiency for the cache subsystem. Besides, clock/power gating circuit can be introduced into our cache design and our proposed cache memory also be extended for low power design implementation.

- Thermal issues are not covered in this thesis. However, thermal issues are becoming increasingly important in the design of modern multi-core embedded systems. The power density of the chip has rapidly increased with every new generation of silicon. Power converts to heat and too much heat can destroy a chip beyond use. To avoid overheating, the overall effect is to force the multi-core chip to shut a part of the cores down and move workload from core to core to spread the heat across the chip. This effect will in turn limits the performance scale of the multi-core. As a future work, we would like to pay more attention on temperature optimization for real-time multi-core systems.

Appendix A

List of Publication

The following list summarizes the publications on which this thesis are based. The pertinent chapters of this thesis are given in brackets.

Journal Paper

1. **Gang Chen**, Biao Hu, Kai Huang, Alois Knoll, Di Liu, Todor Stefanov, Feng Li. Reconfigueable Cache for Real-time MPSOCs: Scheduling and Implementation. In Microprocessors and Microsystems (MICPRO), 2016. **(chapter 2)**

2. **Gang Chen**, Kai Huang, Christian Buckl, and Alois Knoll. Applying Pay-burst-only-once Principle for Periodic Power Management in Hard Real-time Pipelined Multiprocessor Systems. In ACM Transactions on Design Automation of Electronic Systems (TODAES), 2015. **(chapter 4)**

Conference Paper

1. **Gang Chen**, Biao Hu, Kai Huang, Alois Knoll, Kai Huang, and Di Liu. Shared L2 Cache Management in Multicore Real-Time System. In 22nd Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2014. **(chapter 2)**

2. **Gang Chen**, Biao Hu, Kai Huang, Alois Knoll, Di Liu, and Todor Stefanov. Automatic cache partitioning and time-triggered scheduling for real-time mpsocs. In 2014 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2014), 2014 **(chapter 2)**

3. **Gang Chen**, Kai Huang, Jia Huang, and Alois Knoll. Cache partitioning and scheduling for energy optimization of real-time mpsocs. In 24th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2013. **(chapter 3)**

4. **Gang Chen**, Kai Huang, Christian Buckl, and Alois Knoll. Energy optimization with worst-case deadline guarantee for pipelined multiprocessor systems. In Design, Automation and Test in Europe (DATE), 2013. **(chapter 4)**

It follows a list of publications that are not fully covered in this thesis.

Journal Paper

1. **Gang Chen**, Kai Huang, Long Cheng, Biao Hu, and Alois Knoll. Dynamic Partitioned Cache Memory for Real-Time MPSoCs with Mixed Criticality. In Journal of Circuits, Systems and Computers (JCSC), 2016.

2. **Gang Chen**, Kai Huang, and Alois Knoll. Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination. In ACM Transactions on Embedded Computing Systems (TECS), 2014.

Conference Paper

1. Biao Hu, Kai Huang, **Gang Chen**, Long Cheng and Alois Knoll. Adaptive Runtime Shaping for Mixed-Criticality Systems. 2015 ACM International Conference on Embedded Software (EMSOFT), 2015. **(Accepted)**

2. Long Cheng, Kai Huang, **Gang Chen**, Alois Knoll and Biao Hu. Evaluation of Runtime Monitoring Methods for Real-Time Event Stream. In 10th IEEE International Symposium on Industrial Embedded Systems (SIES), 2015.

3. Biao Hu, Kai Huang, **Gang Chen**, and Alois Knoll. Evaluation of Runtime Monitoring Methods for Real-Time Event Stream. In 20th Asia and South Pacific Design Automation Conference (ASP-DAC), 2015.

4. **Gang Chen**, Kai Huang, and Alois Knoll. Adaptive Dynamic Power Management for Hard Real-time Pipelined Multiprocessor Systems. In 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014. **(23% acceptance rate)**

5. Di Liu, Jelena Spasic, Jiali Teddy Zhai, Todor Stefanov, and **Gang Chen**. Resource optimization of csdf-modeled streaming applications with latency constraints. In Design, Automation and Test in Europe (DATE), 2014. **(23% acceptance rate)**

6. **Gang Chen**, Kai Huang, and Alois Knoll. Extended abstract: Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination. In 11th IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia), 2013. **(Best Paper Award)**

7. **Gang Chen**, Kai Huang, Jia Huang, Buckl Christian, and Alois Knoll. Effective online power management with adaptive interplay of dvs and dpm for embedded real-time system. In 16th Euromicro International Conference on Digital System Design (DSD), 2013.

8. Kai Huang, Hardik Shah, Karan Savant, Dexin Chen, **Gang Chen**, Sebastian Klose, and Alois Knoll. A lego/fpga-based platform for the education of cyber-physical/embedded systems. In Workshop on Embedded and Cyber-Physical Systems Education (WESE), 2013.

9. Kai Huang, **Gang Chen**, Christian Buckl, and Alois Knoll. Conforming the runtime inputs for hard real-time embedded systems. In Proceedings of the 49th Design Automation Conference (DAC), 2012. **(Top Conference in EDA Field, 22% acceptance rate)**

10. Kai Huang, **Gang Chen**, Nadine Keddis, Michael Geisinger, and Christian Buckl. Demo abstract: An inverted pendulum demonstrator for timed model-based design of embedded systems. In 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems (ICCPS), 2012.

References

[1] Robert Jones. **Modeling and design techniques reduce 90 nm power**. *EE Times*, August 2004. ix, 8, 9

[2] Chuanjun Zhang, Frank Vahid, and Walid Najjar. **A highly configurable cache for low energy embedded systems**. *ACM Transactions on Embedded Computing Systems*, pages 363–387, 2005. ix, 18, 19, 20, 43, 111

[3] S.M. Martin, K. Flautner, T. Mudge, and D. Blaauw. **Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads**. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design (ICCAD)*, 2002. xi, 82, 101

[4] W.X. Wang and P. Mishra. **Leakage-Aware Energy Minimization Using Dynamic Voltage Scaling and Cache Reconfiguration in Real-Time Systems**. In *Proceedings of the 23rd International Conference on VLSI Design (VLSID)*, 2010. xi, 82, 101

[5] M. Lukasiewycz, S. Steinhorst, F. Sagstetter, Wanli Chang, P. Waszecki, M. Kauer, and S. Chakraborty. **Cyber-Physical Systems Design for Electric Vehicles**. In *Proceedings of 2012 15th Euromicro Conference on Digital System Design (DSD)*, pages 477–484, Sept 2012. 1

[6] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2011. 2

[7] Jiayin Li Meikang Qiu. *Real-Time Embedded Systems: Optimization, Synthesis, and Networking*. CRC Press, 2011. 2

[8] Giorgio Buttazzo. **Research Trends in Real-time Computing for Embedded Systems**. *ACM SIGBED Review*, **3**(3):1–10, July 2006. 2

[9] Hardik Shah, Andreas Raabe, and Alois Knoll. **Challenges of WCET Analysis in COTS Multi-core due to Different Levels of Abstraction**. In *Proceedings of 1st Workshop on High-performance and Real-time Embedded Systems (HiRES)*, 2013. 4

[10] JohnA. Stankovic and Krithi Ramamritham. **What is predictability for real-time systems?** *Real-Time Systems*, **2**(4):247–254, 1990. 4

[11] Lothar Thiele and Reinhard Wilhelm. **Design for Timing Predictability**. *Real-Time Systems*, **28**(2-3):157–177, November 2004. 4

[12] **International Technology Roadmap for Semiconductors**. `http://www.itrs.net/reports.html`. 5, 10

[13] P. Gepner and M.F. Kowalik. **Multi-Core Processors: New Way to Achieve High System Performance**. In *Proceedings of 2006 International Symposium on Parallel Computing in Electrical Engineering (PARELEC)*, pages 9–13, Sept 2006. 5, 8

[14] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013. 5

[15] Goojin Jeong, Young-Ugk Kim, Hansu Kim, Young-Jun Kim, and Hun-Joon Sohn. **Prospective materials and applications for Li secondary batteries**. *Energy and Environmental Science*, **4**:1986–2002, 2011. 5

[16] Yen-Kuang Chen, Chaitali Chakrabarti, Shuvra Bhattacharyya, and Bruno Bougard. **Signal processing on platforms with multiple cores: Part 1 - Overview and methodologies**. *IEEE Signal Processing Magazine*, 2009. 6

[17] D. Dasari, B. Akesson, V. Nelis, M.A. Awan, and S.M. Petters. **Identifying the sources of unpredictability in COTS-based multicore systems**. In *Proceedings of 2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013. 6, 7, 8, 40

[18] B.C. Ward, J.L. Herman, C.J. Kenna, and J.H. Anderson. **Making Shared Caches More Predictable on Multicore Platforms**. In *Proceedings of 2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013. 7, 10, 17, 18, 41, 43

[19] Wm. A. Wulf and Sally A. McKee. **Hitting the Memory Wall: Implications of the Obvious**. *ACM SIGARCH Computer Architecture News*, **23**(1):20–24, March 1995. 7

[20] G. Blake, R.G. Dreslinski, and T. Mudge. **A survey of multicore processors**. *IEEE Signal Processing Magazine*, **26**(6):26–37, November 2009. 7

[21] **ARM Cortex-A15 series**. `http://www.arm.com/products`. 7, 16

[22] **OpenSPARC**. `http://www.opensparc.net/`. 7, 16

[23] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara DÃ¶rr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, AmirH. Moin, Jan Reineke, Bernhard Schommer, and Reinhard Wilhelm. **Impact of Resource Sharing on Performance and Performance Prediction: A Survey**. In *Proceedings of 24th Conference on Concurrency Theory (CONCUR)*. 2013. 8, 17, 40, 42, 72

[24] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. **Cache-aware scheduling and analysis for multicores**. In *Proceedings of 2009 ACM International Conference on Embedded Software (EMSOFT)*, 2009. 8, 10, 17, 41, 42

[25] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. **Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors**. *ACM Computing Surveys*, pages 4:1–4:28, 2012. 8, 16, 17, 18, 72

[26] **Intel 22nm Technology**. `http://www.intel.com/silicon-innovations`. 8

[27] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. **Dark Silicon and the End of Multicore Scaling**. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011. 9, 77

[28] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Ngoc Dao, David Patterson, and Krste Asanovic. **A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency while Preserving Responsiveness**. In *Proceedings of 40th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2013. 10, 18, 41, 42

[29] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. **Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems**. In *Proceedings of IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, 2008. 10, 17, 18, 41, 42, 43

[30] Stuart Fisher. **Certifying Applications in a Multi-Core Environment: The World's First Multi-Core Certification to SIL 4**. White paper, SYSGO AG, 2014. 10, 17

[31] Hyoseung Kim, A. Kandhalu, and R. Rajkumar. **A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems**. In *Proceedings of 2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013. 10, 17, 18, 43, 67

[32] Weixun Wang, Prabhat Mishra, and Ann Gordon-Ross. **Dynamic Cache Reconfiguration for Soft Real-Time Systems**. *ACM Transactions on Embedded Computing Systems*, 2012. 10, 42

[33] Xing Fu, K. Kabir, and Xiaorui. **Cache-Aware Utilization Control for Energy Efficiency in Multi-Core Real-Time Systems**. In *Proceedings of 2011 23rd Euromicro Conference on Real-Time Systems (ECRTS)*, 2011. 10, 40, 42, 43, 74, 110

[34] Tiantian Liu, Yingchao Zhao, Minming Li, and Chun Jason Xue. **Joint task assignment and cache partitioning with cache locking for WCET minimization on MPSoC**. *Journal of Parallel and Distributed Computing*, 2011. 10, 40, 42, 43, 74, 110

[35] Weixun Wang, P. Mishra, and S. Ranka. **Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems**. In *Proceedings of 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011. 10, 11, 37, 40, 41, 42, 43, 44, 65, 69, 74, 109, 110, 111

[36] D.H. Albonesi. **Selective cache ways: on-demand cache resource allocation**. In *Proceedings of 1999 32nd Annual International Symposium on Microarchitecture (MICRO)*, pages 248–259, 1999. 11, 15, 19, 37, 43, 109

[37] G. E. Suh, L. Rudolph, and S. Devadas. **Dynamic Partitioning of Shared Cache Memory**. *Journal of Supercomputing*, **28**(1):7–26, 2004. 11, 15, 19, 37, 109

[38] D. Benitez, J.C. Moure, D. Rexachs, and E. Luque. **A reconfigurable cache memory with heterogeneous banks**. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 825–830, 2010. 11, 15, 19, 37, 109

[39] Karthik T. Sundararajan, Timothy M. Jones, and Nigel Topham. **A Reconfigurable Cache Architecture for Energy Efficiency**. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF)*, 2011. 11, 15, 19, 37, 109

[40] S. Mittal, Zhao Zhang, and J.S. Vetter. **FlexiWay: A cache energy saving technique using fine-grained cache reconfiguration**. In *Proceedings of 2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013. 11, 37, 42, 109

[41] K. Huang, L. Santinelli, J.J. Chen, L. Thiele, and G.C. Buttazzo. **Periodic power management schemes for real-time event streams**. In *Proceedings of the 48th IEEE International Conference on Decision and Control (CDC)*, 2009. 13, 79, 81, 85, 89, 101

[42] K. Huang, J.J. Chen, and L. Thiele. **Energy-Efficient Scheduling Algorithms for Periodic Power Management for Real-Time Event Streams**. In *Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2011. 13, 79, 81

[43] L. Thiele, S. Chakraborty, and M. Naedele. **Real-time calculus for scheduling hard real-time systems**. In *Proceedings of the 2000 IEEE International Symposium on Circuits and Systems*, 2000. 13, 81, 84

[44] Lothar Thiele, Ernesto Wandeler, and Nikolay Stoimenov. **Real-time Interfaces for Composing Real-time Systems**. In *Proceedings of International Conference On Embedded Software (EMSOFT)*, pages 34–43, 2006. 13

[45] J.Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001. 13, 78, 79, 81, 84, 87, 93, 110

[46] Seongbeom Kim, Dhruba Chandra, and D. Solihin. **Fair cache sharing and partitioning in a chip multiprocessor architecture**. In *Proceedings of 2004 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 111–122, Sept 2004. 16

[47] Ravi Iyer. **CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms**. In *Proceedings of the 18th Annual International Conference on Supercomputing*, 2004. 16

[48] Gang Chen, Kai Huang, Jia Huang, and Alois Knoll. **Cache Partitioning and Scheduling for Energy Optimization of Real-Time MPSoCs**. In *Proceedings of 24th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, June 2013. 17, 24

[49] Gang Chen, Biao Hu, Kai Huang, Alois Knoll, Kai Huang, and Di Liu. **Shared L2 Cache Management in Multicore Real-Time System**. In *Proceedings of 22nd Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014. 17, 40

[50] Sangyeun Cho and Lei Jin. **Managing Distributed, Shared L2 Caches Through OS-Level Page Allocation**. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006. 17

[51] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. **Avoiding Conflict Misses Dynamically in Large Direct-mapped Caches**. *ACM SIGOPS Operating Systems Review*, pages 158–170, 1994. 17

[52] Wang Jing and Rui Fan. **The research of Hibernate cache technique and application of EhCache component**. In *Proceedings of 2011 IEEE 3rd International Conference on Communication Software and Networks (ICCSN)*, pages 160–162, 2011. 17

[53] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. **Enigma: Architectural and Operating System Support for Reducing the Impact of Address**

Translation. In *Proceedings of 2010 24th ACM International Conference on Supercomputing (ICS)*, 2010. 17

[54] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. **Real-time cache management framework for multi-core architectures**. In *Proceedings of 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013. 18

[55] N. Suzuki, Hyoseung Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. **Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses**. In *Proceedings of 2013 IEEE 16th International Conference on Computational Science and Engineering (ICESS)*, 2013. 18, 43

[56] A.D.S. Gil, J.I.B. Benitez, M.H. Calvino, and E.H. Gomez. **Reconfigurable Cache Implemented on an FPGA**. In *Proceedings of 2010 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 250–255, Dec 2010. 18, 19, 20

[57] AD. Santana Gil, F.J. Quiles Latorre, M. Hernandez Calvino, E. Herruzo Gomez, and J.I Benavides Benitez. **Optimizing the physical implementation of a reconfigurable cache**. In *Proceedings of 2012 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2012. 18, 19, 20

[58] Afzal Malik, Bill Moyer, and Dan Cermak. **A Low Power Unified Cache Architecture Providing Power and Performance Flexibility**. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED)*, pages 241–243, 2000. 18, 19, 20

[59] Gang Chen, Biao Hu, Kai Huang, Alois Knoll, Di Liu, and Todor Stefanov. **Automatic Cache Partitioning and Time-triggered Scheduling for Real-time MPSoCs**. In *Proceedings of the 2014 9th International Conference on Reconfigurable Computing and FPGAs (ReConfig)*, 2014. 20, 40

[60] **Creating Multiprocessor Nios Systems Tutorial**. `http://www.altera.com`. 21

[61] Arthur Pyka, Mathias Rohde, and Sascha Uhrig. **A Real-Time Capable First-level Cache for Multi-cores**. In *Proceedings of 2013 1st Workshop on High-performance and Real-time Embedded Systems (HiRES)*, 2013. 21, 73

[62] **Adapteva Parallella**. http://www.adapteva.com/parallella/. 22

[63] Jianwei Dai and Lei Wang. **An Energy-efficient L2 Cache Architecture Using Way Tag Information Under Write-through Policy**. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **21**(1):102–112, January 2013. 24

[64] Poonacha Kongetira, K. Aingaran, and K. Olukotun. **Niagara: a 32-way multithreaded Sparc processor**. *IEEE Micro*, **25**(2):21–29, March 2005. 25

[65] Daniel Henderson Jim Mitchell and George Ahrens. **IBM POWER5 Processor-based Servers: A Highly Available Design for Business-Critical Applications**. White paper, IBM, 2005. 25

[66] Nhon Quach. **High Availability and Reliability in the Itanium Processor**. *IEEE Micro*, **20**(5):61–69, September 2000. 25

[67] Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. **FIFO Cache Analysis for WCET Estimation: A Quantitative Approach**. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013. 26

[68] Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. **Hardware Support for WCET Analysis of Hard Real-time Multicore Systems**. In *Proceedings of 2009 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009. 26

[69] **Synopsys Design Compilers**. http://www.synopsys.com. 30, 31

[70] **Semiconductor Manufacturing International Corporation**. http://www.smics.com. 30, 31

[71] **ARM Artisan Physical IP Solutions**. http://www.artisan.com. 31

[72] Martin Lukasiewycz, Sebastian Steinhorst, Florian Sagstetter, Wanli Chang, Peter Waszecki, Matthias Kauer, and Samarjit Chakraborty. **Cyber-Physical Systems Design for Electric Vehicles**. In *Proceedings of 2012 Euromicro Conference on Digital System Design (DSD)*, 2012. 39, 44

[73] Junqing Wei, J.M. Snider, Junsung Kim, J.M. Dolan, R. Rajkumar, and B. Litkouhi. **Towards a viable autonomous driving research platform**. In *Proceedings of 2013 IEEE Intelligent Vehicles Symposium (IV)*, 2013. 40

[74] M.K. Qureshi et al. **Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches**. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006. 42

[75] D. Sanchez et al. **Vantage: Scalable and efficient fine-grain cache partitioning**. In *Proceedings of 2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011. 42

[76] Karthik T. Sundararajan, Timothy M. Jones, and Nigel P. Topham. **Energy-efficient Cache Partitioning for Future CMPs**. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 465–466, 2012. 42

[77] C.M. Patrick, R. Garg, Seung Woo Son, and M. Kandemir. **Improving I/O performance using soft-QoS-based dynamic storage cache partitioning**. In *Proceedings of 2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, Aug 2009. 42

[78] Bach D. Bui, Marco Caccamo, Lui Sha, and Joseph Martinez. **Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems**. In *Proceedings of 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008. 42

[79] Rakesh Reddy and Peter Petrov. **Cache partitioning for energy-efficient and interference-free embedded multitasking**. *ACM Transactions on Embedded Computing Systems*, March 2010. 42

[80] Andrew Wolfe. **Software-based Cache Partitioning for Real-time Applications**. *Journal of Computer and Software Engineering*, pages 315–327, 1994. 43

[81] Frank Mueller. **Compiler Support for Software-Based Cache Partitioning**. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1995. 43

[82] C.E. Lin, Hung-Ming Yen, and Yu-Shang Lin. **Development of Time Triggered hybrid data bus System for small aircraft digital avionic system**. In *Proceedings of IEEE/AIAA 26th Digital Avionics Systems Conference (DASC)*, 2007. 44

[83] S. Baruah and G. Fohler. **Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems**. In *Proceedings of 2011 IEEE 32nd Real-Time Systems Symposium (RTSS)*, 2011. 44

[84] F. Sagstetter, M. Lukasiewycz, and S. Chakraborty. **Schedule integration for time-triggered systems**. In *Proceedings of 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013. 44

[85] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. **Time-triggered implementations of mixed-criticality automotive software**. In *Proceedings of the 15th Conference for Design, Automation and Test in Europe (DATE)*, 2012. 44

[86] Truong Nghiem, George J. Pappas, Rajeev Alur, and Antoine Girard. **Time-Triggered Implementations of Dynamic Controllers**. *ACM Transactions on Embedded Computing Systems (TECS)*, pages 58:1–58:24, 2012. 44, 54

[87] Truong Nghiem, George J. Pappas, Rajeev Alur, and Antoine Girard. **Time-triggered Implementations of Dynamic Controllers**. In *Proceedings of the 6th ACM/IEEE International Conference on Embedded Software (EMSOFT)*, 2006. 44

[88] Jia Huang, J.O. Blech, A. Raabe, C. Buckl, and A. Knoll. **Static scheduling of a Time-Triggered Network-on-Chip based on SMT solving**. In *Proceedings of the 15th Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012. 44

[89] A.K. Gendy and M.J. Pont. **Automatically Configuring Time-Triggered Schedulers for Use With Resource-Constrained, Single-Processor Embedded Systems**. *IEEE Transactions on Industrial Informatics*, pages 37–46, 2008. 44, 56

[90] Nan Guan, Wang Yi, Zonghua Gu, Qingxu Deng, and Ge Yu. **New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms**. In *Proceedings of 2008 Real-Time Systems Symposium (RTSS)*, 2008. 46

[91] 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 *Proceedings of 2001 IEEE International Workshop on Workload Characterization (WWC)*, 2001. 46, 64

[92] **SimpleScalar LLC**. http://www.simplescalar.com. 46

[93] Michael J. Pont. *Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers*. Addison-Wesley Professional, 2001. 54, 69

[94] Richard Barry. *Using the FreeRTOS Real Time Kernel*. Real Time Engineers Ltd, 2010. 57

[95] Jean J Labrosse. *uC/OS-III: The Real-Time Kerne*. Micrium Press, 2009. 57

[96] **Creating a System With Qsys**. https://www.altera.com/. 63

[97] **IBM ILOG CPLEX**. http://www.ibm.com/software/. 64

[98] **CHStone**. http://www.ertl.jp/chstone/. 64

[99] **DSPstone**. http://www.ice.rwth-aachen.de/. 64

[100] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011. 65

[101] **UTDSP**. http://www.eecg.toronto.edu/UTDSP.html/. 65

[102] **Versabench**. http://groups.csail.mit.edu/versabench. 65

[103] H. Nikolov et al. **Systematic and Automated Multiprocessor System Design, Programming, and Implementation**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 542–555, 2008. 65

[104] **Malardalen Real-Time Research Center**. http://www.es.mdh.se/. 65

[105] S. Carta, A. Alimonda, A. Pisano, A. Acquaviva, and L. Benini. **A control theoretic approach to energy-efficient pipelined computation in MPSoCs**. *ACM Transactions on Embedded Computing Systems*, 2007. 78, 80

[106] Y. Yu and V.K. Prasanna. **Power-aware resource allocation for independent tasks in heterogeneous real-time systems**. In *Proceedings of 9th International Conference on Parallel and Distributed Systems*, 2002. 78, 80

[107] R.B. Xu, R. Melhem, and D. Mosse. **Energy-Aware Scheduling for Streaming Applications on Chip Multiprocessors**. In *Proceedings of 28th IEEE International Real-Time Systems Symposium*, 2007. 78

[108] H. Javaid, M. Shafique, S. Parameswaran, and J. Henkel. **Low-power adaptive pipelined MPSoCs for multimedia: An H.264 video encoder case study**. In *Proceedings of 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011. 78, 80

[109] S.L. Shee and S. Parameswaran. **Design methodology for pipelined heterogeneous multiprocessor system**. In *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007. 80

[110] H. Javaid and S. Parameswaran. **A design flow for application specific heterogeneous pipelined multiprocessor systems**. In *Proceedings of 2009 46th ACM/IEEE annual Design Automation Conference (DAC)*, 2009. 80

[111] S.L. Shee, A. Erdos, and S. Parameswaran. **Heterogeneous multiprocessor implementations for JPEG: a case study**. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, 2006. 80

[112] I. Karkowski and H. Corporaal. **Design of heterogenous multi-processor embedded systems: applying functional pipelining**. In *Proceedings of 1997 International Conference on Parallel Architectures and Compilation Techniques*, 1997. 80

[113] A. Alimonda, S. Carta, A. Acquaviva, A. Pisano, and L. Benini. **A Feedback-Based Approach to DVFS in Data-Flow Applications**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2009. 80

[114] H. Javaid, M. Shafique, J. Henkel, and S. Parameswaran. **System-level application-aware dynamic power management in adaptive pipelined MPSoCs for multimedia**. In *Proceedings of 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, nov. 2011. 80

[115] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. **Period Optimization for Hard Real-time Distributed Automotive**

Systems. In *Proceedings of 44th ACM/IEEE Design Automation Conference (DAC)*, 2007. 80

[116] S.Y. Hong, T. Chantem, and X.S. Hu. **Meeting End-to-End Deadlines through Distributed Local Deadline Assignments**. In *Proceedings of 2011 IEEE 32nd Real-Time Systems Symposium (RTSS)*, 2011. 80

[117] P. de Langen and B. Juurlink. **Leakage-aware multiprocessor scheduling for low power**. In *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006. 80

[118] P. de Langen and B. Juurlink. **Leakage-Aware Multiprocessor Scheduling**. *Journal of Signal Processing Systems*, 2009. 80, 82

[119] D. Liu, J. Spasic, J.T. Zhai, T. Stefanov, and G. Chen. **Resource Optimization of CSDF-modeled Streaming Applications with Latency Constraints**. In *Proceedings of Design, Automation and Test in Europe (DATE)*, March 2014. 80

[120] S. Maxiaguine, A. Chakraborty and L. Thiele. **DVS for buffer-constrained architectures with predictable QoS-energy tradeoffs**. In *Proceedings of the 2005 IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2005. 81, 85

[121] J.J. Chen, N. Stoimenov, and L. Thiele. **Feasibility Analysis of On-Line DVS Algorithms for Scheduling Arbitrary Event Streams**. In *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009. 81, 85

[122] F. Yao, A. Demers, and S. Shenker. **A scheduling model for reduced CPU energy**. In *Proceedings of 36th Annual Symposium on Foundations of Computer Science*, 1995. 81

[123] S. Perathoner, K. Lampka, N. Stoimenov, L. Thiele, and J.J. Chen. **Combining optimistic and pessimistic DVS scheduling: An adaptive scheme and analysis**. In *Proceedings of the 2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2010. 81

[124] K. Huang, L. Santinelli, J.J. Chen, L. Thiele, and G.C. Buttazzo. **Adaptive Dynamic Power Management for Hard Real-Time Systems**. In *Proceedings of 2009 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009. 81, 85

[125] K. Huang, L. Santinelli, J.J. Chen, L. Thiele, and G.C. Buttazzo. **Applying real-time interface and calculus for dynamic power management in hard real-time systems**. *Real-Time Systems*, 2011. 81

[126] K. Lampka, K. Huang, and J.J. Chen. **Dynamic counters and the efficient and effective online power management of embedded real-time systems**. In *Proceedings of the 7th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, 2011. 81, 85

[127] H. Nikolov, T. Stefanov, and E. Deprettere. **Systematic and Automated Multiprocessor System Design, Programming, and Implementation**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, **27**(3):542–555, March 2008. 82

[128] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. **System Architecture Evaluation Using Modular Performance Analysis - A Case Study**. *International Journal on Software Tools for Technology Transfer (STTT)*, 2006. 82

[129] R. Jejurikar, C. Pereira, and R. Gupta. **Leakage aware dynamic voltage scaling for real-time embedded systems**. In *Proceedings of 2004 41st ACM/IEEE Design Automation Conference (DAC)*, 2004. 82, 101

[130] H. Oh and S. Ha. **Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints**. In *Proceedings of 10th International Symposium on Hardware/Software Codesign (CODES+ISSS)*, 2002. 84, 102

[131] M. Fidler. **Extending the Network Calculus Pay Bursts Only Once Principle to Aggregate Scheduling**. In *Quality of Service in Multiservice IP Networks*. 2003. 86

[132] V. Jeyakumar, A.M. Rubinov, and Z.Y. Wu. **Sufficient Global Optimality Conditions for Non-convex Quadratic Minimization Problems With Box Constraints**. *Journal of Global Optimization*, 2006. 96

[133] M. Fu, Z. Luo, and Y. Ye. **Approximation Algorithms for Quadratic Programming**. *Journal of Combinatorial Optimization*, 1998. 96

[134] J. Chen and S. Burer. **Globally solving nonconvex quadratic programming problems via completely positive programming**. *Mathematical Programming Computation*, 2012. 96, 101

[135] Ernesto Wandeler and Lothar Thiele. **Real-Time Calculus (RTC) Toolbox**. `http://www.mpa.ethz.ch/Rtctoolbox`, 2006. 101

[136] W. Thies and S. Amarasinghe. **An empirical characterization of stream programs and its implications for language and compiler design**. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*. 102

[137] S. Baruah et al. **Scheduling Real-Time Mixed-Criticality Jobs**. *IEEE Transactions on Computers*, 2012. 111