Doctoral Dissertation

Multicore Task Scheduling and High-Level Synthesis for Embedded Systems

March 2022

Doctoral Program in Advanced Electrical,
Electronic and Computer Systems
Graduate School of Science and Engineering
Ritsumeikan University

NISHIKAWA Hiroki

Doctoral Dissertation Reviewed
by Ritsumeikan University

Multicore Task Scheduling and High-Level Synthesis for Embedded Systems

(組込みシステム設計における
マルチコアタスクスケジューリングと
高位合成)

March 2022
2022年3月

Doctoral Program in Advanced Electrical, Electronic and Computer Systems
Graduate School of Science and Engineering
Ritsumeikan University
立命館大学大学院理工学研究科
電子システム専攻博士課程後期課程

NISHIKAWA Hiroki
西川　広記

Surpervisor: Professor TOMIYAMA Hiroyuki
研究指導教員: 冨山　宏之教授

Abstract

This thesis addresses scheduling techniques of data-parallel tasks and of function-level module sharing for embedded systems. Embedded systems are generally designed for a specific and dedicated application under a variety of constraints according to the requirements of the application. In many cases, there are requirements for performance, price, circuit size, power and energy consumption between hardware and software, and it is necessary to choose the best design method in order to satisfy these requirements for achievement of the best performance exploited from both hardware and software. On the other hand, even though the performance requirements of applications are increasing, the performance improvement of a single core is approaching the limit according to the Dennard scaling law. Multicore architectures have attracted much attention to exploit the performance of systems. Multicore architectures allow a single application to be executed on multiple cores in parallel, achieving higher performance than single-core architectures. This trend has led to the emergence of multi-core systems, including dual- and quad-core systems from IBM, AMD, Intel, ARM, NVIDIA, and others, which have contributed to higher performance not only in embedded systems but also in mainstream computers such as general-purpose servers. However, as the increase in the number of cores, system-design processes become very complex, and further development of parallel computing and design automation technologies is prospective.

Task scheduling is one of system-design processes that determines the execution order of the tasks in an application and the assignment of the tasks on one of the cores. Classical task scheduling assumes that a task is executed on one of the cores, and different tasks can run on different cores in parallel in such a way that the overall schedule length is minimized. On the other hand, this thesis allows each task to run on multiple cores in parallel, and the number of cores assigned to a task is determined during task scheduling simultaneously. Such tasks are generally called as data-parallel tasks, which has recently attracted attention to exploit the parallelism inherent of the tasks (data-parallelism) in many applications and to make efficient use of multicores, and scheduling techniques for the tasks have been excessively expected in parallel computing. This thesis proposes a technique to rapidly find a good schedule based on constraint programming. The experiments demonstrate that our proposed technique reduces the schedule length by up to 81.9%. The proposed technique is further extended towards two directions. One direction is to consider the performance overheads for communications among cores. In this thesis, we propose a communication-aware scheduling technique. This work determines the execution order and the assignment of the tasks, and also schedule the communications among the cores such that the overall schedule length is minimized with mitigating the performance overheads for the communications. The experiments demonstrate that the proposed technique reduces the schedule length by 22.5% on average compared to the state-of-the-art techniques. The other direction is to consider energy efficiency that is one of the most demanding factors in embedded systems. This work targets heterogeneous multicores that consist of two types of cores, where one is high-performance but high-power cores and the other is low-power but low-performance cores. In this work, the proposed technique determines an optimal type for each core at the same time as scheduling of the tasks. In the experimental results, the proposed technique reduces energy consumption by up to 12.4% compared with the state-of-the-art techniques.

Regarding efficient hardware design, high-level synthesis (HLS) that automatically translates software programs into hardware descriptions has now become an indispensable technology. HLS techniques can quickly generate circuits that satisfy design constraints, however unfortunately, the circuit area becomes often larger than that of manually designed circuits. One of the reasons is that common HLS tools basically create multiple instances of a same hardware module when a function is called multiple times in the original software program. Our proposed HLS techniques generate only a single instance of the module which is shared in a time-division manner. The experiments using FPGA shows that the proposed techniques reduce the number of look-up tables by up to 14.9% and flip-flops by up to 19.1%.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and Motivation

Nowadays, there is still increasing demand for better digital electronic systems. Such systems drive the necessity to satisfy even more stringent requirements than ever. In particular, in the domain of embedded systems, the simultaneous optimization of many design metrics such as performance, cost, size, and the energy consumption is so significant that many pieces of research have been studied to solve the issues. Systems-on-a-chip (SoC) has been traditionally adopted to deal with such aforementioned issues since they have achieved things in terms of a significant reduction in energy consumption, chip size, and manufacturing cost due to its high performance per footprint. Unfortunately, however, this pursuit has resulted in the adoption of very complex on-chip architectures, even as growing heterogeneously and/or parallel SoCs, causing prolonging time-to-market, and increasing manufacturing cost and chip size.

As illustrated in Figure 1.1, the number of transistors, thread performance, and clock frequency have been predicted to flatten by 2025 [2]. In other words, the improvement of a single core performance can be no longer possible in the absence of Moore's Law. In addition, embedded software programs have become tremendously complicated as well as SoCs' on-chip design in the context of the emerging Internet of things (IoT)-oriented software applications, artificial intelligence, etc. An important turning point has this circumstance led to reaching for an effective methodology that covers both hardware/software co-design.

An embedded system is different from a general-purpose computer in that the embedded systems are designed to satisfy a variety of requirements such as timing and resource constraints. Mostly, embedded systems are desire to work with low power but prefer if they are small and fast. In addition, the design term of embedded systems should be shortened to meet the time-to-market requirement. From this perspective,

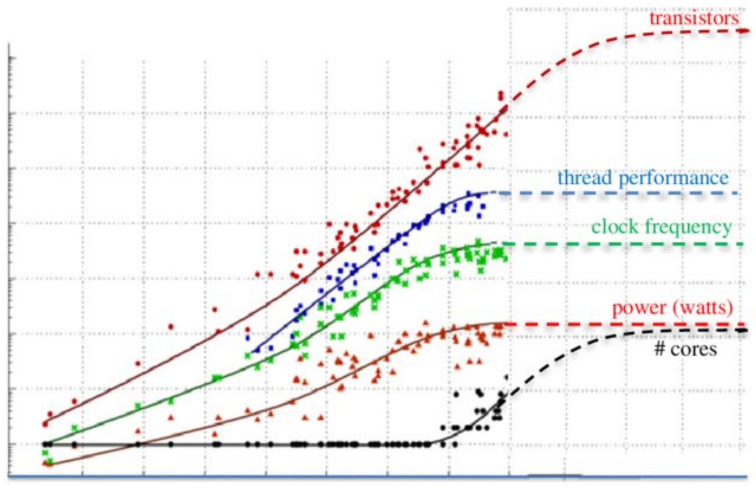


Figure 1.1: Performance stagnation by the end of Dennard scaling [1]

therefore, the efficiency of system design is becoming more and more crucial.

Today, the design of embedded systems and SoC devices is based on multiple processors. Multiprocessor system-on-chip (MPSoCs), which execute concurrent processing on multiple processors, is one of the mainstream in embedded systems to achieve high performance [3, 4]. In addition, tasks of recent modern applications have inherent data parallelism, and which is called as data-parallel tasks [5]. The systems are required to efficiently exploit both task parallelism (inter-task parallelism) and data parallelism (intra-task parallelism). On the other hand, however, system complexities have been growing at an almost exponential rate by the introduction of MPSoC. This has led to the degradation of productivity by the disparity between the required speed of system design and design complexity. One of the major solutions to fill the productivity gap is to raise the level of abstraction in the design process. The purpose of the abstraction is to fully automate the system design process as much as possible.

1.2 Contribution of This Work

This section describes the main contribution of this thesis. There are four contributions, described in the following paragraphs. These contributions address the overall thesis goal and the four specific goals are to explore co-synthesis and hardware synthesis to design embedded systems. Let us briefly look at an overview of the contributions in this thesis, as shown in the following.

- *Contribution 1*: This thesis addresses scheduling of data-parallel tasks on multi-core based on constraint programming (CP) paradigms.
- *Contribution 2*: In addition, the work extends to data-parallel tasks scheduling

with inter- and intra-task communications.

- *Contribution 3*: For energy-efficiency, this thesis presents scheduling of data-parallel tasks on heterogeneous multicores, and extends to simultaneous techniques of scheduling and core-type optimization.

- *Contribution 4*: Finally, this thesis proposes function-level module sharing techniques in high-level synthesis.

Increasing number of cores in embedded devices, multicore task scheduling, which determines the order in which tasks are executed on many cores, has been identified as one of the most important technologies on embedded systems[6, 7]. Multicore task scheduling is classified to NP-hard problems, an optimal solution can hardly be found in practical time. In the literature, many heuristics have been proposed. In order to evaluate such solutions, however, an optimal solution is necessary to be asked once in advance. Many studies that address multicore task scheduling have been proposed for the optimal solution, but most of them have been developed with ILP-based techniques that are very time-consuming to find an optimal solution. This work proposes CP-based scheduling techniques to more quickly find an optimal solution than the state-of-the-art techniques, which has led to contribution 1.

In parallel computing on embedded systems, communications among multicores significantly lead to performance degradation. Inter-task communication such as passing messages, shared memory access, synchronization, etc. frequently occurs as a non-negligible amount of latency [8]. Nevertheless, there have been few works that propose scheduling problems take into account communications on multicore architecture. Even though moldable task scheduling has been studied for the last couple of decades, to the best of our knowledge, there does not exist works that take into account both inter-task and intra-task communication overheads. This thesis extends to multicore task scheduling by taking into account inter-and intra-task communications, which has led to contribution 2.

While pursuing high-performance computing, energy consumption is another important design criterion embedded devices. Low power design techniques have been excessively developed over the last decades. On MPSoCs, there have appeared heterogeneous architectures that have several different types of cores, but system design becomes far more complex than homogeneous architectures [9, 10, 11]. Therefore, the techniques for efficient design of heterogeneous architectures are attracting attention to deal with the issue. Such techniques, however, focus mainly on the optimization of a hardware component in isolation, and there are few techniques for co-design between hardware and software. This work associates with task scheduling and architecture design on heterogeneous multicores. In the past, the techniques for scheduling on heterogeneous multicores have been investigated and a major portion of the works still assumes that a task is assigned a single core [12, 13, 14, 15]. There are, however, few studies that are focused on scheduling multi-threaded tasks on heterogeneous multicores. Furthermore, it extends to an integrated framework that addresses both data-parallel task scheduling and core-type optimization. Through this technique, scheduling and core-type of cores on architecture are simultaneously determined such that the overall energy consumption is minimized, and which is led to contribution 3.

The final contribution, which is referred to as hardware synthesis, represents the techniques in high-level synthesis. To shorten the design time of hardware, there have developed many high-level synthesis tools for a decade. However, such tools can hardly produce a more efficient circuit than manually designed ones. One reason is that high-level synthesis tools often generate multiple instances from a module if a function is invoked more than two times. This thesis attempts to solve this issue and proposes function-level module sharing techniques. The proposed techniques can generate one instance of a shared module even if the shared function is invoked multiple times at different times, which has led to contribution 4.

1.3 Overview of Chapters

The remainder of this thesis is organized as follows. Chapter 2 describes a common design flow of embedded systems and specifies the topics for scheduling and high-level synthesis addressed in this thesis. Furthermore, the overview of the main topics that are addressed in this thesis is introduced in brief.

Chapter 3 to Chapter 6 addresses task scheduling problems. In particular, the scheduling problems addressed in this thesis are classified into multicore task scheduling, and Chapter 7 refers to high-level synthesis in hardware design. In Chapter 3, scheduling problems for data-parallel tasks on homogeneous multicore based on CP are addressed. In this chapter, CP which is one of the recent programming paradigms is adopted to quickly solve the scheduling problem, compared to the state-of-the-art techniques. Furthermore, it extends to address another scheduling problem of data-parallel tasks, which is generalized from the aforementioned ones.

Chapter 4 extends the work in Chapter 3 to scheduling of data-parallel tasks taking into account inter-and intra-task communications. In the real world, embedded systems need to compute with considering communication resources such as data bus, network-on-a-chip, and so on. Such communication overheads may degrade the performance of the systems. This chapter demonstrates that the proposed scheduling techniques with the communication overheads can find an schedule in such a way the overall completion time is minimized.

Chapter 5 presents the scheduling of data-parallel tasks on heterogeneous multicores. Energy-efficient embedded system design has been one of the most crucial topics for several decades. Heterogeneous computing refers to system environments with more than two kinds of computing resources. In order to achieve a low-power design without degrading the performance of the systems, this chapter addresses simultaneous scheduling and core-type optimization techniques in heterogeneous multicores environments.

In Chapter 6, this thesis moves the topic into hardware design. This chapter especially addresses high-level synthesis (HLS) techniques for hardware synthesis. In general, hardware designs are more efficient from the perspective of performance and power consumption than software implementations. However, the design of the hardware is still regarded as a more time-consuming process. HLS is one of the promising techniques to efficiently design hardware, and this chapter presents function-level module sharing techniques. Finally, Chapter 7 summarizes this thesis and refers to the extensions for future work.

Chapter 2

Design of Embedded Systems

2.1 Introduction

The ever-progressing semiconductor processing technique has remarkably increased the number of transistors on a single chip, which leads to efficient and powerful embedded processors. With the increasing functionality and performance of electric systems due to the high degree of integration of LSIs, embedded systems have expanded their application from conventional control devices to information devices such as smartphones and IoT terminals. Highly functional information processing equipment requires diverse functions such as multimedia such as video and audio, sensor processing, and connecting clouds through a network connection to the Internet. In order to meet the diverse requirements, the scale of software on embedded devices tends to expand and become more complex every year. In addition, the shortening of development time and cost reduction due to the expansion of the market size requires developers to produce software efficiently in a short time. From the industrial perspective, most digital functions are implemented by software programs for cost reduction and less power consumption compared to the hardware as shown in Figure 2.1. Furthermore, the performance of a single processor still grows and recent systems employ multicore architecture so that software programs can rapidly run in parallel.

Nonetheless of the improvement of software processors, however, they are often failed to meet the strict performance constraints of embedded systems. Embedded systems usually have to meet their performance constraints; therefore, part of the systems are developed on hardware such as application-specific hardware (ASICs) and field-programmable gate arrays (FPGAs), and currently, graphical processing units (GPUs) are also in fashion. The performance constraints mean the latency to execute a given set of tasks. The hardware performance is dependent on the operation scheduling result, on the other hand, the software performance is determined as the number of clock cycles

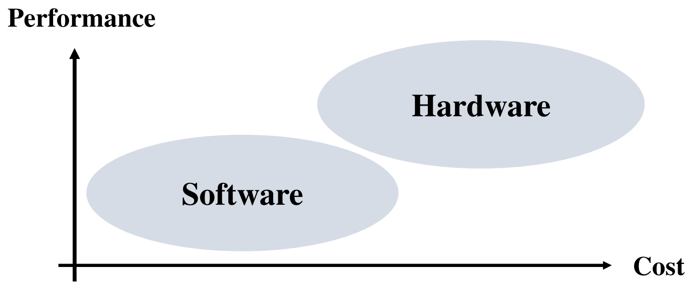


Figure 2.1: Cost and performance trade-off between hardware and software

that takes the processor to execute the tasks. That is why the cycle-per-instruction (CPI) must be a critical metric to evaluate the performance of the processor as software performance. In such literature, efficient embedded system design requires optimization in both hardware and software areas of the application. The common methodology is usually down for the design steps that bring a product idea down to its physical realization, and which is often called a system-level design flow.

2.2 Design of Embedded Systems

This section addresses basic system-level design flow of embedded systems. In this thesis, top-down design process is as shown in the Figure 2.2.

It is characterised by three important components such as *Specification*, *Co-Synthesis*, and *Hardware and Software-Synthesis*. In the following subsections, each step in design flow of embedded systems addressed previously will be explained in detail.

2.2.1 System Specification

The first process specification starts driving with ideas from the designer's brain. Designers have to informally describe what is required for the target embedded systems in a natural language such as English, Spanish, etc. This process is usually called as *requirements*, which is the highest level in system-level design. These requirements are not limited to functional requirements, and the non-functional requirements such as performance, cost, and size should be captured in a specification document [16]. Then, the specification of the functionality for a system can be captured with various specification models [17] such as high-level languages, and more abstract models can be introduced such as task graphs, block diagrams, finite state machines (FSMs), and Petri nets. In this thesis, the task graph model will be employed to represent the system model corresponding to the work.

The functionality equipped in a system with stream data operations can be expressed

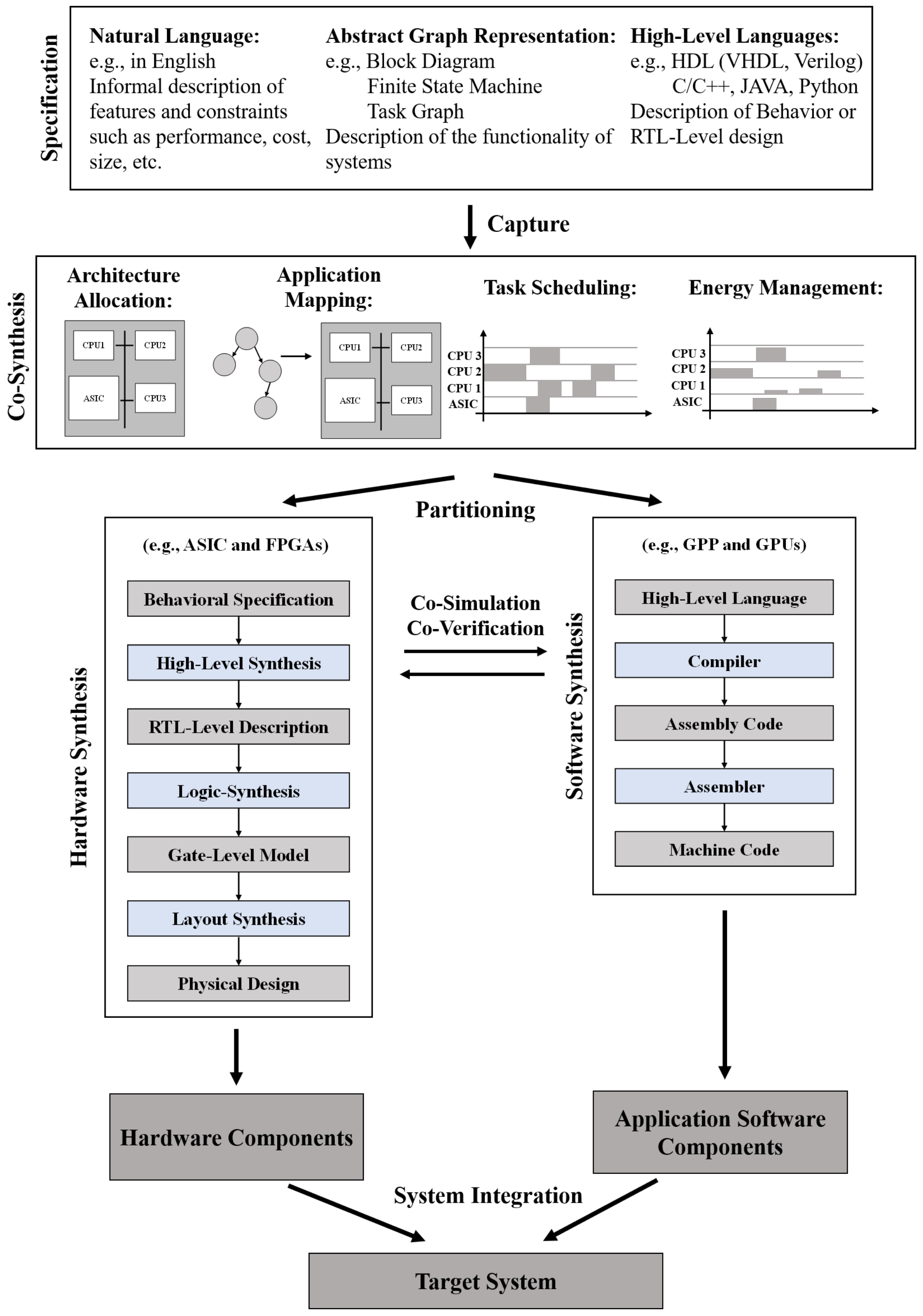
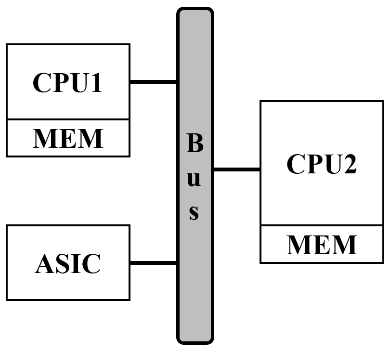
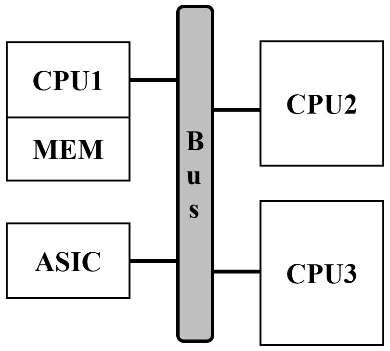


Figure 2.2: Traditional design flow of embedded system

as a graph. In particular, one of the general ways represents a directed acyclic graph (DAG) $G = (V, E)$, where V denotes a set of tasks that should be executed, and E denotes a set of directed edges which refers to flow dependency of communications between the tasks. The task graph can be generated by a variety of techniques such as



(a) Power consumption: 560mW, Production cost: $110



(b) Power consumption: 410mW, Production cost: $230

Figure 2.3: Architecture specification

[18]. According to the specification, each task or a whole of the tasks holds a deadline τ, by which the execution must be finished. In addition, an application in embedded systems usually inherits a repetition period θ that represents the maximum delay between activation of the source tasks.

2.2.2 Co-Synthesis

In *Co-Synthesis*, based on the task graph captured in the previous step, this step consists of the following fourfold: architecture allocation, application mapping, task scheduling in the application, and management of energy consumption.

In the first step, *Architecture allocation*, designers need to determine the system components (i.e., processing elements and communication links) to satisfy the requirements addressed beforehand. There are too many target architectures to determine appropriate ones that can use for the implementation of the desired functionality.

Figure 2.3 indicates a selection problem to determine the appropriate architecture. The goal of this process is to identify the architecture that is the most suitable to the specification. The optimal architecture must provide sufficient performance for the application to meet delivery constraints. In addition, the cost, design time, energy consumption should be reduced as long as possible. In this example, one has two CPUs

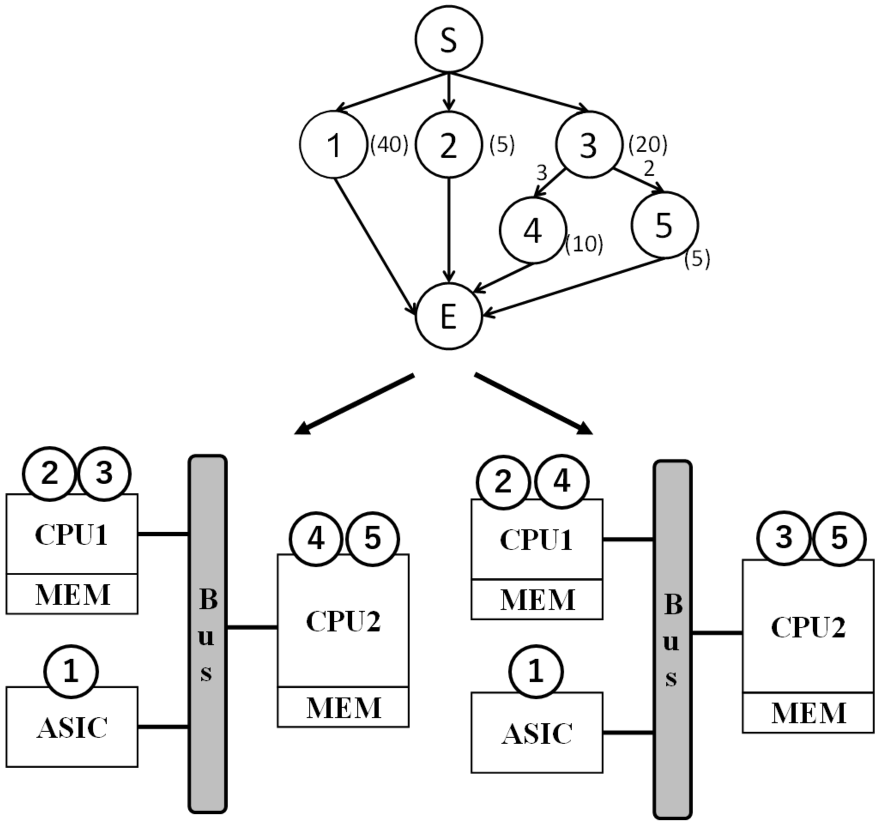


Figure 2.4: Mapping of tasks onto different processing elements

(i.e., CPU 1 and CPU 2) and an ASIC are connected with an internal bus as shown in Figure 2.3(a) and the other has, unlike the former ones, an additional CPU that is implemented in the architecture as shown in Figure 2.3(b). If software implementations are assigned to general-purpose processors such as CPU, they can be flexible and the cost can be reduced to realize than hardware designs. On the other hand, the ASIC can provide higher performance and better energy efficiency than general-purpose processors. Note that ASIC is aimed and designed to process a specific application so that there appears an issue of time-to-market by designing the specific hardware. While balancing between the trade-offs among power consumption, production cost, and design time, and so on, the system designers are required to determine an appropriate architecture design. To efficiently explore an appropriate architecture design, system-level co-synthesis tools are aided to the designers.

Application Mapping

When the architecture specification has been done, the next step is called *Application mapping*. This process play a role of task and communication allocation onto both processing elements and links to communicate on the architecture.

Figure 2.4 indicates two different mappings that tasks in an application are allocated onto identical target architectures. The application is represented as DAG, which is referred to the previous subsection, then the nodes represent tasks and the edges represent precedence dependency among the tasks, respectively [19]. The nodes labeled *S* and *E* are namely dummy tasks, which do not have a computational payload. In addition, the number with parenthesis beside a node represents the execution time of the task.

This example shows two cases where tasks are mapped to the same processing unit in a given architecture. In both figures, Task 1 is determined to be executed on the ASIC. In the left mapping, Tasks 2 and 3 are assigned to CPU 1, while tasks 4 and 5 are assigned to CPU 2. On the other hand, the right figure shows that Task 4 is assigned to CPU 1 instead of CPU 2, and Task 3 is allocated onto CPU 2. Mapping determines whether a task is implemented in either hardware or software. Therefore, the partitioning is mainly conducted in this process [20].

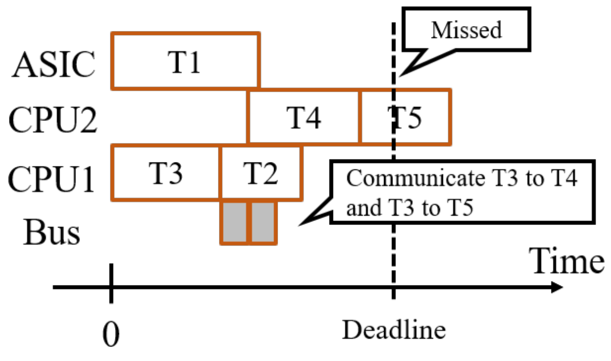
Implementation of Hardware is more efficient from the perspectives of performance and power consumption than the software implementation. In contrast, hardware design is very time-consuming compared with software design. In this context, determining a good solution of mapping is required for a good system design. Without appropriately distributing those activities on the architecture, the performance cannot avoid degradation.

Task Scheduling

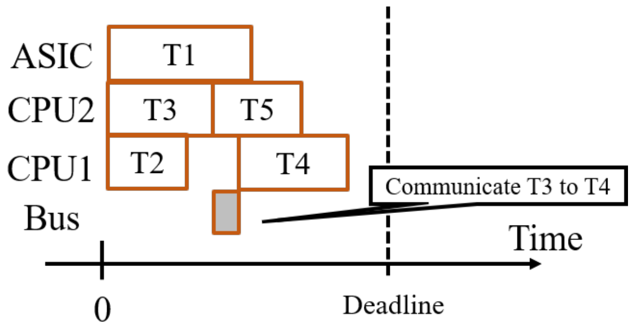
After application mappings onto processing elements, the next step is called as *Task scheduling*. This process determines the execution orders of tasks and communications timing within satisfying deadline constraints. Given an architecture and mapping of tasks and communications on processing elements as well as task graph of applications, the tasks are scheduled on identical processing elements.

Figure 2.5 shows an example of scheduling result based on the application mapping in Figure 2.4. According to the system specification, there is precedence dependency among the tasks. Task 1, 2, and 3 can start independently, but Task 4 and Task 5 are not allowed to start before Task 3 is finished. In this task graph referred to the upper in Figure 2.4, Task 1 requires 40-time units for the execution. Similarly, other tasks require execution times in the parenthesis beside each node.

Figure 2.5 represents two scheduling results based on the application mapping. Figure 2.5(a) indicates a schedule if Task 3 is allocated to CPU 1 and Task 5 is allocated to CPU 2, respectively. There is data flow dependency between Task 3 and Task 5, therefore, it incurs the communication in the internal bus from CPU 1 to CPU 2. Embedded systems often have deadline constraints, which are shown in the figure. The tasks in the application must be finished before the deadline. If the system misses the deadline,



(a) Schedule A



(b) Schedule B

Figure 2.5: Two different schedules based on identical application mapping

the schedule is not invalid and other schedules should be adopted. In the Figure 2.5(a), Schedule A misses the deadline, therefore, this schedule is invalid. On the other hand, Schedule B in Figure 2.5(b) can meet the deadline. In this case, Schedule B is preferred to be employed for a scheduling result. For more detail, communications should be taken into account during scheduling. If either of Task 4 and Task 5 or both are mapped onto different elements from the element that execute Task 3, Task 3 should send data where the successor tasks are assigned. In Schedule A, Task 3 sends data twice to CPU 2 to run Task 4 and Task 5.

Scheduling of the tasks is generally classified as NP-hard problems, and this process is very complex to find an optimal schedule. Therefore, scheduling of tasks onto processing elements is one of the crucial research topics as well as other processes in embedded design.

Energy Management

After allocation an architecture and mapping as well as scheduling, the next step is called as *Energy management* [21]. The step needs to accurately estimate the requirements of the systems, which is for the guidance of the optimization of allocation, map-

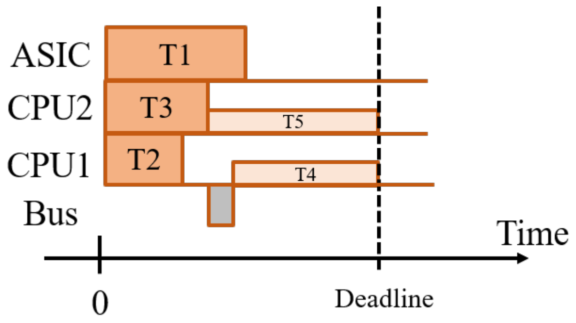


Figure 2.6: Energy management with applying dynamic voltage scaling

ping, and scheduling towards energy-efficient designs [22]. Generally, the techniques for energy management exploit idle and slack times from the scheduling result [23]. Idle times are periods in a schedule that no processing elements execute a task, while slack times are the time between task deadline and the finish time of tasks outgoing edges to dummy task *E*. For such times, there are some techniques such as dynamic voltage scaling (DVS), dynamic power management (DPM) [24]. DPM is a technique that reduces the power consumption of the system by selectively shutting down or sleeping idle elements. On the other hand, the DVS technique reduces supply voltage and clock frequency simultaneously, resulting in shortening idle and slack times.

Figure 2.6 shows the schedule with applying DVS. As shown in the figure, Task 4 and Task 5 are executed with a reduction of supply voltage and clock frequency and slow down the execution so that the execution manages to meet the deadline. However, the effective functioning of DPM and DVS depends on how long the idle time and slack time are. In order to optimize allocation, mapping, and scheduling effectively, idle and slack times optimization must be taken into account.

2.2.3 Hardware and Software Synthesis

Previously, co-synthesis of the systems has been addressed. The systems designed throughout the co-synthesis process are finally split into hardware and software physical implementations [25]. The synthesises of hardware and software are usually conducted concurrently since the interplay between hardware and software can be co-simulated simultaneously.

Hardware Synthesis

The recent design for hardware components has been with synthesis tools and techniques for very-large-scale integration (VLSI). In this step, a behavioral specification is transformed into a description at the register transfer level (RTL). Each component is generated as data paths under the control of a control unit. An RTL description is translated into gate-level expressions by logic synthesis tools. Similar to the data path, the control unit is represented structurally as a netlist of logical gates. Note that power reduction can be handled by the synthesis steps such as high level: clock gating, gate level: logic optimization, mask level: technology selection, etc. However, apart from the low-level power reduction techniques, the energy savings can be further enhanced by applying the previously described energy management techniques such as DPM and DVS at a higher level of abstraction. In general, the higher the abstraction level at which energy minimization is addressed, the greater the energy savings that can be achieved.

Software Synthesis

As with hardware synthesis, all tasks mapped to the software must be translated from high-level descriptions (JAVA, C/C++, SystemC, etc.) to machine code [26]. The software translation hierarchy composes two steps.

Translation of the initial specification in the high-level language into assembly code can be conducted in a general compiler such as GCC or with a special compiler optimized for a particular processor (e.g. DSP). Optimization aims to allocate registers such that operations can be performed without time-consuming memory accesses.

There are also compiler-based techniques to minimize power, such as instruction reordering and memory access reduction. In addition, careful design of algorithms at the source code level can lead to significant power reductions. Such approaches to minimize the power consumed employing system-level energy management techniques do not exclude each other. In actuality, both technologies require to be taken into account for the most energy-efficient system design.

2.3 Scheduling and High-Level Synthesis

2.3.1 Scheduling

In this thesis, we focus on scheduling of tasks in the co-synthesis process. Scheduling basically represents how the processes can be assigned to an available processing element.

This is crucial in terms of multitasking, multiprocessing, multithreading, and real-time operating system design. Modern applications are capable of parallel processing, and parallel processing is one of the promising approaches to efficiently satisfy computational requirements. However, there are problems such as the application partitioning into tasks, the coordination of communication and synchronization, the scheduling of tasks, and their mapping to processing elements. Scheduling and mapping are very

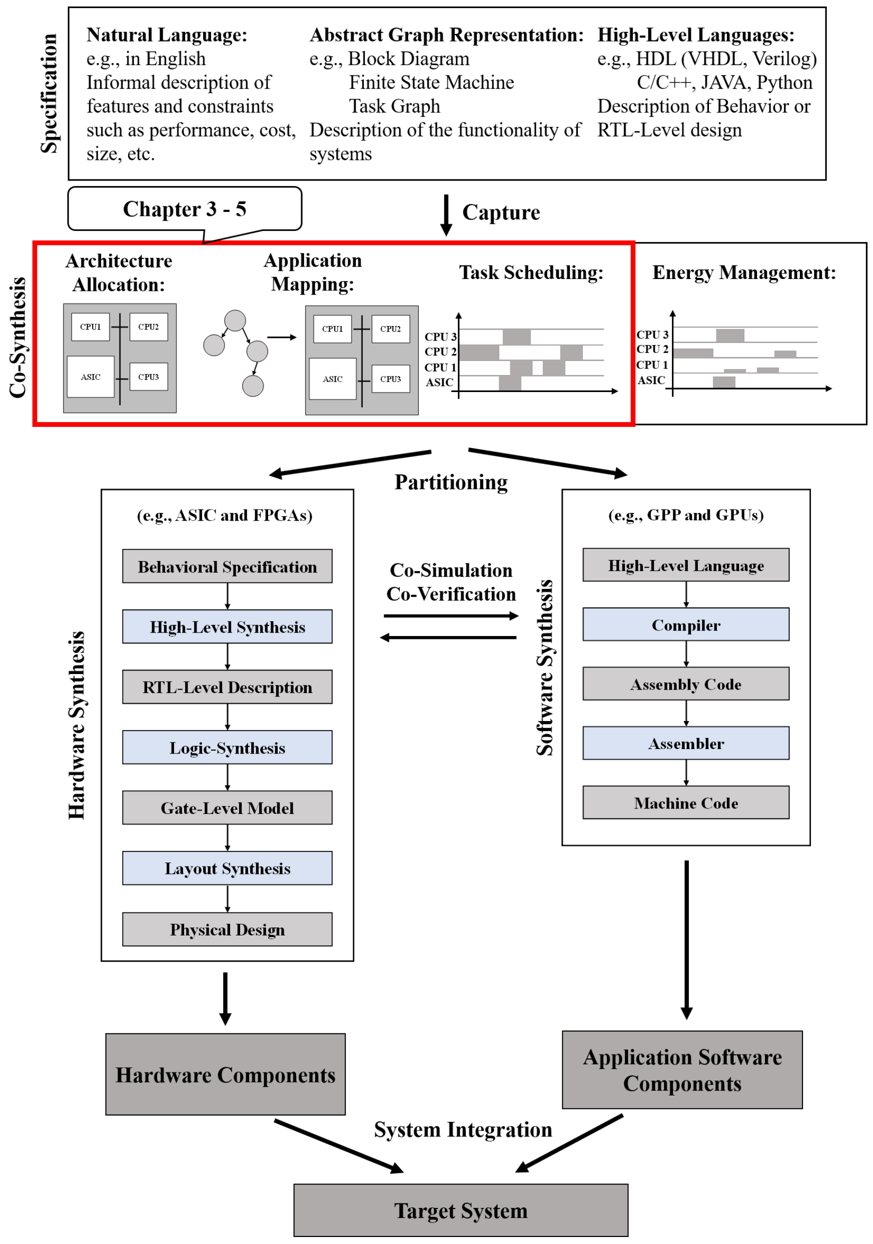


Figure 2.7: Highlighting the target topic in co-synthesis

important issues to achieve minimization of the completion time, of the overall energy consumption, and maximization of quality of services (QoL), because if tasks are scheduled inappropriately, the full potential of the system cannot be exploited, but rather the benefits of parallelism are offset.

Figure 2.7 recalls the embedded systems flow, which notates the works in the thesis. In order to determine whether an application is implemented on hardware or software, the co-synthesis process have a large influence on the partitioning result. Behind the background, this thesis mainly focuses on task scheduling as well as application mapping and architecture allocation. The topics in the figure are addressed from Chapter

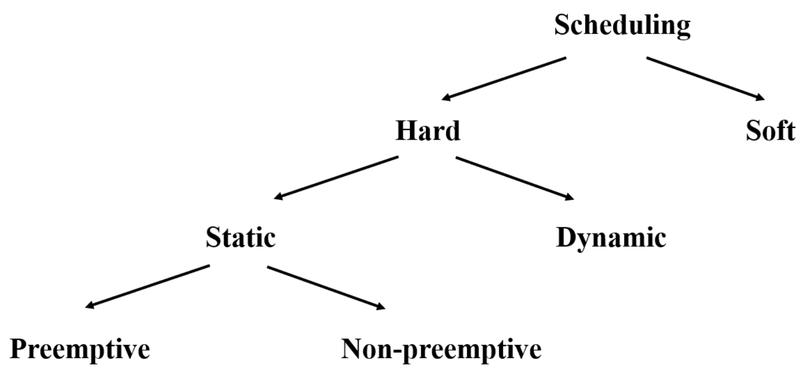
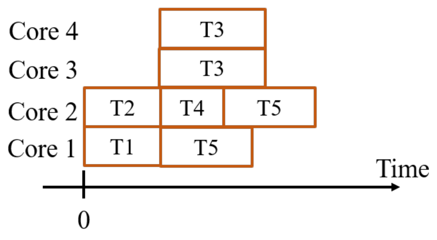


Figure 2.8: Taxonomy of scheduling problems

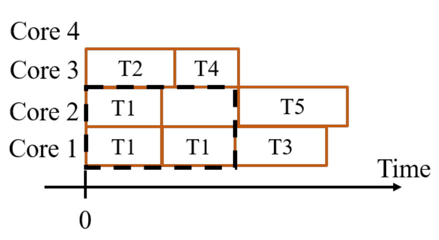
3 to Chapter 5 in detail. The characteristics of the scheduling techniques addressed in this thesis are to handle data-parallel tasks that can be partitioned into several threads and executed in parallel on the multiple cores. The following section briefly mentions that the taxonomies of scheduling problems and data-parallel tasks for comprehension of this topic.

In general, real-time systems can be classified by several types of deadlines. Figure 2.8 represents taxonomy of scheduling problems [27]. The initial separation is based on the type of deadline constraint, which refers to whether hard deadline or soft deadline. Hard deadline constraint means the deadline which all the execution becomes invalid if a schedule violates. On the other hand, a soft deadline is relatively safe in terms of execution even if the deadline is missed. Then, the taxonomy shows static scheduling and dynamic scheduling. Static scheduling is often conducted during compile time. For the parallel program, the profiled characteristics are necessary to be known using profiling tools in advance, such as the execution times, communication delays, data dependencies, synchronization, and so on [28, 29]. Since many parallel programs have data dependencies among tasks, a task graph of the parallel program is usually formed as DAG. On the other hand, the scheduler assigns a task onto processing elements on the fly. Dynamic scheduling algorithms do not know the arrival time of the tasks that a priori can hardly guarantee optimal performance. The objective of dynamic scheduling is not only to minimize the completion time but also to minimize the overhead of communications. If the execution allows preemption, a running task can be suspended by interruption and other tasks can start running. Non-preemptive execution, on the other hand, does not allow the interruption. Once started a task, other tasks cannot be executed instead of the task. In the rest of the thesis, static scheduling is mainly focused on and the preemption is not allowed.

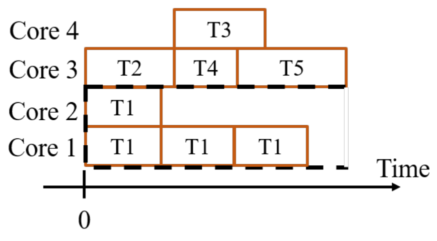
The performance of parallel processing is highly dependent on scheduling of the tasks. In parallel applications, the threads of the same parallel task are allowed to be



(a) Thread tasks



(b) Gang tasks



(c) Federated tasks

Figure 2.9: Different scheduling of parallel task models

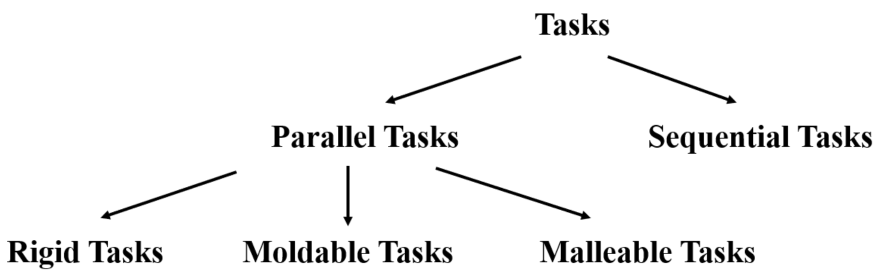


Figure 2.10: Taxonomy of parallel tasks

executed on multiple cores in parallel. As shown in Figure 2.9, there are three main execution models of parallel tasks to run applications in parallel on a multicore such as thread-based, gang, and federated models. The rest of the two models are specialized by thread-based scheduling models. In gang scheduling, multiple parallel threads are categorized and executed with each other, as the name suggests. During the execution of a task, a set of cores assigned to the task is reserved exclusively as shown in Figure 2.9(b). On the other hand, a federated task is a more particular case than a gang task in that the cores are focused on the threads during the lifetime of the systems as shown in Figure 2.9(c). In contrast, thread-based tasks are allowed to be executed on multiple cores independently in Figure 2.9(a). This thesis focuses on thread-based tasks, which is the most complex task model in the parallel execution models, and it addresses thread-based tasks in the following.

Scheduling of parallel tasks requires determining the number of threads for a task. Furthermore, it is necessary to map the tasks onto multicores at the same time as scheduling. Parallel tasks can be classified if the number of threads for a task assigned to the execution is determined, such as rigid, moldable, and malleable tasks as shown in Figure 2.10 [30]. For a rigid task, its parallelism is specified in advance and remains unchanged during scheduling. The number of threads for a moldable task is flexible and decided at the same time as scheduling, but once it is decided, its number is not allowed to be changed during the execution. The number of threads in a malleable task can be changed throughout the runtime. Therefore, the malleable task model is regarded as a generalization of the moldable task model with preemption and migration. Let us recall that this thesis focuses on static scheduling problems that prohibit preemption, and the target of parallel tasks is assumed to be moldable. The detail of the scheduling problems is later addressed from Chapter 3 to Chapter 5.

2.3.2 High-Level Synthesis

In order to efficiently design hardware from a high-level specification, hardware synthesis is very crucial [31]. The design for complicated hardware components is generally based on very-large-scale integration (VLSI) techniques. In the step of hardware synthesis, the design flow can be divided into several subsequent steps; high-level synthesis, logic synthesis, layout synthesis, and the target topic in this thesis is displayed once again as shown in Figure 2.11. The figure represents the fine-grained steps in hardware synthesis, but the work addressed in this thesis is coarse-grained in terms of being higher level in abstraction.

The overview of hardware synthesis is shown in Figure 2.12. High-level synthesis transforms a behavioral specification into an RTL description. The RTL components are generated as data paths execute operations such as arithmetic operations beneath a control unit. Then, the RTL description is synthesized into a gate-level expression as netlists of logic gates. Finally, the layout mask for IC fabrication can be generated through the layout synthesis step. Each physical gate is placed and interconnected wires are implemented and routed.

This thesis focuses on the high-level synthesis step. Typically, designers start an abstract specification that is implemented as a custom processor or a kind of custom hardware in a high-level description. Therefore, the description is usually written as untimed that a function consumes all its input data at the same time without taking into account any delay and obtaining outputs at the same time. High-level synthesis tools transform such untimed high-level specifications into a timed implementation. That is, they automatically generate custom hardware that contains a data path and a controller that satisfy the required specification and design constraints.

High-level synthesis consists of sequential flow as shown in Figure 2.13. Starting

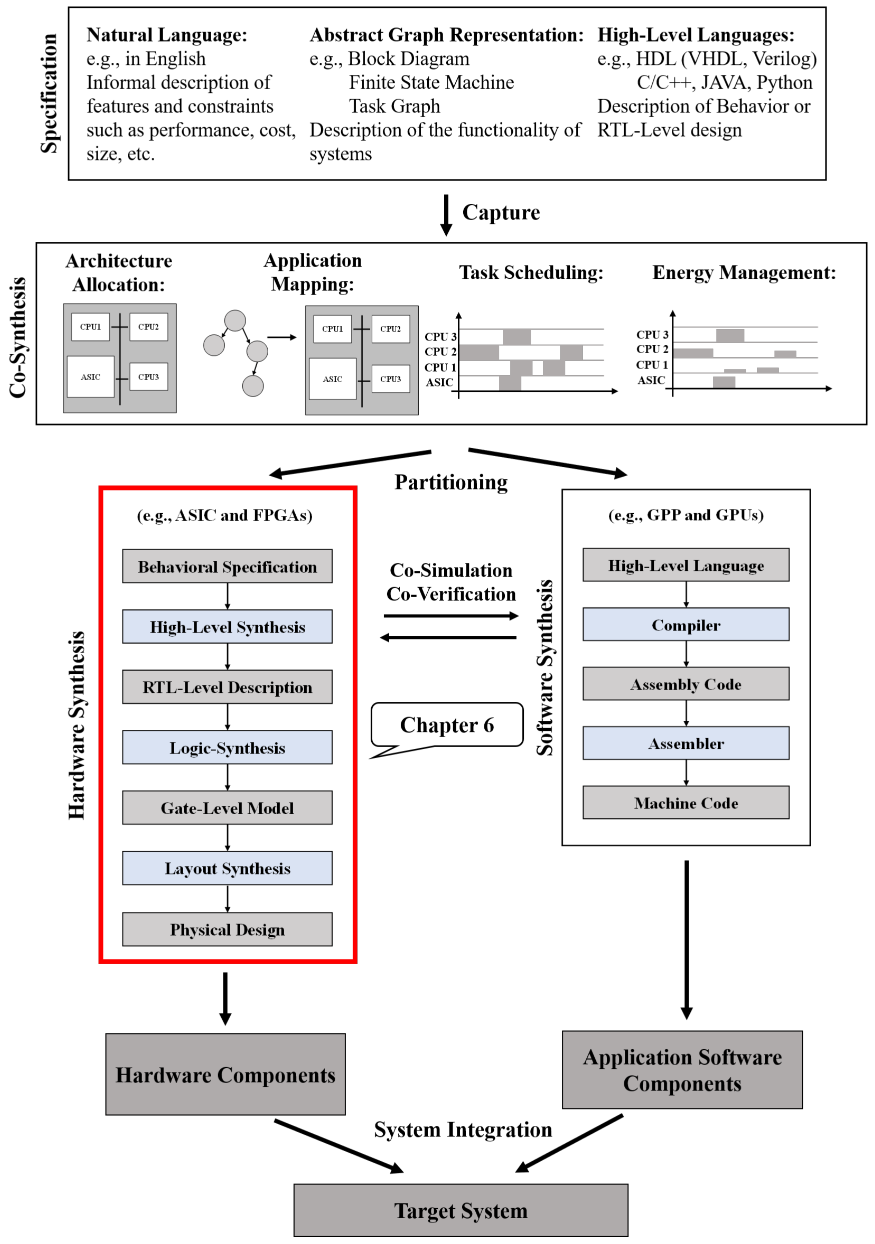


Figure 2.11: Highlighting the target topic in hardware synthesis

from the high-level description of an application, High-level synthesis tools finally output an RTL component. The high-level synthesis starts with compiling the functional specification in the high-level description for code optimization such as dead-code elimination, constant folding, loop transformations, solving false data dependency. The formal model generated by the compilation is often formed as a data flow graph (DFG) with the data flow dependencies between operations. The nodes are represented as the operations and the edges are represented data flow dependencies, respectively. In general, the DFG model is extended to the ones with control dependencies by unrolling all the loops and resolving conditional assignments. Such graphs are called the control data

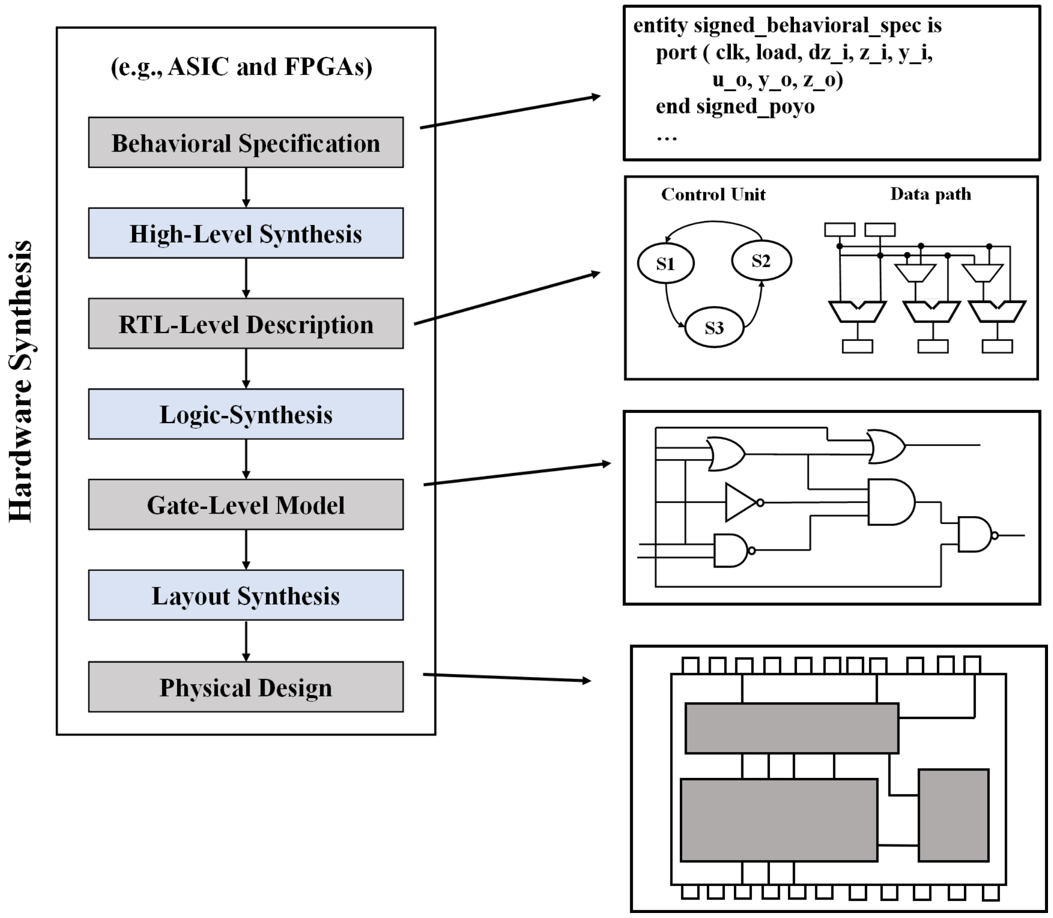


Figure 2.12: Hardware synthesis flow

flow graph (CDFG), in which the edges represent the control flow.

CDFG is more expressive than DFG and can represent loops that can repeat infinitely. However, parallelism is based on a basic-block level. That is, the analysis and transformations must reveal parallelism that exists between basic blocks. For example, it includes loop unrolling, loop pipelining, loop merging, and loop tiling. These techniques are used to optimize latency, throughput, and the size and the times of memory accesses by clarifying the parallelism among loops and between loop iterations.

Allocation process decides the number and the type of resources of hardware such as functional units, storage, and connectivity components to satisfy the resource constraints. The components to use are chosen from the RTL components library. The library includes the characteristics of the components such as delay, gate area, and power consumption. It is crucial to select more than a component for an operation and is also important to describe that this scheduling process is different from the one described in the previous section. This scheduling process requires that all operations listed in the specification are scheduled in clock cycles. This means that each operation must be read from its source (a storage device or a component of a functional unit), carried to

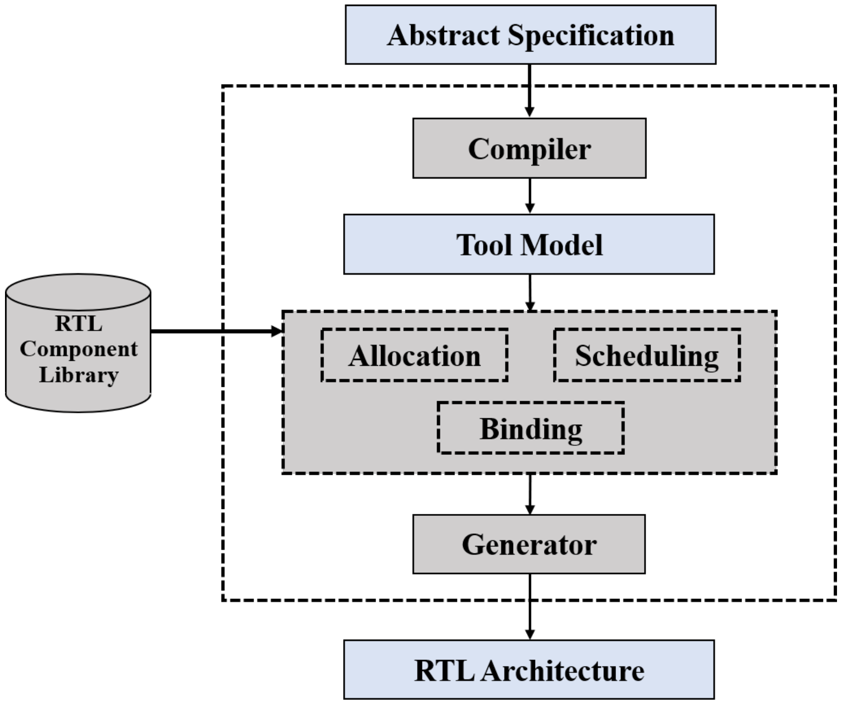


Figure 2.13: High-level synthesis flow

the input of a functional unit capable of performing that operation, and the result carried to its destination. Depending on the functional unit where the operation is mapped, its operation is scheduled either over one clock cycle or over several cycles (multi-cycle). Additionally, the operations may be able to be chained (the output of one operation is fed directly into the input of another operation). Therefore, the operations scheduling can be conducted for parallel execution if there are no data-dependent precedence constraints and if there are adequate resources available at the time.

Multiple Variables that periodically keep a value cannot avoid storage unit bound. A lifetime of the variables can be bound to the same storage unit even if the variables are non-overlapping or completely exclusive. Therefore, every operation must be bound to a functional unit, resulting in that the binding algorithm must optimize its selection if there is a functional unit. Furthermore, the binding of a storage device to a functional unit depends on the connectivity binding. Each transfer from a component to another component should be bound to a connectivity unit such as a bus, crossbar, or multiplexer. In the HLS processes, it estimates the latency and area of connectivity at the earliest possible stage. Another approach is to determine the entire architecture at allocation time in order to use the early floorplanning for both binding and scheduling results.

As mentioned earlier, HLS generally represents a fine-grained process, but what we are dealing with in this paper is a coarse-grained approach using a high-level language rather than an RTL description.

Chapter 3

Scheduling for Moldable Tasks on Homogeneous Multicore

This chapter addresses scheduling problems for moldable tasks on homogenous multiple cores[1].

3.1 Introduction

With the rapid progress of semiconductor manufacturing technology, multicore systems have replaced single-core systems to become the mainstream design paradigm for modern systems. In the literature, multicore task scheduling, which determines the execution order of tasks on multiple cores, has become more important than ever as an increasing number of cores in embedded systems. Each task is classically assumed to be assigned to one of the available cores, and the multiple tasks can be started in parallel on different cores while other tasks are running. [6, 33]. Much recent application includes tasks that can be partitioned into multiple threads in a data-parallel fork-join manner, and the threads can be executed independently on multiple cores [34]. Many scheduling problems that take into account task parallelism (i.e. inter-task parallelism) and data parallelism (i.e. intra-task parallelism) have been remarkably studied [35], [36, 37]. According to [35]-[38], they have assumed that a task is rigid, where the number of threads in each task is fixed. On the other hand, the works in [36, 37] assume that tasks are *moldable* that has an unfixed number of threads. The number of threads is determined during scheduling and its number does not change at runtime.

Task scheduling problems are generally recognized as the NP-hard class of computationally intractable problems [39]. Thus, moldable task scheduling, which is more

[1]This chapter is a refined and reproduced version of the paper originally published in International Journal of Networking and Computing [32].

complex than general task scheduling, is much hard to solve in a practical time. In the past, moldable task scheduling in [36, 37] has been tackled by using integer linear programming (ILP). However, empirically and theoretically, such techniques can find optimal schedules for a small task graph but are hard to find an optimal schedule or even one of the feasible schedules for large task graphs in a time for practical usage.

This work attempts to pursue the issues from a perspective of the programming paradigm, which is called constraint programming. Constraint programming (CP) is one of the promising approaches to combinational optimization problems that include classical task scheduling problems [40, 41, 42]. This work proposes a CP-based scheduling technique for moldable fork-join (MFJ) tasks. Given a set of dependent moldable tasks formed as directed acyclic graph (DAG) and a set of homogeneous multiple cores, the proposed technique determines the number of threads partitioned by a task and assigned the threads to the cores to minimize the overall completion time (a.k.a. makespan). In addition, this work proposes a CP-based scheduling technique assuming that a moldable task offers synchronization, which is called moldable synchronous (MS) tasks. As well as MFJ tasks, an MS task can be partitioned into an unfixed number of threads and the number will be determined during scheduling. On the other hand, threads of a task are prohibited to run independently and need to be mapped at the same time[2]. In contrast, the threads of an MFJ task are allowed to start independently on different cores, or they are even allowed to run sequentially on the same core.

This work contributes to the following threefold:

- This work proposes a CP-based scheduling technique for MFJ tasks.
- This work also proposes a CP-based scheduling technique for MS tasks.
- This work conducts a set of experiments and shows that the proposed CP-based techniques outperform state-of-the-art techniques with ILP.

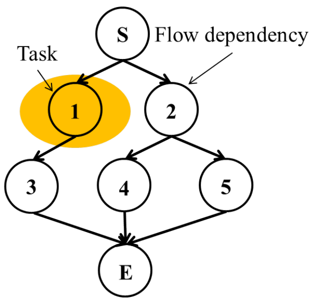
The remainder of this chapter is organized as follows. Section 3.2 describes related work on multicore task scheduling problems. Section 3.3 provides an ILP-based mathematical formulation and the CP-based approach to MFJ task scheduling. This section also presents the experiments and the evaluation. In Section 3.4, we present CP-based scheduling of MS tasks with the introduction of an ILP-based formulation with the experiments. Finally, Section 3.5 concludes this chapter with future plans.

[2]If the threads have to synchronize with each other frequently during their execution on a non-preemptive environment, the task is considered as the synchronous ones.

3.2 Related Work

Classical task scheduling techniques for multicore architectures have been widely investigated in [6]. In general, tasks are allowed to run in parallel on multiple cores if the tasks are independent so that they have not interfered in terms of memory and data. The authors of [33] develop an ILP-based approach to scheduling tasks on multiple processors, considering communication contention. However, these works ignore that a task inherent data-parallelism, resulting in that each task is allowed a single-core execution. In the real-world, applications such as image processing and deep learning-based, include data-parallel tasks as well as task parallelism. In the literature, task scheduling problems, which consider both parallelisms, have been studied in [34]-[37]. In [35], Liu et al. proposed list-based scheduling algorithms for data-parallel tasks. Their work assumes that a set of dependent tasks is assumed in the form of a task graph but tasks are assumed to be rigid, where the number of threads for each task assumes fixed beforehand. The goal of the work is to ask for a schedule to minimize the overall schedule length. Yang and Ha' s work in [38] also focuses on the scheduling of data-parallel tasks. The work in [38] assumes that tasks are called malleable that the number of threads for each task is determined during scheduling and its number is allowed to change at runtime. The goal of the work aims to minimize hardware costs with meeting deadline constraints. Unlike rigid tasks, malleable tasks are intuitively flexible in terms of their granularity, however, malleable tasks offer preemption that a task is necessary to be suspended to change its parallelism, leading to overhead for the interruption and may degrade performance. In [43], the authors take advantage of data-parallelism and present a technique for pipelined task scheduling and mapping on heterogeneous MP-SoCs. Chen and Chu in [44] developed and proved a polynomial-time approximation algorithm for malleable tasks to ask a minimum schedule length. Scheduling of MFJ tasks for real-time systems is studied in [45]. In the scheduling of MFJ tasks, a task is partitioned into multiple threads, and each thread is mapped to one of the multiple cores, independently. The work aims at the evaluation of the tractable and intractable fork-join real-time task model. Lakshmanan et al. in [46] developed an algorithm for malleable fork-join tasks in OpenMP. Saifullah et al. in [47] proposed a real-time task scheduling model which assumes that a task holds the various numbers of threads. Shimada et al. in [36] study MFJ task scheduling with the adoption of an approach based on ILP. They also study in [37] scheduling of moldable tasks with ILP. In this work, a moldable task is assumed that synchronization among all threads of each task is necessary for start time and end time. Their works are with an ILP-based technique, and therefore can hardly be explored due to the excessive time to ask for a solution.

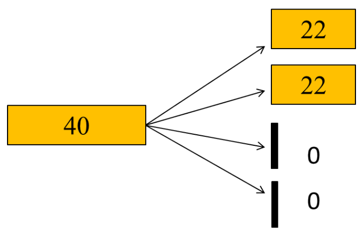
To find a feasible solution in a practical time, heuristic-based approaches have been



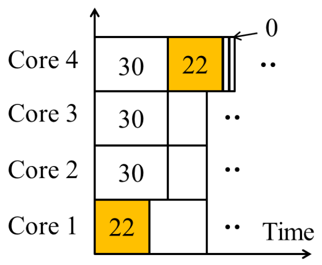
(a) Task graph

Sub-task	Execution time of task 1			
1	40	0	0	0
2	22	22	0	0
3	16	16	16	0
4	14	14	14	14

(b) Execution time of task 1



(c) Split into threads



(d) An optimal schedule

Figure 3.1: A scheduling example for MFJ tasks

developed such as mix integer linear programming (MILP), satisfiability modulo theory (SMT), and CP. In [48], the work studies a MILP approach to scheduling problems of parallel tasks, but it is focused on rigid tasks, where each task is assumed to be executed on a statically fixed number of processors. Liu et al. [49] analyze how efficiently an SAT solver can eliminate the solution space for task scheduling problems with mapping. The SAT-based framework proves a significant improvement in terms of optimality and scalability. Malik et. al in [50] evaluated the performance of scheduling problems with SMT. Furthermore, CP-based approaches are also proposed in the task scheduling area. In [40], the principle of CP techniques for combinational optimization problems has been surveyed. Derived from the work, Baptiste et al. [41] developed a scheduling problem with a CP-based approach. Kuchicinski in [42] also proposes a scheduling problem that takes into account communication cost. The authors both in [51] presented scheduling problems, assuming that tasks are executed on multiple processors. However, the works applied to such a solver are not focused on scheduling for moldable tasks. This work studies a CP-based approach to the scheduling of moldable tasks. We ask for better solutions and much faster than the state-of-the-art techniques for data-parallel task scheduling.

3.3 Scheduling for Moldable Fork-join Tasks

In this section, we first explain MFJ task scheduling using a comprehensive example. Then we describe formulation based on ILP, followed by [36]. Recall that an MFJ task assumes that a task is allowed to be partitioned into a set of threads. The split threads are generally permitted to run independently on multiple cores at different times. In this work, we ignore task migration and preemption. Once determining the number of threads during scheduling, its number will never change at runtime. In addition, one of the threads once assigned to a core cannot be moved to another core at running. For this scheduling problem, given a set of dependent tasks and a set of homogeneous cores, we ask a scheduling result to minimize the overall schedule length.

3.3.1 Problem Description

Figure 3.1 (a), (b), (c), and (d) show an example of MFJ task scheduling on four cores. The table on the top of Figure 3.1 (a) represents a task graph formed as DAG, and (b) shows $Time_{1,k,j}$ values for task 1. In this example, task 1 is partitioned into two threads as shown in Figure 3.1 (c). That is, task 1 is determined to run on dual cores. Then, one of the examples for scheduling results is represented in Figure 3.1 (d). As shown in this figure, the threads of task 1 are assigned to Core 1 and Core 4 at different times, respectively.

3.3.2 ILP Formulation

According to [36], the scheduling problem is based on ILP and the problem is solved by using a commercial solver. In the rest of this section, we briefly address the ILP formulation referred to [36].

This work assumes moldable tasks that the tasks can be partitioned into several threads. For simplicity, we assume that each of the tasks is allowed to be split into threads. Note that the number of threads is no more than the number of cores at most since the efficiency may be degraded due to incurring wait if the number of threads exceeds the number of cores in the target system. Let $split_{i,k}$ denote a 0-1 decision variable. $split_{i,k}$ becomes 1 if task i is split into k threads.

$$\forall i, \qquad \sum_{k} split_{i,k} = 1 \tag{3.1}$$

Recall that the execution time of each thread in a task assumes to be the same. Let $Time_{i,k,j}$ denote the execution time of $j-th$ thread of task i when it is assigned to k cores. $Time_{i,k,j}$ is 0 for $j > k$. $Time_{i,k,j}$ is assumed to be given. This work does not consider how to obtain $Time_{i,k,j}$ values since that is out of scope in this work, but the execution time of each thread in a task can be obtained by many profiles of running the tasks iteratively.

$$\forall i,j, \qquad time_{i,j} = \sum_{k} (split_{i,k} \times Time_{i,k,j}) \tag{3.2}$$

Let $start_{i,j}$ and $finish_{i,j}$ denote the start time and the finish time of the threads of task i, respectively. Note that $start_{i,j}$ is a decision variable and $finish_{i,j}$ is a dependent variable defined by the following equation.

$$\forall i,j, \qquad finish_{i,j} = start_{i,j} + time_{i,j} \tag{3.3}$$

Next, let $core_{i,j}$ be the identified number of the cores which is assigned to $j-th$ thread in task i. If two threads, thread $j1$ in task $i1$ and the thread $j2$ in $i2$, are mapped to the same core, the execution of the two threads cannot be overlapped in time. In other words, multiple threads are prohibited to run on the same core at the same time. This resource constraint is formulated by the following.

$$\begin{aligned} \forall i1,i2,j1,j2, \qquad & core_{i1,j1} \neq core_{i2,j2} \\ \vee \qquad & finish_{i1,j1} \leq start_{i2,j2} \\ \vee \qquad & finish_{i2,j2} \leq start_{i1,j1} \end{aligned} \tag{3.4}$$

This work assumes a set of dependent tasks, where the tasks may have a flow dependency among them. Let $start\_min_i$ and $finish\_max_i$ denote the start time and the finish time of task i, respectively. They are defined as follows.

$$\forall i, \qquad start\_min_i = min_j\{start_{i,j}\} \tag{3.5}$$

$$\forall i, \qquad finish\_max_i = max_j\{finish_{i,j}\} \tag{3.6}$$

Let $Flow_{i1,i2}$ denote a flow dependency from task $i1$ to $i2$. $Flow_{i1,i2}$ is 1 when task $i1$ must be finished before task $i2$ starts. Otherwise, $Flow_{i1,i2}$ is 0. We assume that $Flow_{i1,i2}$ is given. Then, the precedence constraint is expressed as follows.

$$\forall i1,i2, \qquad Flow_{i1,i2} \rightarrow finish\_max_{i1} \leq start\_min_{i2} \tag{3.7}$$

This work aims at minimization of the overall schedule length. Therefore, the objective function of our scheduling problem to be minimized is given as follows.

$$\text{Minimize :} \qquad max_i\{finish_i\} \tag{3.8}$$

Our scheduling problem for MFJ tasks is now formally defined: Given a set of dependent tasks, a set of cores, $Time_{i,k,j}$ and $Flow_{i1,i2}$, find $split_{i,k}$, $core_{i,j}$ and $start_{i,j}$ which minimize the objective function (3.8) subject to the constraints formula (3.1)-(3.7). For simplicity, we do not consider communication among the cores and contention between hardware resources such as memory, bus, and so on. Further, the tasks in this problem are well-ideal in that the execution times of each task are not jitters. Each thread is partitioned from a task assumed to have the same execution time as another thread of the same task. These extensions are out of this work, but they can be easily realized by adding some constraints.

3.3.3 Constraint Programming Approach

To apply CP to an optimization problem, IBM ILOG CP Optimizer is one of the promising solvers, whose expressions and functions are very suitable to solve combinational problems such as job scheduling and nurse scheduling problems in [52] and [53]. ILOG CP Optimizer includes several built-in functions. Referred to IBM ILOG CP Optimizer official publication and the related studies such as [54] and [55], CP Optimizer has been developed for the internal strong search strategies and techniques. By default, the CP optimizer employs a variety of metaheuristic techniques that are dynamically modified according to the problem. The main search techniques include, but are not limited to, Depth First Search (DFS), Large Neighborhood Search (LNS), and Genetic Algorithm (GA). Furthermore, constraint propagation is an internal technique that ILOG CP optimizer performs for searching for a solution to a CP problem. Constraint propagation removes values from regions that are not involved in the solution. CP optimizer then starts to search for a solution under the constraints using a backtracking algorithm. If the CP optimizer succeeds in obtaining a feasible solution, the solution is added to the constraints as an upper or lower bound. Furthermore, the constraint propagation removes the values of the decision variables from the domain which are not involved in the other solutions considered together with the solution. Once again, the CP optimizer restarts to find a solution. In this way, a powerful strategy for solving optimization problems is employed in the CP optimizer. It is these features of the CP that motivate this work. By using the built-in functions, the optimal schedule can be found efficiently. Even in case, optimal schedules are not found in a practical time, the CP optimizer can find better schedules than the ILP-based technique [36]. In the following, we describe the formulation for this scheduling using CP provided by IBM ILOG CP Optimizer.

Interval Variable

One of the unique concepts in ILOG CP Optimizer is called an interval variable. An interval variable internally holds the interval of time between the execution of an activity and is described as the size (length) of the start time, end time, and execution time. Another interesting thing about interval variables is that they include a bool number if it is optional or not. The interval variable may or may not be present. If an interval variable is marked as not existing, then it is ignored during scheduling.

Let $task_{i,j}$ be an interval decision variable for the j-th thread of task i. If the j-th thread of task i is considered, it is present in the scheduling, otherwise, it is not. In the scheduling problem, we are given the execution time of each thread of a task and we attempt to determine the number of threads for each task, to which core it should be mapped and its execution start time.

Precedence Constraints

This scheduling problem assumes a DAG-based task graph. There are the tasks where data-dependent precedence dependency does exist among them. The function, called $endBeforeStart(a, b)$ built in ILOG CP Optimizer, represents the precedence between two interval variables a and b to be true with their presence. Therefore, the precedence constraint that $task_{i1,j1}$ must be finished before $task_{i2,j2}$ starts is expressed as follows.

$$\forall j1,j2, \qquad endBeforeStart(task_{i1,j1}, task_{i2,j2}) \tag{3.9}$$

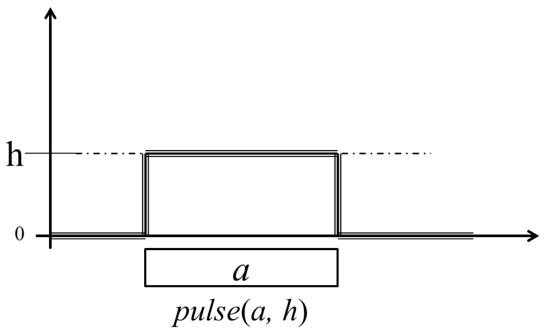
Alternative Constraints

Next, we introduce another concept of a set of variables called as $decision_{i,k,j}$. Let $decision_{i,k,j}$ denote a set of the execution time for $j-th$ thread in task i on k cores. To guarantee that one of $decision_{i,k,j}$ is present for each $task_{i,j}$ where task i is determined to be split into k threads, we use the alternative function.

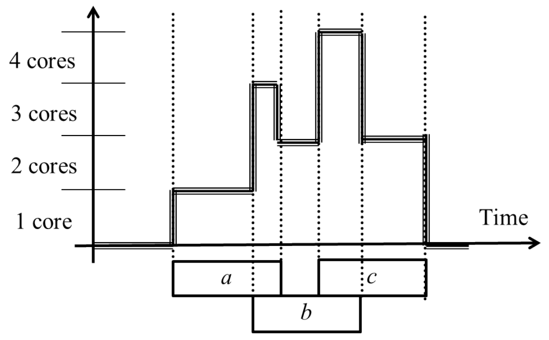
$$\forall i,j \qquad alternative(task_i, \cup_k \{decision_{i,k,j}\}) \tag{3.10}$$

In Formula (3.10), it should be recalled that task i is always present. Therefore, one in a set of $decision_{i,k,j}$ must be present, where the threads in task i must be taken into account. The constraint is expressed in Formula (3.11).

$$\forall i,k,j1,j2, \qquad presenceOf(decision_{i,k,j1}) \rightarrow presenceOf(decision_{i,k,j2}) \tag{3.11}$$



(a) Pulse function



(b) Cumulative function

Figure 3.2: A concept of resource constraint

$presenceOf$ is a built-in function of ILOG CP Optimizer, which returns values 0-1 for the existence of the given interval variable.

Resource Constraints

At any moment in time during scheduling, the total number of cores assigned to active tasks is prohibited to exceed the number of cores. The resource constraint is introduced by a concept called as $pulse$ function and the $cumulfunction$, where they are built-in ILOG CP Optimizer as shown in Figure 3.2. Figure 3.2 (a) shows the concept of the pulse function. Let a be an interval variable and h be a scalar value, respectively. *pulse(a, h)* indicates scalar h while the interval a is active. When a is once absent, the pulse value is down to 0. According to [52] and [53], the cumulative function can accumulate the values generated by its pulse function from pulses over time. As shown in Figure 3.2 (b), it demonstrates an example with the interval variables a, b and c. This case represents that the pulse function is utilized with a, b, and c, respectively. Now, the cumulative function is accumulated with $pulse(a, 1)$ + $pulse(b, 2)$ + $pulse(c, 2)$. Introduction of the pulse and cumulative functions to the resource constraint, we can show the resource constraint about the number of cores as follows.

$$Ncores \geq CumulFunction\{\sum_i \sum_k \sum_j pulse(decision_{i,k,j}, 1)\} \tag{3.12}$$

In the formula, $Ncores$ indicates the total number of cores in the target system. It should be recalled that $decision_{i,j,k}$ is an interval variable and is present if $j-th$ thread of task i split into k threads is active. Formula (3.12) expresses the value of $pulse(decision_{i,j,k}, 1)$ is 1 while $j-th$ thread in task i being active. It represents the value is one if one of the threads is assigned to one of the cores. The right side of the inequality shows that sum of the total number of cores assigned to the active threads in tasks.

Objective Function

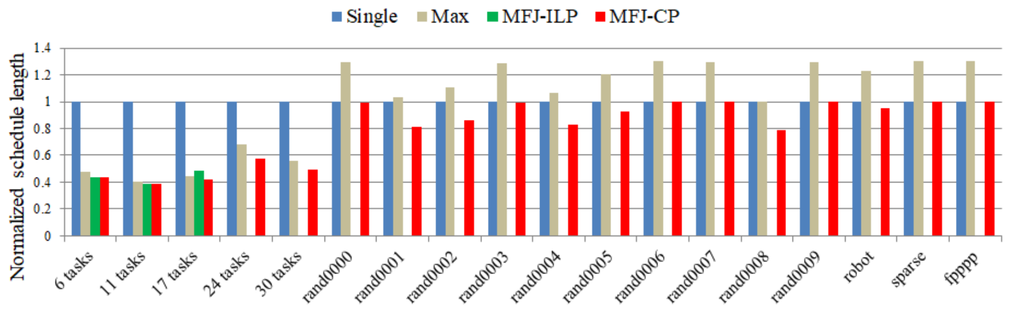
Our objective function is to minimize the overall schedule length, and is defined as follows.

$$\text{Minimize :} \qquad max_{i,j}\{endOf(task_{i,j})\} \tag{3.13}$$

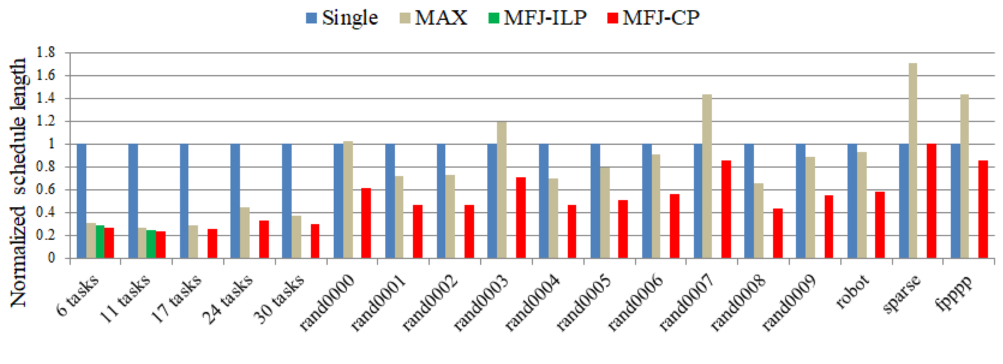
$endOf$ is a built-in function in ILOG CP Optimizer, which returns the end time of the given interval variable. In general, constraint programs do not have any objective functions to be minimized or maximized. Constraint programs search for values for variables which satisfy all of the specified constraints. However, many state-of-the-art CP solvers including ILOG CP Optimizer are capable of finding the best values for variables to optimize a specified objective function. In this work, we take advantage of the optimization capability in ILOG CP Optimizer.

3.3.4 Experiments

In order to compare the performance of the proposed scheduling techniques, we have conducted a series of experiments. The scheduling of MFJ tasks is much more complex than general scheduling problems in terms of computational cost since a task is allowed to be parallelized. Thus we have prepared a small set of task graphs consisting of 6 to 30 randomly generated tasks in DAGs [56]. The rest of the task graphs are derived from the $Standard\,Task\,Graph(STG)$ developed at Waseda Univerity [57]. Ten out of thirteen task graphs, called $rand0000$ to $rand0009$ are randomly generated with exact 50 tasks. The three task graphs out of them are called $robot$, $sparse$, and $fpppp$, respectively. They are well-known as actual applications in the real world. For scheduling, the applications are detailed as follows: robot control application includes 88 tasks and



(a) Scheduling results on 4 cores

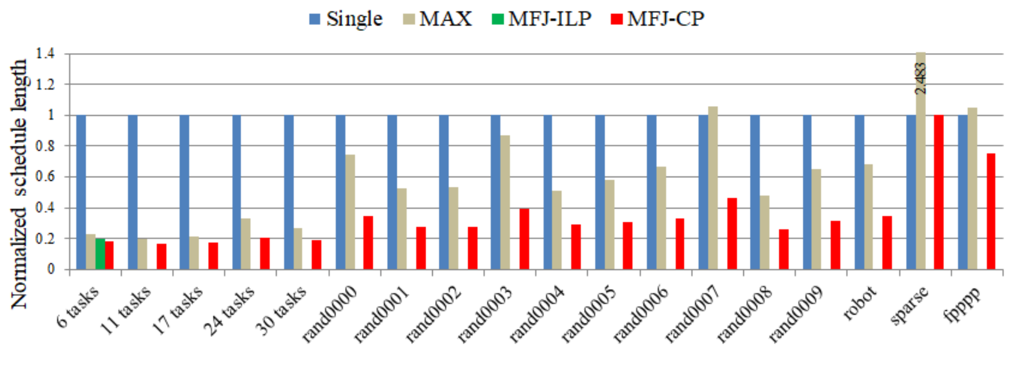


(b) Scheduling results on 8 cores

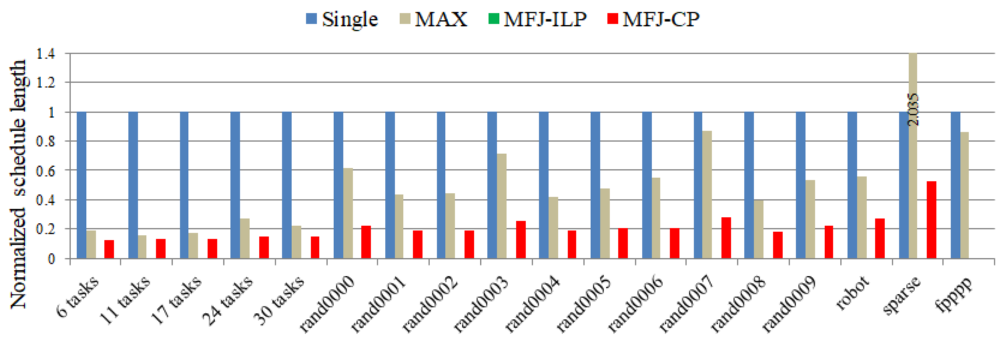
Figure 3.3: Scheduling results for MFJ tasks on 4 and 8 cores

131 edges, sparse matrix solver application includes 96 tasks and 67 edges, and SPEC fpppp application includes 334 tasks and 1145 edges. The experiments explicitly ignore communication costs among the cores due to memory accesses and bandwidth. In addition, hardware resource contention is also neglected. Each application program assumes that there is a data-dependent precedence constraint between tasks. STG's original task graph assumes that each task is expected to run only on a single core so that the execution times are given assuming execution on a single core. We have also multiplied them by 100, as the execution times may be less than 1 if we assume that the task runs on 32 cores. Therefore, in our experiments, the execution times of MFJ tasks on multiple cores are assumed to be $Time_{i,k,j} = 100 \times Time_{i,1,j} \times (0.1 + 0.9/k)$, for $k \geq 1$, where $Time_{i,k,j}$ denotes the execution time of $j-th$ thread in task i assigned to k cores, where each thread is assumed to have 10% overhead for parallelization of a task. The number of cores in the target systems is varied from 4 to 32.

- *Single (optimal)*: Optimal scheduling on the assumption that a task is assigned to a single core. The optimal solutions are provided in the STG package [57].
- *Max*: Every task is executed on all cores, and the tasks are sequentially executed.
- *MFJ-ILP*: ILP-based MFJ task scheduling presented in [36].



(a) Scheduling results on 16 cores



(b) Scheduling results on 32 cores

Figure 3.4: Scheduling results for MFJ tasks on 16 and 32 cores

- *MFJ-CP*: CP-based MFJ task scheduling presented in this work.

Figure 3.3 (a) and (b) and Figure 3.4 (a) and (b) show scheduling results on 4, 8, 16 and 32 cores, respectively. The obtained results are formally normalized to the Single technique. For the case of 4-core systems, as shown in Figure 3.3(a), the MFJ-CP technique can find schedules in a practical, but they are slightly different from the results by the Single technique due to poor parallelism against the number of tasks. With decreasing number of tasks, we can find much shorter schedules even on 4 cores in the systems. On the other hand, the results by the MFJ-ILP technique are almost missed in the graphs, which means that the technique is failed to find any of the schedules in time. In the cases with a small number of tasks, the MFJ-ILP may be possible to find better schedules than the Single technique and the Max technique. However, the schedule lengths are the same or longer by 7.1% than the MFJ-CP technique. From the results, the CP-based scheduling technique is superior to the state-of-the-art scheduling techniques in terms of the quality of schedule length.

In the 8-core systems as shown in Figure 3.3(b), much significant improvement is observed. The MFJ-CP technique is successful in obtaining shorter schedules than the other techniques. On the other hand, the MFJ-ILP technique can no longer find a schedule in many cases. In the sparse benchmark, the best schedule is obtained by assigning each task to a single core. The MFJ-CP technique demonstrates that it can reduce the schedule length by up to 56.6%, compared to the Single technique.

With the increase in the number of cores, the MFJ-CP technique can reduce the length of the schedule by 65% on average, while the MFJ-ILP technique is no longer able to solve this problem. As for the Max technique, it is possible to find a shorter schedule than the Single technique. However, the Max technique always finds worse schedules than the MFJ-CP technique. On the other hand, this technique has the advantage that it can find schedules instantly. In Sparse, the parallelism of the tasks is relatively high for the number of cores due to the priority dependency between tasks. Also, the overhead due to parallelism affects the length of the schedule. Therefore, for this benchmark, when the parallelism of the number of cores is low compared to the parallelism of the priority dependencies, running on a single core is more suitable.

In the 32-core systems in Figure 3.4(b), the MFJ-ILP technique is impossible to find any of the solutions due to the high complexity of scheduling on multicore. The same is true for the case of fpppp tried by the MFJ-CP technique, but it is successfully able to improve the effectiveness by up to 81.9% concerning minimization of the schedule lengths in the overall cases.

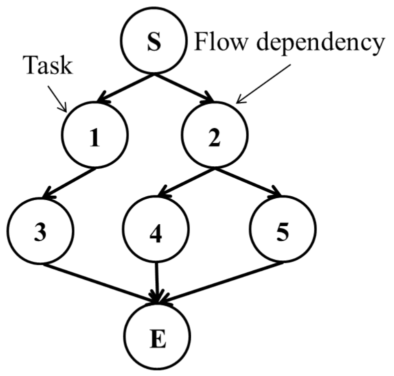
3.4 Scheduling for Moldable Synchronous Tasks

Similar to the scheduling presented in the previous section, MS tasks can be partitioned into an unfixed number of threads. However, the threads cannot be executed independently, and need to be scheduled synchronously with each other. The threads of an MS task run at the same time on different cores. In this section, we describe a CP-based scheduling technique for MS tasks. Scheduling of MS tasks properly asks the execution order of tasks and the number of threads for each task during scheduling. The objective of this scheduling problem aims to minimize schedule length as well as the previous section.

3.4.1 Problem Description

Figure 3.5 (a), (b), and (c) show an example of scheduling for MS tasks. In Figure 3.5 (a), a set of dependent tasks is represented as DAG, called a task-graph. Each of the tasks is associated with the execution time which is a function of the number of cores to execute the task. A table of execution time for task 1 is shown in Figure 3.5 (b).

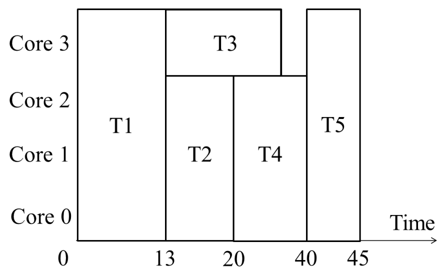
For example, if task 1 is to be run on a single core, then the execution time for task 1 will consume 45-time units. It is also assumed that each task must need to synchronize



(a) Task graph

	Cores to use			
Task	1	2	3	4
1	45	25	15	13
2	15	10	7	4
3	20	10	17	10
4	40	22	20	11
5	19	13	7	5

(b) Execution time of task 1



(c) An optimal schedule

Figure 3.5: A scheduling example for MS tasks

with the other threads in the task and that all threads start running in parallel on multiple cores at the same time. When running on dual cores, the execution time of task 1 will take 25-time units. It should be noted that task partitioning incurs overhead for parallelization, so the assumption is made that execution time will not be linearly reduced on multicore. Consider the function of the execution time of a task is given, and it is outside the scope of this study to determine how to obtain that value. The number of cores allocated to a task is determined at the same time as during scheduling.

Figure 3.5 (c) shows one of the optimal schedules for the task-graph in Figure 3.5 (a). The overall scheduling length of the schedule is seen as 45 time units.

3.4.2 ILP Formulation

The scheduling for MS tasks presented in [37] is with ILP. To compare with the proposed technique, the solutions are obtained by a commercial solver. The rest of this section briefly describes the ILP formulation presented in [37].

Let $map_{i,j}$ be a 0-1 decision variable. $map_{i,j}$ becomes 1 if task i is mapped to core j, otherwise 0. Let $cores_{i,k}$ be 0-1 variable, which becomes 1 if task i uses k cores; k is ranged from one to the maximum number of cores in the target system, and otherwise 0. Note that $cores_{i,k}$ depends on $map_{i,j}$ and is determined as follows.

$$\forall i, \qquad \sum_k cores_{i,k} = 1 \tag{3.14}$$

$$\forall i, \qquad \sum_j map_{i,j} \; = \; \sum_k cores_{i,k} \times k \tag{3.15}$$

Let $Time_{i,k}$ denote the execution time of task i on k cores. $Time_{i,k}$ is assumed to be given as mentioned earlier. The execution time of task i is given by the following equation.

$$\forall i, \qquad time_i = \sum_k \{cores_{i,k} \times Time_{i,k}\} \tag{3.16}$$

Next, let $start_i$ and $finish_i$ denote the start time and finish time of task i, respectively. Note that $start_i$ is a decision variable and $finish_i$ is a dependent variable defined by the following equation.

$$\forall i, \qquad finish_i = start_i + time_i \tag{3.17}$$

The two tasks, $i1$ and $i2$, cannot be mapped to a core at the same time. That is, the execution of the two tasks cannot be overlapped in time. This resource constraint of cores is formulated by the following inequality.

$$\begin{aligned} \forall i1{,}i2{,}j, \qquad & map_{i1,j} + map_{i2,j} < 2 \\ \vee \qquad & finish_{i1} \leq start_{i2} \\ \vee \qquad & finish_{i2} \leq start_{i1} \end{aligned} \tag{3.18}$$

This work assumes a set of dependent tasks, and some tasks have precedence dependencies. As a precedence dependency between task $i1$ and task $i2$, $Flow_{i1,i2}$ becomes 1 if task $i1$ is ended before task $i2$ starts, and otherwise 0.

$$\forall i1{,}i2, \qquad Flow_{i1,i2} \rightarrow finish_{i1} \leq start_{i2} \tag{3.19}$$

The objective is to minimize the overall scheduling length. Therefore, the objective function of the scheduling problem is as follows.

$$\text{Minimize :} \qquad max_i\{finish_i\} \tag{3.20}$$

Now, the scheduling problem for MS tasks is formally defined: Given a task-graph, a set of cores, $Time_{i,k}$ and $Flow_{i1,i2}$, it decides $map_{i,j}$ and $start_i$ which minimize the objective function (3.20) subject to the six constraints (3.14)-(3.19). Although some of the expressions are not linear, they can be easily transformed into linear forms as presented in [36].

3.4.3 Constraint Programming Approach

Let $task_i$ denote an interval decision variable for task i. It must be present in the scheduling problem, and thus the presence status of $task_i$ is true. Next, let $decision_{i,k}$ denote an interval decision variable which decides the number of cores to execute $task_i$. $decision_{i,k}$ is present if k cores are assigned to $task_i$, otherwise absent. This work assumes that the execution time of $decision_{i,k}$ is given, similarly to $Time_{i,k}$ in Section 3.3.2.

This work assumes a set of dependent tasks, where tasks may have a given precedence dependency. The function, called $endBeforeStart(a, b)$ built-in ILOG CP Optimizer, represents the precedence constraint between interval variables a and b are considered to be true provided that both of the two-interval variables are present. Therefore, the precedence dependency constraint that $task_{i1}$ must be finished before $task_{i2}$ starts is expressed as follows.

$$\forall i1{,}i2, \qquad endBeforeStart(task_{i1}\ , task_{i2}) \tag{3.21}$$

An alternative constraint is one of the functions in ILOG CP Optimizer. Let denote a and b_i interval variables. The function $alternative(a, b_1...b_n)$ represents exactly one of a set of intervals $b_1...b_n$ is present on the condition that interval a is present. The start and the end time of interval a are synchronized with those of b_i which is chosen to be present. If a is absent, every b_i is also absent. Using the alternative function, we can guarantee that one of $decision_{i,k}$ is present for each $task_i$, as shown in Formula (3.22).

$$\forall i, \qquad alternative(task_i, \cup_k \{decision_{i,k}\}) \tag{3.22}$$

In Formula (3.22), it should be recalled that $task_i$ is always present. Therefore, one of $decision_{i,k}$ must be present.

This work tries to minimize the schedule length under a resource constraint on several cores. At any moment in time, the total number of cores assigned to active tasks cannot exceed the number of cores in the target system. This resource constraint is expressed with the pulse function in ILOG CP Optimizer. Figure 3.2, shown in the previous section, shows the concept of the pulse function. The resource constraint is expressed as follows.

$$Ncores \geq CumulFunction\{\sum_i \sum_k pulse(decision_{i,k}, k) \tag{3.23}$$

In the formula (3.23), $Ncores$ denotes the number of cores. It should be recalled that $decision_{i,k}$ is an interval variable and is present if k cores are assigned to task i. Formula (3.23) means the value of $pulse(decision_{i,k}, k)$ is k while task i being active. The value k is the number of cores assigned to a task, that is, the right side of the inequality shows that sum of the number of cores assigned to the active tasks. Therefore, this formulation does not take into account mapping of the tasks onto the cores but simply does the number of cores assigned to them. Formula (3.23) does not identify what cores to be assigned to a task, thus, the number of combinations of the cores assigned to a task is significantly reduced unlike the formulation addressed in the ILP technique.

Our objective function is to minimize the overall schedule length, and is defined as follows.

$$\text{Minimize :} \qquad max_i\{endOf(task_i)\} \tag{3.24}$$

As earlier mentioned in the previous section, $endOf$ is a built-in function in ILOG CP Optimizer, which returns the end time of the given interval variable. We try to minimize the overall schedule length with the benefit of ILOG CP Optimizer that are capable of finding the best values for variables to optimize a specified objective function.

3.4.4 Expertiments

To evaluate and compare the performance of the proposed approach, we have conducted a set of experiments. we have set task-graphs that are composed of a small number of tasks from 6 to 30 tasks in [56]. The rest of the task graphs are derived from the $Standard\,Task\,Graph(STG)$ developed at Waseda Univerity [57]. Ten out of thirteen task graphs, called $rand0000\ to\ rand0009$ are randomly generated with exact 50 tasks, and the others, called $robot$, $sparse$, and $fpppp$, are modeled into DAG as benchmarks of actual application.

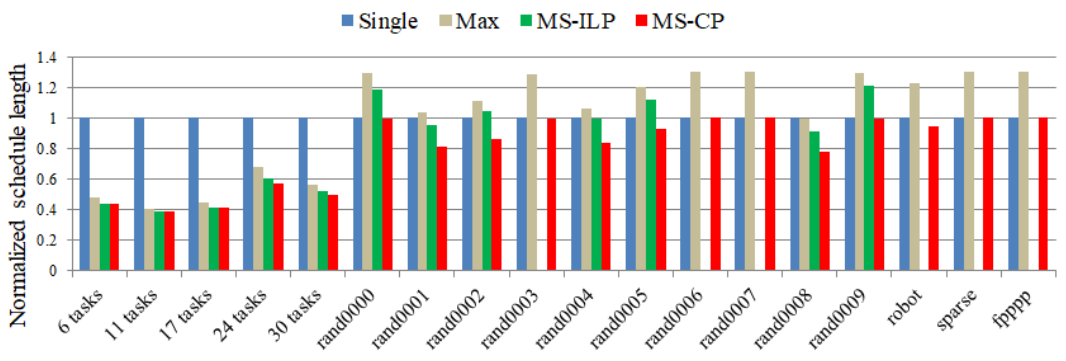
The details of each application are as follows. Robot control application has 88 tasks and 131 edges, sparse matrix solver application has 96 tasks and 67 edges, and fpppp application includes 334 tasks and 1145 edges. Each application is formed as a DAG-based task graph, therefore, tasks have data-dependent precedence dependency. STG's original task graphs assume that each task is only executed on a single core, and therefore provide execution times that assume execution on a single core. In the experiments, the execution times of each tasks are given as follows: $Time_{i,k} = 100 \times Time_{i,1} \times (0.1 + 0.9/k)$, for $k \geq 1$, where $Time_{i,k}$ denotes the execution time of task i assigned to k cores with 10% for an overhead by parallelization of tasks. The number of cores in the target systems is varied from 4 to 32.

- *Single (optimal)*: Optimal scheduling on the assumption that a task is assigned to a single core. The optimial solutions are the same as the ones multiplied by 100 in the STG package [57].
- *Max*: Every task is executed on all cores, and the tasks are sequentially executed.
- *MS-ILP*: ILP-based MS task scheduling presented in [37].
- *MS-CP*: CP-based MS task scheduling presented in this work.

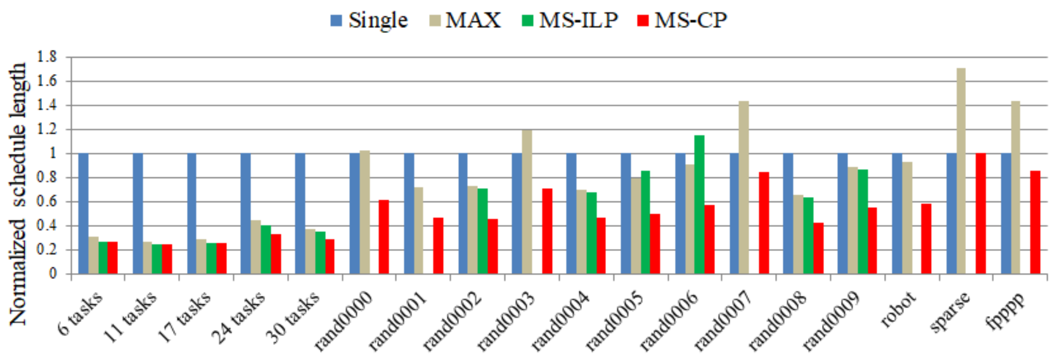
A commercial solver, IBM ILOG CPLEX 12.8 is used to solve for the MS-ILP and MS-CP methods. CPLEX has two optimization engines, i.e., a mathematical programming engine and a constraint programming engine. The mathematical programming engine is used for the MS-ILP method and the CP engine is used for the MS-CP method, respectively. CPLEX is run on an Intel Core i9 7980XE (2.6GHz) processor with 128GB of memory; CPLEX execution time is limited to 10 hours in CPU time, after which the best schedule found at that time is used for comparison.

Figure 3.6 (a) and (b) and Figure 3.7 (a) and (b) show scheduling results for MS tasks on 4, 8, 16 and 32 cores, respectively. The X-axis of the graphs shows the task-graphs, where $rand0000$ to $rand0009$ include 50 tasks for each task-graph as well as in the previous section. The Y-axis shows the schedule length obtained by the four techniques. The schedule lengths are normalized to the Single technique. In some cases, no solution is found by the MS-ILP technique. There is a reason that the MS ILP technique is almost failed to obtain even one of the feasible solutions in a practical time.

Figure 3.6 (a) shows the results of the 4-core systems. The results in the MS-CP technique surpass the ILP-based technique with an improvement of 11% on average, nevertheless, there is poor parallelism. As shown in the cases of the applications, the effectiveness of multi-threaded tasks is no more improved than the traditional technique. On the other hand, the Max technique finds longer schedules in many cases due to the overhead by parallelization.



(a) Scheduling results on 4 cores

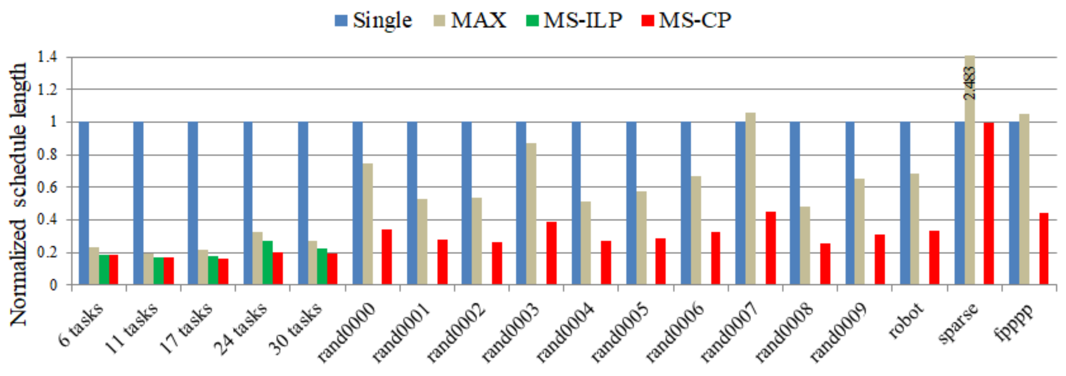


(b) Scheduling results on 8 cores

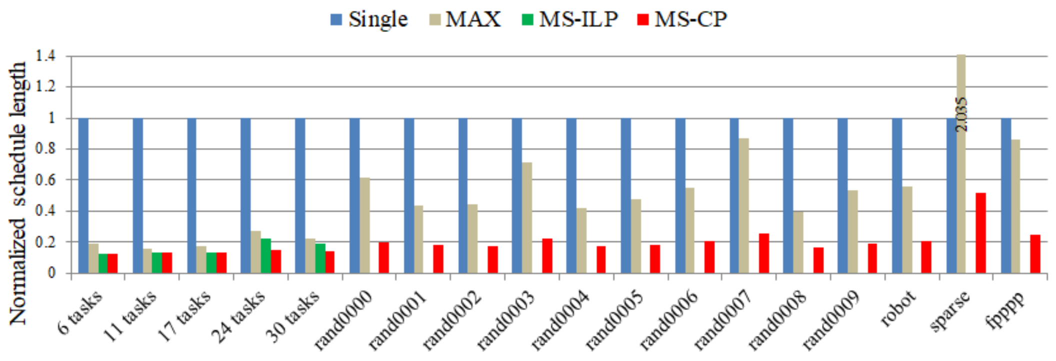
Figure 3.6: Scheduling results for MS tasks on 4 and 8 cores

In the 8-core systems as shown in Figure 3.6 (b), the effectiveness of the MS-CP technique is observed. Although the MS-ILP technique fails to obtain a shorter schedule in some cases, the MS-CP technique is still successfully obtained with better solutions than the other techniques. For small task graphs of 6 to 30 tasks, there is a slightly different schedule length by the MS-CP technique is shorter than the schedule by the MS-ILP technique by up to 7.6%. On average, the MS-CP technique finds shorter schedules by 18.6% over the results.

In Figure 3.7 (a) and (b), the results found by the MS-ILP technique are almost missed. With an increasing number of cores, the number of threads of each task can be large. On the other hand, the MS-CP technique can find feasible schedules, which means that our CP-based approach can find good feasible schedules in a short time. In the results, we can obtain benefits attributed to a CP-based approach.



(a) Scheduling results on 16 cores



(b) Scheduling results on 32 cores

Figure 3.7: Scheduling results for MS tasks on 16 and 32 cores

3.5 Conclusions

In this chapter, we have presented the scheduling technique using constraint programming for the MFJ task and MS task. In the scheduling of MFJ tasks, the number of threads and their execution order are determined during scheduling and multiple threads can run in parallel independently at different times. In the scheduling of the MS task, the number of threads is determined during scheduling as well, on the other hand, the threads require synchronization and run in parallel at the same time. The experimental evaluation demonstrates the effectiveness of the proposed techniques. For all the experiments, we have obtained much better schedules compared to the existing state-of-the-art techniques. Thus, we have confirmed that CP-based scheduling can quickly find good schedules for formable tasks. In the future, we will take into account more complex issues such as memory accesses such as cache and main memory. In addition, we need to consider resource conflicts between tasks.

Chapter 4

Scheduling of Moldable Tasks with Inter- and Intra-Task Communications

This chapter addresses a communication-aware scheduling problem that consider inter- and intra-task communications on homogeneous multicore[1].

4.1 Introduction

Due to the growing demand for high performance in embedded systems, parallel computing is a promising technique for multicore systems [6]. In the domain of parallel computing, one of the main challenges faced by designers is to obtain an efficient schedule to speed up or reduce energy consumption. To satisfy the requirements, an essential technique in parallel computing is multicore task scheduling, which determines the execution order and mapping of the tasks on multiple cores. Many existing scheduling techniques classically have in the common assumption that each task is assigned to one of the multiple cores; however, most programs have recently included inherent data parallelism. A task in a modern program can be partitioned into multiple threads by taking advantage of the task structure in a data-parallel manner so that the threads are executed in parallel. In other words, many current works study the scheduling of parallel tasks that consider both task parallelism and data parallelism.

According to [27] and [60] applications in the real world for parallel tasks can be classified into three types, i.e., rigid tasks, moldable tasks, and malleable tasks. For a rigid task, its parallelism is specified in advance and remains unchanged during scheduling. The number of threads for a moldable task is flexible and decided at the same time as scheduling but, once decided, this number cannot be changed during the ex-

[1]This chapter is a refined and reproduced version of the papers originally published in International Journal of Embedded Systems [58] copyrighted by Inderscience Publishers and in International Workshop on Software and Compilers for Embedded Systems (SCOPES) [59].

ecution. The number of threads in a malleable task can be changed throughout the runtime. Therefore, the malleable task model is the one generalized from the moldable task model with preemption and migration.

In the real world, many tasks for parallel applications are moldable [6]; however, there remains little research that focuses on the scheduling of moldable tasks. Moreover, most of the existing works in scheduling problems for moldable tasks have assumed a strongly idealized model of the parallel systems. One of the classical assumptions in the scheduling of parallel tasks has been to ignore communication costs between the cores. Based on this premise, it can easily maximize the utilization of multicore to map tasks to each core; however, it incurs the communications among cores in the real world and cannot ignore.

There were several works such as [61, 62, 63], and [64] assume that communication to incur among the cores for data transfer. In parallel programs, communication between the cores has a strong impact on a scheduling result since the cores carry out various threads of a task with different sizes of data. For a moldable fork-join (MFJ) task, internal communication between the cores for shared memory accesses or message passing inevitably leads to the degradation of latency.

In the scheduling of moldable tasks, Shimada et al. proposed a scheduling method with inter-task communication and tried to reduce the communication overhead to the overall execution time [65]. This study assumes a synchronous task model where threads in a task are considered to simultaneously run in parallel on different cores, but intra-task communication is not considered. MFJ tasks are multi-threaded, with each thread running independently on a core. If multiple threads are distributed across several different cores, the threads need to transfer data frequently between the cores. Also, even if multiple cores are provided for parallel execution, the speed of task execution is limited by Amdahl's Law. Therefore, inter- and intra-task communication for task scheduling is important to improve the performance of parallel computing systems. In addition, although this research assumes a synchronous parallel task model, it has a drawback that most tasks in real parallel programs have a fork-join structure, and threads can be scheduled independently.

The contributions of this work are as follows: Our work formulates scheduling of MFJ tasks considering inter- and intra-task communication costs for the first time. Data-parallel tasks are nowadays popular, but there are few works to tackle communication overheads among cores. Moreover, there have never been such scheduling problems for moldable tasks. Another contribution indicates we propose a two-step approach to the scheduling problem for efficiently finding good solutions in a practical time. Throughout the experiments, we show the effectiveness of the scheduling techniques for task graphs generated at random and several real-world applications. Furthermore, to evaluate the impact of communications among the cores on makespan, we evaluate the scheduling results with various computational communication ratios. The rest of the chapter is organized in the following: Section 4.2 describes related the literature of scheduling techniques for parallel tasks. Section 4.3 presents the proposed techniques for MFJ task scheduling with inter- and intra-task communications. Section 4.4 experiments performance evaluation with the state-of-the-art scheduling techniques and the proposed techniques and for the effect on communication cost to computation cost ratio (CCR) on multicore architecture. Finally, Section 4.5 concludes the chapter.

4.2 Related Work

Scheduling of parallel tasks on multicore architectures has been widely investigated by the works [34, 66, 67]. Scheduling of tasks on multicore is classically assumed to assign a single core to a task, and multiple tasks are executed on the cores in parallel, independently. On the other hand, a modern task called a parallel task is assumed to be executed on multiple cores. In other words, both the task parallelism and the data parallelism are necessary to be exploited [68] and [69] so that the potential of parallelization in a task is fully utilized. In the execution of parallel tasks, there are well-known two main models, called Gang and Thread [70] and [71]. The gang-constrained task model is assumed that all parallel computational elements of instances for a task start and end the execution with synchronization. On the other hand, there is no such constraint for the order-constrained model; therefore, each of the threads does not have to be synchronized with the other threads and can be started independently in a fork-join manner. Since our research focuses on multi-threaded tasks in the fork-join task model, the following works are aimed at the thread model of parallel tasks.

Scheduling techniques for multi-threaded tasks have been widely studied for a decade. In [35], they presented list-based PCS scheduling algorithms oriented to data-parallel tasks. Their work assumes that a set of dependent tasks is given formed as a DAG-based task-graph, where the number of threads for a task is not determined beforehand. The work attempts to minimize the makespan. Furthermore, they extended the work to solve the scheduling problem efficiently by a dual-mode algorithm [71] and by a branch-and-bound approach [72]. The work [38] also focused on scheduling of data-parallel tasks. Unlike these works by [71] and [72], the work [38] assumed that tasks are moldable, where the parallelism for a task is unfixed in advance and is determined during scheduling. The goal of the work aims to minimize hardware costs under deadline constraints. In [45], the authors proposed a technique for scheduling and mapping pipelined tasks on a heterogeneous MPSoC using data parallelism. Jansen et al. [73] tried to exploit the monotony of a moldable job. They developed polynomial approximate algorithms in m to polynomial in log m, where m denotes the number of machines. Chen et al. [44] designed a polynomial-time approximation algorithm for malleable tasks to find a minimum schedule length. In [46], they developed an algorithm for tasks in fork-join structure using OpenMP. The work in [32] developed scheduling techniques for moldable tasks based on constraint programming. Compared to existing techniques that are based on ILP formulation [37], the constraint programming approach successfully shortened the schedule length.

One of the drawbacks of the works mentioned above, they were extremely idealized and did not consider communication for data transfer among multiple cores. However, communication overhead becomes significantly large and seems to badly affect the performance of parallel computing. Hwang et al. [74] proposed a non-preemptive list scheduling problem that involves inter-processor communication delay, based on the earliest task first policy. They aimed at minimization of the schedule length. Yang et al. [75] developed heuristic list-scheduling techniques that introduce not only critical path scheduling but also ready list priority. In this literature, the communication overhead is generally supposed to incur when data-dependent tasks are mapped on different cores. The data transfer rate between two cores has been assumed to be constant in general [76]; therefore, the communication costs are almost equal to the amount of data for transfer. The work in [61] developed task scheduling techniques on multiprocessors with inter-processor communication as well as dynamic and static load balancing strategies. In this work, they employed a concept of the communication cost that refers to the amount of time necessary for data transfer, called CCR. Davidovic et al. [62] studied a set of benchmark instances that includes communication overhead. The work in [64] proposed MILP formulation for scheduling with communication delays on parallel systems. Morady et al. [77] studied scheduling of tasks considering communication costs. They employed a genetic algorithm as a metaheuristic technique to solve the scheduling problem since task scheduling is well-known as an NP-hard problem. Roy et al. [78] developed an ILP-based strategy for an optimal solution of precedence-constrained task-graphs (PTG) on a heterogeneous environment. They argued that heuristic strategies for scheduling PTG generally assume that processors are fully connected to avoid communication contention, but the heterogeneous multiple cores are usually connected with heterogeneous communication resources. Typically, parallel applications have long execution times. In other words, the data size in the programs to execute is large, and the CCR of the tasks in such a parallel program is high. Therefore, such high CCRs lead to a scheduling result critically different from the one with low CCRs.

Most of the existing works aware of communication costs, unfortunately, have assumed that a task is run on only a single core. On the other hand, the work in cites shimada2019communication developed moldable task scheduling techniques and considers inter-task communication; however, multi-threaded tasks frequently incur both inter- and intra-task communications. As increasing the number of threads for a task, intra-task communications may lead to non-negligible interference for efficient execution. Moreover, the communications further strongly affect a scheduling result if the CCR of the tasks is high. Therefore, we study the scheduling problem for moldable tasks with the communications and evaluate how much scheduling by different CCRs affects the results of schedule lengths.

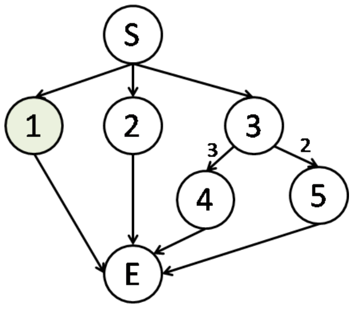
4.3 Scheduling Problem

In this section, we present the scheduling problem for MFJ tasks on homogeneous multiple cores. The fork-join task model satisfies the fork-join programming manner, where a single master thread partitions into multiple computational threads with a fork command and gathers the results of the parallel computation with a join command. Namely, scheduling of MFJ tasks determines the number of the computational threads and assignment of all the threads on multicore at the same time. Since this work takes into account inter- and intra-task communication costs for data transfer, our proposed technique also schedules the communications.

4.3.1 Problem Description

Figure 4.1 shows the example of scheduling of MFJ tasks on homogeneous multiple cores. In Figure 4.1(a), a set of dependent tasks is represented as DAG-based task-graph. The tasks labelled “S” and “E” are dummy tasks that represent entry and exit points, respectively and these nodes have no computational workloads. Edges represent data-dependent precedence constraints for the tasks and the weights on the edges indicate inter-task communication between two tasks if they are executed on different cores respectively. The nodes between the precedence constraints imply that a predecessor task must be finished before a successor task starts. In this work, the following classical assumptions are adopted; local communication cost is free, there is a communication subsystem so that cores are not involved in communication overhead, and communications can incur concurrently so that there is no contention for the communication resources. No that this example consider no communication time for outgoing edges from the entry node and incoming edges to the exit node for simplicity but without loss of generality. Also, each task is allowed to be multi-threaded as a fork-join structure.

In this work, we assume fork-join structure as follows: A task consists of three types of threads, which are called as a pre-processing, body and a post-processing threads,



(a) Task-graph

	Execution times of task 1			Communications	
#par	Pre-process	Body	Post-process	Pre-intra	Post-intra
1	0	40	0	0	0
2	3	21	3	2	2
3	5	14	5	1	1
4	6	12	6	1	1

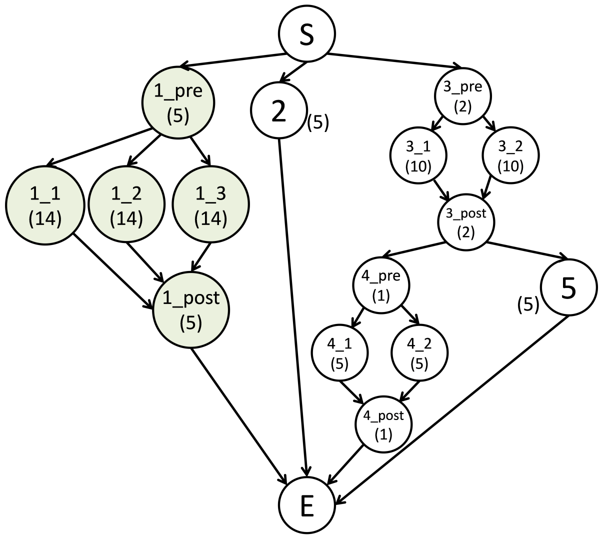
(b) Computation times and intra-task communication times for Task 1

Figure 4.1: An inputs of MFJ task scheduling with inter and intra-task communications

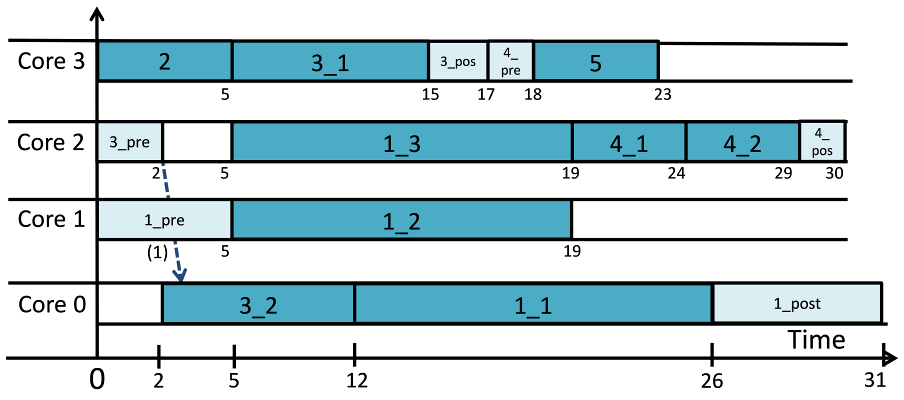
respectively. The pre-processing thread is called one master thread, which is divided into several parallel threads called body threads. The body threads are allowed to run in parallel on multiple cores at different cores at different time. When the execution of the body threads is done, the post-process thread synchronously terminates these body threads and resumes the execution of the master thread. It is assumed that the execution time of a thread depends on the number of body threads into which the task is split. Assume that the execution time of each task and the communication time (or data transfer time) between the tasks in the priority dependency are profiled and given in advance. There are several works for estimation of such information [79], but the topic is out of scope in this work. For simplicity, Figure 4.1(b) shows the execution times of each of the threads for Task 1.

If Task 1 is not parallelized, the execution time will be 40. In this case, there will be no pre-processing thread and no post-processing thread respectively. There will be no thread of processing and no thread of post-processing respectively. For example, given a task-graph like the one in Figure 4.1(a). In addition, we are given a table of computation time per thread and communication time between threads as shown in Figure 4.1(b). Then our scheduling technique simultaneously determines the number of body threads in each task and the mapping of the threads in each task. By simultaneously determining the number and mapping of threads for each task, the overall length of the schedule can be minimized.

Assume that neither inter- nor intra-task communications, Task 1 is split into three threads is represented in Figure 4.2(a); the computation times of every body thread is decided as 14. Moreover, of necessity are the pre-process thread and the post-process thread, each of which respectively takes 5 time-units. Figure 4.2(b) presents the scheduling result of Figure 4.2(a), and it shows an optimal overall schedule length is 31. If the communications are taken into account besides scheduling, the intra-task communication incurs by 1 from Core 2 to Core 0, where Core 2 transfers the data of Task 3 to



(a) Task 1 is split into three parallel threads without the communication time

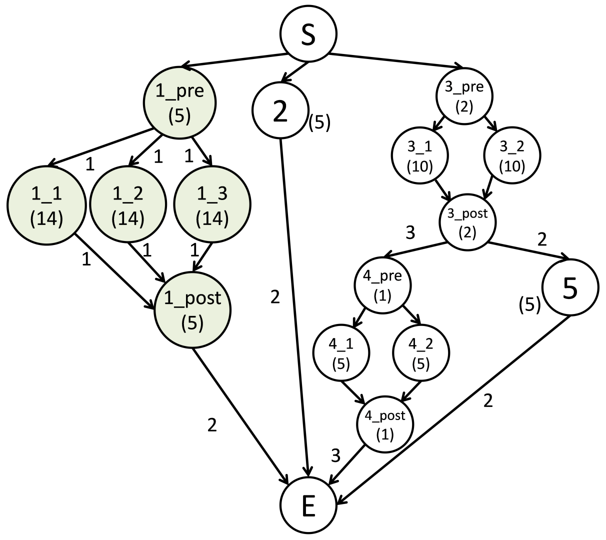


(b) Communication time affects the overall schedule length

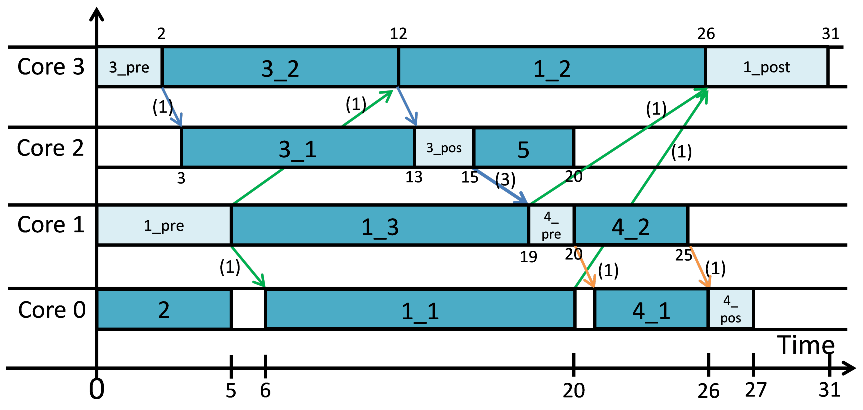
Figure 4.2: A scheduling result without communication delay

Core 2. Then, the schedule length is extended to 32. On the other hand, Figure 4.3(a) represents the problem with considering the communications. Each edge from the pre-process to the body threads in Task 1 has communication time of 1 in Figure 4.1(b).

The communication times are diverse depending on the parallelism of a task due to data size fo transfer depending on the number of body threads, followed by the CCR [61] and [64]. This example assumes the CCR is less than 0.1 for comprehension. Otherwise, if the CCR is higher than 0.1, the communications could incur degradation of latency on the schedule. Figure 4.3(b) shows the scheduling result of Figure 4.3(a). The pre-process thread of Task 1 is assigned to Core 1. Out of the three-body threads of Task 1, the third thread is mapped on the same core (Core 2), and no communication is incurred. On the other hand, the first body thread of Task 1 is mapped on Core 0,



(a) Task 1 is split into three parallel threads with the communication time

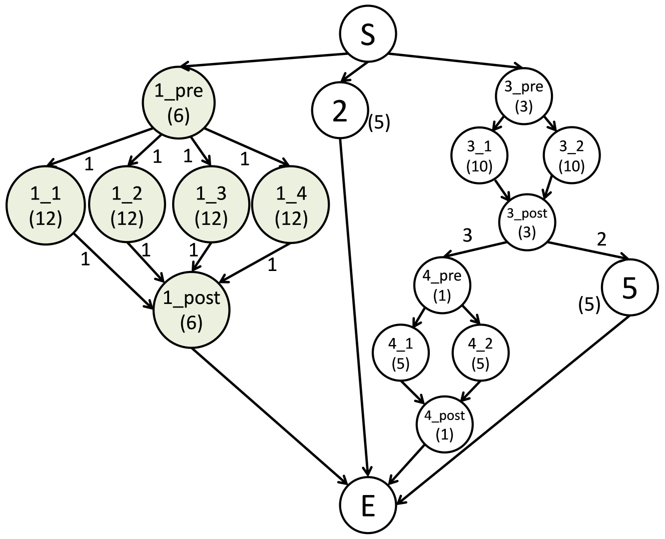


(b) An optimal scheduling result of the task-graph

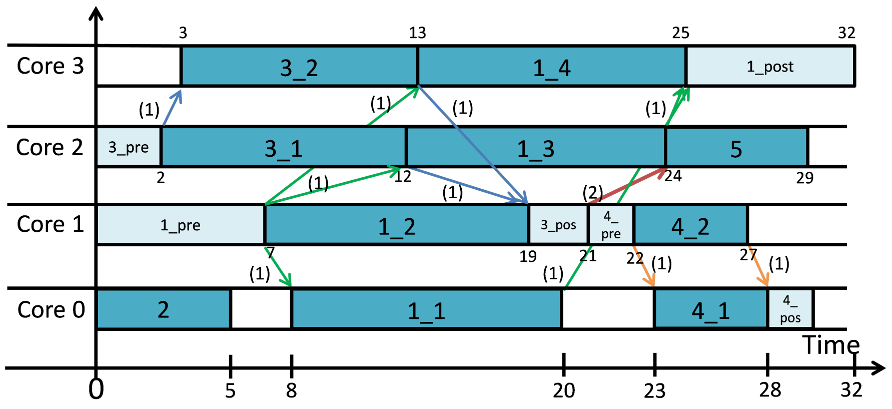
Figure 4.3: A scheduling result taken into account communication delay

and which is different from the pre-process thread. Therefore, the first body thread is delayed by a 1-time unit due to transferring data from Core 1 to Core 0. Similarly, the pre-process of Task 4 on Core 1 is delayed by 3-time units due to data transfer from the post-process of Task 3 on Core 2. Compared to Figure 4.2(b), the shorter schedule length is obtained. Figure 4.4(a) shows the case if Task 1 is split into four threads; however, the schedule length becomes longer, compared to the case with three threads as shown in Figure 4.4(b). Through the example, each degree of parallelism of a task and the communications could affect the scheduling result. As the task graph becomes even larger, the impact of parallelism and communication is expected to increase.

The scheduling problem presented above is much more complex than the existing scheduling problems. The thread execution time for each task depends on how many



(a) A task-graph with multi-threaded tasks



(b) The longer schedule than the previous result

Figure 4.4: A optimal scheduling result without considering communication delay

body threads the task is parallelized into. The intra-task communication time depends not only on how many tasks are parallelized but also on which core the task is mapped to. The inter-task communication time also depends on the mapping of the post-processing thread of the preceding task to the pre-processing thread of the following task. To minimize the schedule length, all sub-problems need to be solved in one optimization framework.

4.3.2 IP Formulation

The rest of this section formulates our scheduling problem as an IP problem. Let $par_{i,k}$ denote a 0-1 decision variable, and which becomes 1 if task i is determined to be split into k body threads, otherwise 0.

$$\forall i, \qquad \sum_k par_{i,k} = 1 \tag{4.1}$$

Let $Time\_pre_{i,k}$, $Time\_body_{i,k}$ and $Time\_post_{i,k}$ denote execution times of the pre-process thread, body threads and the post-process thread of task i, respectively, if the task is partitioned into k body threads. $Time\_pre_{i,k}$, $Time\_body_{i,k}$ and $Time\_post_{i,k}$ are assumed to be given. The execution times of the pre-process thread, body threads and the post-process thread of task i are given by:

$$\forall i, \qquad time\_pre_i = \sum_k Time\_pre_{i,k} \times par_{i,k} \tag{4.2}$$

$$\forall i, \qquad time\_post_i = \sum_k Time\_post_{i,k} \times par_{i,k} \tag{4.3}$$

$$\forall i, \qquad time\_body_i = \sum_k Time\_body_{i,k} \times par_{i,k} \tag{4.4}$$

Let $Comm\_intra\_pre_{i,k}$ and $Comm\_intra\_post_{i,k}$ denote communication time from the pre-process thread to the body threads and from the body threads to the post-process thread, respectively, if the task is split into k body threads. $Comm\_intra\_pre_{i,k}$ and $Comm\_intra\_post_{i,k}$ are assumed to be given. Let $pre\_intra_{i,j}$ be a 0-1 decision variable, which becomes 1 if the intra-task communication is necessary between the pre-process thread and the $j-th$ body thread in task i. Similarly, let $post\_intra_{i,j}$ be a 0-1 decision variable, which becomes 1 if the intra-task communication is necessary between the $j-th$ body thread and the post-process thread in task i. Then, the communication time from the pre-process thread and the $j-th$ body thread and the communication time from the $j-th$ body thread and the post-process thread in task i are given by:

$$\begin{aligned} \forall i,j \qquad & comm\_intra\_pre_{i,j} \\ & = \textstyle\sum_k Comm\_intra\_pre_{i,k} \times par_{i,k} \times pre\_intra_{i,j} \end{aligned} \tag{4.5}$$

$$\begin{aligned} \forall i,j \qquad & comm\_intra\_pre_{i,j} \\ & = \textstyle\sum_k Comm\_intra\_pre_{i,k} \times par_{i,k} \times pre\_intra_{i,j} \end{aligned} \tag{4.6}$$

$pre\_intra_{i,j}$ is 1 if the pre-process thread and the $j-th$ body thread of task i are mapped to different cores. Similarly, $post\_intra_{i,j}$ is 1 if the $j-th$ body thread and the post-process thread of task i are mapped to different cores. Let $map\_pre_{i,k}$ and $map\_post_{i,k}$ be 0-1 decision variables, which are 1 if the pre-process thread and the post-process thread of task i are mapped to the $k-th$ core, respectively. Let $map\_body_{i,j,k}$ be a 0-1 decision variable, which is 1 if the $j-th$ body thread of task i is mapped to the $k-th$ core.

$$\forall i,j \qquad pre\_intra_{i,j} = \begin{cases} 0 \; if \; map\_pre_{i,k} = map\_body_{i,j,k} \; for \; any \; k \\ 1 \; otherwise \end{cases} \tag{4.7}$$

$$\forall i,j \qquad post\_intra_{i,j} = \begin{cases} 0 \; if \; map\_body_{i,k} = map\_post_{i,k} \; for \; any \; k \\ 1 \; otherwise \end{cases} \tag{4.8}$$

Let $start\_pre_i$ and $finish\_pre_i$ denote start and finish times of the pre-process thread of task i, respectively. Similarly, $start\_body_{i,j}$, $finish\_body_{i,j}$, $start\_post_i$ and $finish\_post_i$ are defined for the $j-th$ body thread and the post-process thread:

$$\forall i,j \qquad finish\_pre_i = start\_pre_i + time\_pre_i \tag{4.9}$$

$$\forall i,j \qquad finish\_body_{i,j} = start\_body_{i,j} + time\_body_i \tag{4.10}$$

$$\forall i, \qquad finish\_post_i = start\_post_i + time\_post_i \tag{4.11}$$

For a task, the pre-process thread must be finished before any of the body threads start, and all the body threads must be finished before the post-process thread starts. Then, precedence constraints inside of a task which consider intra-task communication are expressed by:

$$\forall i,j \qquad finish\_pre_i + comm\_intra\_pre_i \leq start\_body_{i,j} \tag{4.12}$$

$$\forall i,j \qquad finish\_body_{i,j} + comm\_intra\_post_{i,j} \leq start\_post_i \tag{4.13}$$

The body threads of a task can run independently on multiple cores in parallel. The body threads may be mapped on different cores for parallel execution, or they may be mapped on a same core. $map\_body_{i,j1,k}$ represents that the value becomes 1 if $j1-th$ body thread of task i is mapped on $k-th$ core, otherwise 0. Thus, if two of the body threads of task i are mapped on the same core, the resource constraint for prohibiting overlapping must be met in the following:

$$\begin{aligned} \forall i, j1, j2, k, j1 \neq j2 \qquad & map\_body_{i,j1,k} = map\_body_{i,j,k} \\ \rightarrow \qquad & finish\_body_{i,j1} \leq start\_body_{i,j2} \\ \vee \qquad & finish\_body_{i,j2} \leq start\_body_{i,j1} \end{aligned} \tag{4.14}$$

Similar to the formula (4.14), the overlap constraints for each thread of different two tasks are needed to be satisfied. If a core that executes a pre-process thread of task $i1$ is the same core where a pre-process thread of task $i2$ is assigned, they cannot be overlapped at the same time. Let $map\_pre_{i,k}$ be 0-1 decision variable, which becomes 1 if the pre-process thread of task i is mapped on $k - th$ core, otherwise 0. In order to avoid such an overlap, the formula is given by:

$$\begin{aligned} \forall i1, i2, k, i1 \neq i2 \qquad & map\_pre_{i1,k} = map\_pre_{i1,k} \\ \rightarrow \qquad & finish\_pre_{i1} \leq start\_pre_{i2} \\ \vee \qquad & finish\_pre_{i2,j2} \leq start\_pre_{i1} \end{aligned} \tag{4.15}$$

If $j - th$ body thread in task $i2$ and the pre-process thread of task $i1$ are mapped on the same core, they are assigned to the core so that their executions are prohibited to overlap at the same time as well.

$$\begin{aligned} \forall i1, i2, k, i1 \neq i2 \qquad & map\_post_{i1,k} = map\_post_{i1,k} \\ \rightarrow \qquad & finish\_post_{i1} \leq start\_post_{i2} \\ \vee \qquad & finish\_post_{i2,j2} \leq start\_post_{i1} \end{aligned} \tag{4.16}$$

Let $map\_post_{i,k}$ be 0-1 decision variable that becomes 1 if the post-process thread of task i is mapped on $k - th$ core. For the post-process thread of task i2, the overlap constraint is also satisfied.

$$\begin{aligned} \forall i1, i2, k, i1 \neq i2 \qquad & map\_post_{i1,k} = map\_post_{i1,k} \\ \rightarrow \qquad & finish\_post_{i1} \leq start\_post_{i2} \\ \vee \qquad & finish\_post_{i2,j2} \leq start\_post_{i1} \end{aligned} \tag{4.17}$$

Each of the body threads of two different tasks $i1$ and $i2$ are not executed on the same core at the same time. The overlap constraint is in the following:

$$\begin{aligned}\forall i1,i2,j1,j2,k,i1 \neq i2 \qquad & map\_body_{i1,j1,k} = map\_body_{i2,j2,k} \\ \rightarrow \qquad & finish\_body_{i1,j1} \leq start\_body_{i2,j2} \\ \vee \qquad & finish\_body_{i2,j2} \leq start\_body_{i1,j1} \end{aligned} \tag{4.18}$$

If a body thread in task $i1$ and the post-process thread of task $i2$ are mapped on same core, the overlap constraint is met as follows:

$$\begin{aligned}\forall i1,i2,k,i1 \neq i2 \qquad & map\_body_{i1,j,k} = map\_post_{i2,k} \\ \rightarrow \qquad & finish\_body_{i1,j,k} \leq start\_post_{i2,k} \\ \vee \qquad & finish\_post_{i2,k} \leq start\_body_{i1,j,k} \end{aligned} \tag{4.19}$$

The following formula is defined from the post-process thread of task $i1$ to the post-process thread of task $i2$.

$$\begin{aligned}\forall i1,i2,k,i1 \neq i2 \qquad & map\_post_{i1,k} = map\_post_{i2,k} \\ \rightarrow \qquad & finish\_post_{i1} \leq start\_post_{i2} \\ \vee \qquad & finish\_post_{i2} \leq start\_post_{i1} \end{aligned} \tag{4.20}$$

So far, we have focused on precedence constraints and resource constraints within a task. Similarly, the constraints for inter-task also are required to be satisfied. Let $inter_{i1,i2}$ denote 0-1 decision variable that becomes 1 if the post-process thread of task $i1$ is mapped on the different core that is assigned task $i2$.

$$\forall i1,i2 \qquad inter_{i1,i2} = \begin{cases} 0 \; if \; map\_post_{i1,k} = map\_pre_{i2,k} \; for \; any \; k \\ 1 \; otherwise \end{cases} \tag{4.21}$$

Let $Comm\_inter_{i1,i2}$ denote communication time from the post-process thread of task $i1$ to the pre-process thread of task $i2$, which is given in advance. If inter-task communication is incurred between task $i1$ and $i2$, the formula for the inter-task communication is given by:

$$\forall i1,i2 \qquad comm\_inter_{i1,i2} = Comm\_inter_{i1,i2} \times inter_{i1,i2} \tag{4.22}$$

In the formula (4.22), $iner\_i1, i2$ is 0-1 decision variable. $inter\_i1, i2$ becomes 1 if the post-process thread of task $i1$ and the pre-process thread of task $i2$ are executed on different cores, otherwise it becomes 0.

If there is a precedence dependency from task $i1$ to $i2$, the start time of successor task $i2$ can be executed after the post-process thread of predecessor task $i1$. Moreover, the inter-task communication may incur at the time. Then, a precedence constraint which considers inter-task communication is expressed by:

$$\forall i1,i2 \qquad finish\_post_{i1} + comm\_inter_{i1,i2} \leq start\_pre_{i2} \tag{4.23}$$

If a task is partitioned into threads, inter-task communication may incur among the cores due to data transfer. In this work, the communication is incurred if any of the two threads in a task are mapped on different cores respectively. In contrast, if the task is determined as single-core execution, there is no communication that incurs among the cores. The following formula shows that the case that the parallelism of a task is 1. Single-core execution indicates pre-processing, body, and post-processing threads do not exist. Therefore, the pre-procesing thread and the post-processing thread requires no computational workload, respectively but assume to be mapped on the same core in order to ignore the communication time.

$$\begin{aligned} \forall i,j \qquad & par_{i,1} = 1 \\ \rightarrow map\_pre_{i,k} \wedge & map\_body_{i,j,k} \wedge map\_post_{i,k} \end{aligned} \tag{4.24}$$

The goal of the scheduling problem is to minimize the overall schedule length. In other words, we aim to minimize the overall completion time of the tasks. The objective function is as follows:

$$\text{Minimize :} \qquad max_i\{finish\_post_i\} \tag{4.25}$$

As mentioned above, our scheduling problem is based on an integer programming. Although some of the formulas above are not in a linear form, they can be easily linearized by simple transformation techniques. Therefore, this problem is able to be solved by employing general-purpose ILP solver software. For simplicity, we do not consider contention between hardware resources such as memory bandwith, bus delay, and so on. Further, the tasks in this problem are well-ideal in that the execution times of each task are not jitters. Each thread partitioned from a task assumes to have the same execution time as another thread of the same task. These extension are out of this work, but they can be easily realized by adding some constraints.

4.3.3 Two-Phase Heuristic Approach

In general, task scheduling is well known as an NP-hard problem. The scheduling technique presented in the previous section is much more complicated than existing task scheduling problems due to considering multi-threading and communication. Therefore, our scheduling problem includes the issue that it can hardly find a good schedule within a practical time. To overcome the issue, we propose a two-step heuristic technique for the scheduling technique. The first step represents moldable synchronous (MS) task scheduling. Unlike the scheduling technique presented in the previous section, the MS task scheduling does not allow the body threads to run on multicore independently but run simultaneously with synchronization. This scheduling technique generally determines the number of the body threads and the execution order of the threads. Through the MS task scheduling, we obtain a schedule, where the number of threads for each task and mapping of the threads on multiple is determined. Afterward, fork-join (FJ) task scheduling is utilized with given the parallelism for each task obtained by the MS task scheduling. Therefore, the FJ task scheduling simply determines the execution order of the tasks. Note that both techniques consider inter- and intra-task communications during scheduling.

Since MS task scheduling is a subset of our technique described in the previous section optimal in terms of synchronization to execute the threads, we can easily obtain the formulation for MS task scheduling by adding the following formula to the formulation earlier presented in the previous section.

$$\forall i, j1, j2 \qquad start\_body_{i,j1} = start\_body_{i,j2} \tag{4.26}$$

MS task scheduling, in the first step, performs to decide the number of body threads for each task for a certain period, and we can obtain $par_{i,k}$ that is the degree of parallelism for each task. The tasks with the fixed parallelism are represented as rigid. Then, for a certain period, the threads are scheduled with the fixed degree of parallelism of the tasks during FJ task scheduling.

4.3.4 Limitations

There are several limitations to this scheduling technique. We assume that each task consists of several threads, and that these threads are running on a homogeneous multicore. We then assume that the execution times of the threads are computationally equivalent, even on different cores. Therefore, the scheduling techniques in this work are largely inapplicable to heterogeneous multicore platforms. In addition, this work assumes that the communication links between the cores are ideal. This work also assumes that the communication link between the cores is ideal, does not consider contention between simultaneous communications, and the communication time does not depend on the physical distance between the two cores. These limitations will be relaxed in the future.

4.4 Experiments

This section presents the experiments of the proposed techniques. In order to evaluate this work, we conduct a set of experiments. Thirteen random task-graphs generated by TGFF [56] are scheduled on 4-core and 8-core target systems, respectively. Besides the randomly generated DAGs, the performance of our proposed techniques is also evaluated for three real-world applications, i.e., the Broadband workflow, the Cybershake workflow, and the Montage workflow in [80] Each task in the task-graphs can be split into several threads, where the degree of parallelism for each task is varied from one to the number of cores on the target system. As a solver, ILOG CP Optimizer 12.8 is employed to find solutions. Since task scheduling problem is generally very complex, it is impossible to find exactly optimal solutions in practical time. Thus, we limit the runtime of the solver up to five hours in wall-clock time, and the best solutions found at the moment are selected for the evaluation. The experiments are performed on Intel Core i9 7980XE (2.6GHz) processor with 128 GB memory.

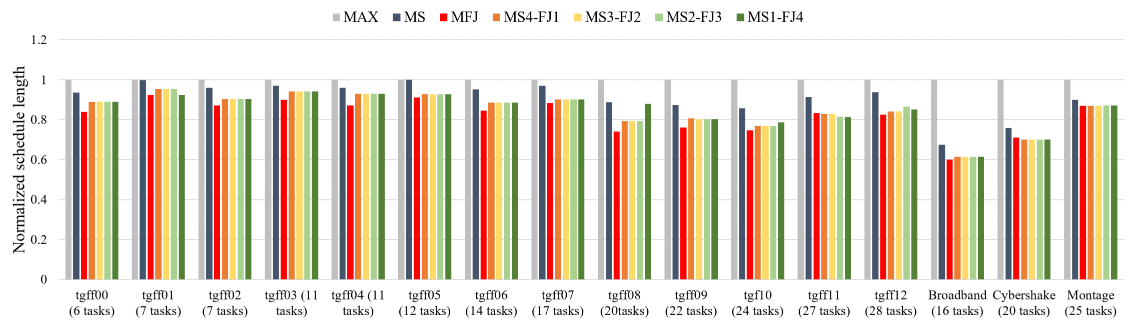
We compare the four scheduling techniques as follows.

- *Max*: Each task is assigned to all cores. In other words, each task is split into N body threads for the N-core target system. Then, the tasks are sequentially scheduled. If master threads are always executed on the same core, the inter-task communication is not incurred in the schedule. On the other hand, intra-task communication must be incurred due to the sequential execution of the tasks.

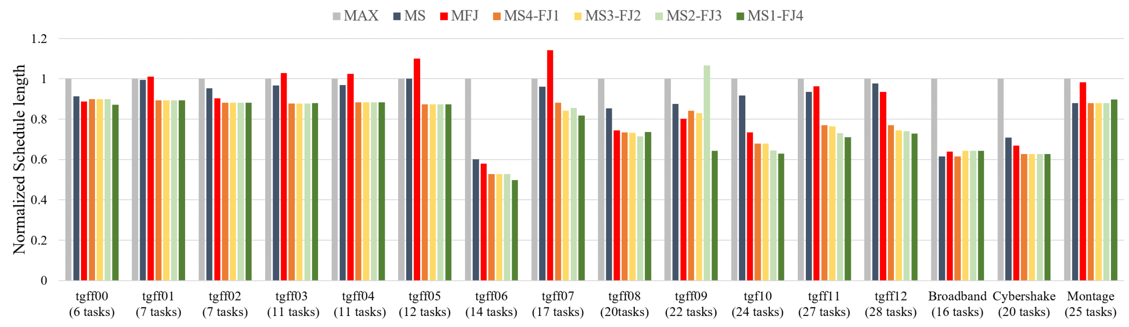
- *MS*: Moldable task scheduling technique in [36]. Unlike this work, all the body threads in a task are started at the same time with synchronization. Thus, intra-task communication must be incurred for each task.

- *MFJ*: Moldable fork-join task scheduling technique proposed in this work. This technique schedules not only threads but also the inter and the intra-task communications. The threads can be executed on multicore independently.

- *MS-FJ*: A two-step heuristic approach to the MFJ technique. For the first several hours, the MS technique is performed to decide the number of body threads for each task. Then, for the remaining the hours, the threads are scheduled with the determined parallelism.



(a) Scheduling results on 4 cores



(b) Scheduling results on 8 cores

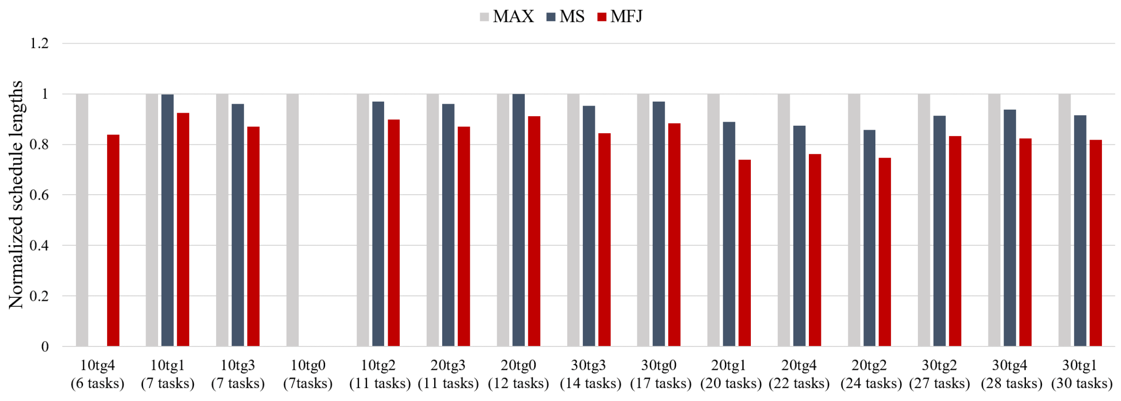
Figure 4.5: Performance evaluation with the proposed techniques

In the experiments, we employ four techniques with different parameters to investigate the effectiveness of the MS-FJ technique, called as MS4-FJ1, MS3-FJ2, MS2-FJ3, and MS4-FJ1, respectively. For example, the MS4-FJ1 technique represents that the MS technique is performed in four hours, and the MFJ technique performs in an hour with the parallelism.

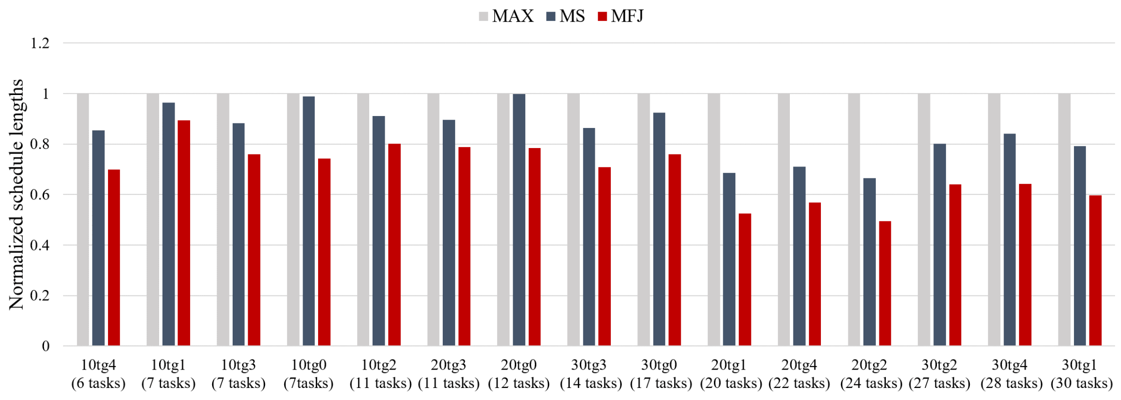
4.4.1 Performance Results

The results of scheduling on 4 and 8 cores for performance evaluation are shown in Figure 4.5 (a) and (b), respectively. The x-axis of the graph represents the task-graph, and the numbers in the suffixes mean the identification of the task-graph. The numbers in parentheses represent the number of nodes in the task-graph. The Y-axis represents the length of the schedule obtained by the seven techniques. The Y-axis shows the length of the schedule obtained by the seven techniques, normalized by the MAX technique.

Figure 4.5(a) shows the scheduling results on four cores. The MFJ technique yields shorter schedules than the other techniques. Due to the huge solution space, the MFJ technique can flexibly schedule the threads in such a way that the inter- and intra-task communications are hidden in the schedule. On average, the MFJ technique achieves the short schedules by 22.5%. In cases such as 27 tasks and the Cybershake workflow, which consists of 20 tasks, the MFJ technique shows a long schedule than the heuristic



(a) 10% of inter-task and 5% of intra-task for communication overheads

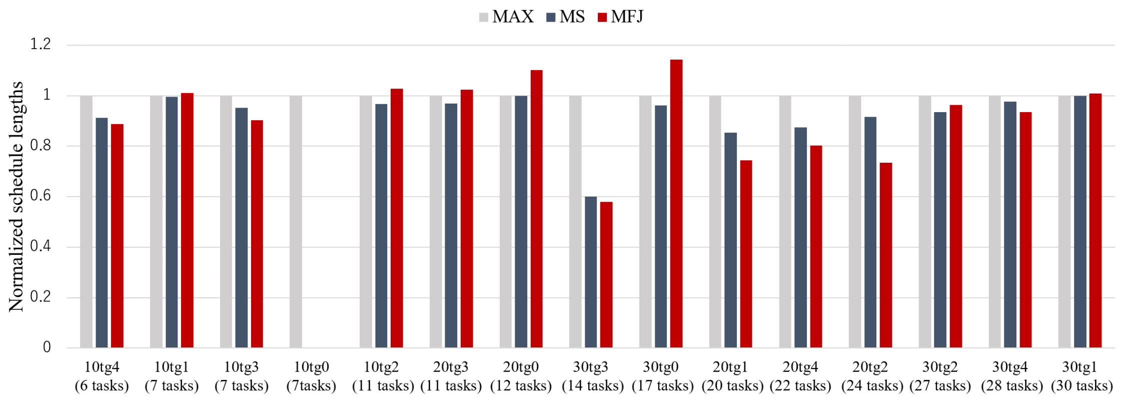


(b) 50% of inter-task and 25% of intra-task for communication overheads

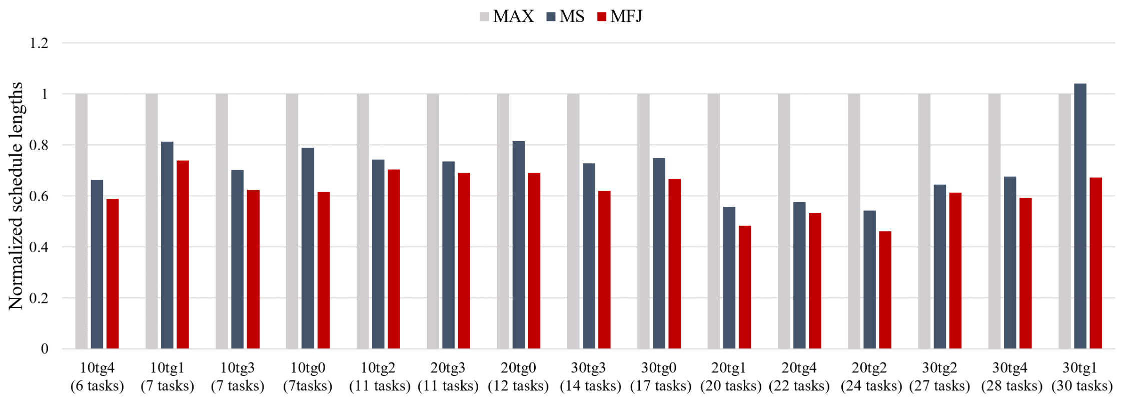
Figure 4.6: Scheduling results on 4 cores

techniques, on the other hand, the MS1-FJ4 finds the best solution. The overall results show that more time is spent on determining parallelism than on a schedule. For 20, 24, 27, and 28 tasks, the heuristic techniques show slightly different results. The results imply that the most effective heuristic method depends on the task graph. Furthermore, we have conducted experiments on three applications and shown that the proposed technique is effective compared to the state-of-the-art techniques.

Figure 4.5(b) presents the results on eight cores. As a result, the MFJ technique cannot find a better schedule than the other techniques within 5 hours due to its huge solution space. On the other hand, for tasks 7 and 11, the heuristic techniques can obtain a good schedule. This result shows empirically that the scheduling of threads and communication has a greater influence on the length of the schedule than the degree of parallelism of the tasks. As the number of tasks in the task-graph increases, the heuristic method yields shorter schedules, but when the number of tasks is 22, the MS2-FJ3 method yields the longest schedule length. This shows that the parallelism of the tasks has a stronger influence on the results than the scheduling of threads and communication.



(a) 10% of inter-task and 5% of intra-task for communication overheads



(b) 50% of inter-task and 25% of intra-task for communication overheads

Figure 4.7: Scheduling results on 8 cores

In addition, the MS technique finds good solution in the Montage workflow compared to the MFJ technique since the task-graph of the Montage workflow is symmetry in data dependencies so that the sub-tasks are preferred to run simultaneously. On the other hand, heuristic approaches such as MS4-FJ1, MS3-FJ2, and MS2-FJ3 find the same schedule as the MS technique.

4.4.2 Effect of CCRs

To evaluate the impact of the communications on scheduling results, we conduct the experiments with various. Of the twelve task-graphs, the CCRs for inter-task communications are 0.1 and 0.5 and those for intra-task communications are 0.05 and 0.25, respectively.

Figures 4.6 and 4.7 are scheduling results on four cores and eight cores, respectively. Figure 4.6(a) shows the scheduling results, where the CCR of inter-task is 0.1 and of intra-task is 0.05. On the other hand, Figure 4.6(b) shows the results, where the CCR of inter-task is 0.5 and of intra-task communication is 0.25. In Figure 4.6(a), the MFJ technique shortens the schedules by 16.0% on average, compared to the MAX technique. The MS technique obtains the schedules shortened by 6.3% on average. From the results on the eight cores, it is observed that an increase in CCR leads to improvement of the performance in all the cases by the MS technique and the MFJ technique.

Figure 4.7(a), where the CCR of inter-task is 0.1 and the CCR of intra-task is 0.05, shows that the MFJ technique seems difficult to find the better schedules within five hours due to the complexity of the scheduling problem. On average, the MFJ technique and the MS technique find shorter schedules by 8.1% and 7.8%, respectively. In Figure 4.7(b), on the other hand, the MFJ technique achieves shorter schedules by 38.0%, and the MS technique shortens by 28.7% on average. The results imply that the MS and the MFJ techniques can relatively shorten the schedule lengths since the MAX technique must incur intra-task communications for every task due to using all the cores at the same time. In contrast, the MS and the MFJ techniques can schedule the multi-threaded tasks in such a way that the effect of both inter- and intra-task communications are mitigated by hiding the communications in the schedule.

4.5 Conclusions

This work have presented MFJ task scheduling techniques based on IP, considering intra- and inter-task communications. This scheduling technique integrates both deciding the parallelism for each task during scheduling. Also, we evaluated the performance of the proposed techniques with several two-step heuristic approaches and studied the effect of different CCRs on scheduling results. The experimental results demonstrate that the proposed techniques can find greater schedules than existing state-of-the-art techniques. Regarding the study for the different CCRs, we show the effectiveness our scheduling technique can shorten schedule lengths in an increase of CCR. In future, we plan to develop rapid heuristic algorithms available for large task-sets.

Chapter 5

Simultaneous Scheduling and Core-type Optimization for Moldable Tasks on Heterogeneous Multicores

This chapter addresses energy-aware scheduling of moldable tasks with core-type optimization in heterogeneous multiple cores[1].

5.1 Introduction

Task scheduling on multicore that determines the execution order of tasks on multiple cores has become more important than ever due to the increasing number of cores in embedded systems. In general, task scheduling problems are NP-hard [39], and a large number of researchers have studied task scheduling problems over several decades. Task scheduling problems classically assume that tasks are scheduled in such a way that the tasks are executed in parallel on the different core while every task is mapped on one of the cores [6]. However, modern, realistic applications include data parallelism within the task. Tasks are essentially decomposed in a fork-join fashion, where the data is split into multiple small pieces which can be processed independently of each other. From this point of view, the scheduling of fork-joined tasks, in which each task is divided into multiple sub-tasks and executed on multiple cores, has been studied. In this study, we propose a method for scheduling moldable fork-join (MFJ) tasks based on integer linear programming (ILP) on a heterogeneous architecture consisting of big and little cores. In this work, we assume that the tasks are moldable. This implies that the number of cores allocated to a task is flexible. This research aims to meet the deadline constraint while

[1]This chapter is a refined and reproduced version of the paper to be published in IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences [81] copyrighted by IEICE.

minimizing energy consumption. We also propose a method to simultaneously perform task scheduling and multicore architecture customization. In this method, the type of core (large or small) is determined simultaneously with the task scheduling of the MFJ. Note that this work is an extended version of [82], and we have extended the work as follows.

- The state-of-the-art work in [82] compares the energy consumption from the perspective of dynamic energy consumed on the target systems by the cores used during executing tasks. In this work, we consider not only dynamic energy but also static energy consumption which is constantly consumed until all tasks end.
- We propose a warm-start approach to the scheduling problem since the previous work in [82] addresses that the proposed technique cannot find worse solutions than the traditional technique due to its huge solution space.

The rest of this chapter is organized as follows. Section 5.2 describes the related work on task scheduling. Section 5.3 proposes a MFJ task scheduling technique. Section 5.4 proposes a codesign technique of simultaneous MFJ task scheduling and core-type optimization. In addition, we present a warm start approach to the codesign technique. Section 5.5 describes experiments and the comparison, and Section 5.6 concludes this chapter.

5.2 Related Work

In [6], classic techniques on task scheduling for multicore architectures are extensively surveyed. Multiple tasks which are independent of each other are executed in parallel on different cores. However, it is assumed that each task is not parallelized and is executed on a single core. In [35], Liu et al. proposed list-based scheduling algorithms for data-parallel tasks. Their work assumes that a set of dependent tasks is given formed as a DAG-based task-graph, where each task is assigned a fixed number of cores. Then, they attempt to minimize the overall schedule length (a.k.a. makespan). Yang and Ha’ s work in [38] also focuses on the scheduling of data-parallel tasks. Unlike the work in [35], their work in [38] assumes that tasks are moldable, where the number of cores for a task is unfixed but is determined during scheduling. The objective of this work asks to find a solution that minimizes hardware cost by meeting deadline constraints. In [43], the authors take advantage of data-parallelism and proposed a technique for pipelined task scheduling and mapping on heterogeneous MPSoCs. Chen and Chu in [44] designed a polynomial-time approximation algorithm for malleable tasks to shorten schedule lengths The authors of [36] studied scheduling of moldable tasks based on integer linear programming. Furthermore, they proposed a moldable fork-join task scheduling technique [37]. They argued that the problems can hardly be solved with an ILP-solver in practical time. To deal with the issue, Nishikawa et al. proposed the same scheduling problems but was based on constraint programming in [36] and [37] so those better solutions are explicitly found in a practical time [32]. In [45], fork-join task scheduling for real-time systems has been studied. The work aims at the evaluation of the tractable and intractable fork-join real-time task model. Lakshmanan et al. in [46] developed an algorithm for moldable fork-join tasks in OpenMP. Saifullah et al. in [47] proposed a real-time task scheduling model, which assumes that a task holds the various numbers of threads. Another direction of studies on multicore task scheduling is for heterogeneous multicores [15, 13, 83], and [10]. In [15], Yan et al. studied a task scheduling problem on heterogeneous multiple processors for real-time applications, which tries to minimize whole energy consumption with two heuristic algorithms under deadline constraints.

Bridi et al. [13] developed a scheduler based on constraint programming for heterogeneous high-performance computing machines. In this work, a commercial scheduler is modified with a greedy approach to maximize performance and quality of service. We improve a commercially available scheduler with a greedy approach to maximize performance and quality of service. Task scheduling on heterogeneous multicores is also studied in [14]. Studying task scheduling in heterogeneous multicores, Barbosa et al. [83] propose a list-based static scheduling algorithm for moldable tasks. A list-based static scheduling algorithm is proposed to minimize the schedule length of dependent tasks. It aims at minimizing the schedule length of moldable tasks on heterogeneous clusters, where the dependencies between tasks are represented by DAGs. In addition, modern processors are Modern processors incorporate Dynamic Voltage Frequency Scaling (DVFS) technology, which dynamically adjusts supply voltage and frequency. There is a large body of literature on multi-task scheduling using DVFS. For example, Qin et al. [84] developed an ILP-based energy-aware task allocation technique for real-time tasks, which employs DVFS scheduling within a task. On the other hand, in this work, we assume a heterogeneous multi-core architecture without DVFS functionality. Different cores may have different frequencies and voltages, but each core cannot change its frequency and voltage at runtime. In this work, we study moldable fork-join task scheduling for energy minimization under deadline constraints using heterogeneous multicore architectures. To the best of our knowledge, this is the first work on this topic. It also differs from the previous literature in that it proposes both MFJ task scheduling and heterogeneous multicore customization at the same time.

5.3 Scheduling of Moldable Fork-Join Tasks on Heterogeneous Multicores

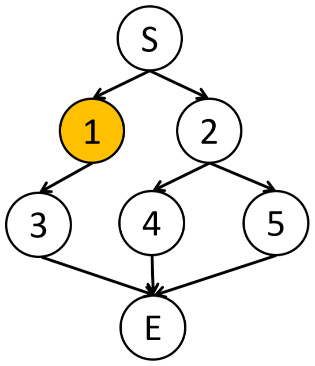
This section presents a problem description and an ILP-based formulation for scheduling moldable fork-join tasks on heterogeneous multicores. Unlike [82], this work classifies the energy consumption into dynamic and static energies. Dynamic energy is assumed to be consumed by the cores to perform tasks in the target system. Static energy, on the other hand, is always consumed until all tasks have been completed. The power consumption of the static energy depends on the type of cores in the architecture. Therefore, static energy is assumed to be generated when the core is idle unless all tasks are finished.

5.3.1 Problem Description

Figure 5.1 shows an example of scheduling of moldable fork-join (MFJ) tasks on heterogeneous multicores. In Figure 5.1(a), a set of dependent tasks is represented as a directed acyclic graph, so called a task-graph. Each task is associated with the execution time which is a function of the number and the type of cores to execute the task. The tasks labelled "S" and "E" are empty nodes which represent entry and exist points, respectively.

Figure 5.1(b) shows a table of the execution time of Task 1. If Task 1 is assigned to a single little core, its execution time is 32-time units. The execution time of Task 1 on a single big core is 21. It is assumed that Task 1 is parallelized and split into several sub-tasks. If Task 1 is split into two sub-tasks, the execution time of the sub-task on the little core is 18 and the execution time of the sub-task on the big score is 12. These two sub-tasks can be executed on two little cores, two big cores, or one big core and one little core. This work assumes that, for each task, a table of execution time as shown in Figure 5.1(b) is given. Task 1 consumes 32 units of dynamic energy when the task is executed on a little core without parallelization as shown in Figure 5.1(c). If task 1 is partitioned into two sub-tasks and executed on a little core and a big core respectively, the task consumes total dynamic energy of 59 (= 18 + 41). Here, we assume that the execution time and the energy consumption, which represent in Figure 5.1(b) and 5.1(c), are profiled in advance throughout running each task repeatedly on multicores. To simply reduce energy consumption without concern for performance, from a dynamic energy point of view Task 1 should not be parallelized but run on a single small core.

However, from a static energy point of view, dividing up the tasks will, on the contrary, reduce the overall energy consumption since the cores consume energy even when



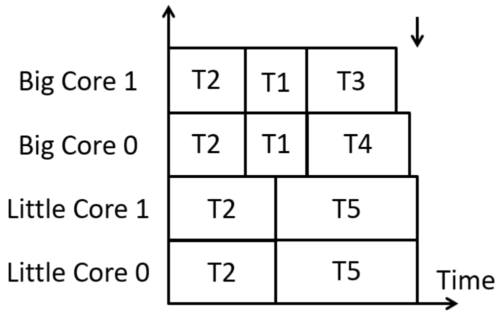
(a) Task graph

# sub-tasks	Little	Big
1	32	21
2	18	12
3	13	9
4	11	7

(b) Execution time of Task 1

# sub-tasks	Little	Big
1	32	72
2	18	41
3	13	30
4	11	25

(c) Dynamic energy consumption of Task 1



(d) A scheduling result

Figure 5.1: An example of scheduling of moldable fork-join tasks

they are idle until the task is finished. On the other hand, if performance is prioritized, the task should be parallelized and run on as many cores as possible. Between the minimum energy solution and the maximum performance solution, many alternatives take dynamic and static energy into account. Therefore, when it comes to scheduling, the trade-off between performance and energy consumption needs to be considered. In this work, we assume that the deadline constraint is given as fixed. In other words, the makespan must be shorter than or equal to the given deadline. Given a set of moldable tasks and a deadline constraint, we determine the number of sub-tasks for each task and schedule the sub-tasks on a heterogeneous multicore in such a way that the total energy consumption is minimized while meeting the deadline constraint. We also assume that the tasks are non-preemptive and that task migration is not possible. Therefore, each task dispatched to one core will not be interrupted during execution, nor will it be migrated to another core. As a first step, our work does not take into account dynamic features of system behavior such as cache misses, page faults, interconnection congestion, and memory contention. In the future, these dynamic features should be taken into account as they affect the performance and energy consumption of the system. Although this work is based on unrealistic simplifications, it is an important step towards the practical scheduling of moldable tasks.

An example of the scheduling result for the task-graph in Figure 5.1(a) is shown in Figure 5.1(d). In the figure, it is assumed that the hardware consists of two little cores and two big cores. Task 2 is parallelized on all four cores, while Task 3 is executed on a single big core. This work assumes that a deadline constraint is given to the entire task graph. However, it should be noted that this work can be easily extended in such a way that individual tasks in a task graph have their deadline constraints.

5.3.2 ILP Formulation

For an architectural platform containing a given set of tasks and a number of cores per type and type of core, the energy-aware scheduling problem for moldable fork-join tasks can be addressed by determining the number of cores for each task and allocating sub-tasks to multiple heterogeneous cores. In this subsection, we begin our ILP formulation in order to explore the problem of energy-aware scheduling of moldable fork-join tasks on heterogeneous multicore. Let $par_{i,k}$ denote a binary decision variable to determine the number of cores to assign task i. If $par_{i,k}$ becomes 1, task i is assigned to k cores.

$$\forall i, \qquad \sum_{k} par_{i,k} = 1 \tag{5.1}$$

Each sub-task is assigned to a core, and this work assumes that neither task nor thread can migrate to a different core. Thus, the sub-task can uniquely decide to be allocated on which type of the cores. Let $type_{i,j,l}$ denote type l of cores for $j-th$ sub-task of task i. Note that we assume the cores consist of two types (i.e., big and little), but the following formula can be employed to use no matter how many types of cores.

$$\forall i,j \qquad \sum_{l} type_{i,j,l} = 1 \tag{5.2}$$

If the number of cores and the type of cores to assign a task are decided, the execution time for each sub-task can be introduced. It should be noted that the execution time for the tasks is well-profiled beforehand and is not assumed to be jitter. As addressed in the previous section, we are given the execution time depends on the number of cores and the type of cores. The execution time of $j-th$ sub-task in task i on core of type l is profiled in advance and given as $Time_{i,k,l}$. $time_{i,j}$ is a decision variable for the execution time of $j-th$ sub-task in task i.

$$\forall i,j \qquad time_{i,j} = \sum_{k}\sum_{l} time_{i,k,l} \times par_{i,k} \times type_{i,j,l} \tag{5.3}$$

However, the formula includes multiplications by two decision variables. Let $mode_{i,k,j,l}$

denote the execution mode for each sub-task. $mode_{i,k,j,l}$ becomes 1 if $par_{i,k}$ and $type_{i,j,l}$ respectively become 1. The expression is shown as follows:

$$\forall i,j \qquad mode_{i,k,j,l} = \begin{cases} 0\ if\ par_{i,k} +\ type_{i,j,l} > 1 \\ 1\ otherwise \end{cases} \tag{5.4}$$

Still, the above formula is not linearized. The formula can be easily linearized by replacing with the following inequalities.

$$\forall i,k,j,l \qquad U \cdot mode_{i,k,j,l} - (par_{i,k} + type_{i,j,l} - 1) \geq 0 \tag{5.5}$$

$$\forall i,k,j,l \qquad U(1 - mode_{i,k,j,l}) + (par_{i,k} + type_{i,j,l} - 1) \geq 0 \tag{5.6}$$

Note that U is a large constant number. With the determined execution mode, we can calculate the execution time for each sub-task. Recall that jitters of execution time of the tasks is out of scope in this work. We also assume that the execution times of the sub-tasks for a task are the same. Recall that we are given the execution time dependent on the number of cores and the type of cores. Now, $time_{i,j}$, which is a decision variable for the execution time of $j-th$ sub-task in task i, can be expressed instead of the formula (5.3):

$$\forall i,j \qquad time_{i,j} = \sum_{k}\sum_{l} Time_{i,k,l} \times mode_{i,k,j,l} \tag{5.7}$$

Let $finish_{i,j}$ denote the finish time of $j-th$ sub-task of task i. Similarly, $start_{i,j}$ is the start time of $j-th$ sub-task of task i. The finish time of sub-task is represented as follows:

$$\forall i,j \qquad finish_{i,j} = start_{i,j} + time_{i,j} \tag{5.8}$$

The sub-tasks are assumed to be run on multicore, independently. Thus, the finish time of task i is defined as the time when all the sub-tasks of task i are finished.

$$\forall i,j \qquad finish\_max_i \geq finish_{i,j} \tag{5.9}$$

Similarly, $start\_min_i$ represents the start time of task i. The start time of task i is defined as the time when a first sub-task of task i is started on a core. Let $start_{i,j}$ denote the start time of $j-th$ sub-task in task i, then they are given by:

$$\forall i,j \qquad start\_min_i \leq start_{i,j} \tag{5.10}$$

Next, let $map_{i,j,k,l}$ denote a binary decision variable for mapping of $j-th$ thread of task i on $k-th$ cores of type l. $map_{i,j,k,l}$ becomes 1 if $j-th$ sub-task of task i is run on $k-th$ core of l type:

$$\forall i,j,l \qquad \sum_{k} map_{i,j,k,l} - type_{i,j,l} = 0 \tag{5.11}$$

To avoid overlapping, multiple sub-tasks cannot be assigned to the same core at the same time. They cannot be assigned to the same core at the same time. If the $j1-th$ sub-task of task i1 is mapped to $j2-th$ sub-task of task $i2$, or two different sub-tasks of a task are mapped to run, they will be mapped to different cores. If $j1-th$ sub-task of task $i1$ and $j2-th$ sub-task of task $i2$, or two different sub-tasks of a task, are mapped to be executed, they will be mapped to different cores. If two sub-tasks of a task are mapped to run, they are mapped to different cores because sub-tasks are not allowed to overlap on the same core at the same time. Otherwise, after a sub-task has finished on a core, it will be assigned to that core. It will be assigned to the core. This means that it must be satisfied that the $j1-th$ sub-task of task $i1$ and $j2-th$ sub-task of task i2 exist simultaneously. It must be satisfied that $j1-th$ sub-task of task $i1$ and $j2-th$ sub-task of task $i2$ are not mapped to the same core at the same time, or that they are executed after some sub-task has finished, which is as shown in the following logical expression:

$$\begin{gathered}\forall i1,i2,j1,j2,k,l,\,(i1 \neq i2 \vee j1 \neq j2),\\ \neg(map_{i1,j1,k,l} \vee map_{i2,j2,k,l})\\ \vee finish_{i1,j1} \leq start_{i2,j2}\\ \vee finish_{i2,j2} \leq start_{i1,j1}\end{gathered} \tag{5.12}$$

Formula (5.12) is not expressed as linear algebra. For the linearization, this expression can be transformed by De Morgan' s low as follows:

$$\begin{gathered}\forall i1,i2,j1,j2,k,l,\,(i1 \neq i2 \vee j1 \neq j2),\\ \neg map_{i1,j1,k,l}\\ \vee \neg map_{i2,j2,k,l}\\ \vee finish_{i1,j1} \leq start_{i2,j2}\\ \vee finish_{i2,j2} \leq start_{i1,j1}\end{gathered} \tag{5.13}$$

The formula (5.13) is still expressed as a logical constraint, and we can linearize it with the four formulas (5.14)-(5.16). Here, x,y are 0-1 auxiliary decision variables to describe the following disjunctive constraint and U is a constant large number as well as in the formula (5.15):

$$\begin{aligned} &\forall i1,i2,j1,j2,(i1 \neq i2 \vee j1 \neq j2), \\ &finish_{i1,j1} \leq start_{i2,j2} + U(1 - x_{i1,i2,j1,j2}) \end{aligned} \tag{5.14}$$

$$\begin{aligned} &\forall i1,i2,j1,j2,(i1 \neq i2 \vee j1 \neq j2), \\ &finish_{i2,j2} \leq start_{i1,j1} + U(1 - y_{i1,i2,j1,j2}) \end{aligned} \tag{5.15}$$

$$\begin{aligned} &\forall i1,i2,j1,j2,(i1 \neq i2 \vee j1 \neq j2), \\ &(1 - map_{i1,j1,k,l}) + (1 - map_{i2,j2,k,l}) + x_{i1,i2,j1,j2} + y_{i1,i2,j1,j2} > 0 \end{aligned} \tag{5.16}$$

In this problem, we assume that each task has precedence constraint. If task $i2$ must be started after task $i1$ ends, which is found in a task graph, $Pred_{i1,i2}$ is set to 1. The start time and the finish time of a task has been already defined at the formulas (5.8) and (5.9).

$$\forall i,j \qquad Pred_{i1,i2} = \begin{cases} 1\ if\ finish\_max_{i1} \leq\ start\_min_{i2} \\ 0\ otherwise \end{cases} \tag{5.17}$$

Similar to the formula (5.3), the formula (5.17) is also replaced to be linearized as follows:

$$\forall i1,i2, \qquad U \cdot Pred_{i1,i2} - (start\_min_{i2} - finish\_max_{i1}) \geq 0 \tag{5.18}$$

$$U(1 - Pred_{i1,i2} + (start\_min_{i2} - finish\_max_{i1}) \geq 0 \tag{5.19}$$

Another important constraint is deadlines. All tasks must be completed by a deadline. In this work we have constrained the deadline to the entire task since we need to meet the overall completion time of the scheduling. The deadline constraint is given as follows:

$$Deadline \geq max_i\{finish\_max_i\} \tag{5.20}$$

In this scheduling problem, the dynamic energy is consumed during the time dedicated to the execution of every sub-task for each type of cores. Note that α_l is dynamic power consumption on l type of cores. Then, the energy consumed on a l type core is expressed as follows:

$$dynamic\_energy = \sum_i \sum_k \sum_j \sum_l \alpha_l \times Time_{i,k,l} \times mode_{i,k,j,l} \tag{5.21}$$

On the other hand, the static energy consumption is attributed to the makespan. The makespan is obtained by the following formula (5.22). Each type of cores is assumed to have own static power consumption β_l, therefore, the total static energy consumption is calculated for each of the cores k for each type l in the formula (5.23).

$$\forall i \qquad makespan \geq finish\_max_i \tag{5.22}$$

$$static\_energy = \sum_k \sum_l \beta_l \times makespan \tag{5.23}$$

Therefore, this work aims to minimize the total energy consumption with consideration both dynamic and the static energy consumption.

5.4 Simultaneous Scheduling and Core-type Optimization

The scheduling problem discussed in the previous section assumes that the architecture is fixed before scheduling. However, in the design of embedded systems, there are many cases where the hardware architecture has to be customized not only for high performance but also for energy efficiency for application programs to run more effectively. Therefore, in this section, we introduce a hardware/software code design methodology for heterogeneous multi-core systems. In this work, the type of cores deployed in the architecture and the scheduling of the moldable tasks are simultaneously optimized to minimize the total energy consumption under deadline constraints. In this study, the total number of cores in the system of interest is fixed in advance, but the type of cores can be changed flexibly. Given a set of tasks, a total number of cores, and a deadline, the framework simultaneously optimizes the type of cores and schedules the moldable tasks such that the total energy consumption is minimized.

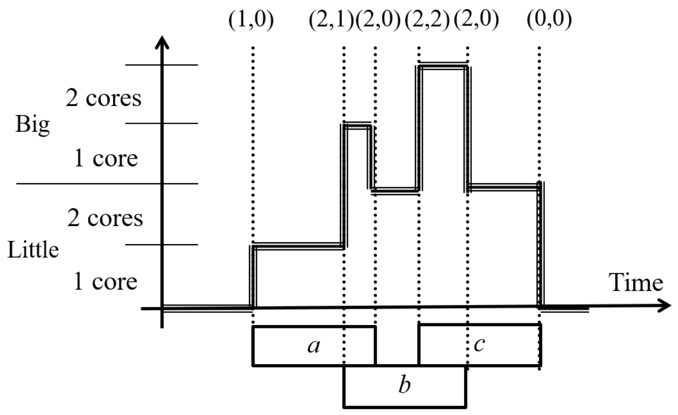


Figure 5.2: Cores usage during the execution of task a, b, and c

$$\text{Minimize :} \qquad dynamic\_energy + static\_energy \tag{5.24}$$

5.4.1 An Integrated Framework

Simultaneous core-type optimization and scheduling can be developed by slightly extending the formulation presented in Section 3. It should recall that the number of cores for each type such as little cores and big cores are limited. Let denote t, which is the time during scheduling $0 \leq t \leq Deadline$, we define a set of tasks $S(t)$ that the tasks $or sub - tasks$ run on the cores at time t. Figure 5.2 shows the concept of $S(t)$. This example is given two cores for little cores and big cores, respectively.

First, task a starts running on a little core. At the time, task a is included in $S(t)$ $i.e., a \in$ S(t). Then, task b starts on a little core and a big core during the execution of task a $i.e., \{a, b\} \in$ S(t). At the time, three cores are occupied in total. When task a is finished, the task is removed from $S(t)$. When task c is arrived, four cores are used and $S(t)$ includes the tasks $\{b, c\}$, which are during running. The parenthesis upside of Figure 5.2 shows the number of cores for each type during running. At the time starting task a, one little core and no big core are employed for the execution. In the example, the number of each type of cores to execute tasks cannot be exceed the number of cores in the target system. Given the total number of l-type of cores in the target system as $Ncores_l$, the resource constraint is expressed as follows:

$$\sum_{i \in S(t)} \sum_{j} \sum_{k} map_{i,j,k,l} \leq Ncores_l \tag{5.25}$$

The formula (5.25) can handle both big cores and little cores. In addition, it can easily extend to more types of cores than three by increasing the core types l. Now, the total number of cores for the target system is given as $Ncore$. The sum of the number of cores for each type is the same as the total number of cores in the target system. The number of cores for each type is constrained by the formula as follows:

$$\sum_{l} Ncores_l = Ncore \tag{5.26}$$

By the equations presented in Section 5.3, we formulate the problem of scheduling and core-type optimization. In this paper, we assume a heterogeneous multi-core architecture with two types of cores: big cores and little cores. However, this formulation can be comprehensively extended to architectures with more than two types of cores.

5.4.2 Two-Phase Approach based on Warm Start Technique

This section demonstrates a warm start approach to the scheduling problem presented in the previous section. In general, task scheduling problems are classified as NP-hard problems. Our presented works in Section 5.3 and 5.4 are more complex than general scheduling problems since both the number of cores assigned to each task and the core-type on the system are decided throughout scheduling. An optimal schedule can hardly be found in practical time, and even a feasible schedule is rather difficult to be found. To address this problem, we propose a two-step warm-started scheduling. This approach takes advantage of the fact that our proposed technique can determine the type of cores to be either a big core or a little core. Initially, the tasks are scheduled on only big cores [32] and obtain a solution. Here, the solution consists of the degree of parallelism of each task and the start time of each sub-task. The solution in the initial scheduling is utilized as the initial solution for the second step. In the second step, our technique that is presented in Section 5.3 starts to find a good schedule. Note that a solution from the first step scheduling must satisfy the constraints since our technique presented in Section 5.3 can flexibly determine the type of each core to be either a big core or a little core. Therefore, this technique can be applied only to the presented technique in Section 5.3.

5.5 Experiments

5.5.1 Setup

To evaluate this work, we have conducted a set of experiments. Twelve random task-graphs generated by TGFF [56] are used as benchmark task-graphs.

Each task graph consists of between 6 and 30 tasks. In the experiment, the number of tasks was limited to 30, since problems consisting of a larger number of tasks than 30 cannot be solved in a practical time. The execution time of each task is determined randomly. For parallelization, we assume an overhead of 10% by dividing the tasks into sub-tasks. We also set the number of cores of the target system to 4 and 8, and the maximum parallelism of each task was set within the range of the number of cores $i.e., four and eight, respectively$. There is no scheduling algorithm that solves the same problem as in this study. Therefore, we compare the following five methods although they have different hardware architectures.

- *All-Big*: MFJ task scheduling on big-only homogeneous multicores. This scheduling is solved with constraint programming [32], for deadline-constrained energy minimization.
- *All-Little*: MFJ task scheduling on little-only homogeneous multicores. This scheduling is solved in the same way as All-Big above.
- *Big-and-Little*: MFJ task scheduling on heterogeneous multicores presented in Section 3 of this chapter. Half cores are little, and another half are big in [82].
- *Big-and-Little-Customized*: Simultaneous scheduling and core-type customization technique presented in [82].
- *Warm-Start-BLC*: A warm start approach, which is combined with the All-Big technique and the Big-and-Little-Customized technique. First, the All-Big technique is employed to obtain an initial schedule in the first ten hours in CPU time, and the Big-and-Little-Customized technique starts in the remaining forty hours.

All five scheduling methods are run using ILOG CP Optimizer 12.9 on a Ryzen Threadripper 3970X (3.7GHz, 32 cores, 64 threads in total) with 256GB of memory. In general, ILOG CP Optimizer is able to find the exact optimal solution. However, when the number of tasks increases, it is not possible to find the exact optimal solution in a practical time. Therefore, in this experiment, we decided to limit the total CPU execution time of ILOG CP Optimizer to 50 hours and adopt the optimal solution found at that time. In this experiment, we vary the time limit constraint with the following equation.

$$Deadline = XB + (XL - XB) \times D \tag{5.27}$$

In this formula, XL and XB denote the shortest schedule lengths on little-only multicores and big-only multicores, respectively. XL and XB are obtained with ILOG CP Optimizer (up to 10 hours in CPU time). D is a parameter indicating the tightness of deadline. The smaller D is, the tighter the deadline constraint is. In our experiments,

D is set to be 100%, 87.5% 75% and 50%. These parameter settings are based on simplified assumptions as follows; the dynamic power accounts for 70% of the total power on the big cores; the clock frequency $i.e., performance$ of the big cores are 1.5 times higher than that of the little cores; the supply voltage of the big cores are 1.5 times higher than that of the little cores; the dynamic power is proportional to the cube of voltage; the static power is proportional to voltage. Since values of the parameters vary depending on various factors such as the processor architectures, process technology, temperature and so on, more extensive experiments with various parameter values are one of our future works.

$$\alpha_{big} : \beta_{big} = 7 : 3 \tag{5.28}$$

$$\alpha_{big} : \alpha_{little} = 3.375 : 1 \tag{5.29}$$

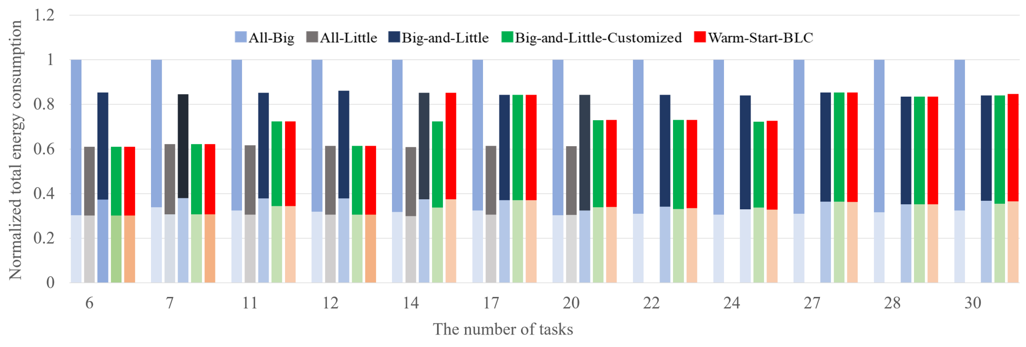
$$\beta_{big} : \beta_{little} = 1.5 : 1 \tag{5.30}$$

5.5.2 Results

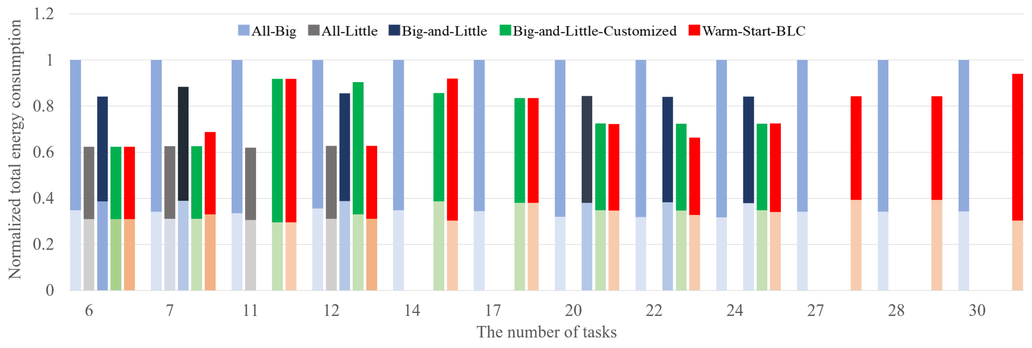
Experimental results are shown in Figures 5.3, 5.4, 5.5, and 5.6. The X-axis of the graph represents the task graph and each label indicates the number of nodes in the task graph. The Y-axis represents the total energy consumption of the scheduling results obtained by the five methods. Each graph also contains the dynamic energy, shown in darker colours on the top, and the static energy, shown in lighter colours on the bottom. The overall energy consumption has been normalized using the All-Big technique. In some cases, no solution can be found, with the exception of the Warm-Start-BLC technology. There are two possible reasons for this. One is that there is no feasible solution to the deadline constraint. Another reason may lay that the CP solver cannot find any feasible solution within the limited time even if feasible solutions do exist.

Figure 5.3 (a) and (b) show the results under the deadline constraint D=100% on four cores and eight cores, respectively. Due to the looseness of the dead-line constraint, the All-Little technique achieves the lowest energy in most cases.

The graphs show that the proposed method achieves the same or smaller values of static and dynamic energies than the Big-and-Little technique. Theoretically, the Big-and-Little-Customized technique should be the best because the solution space of the Big-and-Little method covers the solution space of the other technique. However, due to the limited CPU runtime of the ILOG CP Optimizer, the Big-and-Little-Customized technique may not provide as good a solution as the All-Little technique. For the 14 tasks with 4 and 8 cores, the Warm-Start-BLC technique did not find a better solution than the Big-and-Little-Customized technique, while the Big-and-Little-Customized technique did not find any solution. However, the Big-and-Little-Customized technique could not find a single solution, whereas the Warm Start-BLC technique was able to find a feasible solution in all cases. The scheduling results from the Warm-Start technique depend on how good the initial solution is, and therefore the results depend on the task graph. Since the deadline constraint is loose, the All-Little technique achieves the lowest energy in many cases. The graphs show that our proposed technique obtains the same or smaller values for both static and dynamic energies compared to the Big-and-Little technique. Theoretically, the Big-and-Little-Customized technique should be the best because the solution space of the Big-and-Little technique covers the solution space of the other techniques. However, due to the limited CPU runtime of the ILOG CP Optimizer, the Big-and-Little-Customized technique may not find as good a solution as the All-Little techniques. For the 14 tasks with 4 and 8 cores, the Warm-Start-BLC technique did not find better solutions than the Big-and-Little-Customized technique, but the Big-and-Little-Customized technique does not find any solutions, whereas the Warm Start-BLC technique was able to find a feasible solution in all cases. The scheduling results from the Warm- Start technique depend on how good the initial solution is, and therefore the results depend on the task graph.

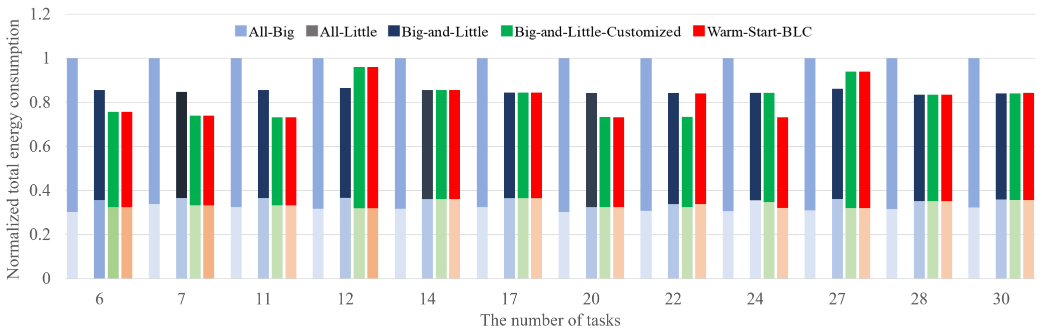


(a) Scheduling results on 4 cores

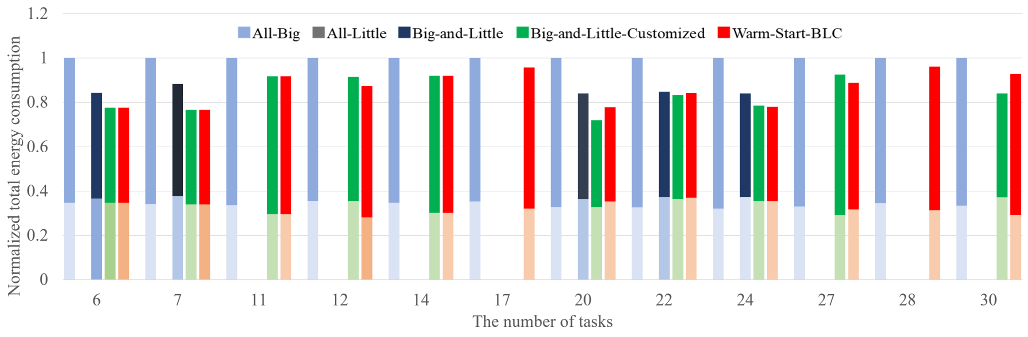


(b) Scheduling results on 4 cores

Figure 5.3: Dynamic and static energy consumption under deadline constraint D=100%



(a) Scheduling results on 4 cores

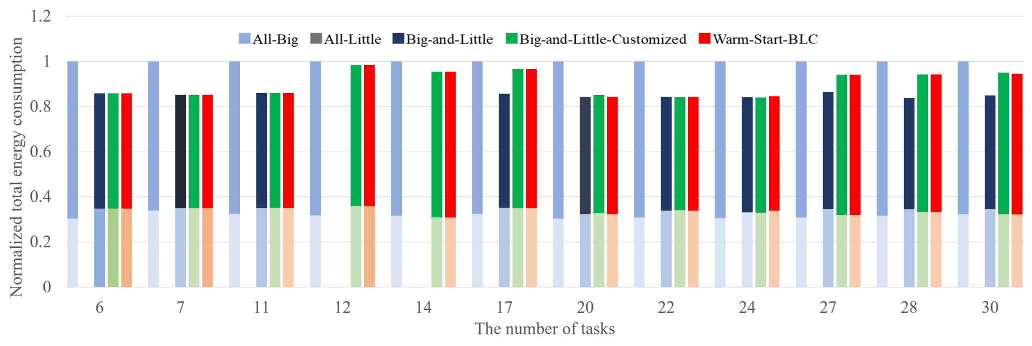


(b) Scheduling results on 4 cores

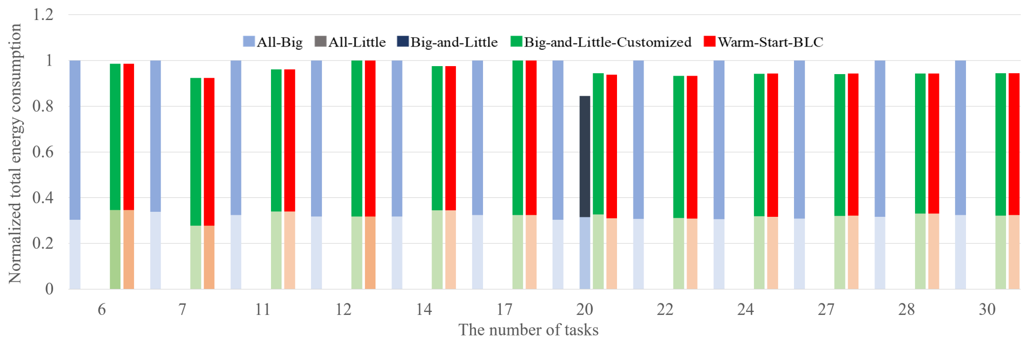
Figure 5.4: Dynamic and static energy consumption under deadline constraint D=87.5%

When the deadline constraint D is 87.5% as shown in Figure 5.4, the All-Little technique cannot find any solution since the technique cannot meet the deadline anymore. On four cores, the Big-and-Little technique consumes lower energy than the All-Big technique by up to 15.2%. Compared to the All-Big technique, the Warm-Start-BLC technique finds a schedule with lower energy by up to 19.5%. From the perspective of static energy, the Warm-Start-BLC finds longer schedules than the Big-and-Little-Customized so that the static energy becomes larger. Most of the results show that the total energy consumption largely depends on dynamic energy consumption. However, in terms of the impact of static energy, the static energy of the Big-and-Little-Customized technique for 12 tasks in Figure 5.4(b) surpass that of the Warm-Start-BLC technique, resulting in that the Warm-Start-BLC technique obtaining the better schedule.

When the deadline constraint is 75% and 50% as shown in Figures 5.5 and 5.6, the Big-and-Little technique fails to find any solution in most cases since deadlines would be missed. Still, the Big-and-Little-Customized technique finds good solutions in many cases, however, there are cases with no solution due to the huge solution space. On the other hand, the Warm-Start-BLC technique is successfully obtaining good solutions for all the cases, and the total energy required in the target system is almost the same as that by the Big-and-Little Customized. As shown in Figures 5.3, 5.4, 5.5, and 5.6, the static energy consumption accounts for a portion by 35.3% of the total energy consumption



(a) Scheduling results on 4 cores



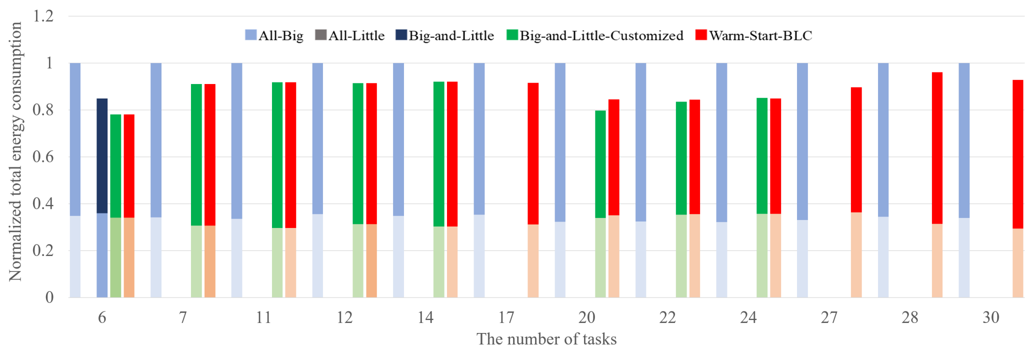
(b) Scheduling results on 4 cores

Figure 5.5: Dynamic and static energy consumption under deadline constraint D=75%

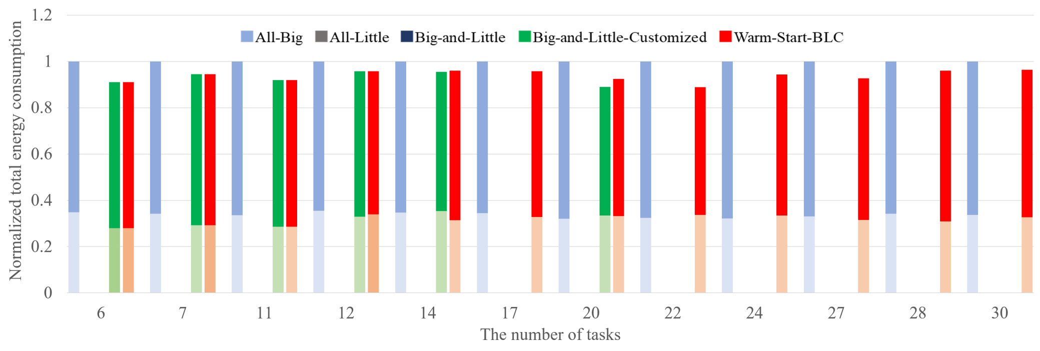
on average, and the static energy has less impact than the dynamic energy. In the experiments, we do not consider overheads such as memory accesses and communications. The scheduled length of the real-world applications should be longer than that of the experiments, and the static energy is expected to be increased.

5.6 Conclusions

This work proposes a scheduling technique for energy-aware typable fork-join tasks on heterogeneous multicores. We also propose a technique for multicore customization and task scheduling simultaneously. Furthermore, we propose a warm-start technique that simultaneously performs scheduling and core-type optimization to efficiently find a schedule that can be executed in a practical time. Through experiments, we have demonstrated the effectiveness of our proposed techniques. In the future, we plan to develop a fast heuristic algorithm for the scheduling problem and consider more assumptions such as jitter in the execution time of each task, different overheads due to parallelization of tasks, and overheads due to communication. In addition, we plan to try the DVFS approach to evaluate whether energy consumption can be further reduced and to develop heuristic algorithms to solve the scheduling problem in a practical architecture-aware



(a) Scheduling results on 4 cores



(b) Scheduling results on 4 cores

Figure 5.6: Dynamic and static energy consumption under deadline constraint D=50%

manner.

Chapter 6

Function-Level Module Sharing in High-Level Synthesis

This chapter addresses module sharing techniques in high-level synthesis[1].

6.1 Introduction

High-level synthesis (HLS) is one of the techniques which automatically translates software programs that are typically described in high-level codes such as C/C++ into register transfer level (RTL) specifications in hardware description languages [86, 87]. Recently, HLS has been becoming increasingly appealing due to its potential to improve productivity not only for hardware designers but software designers. In HLS, it is often necessary to satisfy various constraints such as resource constraints or time constraints due to area limitation or responsivity of the circuit by minimizing the area or delay of the generated circuit, respectively. However, the quality of HLS-generated circuits that meet such constraints is still lower than human-designed circuits in many cases. Our preliminary experiments with a state-of-the-art HLS tool and standard benchmark programs showed that the HLS tool can share functional units (such as adders and multipliers). Vivado HLS, which can effectively share function units, is one of the most popular HLS tools around the world. Vivado HLS is capable of exploiting the sharing of coarse-grained modules, however, can hardly take advantage of sharing coarse-grained modules at the function level in practical use. In such conventional HLS tools, multiple instances of the same module synthesized from a function are generated even if the function is exclusively called from different functions. To avoid the issue, a common technique that realizes function-level module sharing is function inlining. Function inlining may

[1]This chapter is a refined and reproduced version of the paper originally published in ETRI Journal [85] copyrighted by ETRI.

enable improvement latency by exploiting parallelism at the instruction level and optimizing code transformation by code elimination. Moreover, unlike software, hardware resources are capable of being shared at the function level by removing boundaries among functions [88, 89].

However, in general, excessive function inlining often leads to degradation of performance, as it creates a large number of huge modules of control state and significantly increases the number of operations in a function. The more complex the controller, the longer the wiring, and the larger the load capacity. Therefore, function inlining can degrade the overall performance and also violate timing constraints. In this paper, we present two HLS techniques for module sharing at the function level. These techniques create a single module instance from a function, even if the function is invoked from a different function. The remainder of this paper is organized as follows. In Section 2 we describe related work. In Section 3, we introduce the concept of module sharing at the function level and explain how to share a module without creating multiple instances of the same module. Section 4 presents the experimental results, and Section 5 concludes the paper.

6.2 Related Work

Recent previous works in the domain of HLS have been investigated. The work in [90] addresses pipeline synthesis techniques for field-programmable gate array (FPGA) and introduces two algorithms to automatically explore the large design spaces and parallelizes C-based description to maximize throughput or minimize area. Zhao et al. introduced mixed-integer linear programming (MILP) formulation for mapping-aware pipeline module scheduling [91]. Alle et al. presented source-to-source transformation using dynamic scheduling strategies, yet this work needs additional logic for a complex decision that incurs the overhead of resources [92]. Hara et al. studied function clustering and proposed integer linear programming (ILP) formulation to optimize for minimizing clustering size underperformance and area constraints, and they extended the work to introduce clustering techniques for similar functions merge [93, 94].

In order to reduce the area in a circuit at the register transfer level, a well-known methodology especially in the binding for HLS processes is called resource sharing. Recently, there have been several types of research on resource sharing. In resource sharing, a single functional unit (FU) is shared among different operations by adding multiplexers (MUXes) into the shared functional unit [95]. The finite state machine (FSM) controls the MUXes for steering the data to the adder that depends on the state. Cardoso in [96] presented an algorithm including temporal partitioning, resource sharing, scheduling, and allocation and binding to efficiently use resources of FUs. Cong and Jiang developed a framework for pattern-based behavior synthesis [97]. They proposed graph-based techniques to efficiently extract patterns of operators in the HLS of FPGA circuits, and share blocks of functional units according to the patterns in binding. However, the approach works only at the functional unit level within a module. Intra-module sharing of functional units cannot be allowed. To overcome the limitations, the work in [98] employed function proxies that enable sharing blocks of common operations across the boundaries of modules, but the approach needs the support for the synthesis of function pointer. The authors refer that their proposed techniques are necessary to be combined with other optimization strategies such as inlining. The authors in [99] presented a module sharing technique but it can hardly be used if functions are invoked in conditional statements.

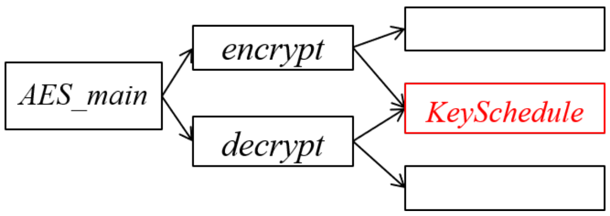
6.3 Function-Level Module Sharing Techniques

6.3.1 Motivation

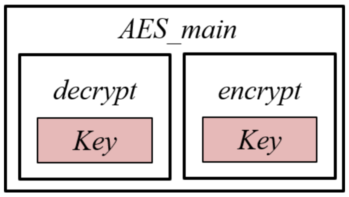
In this section, we describe our motivation and proposed function-level module sharing techniques. Let us consider a C program whose function call graph is shown in Figure 6.1(a). Function AES_main calls two functions encrypt and decrypt, each of which calls function Key. From this program, a state-of-the-art HLS tool such as Xilinx Vivado HLS generates a circuit as shown in Figure 6.1(b).

The top module AES_main contains an encrypt module and a decrypting module, each of which contains a Key module. Thus, the generated circuit contains in total two instances of the Key module. Due to the sequential nature of the C program, however, the two Key modules are not activated simultaneously. In this case, generating multiple instances of the identification module is a waste of silicon area. Our function-level module sharing techniques proposed in this paper create a single instance of a module from a function even if the function is called from different functions. One of the ways to module sharing is function inlining. By function inlining, the functions between the AES_main and the Key function; encrypt and decrypt are inlined to the AES_main function, respectively. The AES_main function becomes the only caller of Key so that a single instance of the Key module is created. In this way, function inlining enables function-level module sharing. However, excessive function inlining often leads to performance degradation. Function inlining may create huge modules with a large number of control states, resulting in timing violation [94].

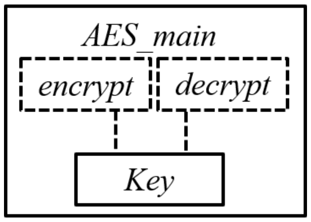
In this paper, two presented techniques focus on module sharing at the function level. Our techniques both are based on source-to-source transformation at the C-code level, and it is not necessary to modify an RTL circuit.



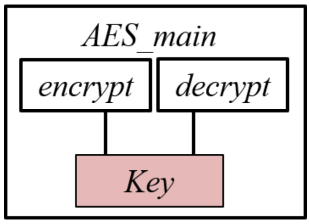
(a) Function call graph



(b) HLS-generated circuit



(c) Circuit with function inlining; encrypt and decrypt are inlined into AES_main



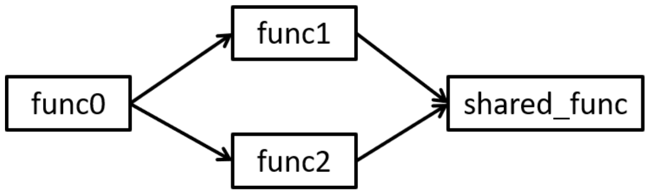
(d) Circuit with module sharing

Figure 6.1: An example of the concept for module sharing

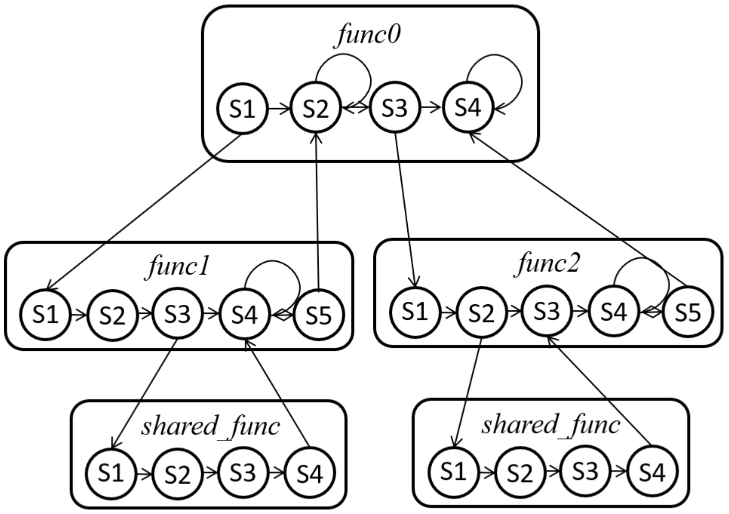
6.3.2 Module Sharing Technique by Selective Function Inlining

To enable sharing of functions in C-code, we first propose a module sharing technique by function inlining. In general, function inlining itself may lead the performance degradation due to the aforementioned characteristics. In this paper, we introduce a technique for module sharing by inlining functions except a shared function. Figure 6.2(a) represents a call graph which has a same call hierarchy. func0 calls both func1 and func2, and each of func1 and func2 calls the $shared\_func$, respectively.

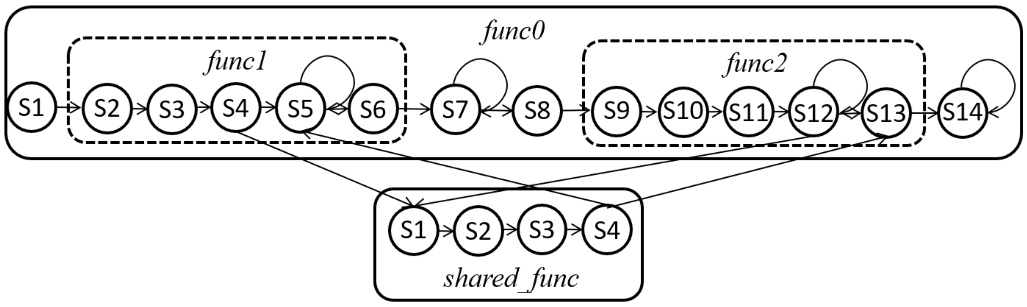
Figure 6.2(b) shows the FSM of a circuit without module sharing. In the figure, the func0 function calls two functions represented as func1 and func2 which call $shared\_func$, respectively. Note that the circuit is generated by the state-of-the-art HLS tool in such a way that two instances of the $shared\_func$ module are created. In order



(a) Call graph with a same call hierarchy



(b) FSM without module sharing



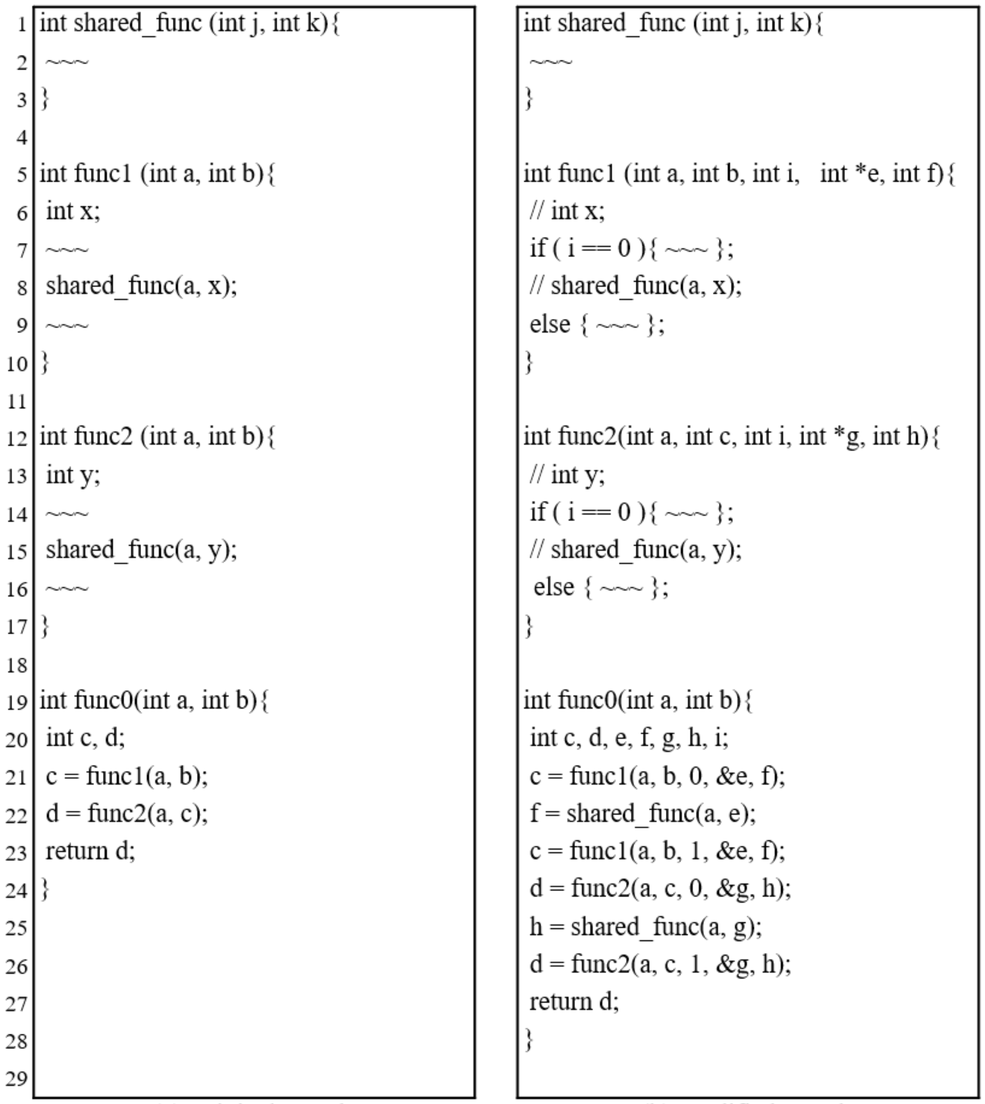
(c) FSM with module sharing by function inlining

Figure 6.2: State Transition Diagram

to reduce the area of the circuit, the $shared\_func$ function is necessary to be shared since the two $shared\_func$ are called in a mutually exclusive way. In our proposal, a C-code for this circuit is modified so that an HLS tool generates a circuit whose FSM is as shown in Figure 6.2(c). In this figure, each of func1 and func2 is inlined into func0, and the func0 calls the $shared\_func$. The caller of the $shared\_func$ is only the func0, and the HLS tool generates an instance of the $shared\_func$ module.

6.3.3 Module Sharing Technique by Multiple Function Calls

Another way for function-level module sharing is to move the shared function into the upper level. This transformation is performed at the C source level. Figure 6.3(a) shows an example of C code. The func0 function calls two functions func1 and func2, each of which calls $shared\_func$. As aforementioned, the state-of-the-art HLS tool generates a circuit whose FSM is similar to the figure as shown in Figure 6.2(a). Our technique transforms the code in Figure 6.3(a) into the one in Figure 6.3(b). The func1 is partitioned into two parts at the point of calling the $shared\_func$, and an if-then-else statement is inserted. The first part of the func1 is moved to the then block, and the second part is moved to the else block.



```
int shared_func (int j, int k){
 ~~~
}

int func1 (int a, int b){
 int x;
 ~~~
 shared_func(a, x);
 ~~~
}

int func2 (int a, int b){
 int y;
 ~~~
 shared_func(a, y);
 ~~~
}

int func0(int a, int b){
 int c, d;
 c = func1(a, b);
 d = func2(a, c);
 return d;
}
```

(a) Original C code

```
int shared_func (int j, int k){
 ~~~
}

int func1 (int a, int b, int i,   int *e, int f){
 // int x;
 if ( i == 0 ){ ~~~ };
 // shared_func(a, x);
 else { ~~~ };
}

int func2(int a, int c, int i, int *g, int h){
 // int y;
 if ( i == 0 ){ ~~~ };
 // shared_func(a, y);
  else { ~~~ };
}

int func0(int a, int b){
 int c, d, e, f, g, h, i;
 c = func1(a, b, 0, &e, f);
 f = shared_func(a, e);
 c = func1(a, b, 1, &e, f);
 d = func2(a, c, 0, &g, h);
 h = shared_func(a, g);
 d = func2(a, c, 1, &g, h);
 return d;
}
```

(b) Modified C code

Figure 6.3: An example of function module sharing

A new argument $i.e.,$ int i is added to func1, indicating which part of func1 should be executed. If i is set to 0, the then block of func1 is executed. If it is set to 1, the else block is executed. The call to the $shared\_func$ is moved from the func1 to the func0. In the func0, func1 is called first with *i*=0, the $shared\_func$ is called next, then, func2 is called again with i=1. The same transformation is applied to func2 as well. In the modified code in Figure 6.3(b), the $shared\_func$ is called twice but from the same function of the func0. Therefore, only a single instance of the $shared\_func$ module is created by the HLS tool. Figure 6.4 shows the FSM of the circuit where the module sharing technique is applied. In this case, the number of function calls is increased, however, each function keeps its size without growing the complexity of control states and the operations.

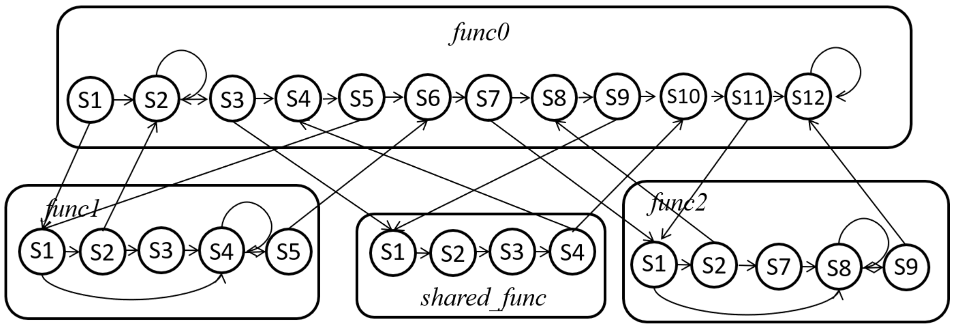


Figure 6.4: FSM of the circuit with module sharing

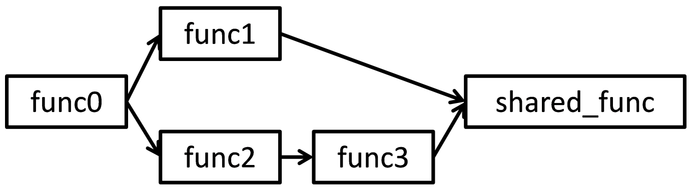


Figure 6.5: Call graph with a different call hierarchy

This technique is sometimes unable to be employed in such a case as shown in Figure 6.5. Figure 6.5 represents a call graph in the case in which the $shared\_func$ cannot be shared. The figure shows that the func0 calls the func1 and the func2, and the func2 calls the func3. Each of the func1 and the func3 calls the $shared\_func$. To share the $shared\_func$ module, the level of the call hierarchy is necessarily the same as the func0. In other words, the number of functions in the higher place of call hierarchy from the mutual caller function needs the same to share the shared function. The issue is solvable if the func3 is inlined to the func2. If either the func3 is inlined to the func2 or the func2 is inlined to the main function, the hierarchy of the shared function from the mutual caller function becomes the same.

6.4 Experiments

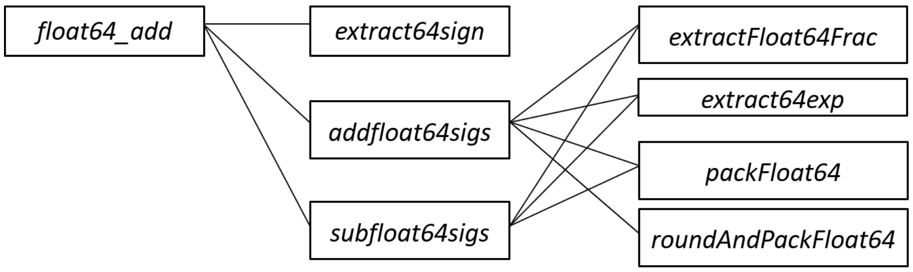
We have conducted experiments to compare our module sharing technique with the four techniques as follows.

- *Default*: All functions are synthesized based on the default configuration of Vivado HLS.
- *Inline-off*: HLS-generated circuits without module sharing, and it generates multiple instances for the same module without inlining.
- *Inline-all*: Inlining all the functions into a main function.
- *Share-inlining*: Module sharing technique by function inlining
- *Share-calls*: Module sharing technique by multiple function calls

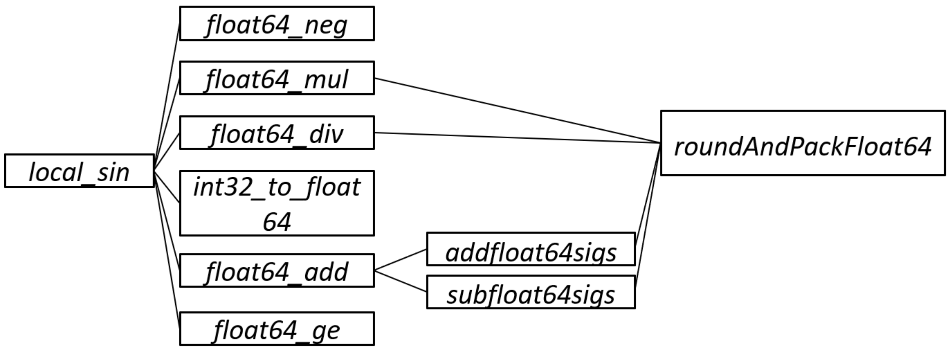
We used Vivado HLS 2019.1 as an HLS tool and Xilinx Zynq-7000 as a target board. Clock frequency was set to 100MHz; thus, timing constraint was within 10ns. We characterize area requirements by reporting the number of look-up tables (LUTs), flip-flops (FFs), digital signal processing (DSPs), block RAM (BRAMs), execution clock cycles, and critical path delay (CP delay). In the experiments, three of four benchmarks programs derived from CHStone benchmark suite [88] were employed. CHStone benchmark suite is easy to use since the programs of CHStone are written in the standard C language and any of the extensions is unnecessary. In this paper, we selected aes, $dfadd$, and $dfsin$.

Figures 6.6(a) and 6.6(b) represent call graphs of $dfadd$ and $dfsin$, respectively. For aes, the call graph has been mentioned in Figure 6.3. The rest of the benchmarks called $dfaddsub$, is introduced by the work in [93]. This is based on Soft Float library [100] for software implementation of binary representation. A call graph of $dfaddsub$ is represented in Figure 6.6(c). $dfaddsub$ has three arguments; a, b, and 1-bit id. id is used for if-then-else statement to branch off $float64add$ addition function and $float64sub$ subtraction function from the caller function called as $float64addsub$. Both $float64add$ and $floaf64sub$ call three functions; extract64sign, addfloat6sigs, and subfloat64sigs, respectively. There also exist other functions, but their details are omitted since they are very small. Note that, the functions to be shared are determined following an objective such as area minimization and performance maximization by designers in advance.

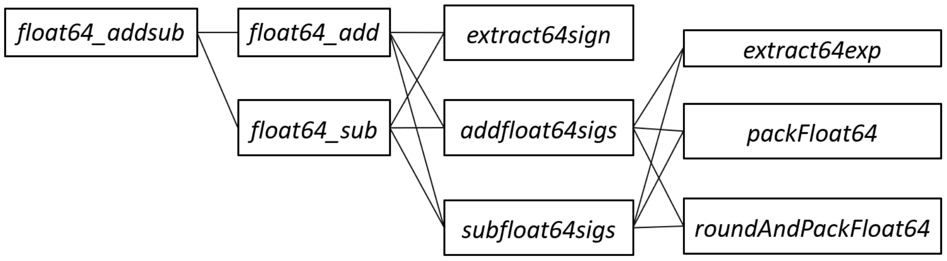
Table 1 shows the synthesis and simulation results in terms of the number of LUTs, FFs, execution cycles, DSPs, and BRAMs, where the numbers in parentheses denote the normalized values to Default. For the aes, $Share\text{-}inlining$ can reduce the number of



(a) $dfadd$



(b) $dfsin$



(c) $dfaddsub$

Figure 6.6: Call graphs of the benchmark programs

LUTs by 6.4%. The technique effectively reduces the area of the circuit by generating a single instance of a shared module.

On the other hand, $Share$-$calls$ increases LUTs by 63.7% due to the additional circuits for if-then-else statement in each module, and the overhead of multiple functions calls in this technique to invoke the shared function grows the number of clock cycles. $Share$-$inlining$ obtains the smallest number of clock cycles due to the elimination of the overhead for function calls. In $dfadd$, both $Share$-$inlining$ and $Share$-$calls$ have achieved reducing the circuit area compared with $Default$ due to a large function which is shared. $Share$-$calls$ reduces 4.6% and $Share$-$inlining$ also reduces by 14.9% of the number of LUTs. If all the functions are inlined into the main function, the result shows 9.1% reduction of LUTs against $Default$. According to the result, $Default$ may generate multiple instances of modules with several LUTs. Our pro-

Table 6.1: Synthesis and simulation results

	Default	**Inline-off**	**Inline-all**	**Share-inlining**	**Share-calls**
aes					
LUTs	1460 (1)	2769 (1.897)	1825 (1.250)	1366 (0.936)	2390 (1.637)
FFs	1764 (1)	2721 (1.542)	1478 (0.838)	1639 (0.929)	2313 (1.311)
DSPs	2 (1)	2 (1)	0 (-)	2 (1)	2 (1)
BRAM	10 (1)	10 (1)	9 (0.9)	10 (1)	10 (1)
Clock cycles	3071 (1)	3193 (1.040)	2996 (0.976)	3062 (0.997)	3073 (1.001)
CP delay (ns)	6.192	7.127	6.342	6.666	6.652
dfadd					
LUTs	4650 (1)	4497 (0.967)	4228 (0.909)	3960 (0.851)	4437 (0.954)
FFs	2425 (1)	2439 (1.006)	1857 (0.766)	1962 (0.809)	2615 (1.078)
DSPs	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
BRAM	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
Clock cycles	5 (1)	6 (1.2)	4 (0.8)	5 (1)	7 (1.4)
CP delay (ns)	7.401	7.917	9.765	8.249	8.328
dfsin					
LUTs	10390 (1)	10510 (1.012)	9211 (0.887)	8896 (0.856)	9858 (0.949)
FFs	6815 (1)	7837 (1.150)	6033 (0.885)	6475 (0.950)	7946 (1.166)
DSPs	43 (1)	43 (1)	59 (1.372)	59 (1.372)	43 (1)
BRAM	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
Clock cycles	1308 (1)	1308 (1)	1309 (1.001)	1304 (0.997)	1340 (1.024)
CP delay (ns)	9.975	9.975	10.328	9.975	9.975
dfaddsub					
LUTs	4664 (1)	9015 (1.933)	4685 (1.005)	4664 (1)	4661 (0.999)
FFs	2426 (1)	5011 (2.066)	2426 (1)	2426 (1)	2429 (1.001)
DSPs	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
BRAM	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
Clock cycles	5 (1)	7 (1.4)	5 (1)	5 (1)	6 (1.2)
CP delay (ns)	7.401	7.917	7.401	7.401	7.401

posed techniques, however, degrade the performance of the circuits, which is found in the CP delay. The techniques for module sharing increase the control states and the number of operations, and these overheads incur a longer critical path delay than the others. $dfsin$ is a sine function for double floating-point numbers and is defined as a quite large circuit in the experiments. In this case, $Share$-$calls$ can reduce by up to 5.1% of the number of LUTs without performance degradation. In the result obtained by $Share$-$inlining$, LUTs and FFs are reduced by up to 14.4% and 15.0%, respectively. Due to the increased overhead for multiple function calls, $Share$-$calls$ needs the largest number of clock cycles but meets the timing constraint. On the other hand, Inline-all violates the constraint within 10ns. In the results, function inlining with module sharing can effectively reduce the area and satisfy the constraint. For $dfaddsub$, $Share$-$calls$ is successful in area reduction of the circuit. In the case, $Share\text{-}calls$ shares $float64\_add$ addition function and $float64\_sub$ subtraction function and invokes one-by-one by multiple function calls without function inlining. Therefore, a single instance for each of the other callees invoked by $float\_add$ and $float\_sub$ is generated. On the other hand, $Share\text{-}inlining$ inlines the $float64\_add$ function and the $float64\_sub$ function into the caller function. The callees for $float\_64add$ and $float64sub$ are inlined as well so that the created circuit becomes a large number of control states is required. The overhead from the increased number of states by function inlining exceeds that by the function calls. $Inline\text{-}off$ needs more LUTs and FFs than the others since it synthesizes double instances of the callee functions as well as the shared function. Future work includes the evaluations with further larger programs applied to our proposed techniques are necessary.

6.5 Conclusions

In this paper, we have proposed techniques for function-level module sharing in high-level synthesis. Since our techniques are source-level transformation, it can be applied as a front-end of existing HLS tools. Our experimental results show a significant reduction in look-up tables. In future, we plan to improve our techniques in order to automatically determine which technique to employ. Also, we plan to extend our technique towards more general programs. On another direction, our proposed techniques are tool dependent so far, and the techniques can hardly be applied to other HLS tools in the same way. Therefore, we plan to develop an extension of our techniques that are available without stuck to particular HLS tools.

Chapter 7

Conclusions

7.1 Summary

Today, embedded systems are found everywhere in smartphone, digital cameras, many kinds of wearable devices, and other devices/gadget. As the characteristics of such devices are totally different, the requirements for the design also differ, and there have appeared advancement of technologies, especially efficient system-level design technology, that enable designers to automatically make the development and production of the systems in such a way that the requirements are fully satisfied under timing and resource constraints specialized to the embedded systems. To further improve performances with reduction of energy, size, and cost, Systems-On-a-Chip (SoC) have been grown rapidly. Recently, the need for more computing power as well as increasing demand for the higher performance leads to SoC with multiple computation cores, which is called Multi-Processor System-On-Chip (MPSoC).

MPSoC platform has been developed by many industrial companies, however, the time-to-market has often prolonged due to increasing the difficulty caused by complexity of system-level design. Therefore, the technology for efficient design are focused to optimize both perspectives of software and hardware, which has led to co-design technology. One of major challenges faced by designers in software design is the scheduling and mapping of embedded parallel applications on multicore so that they can improve their performance and scale over more and more cores. In general, application performance is hitting the ceiling due to multiple bottlenecks including contention for shared resources such as caches, memory and internal connections. It results in time consuming for developers to identify where the bottleneck degrades the performance from the source code. On the other hand, another of the challenges refers to hardware design. Hardware implementation is very efficient in terms of performance and power consumption compared to software implementation, however, the design of hardware has been regarded as more time consuming process among embedded system processes.

To overcome the issues, this thesis has worked on task scheduling techniques that determine the execution order of tasks in an application and allocation the tasks on multicore. The tasks in the parallel programs of the application can be run at the same time in parallel on multicore. In addition, the tasks in recent applications usually have inherent parallelism in the data parallelism fashion, and such tasks are called data-parallel tasks. Data-parallel tasks are allowed to split into multiple threads (or sub-tasks), and each thread is assigned to one of the cores. Data-parallel tasks can be classified into $rigid$, $moldable$, and $malleable$, and this thesis focuses on $moldable$ tasks, where the number of threads are determined during scheduling and not changed during runtime. Thus, this thesis has focused on scheduling of moldable tasks on multicore, which determine the number of threads for each task and the schedule the tasks on multicore. Behind the literature, this thesis have proposed several scheduling techniques. In addition, this thesis has tackled to hardware design issues in high level synthesis (HLS). HLS techniques, which automatically generate register transfer level (RTL) design from the behavior level languages such as C/C++ description. The common HLS tools are able to generate the circuit that can satisfy the timing and resource constraints, but they are not efficient enough to create the better circuits since the HLS tools generate the multiple instances even if the instances are invoked at the different time. This thesis have proposed modules sharing techniques to generate a better circuit than existing techniques do.

Let us summarize the overview of the chapters. Chapter 2 has described the comprehensive design flow of embedded systems. Chapter 3 has addressed the scheduling techniques that schedule moldable tasks on multicore, which is based on constraint programming (CP) paradigm. The proposed techniques are contributed to more quickly find an optimal scheduling solution than the state-of-the-art techniques. On four-core architecture, the proposed technique for moldable fork-join tasks (MFJ) tasks can find solutions although the state-of-the-art techniques fails to find any solution. In addition, On 8 to 32 cores, the results of the proposed technique can still find a good solution compared to existing techniques. The proposed scheduling technique for moldable synchronous tasks (MS) tasks can achieve the improvement by up to 11% on average compared to the state-of-the-art technique. The state-of-the-art technique fails to find any solution in a practical time, therefore, the proposed techniques are able to quickly find a solution in a practical time.

Chapter 4 has extended the scheduling techniques to take into account communication delays. The experiments have been conducted in terms of performance, and the effect of communication to computation rate (CCR) has been evaluated. The experimental results show our proposed techniques can obtain greater schedules than the state-of-the-art techniques on multicore. Regarding the study for the different CCRs, we show the effectiveness our scheduling technique can shorten schedule lengths in increasing CCRs.

Chapter 5 has addressed the scheduling problems that assume a heterogeneous architecture from the perspective of energy efficiency. The proposed techniques determine the number of threads and schedule for reduction of the overall energy consumption in the system. Furthermore, it optimizes the types of cores as well as the schedule for further reduction of energy consumption. The experimental results show that the proposed scheduling techniques for heterogeneous multicore architecture can reduce the energy consumption compared to the techniques for homogeneous multicore architecture. With regard to the comparison on the heterogeneous multicore architecture, the state-of-the-art technique, where the architecture is fixed and determined in advance, is less of flexibility in terms of mappings, while the proposed technique can flexibly determine the architecture (i.e., the types of cores on the architecture). If the deadline is loose, the proposed technique can remarkably reduce the energy consumption, and the two-phase heuristic technique is able to find a feasible solution every case. As the deadline becomes tight, the proposed technique can find better solutions than the state-of-the-art techniques in many cases, while there are some cases that the state-of-the-art techniques fails to find a good solution due to the huge solution space. It should be noted that the architecture determined through the proposed technique become closer to the homogeneous architectures since the performance becomes crucial for satisfying the deadline. The proposed heuristic approach demonstrates that it can obtain the solutions in all the cases within the runtime even if the deadline becomes tight.

Chapter 6 has mentioned the HLS techniques to deal with the conventional issues lurking in the HLS tools. The experimental results show the module sharing techniques can reduce the number of hardware resources, while the traditional techniques require more LUTs and FFs than the others since it synthesizes double instances of the callee functions as well as the shared function. In contrast, the proposed techniques effectively reduce the area of the circuit by generating a single instance of a shared module. To share a module, however, it is necessary for each module to own a branch statement such as an if-then-else statement so that the created circuit becomes huge and the large number of control states is required. It should suggest that the overhead of area incurred by the circuit for the branch statement can be ignored if the modules are relatively large. Module sharing with function inlining indicates that it can achieve the reduction of area since the circuit for the branch statement is almost unnecessary, while the inlined module becomes large.

7.2 Future Directions

The works presented in this thesis will be able to extend in several directions. Chapter 3 has proposed scheduling techniques for two kinds of moldable tasks. One is called fork-join task and another is called synchronous task, respectively. The techniques are, however, simplified in order to evaluate the performance improvement with exploiting data-parallelism of the tasks. Thus, the work does not consider main memory access and cache misses that are excessively crucial bottlenecks in parallel programs. In addition, this thesis have not mentioned a heuristic technique based on meta-heuristics.

Chapter 4 has extended the work in Chapter 3 with taking into account inter- and intra-task communications. Furthermore, this work introduce the concept of computation and communication ratio, and it suggests that scheduling aware of the size of data is very crucial in communication overheads. This work has assumed that communications are incurred between the cores. For simplicity, this work has ignored the number of resources for communication, such as network interfaces and links, however, it is limited in the real-world. The conflicts among communications often result in prolong the overall execution time. Thus, resource contentions and network typologies will be taken into account in the future.

Chapter 5 has addressed that energy-aware scheduling techniques and optimization of core-types on heterogeneous multicores. The work attempts to minimize the overall energy consumption throughout scheduling the tasks, mapping, and core-type optimization. In core-type optimization, the technique determines weather each core is either high-performance core and power-efficient cor during scheduling. However, this technique has not employed dynamic voltage and frequency scaling (DVFS) and dynamic power management (DPM). As well as the mentions in Chapter 4, memory accesses also largely affect energy consumption such as cache misses and main memory accesses. These metrics should be taken into account in order to make this work practical in reality, and considering the dynamic features is one of future works.

Chapter 6 has proposed function-level module sharing techniques in high-level synthesis (HLS). This work can be applied to a front-end of HLS tools, however, the technique currently is necessary to be done manually. Therefore future work includes automatic determination for which sharing techniques to employ. In addition, this work have been conducted in several programs. In the future, more general programs are employed to evaluate our sharing techniques.

Acknowledgment

I would like to express my greatest gratitude to Professor Hiroyuki Tomiyama in Ritsumeikan University. He gave me a lot of opportunities to dive into the depths in the field of embedded systems. It is my fortune to have studied under a guidance of Professor Tomiyama for five years. I also thank to Ittetsu Taniguchi, Associate Professor of Osaka University, for his expert advice and encouragement throughout the life in Ph.D course. Associate Processor Meng Lin in Ritsumeikan University has induced me to pursue Ph.D degree. Actually, I had never thought to enter Ph.D course before I was assigned to this laboratory and met those professors. Of course, I must thank to Assistant Professor Xiangbo Kong, for having many discussions and invaluable suggestions. It is one of my greatest opportunities to have discussed for five years with Assistant Professor Kong as a student and as an instructor. They are always open whenever I got in trouble or had questions not only about my research or writing as many as my personal trouble. They consistently steered or guided me in the right direction whenever and wherever they thought I needed it. Professor Samarjit Chakraborty, Professor of The University of North Carolina at Chapel Hill (ex. Technical University of Munich), warmly accepted me visiting his laboratory for three months from August in 2018. I would like to thank Prof. Dr. Sangyoung Park, Assistant Professor of Technical University of Berlin, for their hospitality and giving precious opportunities to discuss optimization design of cell-balance with batteries during my visit. Dr. Swaminathan Narayanaswamy, Researcher of Technical University of Munich, taught me what is the high-level technical research, and the way of making progress of research. The opportunity in TUM was very stimulating and precious in my life, and such experiences resulted in having encouraged me to enter Ph.D course. I must give my greatest appreciation, of course, goes to all of the members in Tomiyama-lab. Special thanks are to be given especially my colleagues, Takafumi Miyazaki, Hayato Hidari, Kazumasa Kadota, Yusuke Funabashi, and Shunsuke Negoro for supporting our daily lives and solving a lots of technical problems as well. Along with my master course, I really thank to Kana Shimada, Ryohei Nozaki, and Kenta Shirane to discuss, draft and submit manuscripts together. We can have had many discussions to achieve submitting manuscripts. These experiments could lead me to Ph.D course and actually result in changing the direction for my life. Along with my Ph.D course, I have been supported by Satoshi Ito, Yuho Toku, Wanyin Shi, Qidi Zhang, Kazuki Shima, Koyu Ohata, Masaki Sano, Tomoyasu Shimada, Chiharu Shiro, Eiji Sugahara, Yilin Zhao, Takumi Mizuno, Mao Nishira, and Tomoki Shimizu. Of course, I am truly grateful for the warmth of the other members of laboratory. As far as I am concerned, they must be the best company that I have ever known. They have always been there for me, sometimes encouraging me to come back to research with their patient support. I would like to express my greatest gratitude and respect to all people who still have been working hard for the degree of their philosophy through the world. Finally, I am eternally grateful to my family and precious for their endless support and encouragement, and to all the wonderful friends I have encountered so far. Thank you for your endless support and encouragement. I could not have done this work without the many supportive and important people in my life.

References

[1] John Shalf, “The future of computing beyond moore’s law,” *Philosophical Transactions of the Royal Society. A, Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2166, 2020.

[2] Igor L. Markov, “Limits on fundamental limits to computation,” *CoRR*, vol. abs/1408.3821, 2014.

[3] Grant Martin, “Overview of the mpsoc design challenge,” *2006 43rd ACM/IEEE Design Automation Conference*, pp. 274–279, 2006.

[4] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and J”org Henkel, “Mapping on multi/many-core systems: Survey of current and emerging trends,” *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–10, 2013.

[5] Jaspal Subhlok and Gary Vondran, “Optimal mapping of sequences of data parallel tasks,” *ACM SIGPLAN Notices*, vol. 30, no. 8, pp. 134–143, 1995.

[6] Maciej Drozdowski, “Scheduling multiprocessor tasks—an overview,” *European Journal of Operational Research*, vol. 94, no. 2, pp. 215–230, 1996.

[7] Adel Manaa and Chengbin Chu, “Scheduling multiprocessor tasks to minimise the makespan on two dedicated processors,” *European Journal of Industrial Engineering*, vol. 4, no. 3, pp. 265–279, 2010.

[8] Dirk Bouwmeester and Anton Zeilinger, “The physics of quantum information: basic concepts,” *The physics of quantum information*, pp. 1–14, 2000.

[9] Rakesh Kumar, Dean M Tullsen, Norman P Jouppi, and Parthasarathy Ranganathan, “Heterogeneous chip multiprocessors,” *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[10] Jing Liu, Kenli Li, Dakai Zhu, Jianjun Han, and Keqin Li, “Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 2, pp. 1–25, 2016.

[11] Hui Cheng, "A high efficient task scheduling algorithm based on heterogeneous multi-core processor," *2010 2nd International Workshop on Database Technology and Applications*, pp. 1–4, 2010.

[12] C'edric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andr'e Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[13] Thomas Bridi, Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini, "A constraint programming scheduler for heterogeneous high-performance computing machines," *IEEE transactions on parallel and distributed systems*, vol. 27, no. 10, pp. 2781–2794, 2016.

[14] Shaikhah AlEbrahim and Imtiaz Ahmad, "Task scheduling for heterogeneous computing systems," *The Journal of Supercomputing*, vol. 73, no. 6, pp. 2313–2338, 2017.

[15] Yan Wang, Kenli Li, Hao Chen, Ligang He, and Keqin Li, "Energy-aware data allocation and task scheduling on heterogeneous multiprocessor systems with time constraints," *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 2, pp. 134–148, 2014.

[16] AA Jerraya, M Romdhani, Phillipe Le Marrec, Fabino Hessel, Pascal Coste, C Valderrama, GF Marchioro, JM Daveau, and Nacer-Eddine Zergainoh, "Multilanguage specification for system design and codesign," *System Level Synthesis, NATO ASI*, 1999.

[17] Daniel D Gajski, Jianwen Zhu, and Rainer Dömer, "Essential issues in codesign," *Hardware/Software Co-Design: Principles and Practice*, pp. 1–45, 1997.

[18] Keith S Vallerio and Niraj K Jha, "Task graph extraction for embedded system synthesis," *16th International Conference on VLSI Design, 2003. Proceedings.*, pp. 480–486, 2003.

[19] Daniel D Gajski and Loganath Ramachandran, "Introduction to high-level synthesis," *IEEE Design & Test of Computers*, vol. 11, no. 4, pp. 44–54, 1994.

[20] Frank Vahid and Daniel Gajski, "Specification partitioning for system design," *DAC*, vol. 92, pp. 219–224, 1992.

[21] Gustavo Callou, Paulo Maciel, Eduardo Tavares, Ermeson Andrade, Bruno Nogueira, Carlos Araujo, and Paulo Cunha, "Energy consumption and execution time estimation of embedded system applications," *Microprocessors and Microsystems*, vol. 35, no. 4, pp. 426–440, 2011.

[22] Vasilios Konstantakos, Alexander Chatzigeorgiou, Spiridon Nikolaidis, and Theodore Laopoulos, "Energy consumption estimation in embedded systems," *IEEE Transactions on instrumentation and measurement*, vol. 57, no. 4, pp. 797–804, 2008.

[23] Khurram Bhatti, Cecile Belleudy, and Michel Auguin, "Power management in real time embedded systems through online and adaptive interplay of dpm and dvfs policies," *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pp. 184–191, 2010.

[24] Gang Chen, Kai Huang, and Alois Knoll, "Energy optimization for real-time multiprocessor system-on-chip with optimal dvfs and dpm combination," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 3s, pp. 1–21, 2014.

[25] Wayne H Wolf, "Hardware-software co-design of embedded systems," *Proceedings of the IEEE*, vol. 82, no. 7, pp. 967–989, 1994.

[26] Rajesh Kumar Gupta, *Co-synthesis of hardware and software for digital embedded systems*, vol. 329, Springer Science & Business Media, 2012.

[27] Tobias Langer, Lukas Osinski, and Juergen Mottok, "A survey of parallel hard-real time scheduling on task models and scheduling approaches," *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, pp. 1–8, 2017.

[28] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to parallel computing*, vol. 110, Benjamin/Cummings Redwood City, CA, 1994.

[29] David Culler, Jaswinder Pal Singh, and Anoop Gupta, *Parallel computer architecture: a hardware/software approach*, Gulf Professional Publishing, 1999.

[30] Joël Goossens and Vandy Berten, "Gang ftp scheduling of periodic and parallel rigid real-time tasks," *arXiv preprint arXiv:1006.2617*, 2010.

[31] Rainer Dömer, Daniel D Gajski, and Jianwen Zhu, "Specification and design of embedded systems," *it-Information Technology*, vol. 40, no. 3, pp. 7–12, 1998.

[32] Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, “A constraint programming approach to scheduling of malleable tasks,” *International Journal of Networking and Computing*, vol. 9, no. 2, pp. 131–146, 2019.

[33] Sarad Venugopalan and Oliver Sinnen, “Optimal linear programming solutions for multiprocessor scheduling with communication delays,” *International conference on algorithms and architectures for parallel processing*, pp. 129–138, 2012.

[34] Pierre-François Dutot, Grégory Mounié, and Denis Trystram, “Scheduling parallel tasks: Approximation algorithms,” 2004.

[35] Yang Liu, Lin Meng, Ittetsu Taniguchi, and Hiroyuki Tomiyama, “Novel list scheduling strategies for data parallelism task graphs,” *International Journal of Networking and Computing*, vol. 4, no. 2, pp. 279–290, 2014.

[36] Kana Shimada, Shogo Kitano, Ittetsu Taniguchi, and Hiroyuki Tomiyama, “Ilp-based scheduling for parallelizable tasks,” *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. 100, no. 7, pp. 1503–1505, 2017.

[37] Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, “Ilp-based scheduling for malleable fork-join tasks,” *ACM SIGBED Review*, vol. 16, no. 3, pp. 21–26, 2019.

[38] Hoeseok Yang and Soonhoi Ha, “Ilp based data parallel multi-task mapping/scheduling technique for mpsoc,” *2008 International SoC Design Conference*, vol. 1, pp. I–134, 2008.

[39] Juris Hartmanis, “Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson),” *Siam Review*, vol. 24, no. 1, pp. 90, 1982.

[40] Francesca Rossi, Peter Van Beek, and Toby Walsh, *Handbook of constraint programming*, Elsevier, 2006.

[41] Philippe Baptiste, Claude Le Pape, and Wim Nuijten, *Constraint-based scheduling: applying constraint programming to scheduling problems*, vol. 39, Springer Science & Business Media, 2001.

[42] Krzysztof Kuchcinski, “Constraints-driven scheduling and resource assignment,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 3, pp. 355–383, 2003.

[43] Hoeseok Yang and Soonhoi Ha, “Pipelined data parallel task mapping/scheduling technique for mpsoc,” *2009 Design, Automation & Test in Europe Conference & Exhibition*, pp. 69–74, 2009.

[44] Chi-Yeh Chen and Chih-Ping Chu, “A 3.42-approximation algorithm for scheduling malleable tasks under precedence constraints,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1479–1488, 2012.

[45] Jinghao Sun, Nan Guan, Yang Wang, Qingxu Deng, Peng Zeng, and Wang Yi, “Feasibility of fork-join real-time task graph models: Hardness and algorithms,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 1, pp. 1–28, 2016.

[46] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” *2010 31st IEEE Real-Time Systems Symposium*, pp. 259–268, 2010.

[47] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill, “Multi-core real-time scheduling for generalized parallel task models,” *Real-Time Systems*, vol. 49, no. 4, pp. 404–435, 2013.

[48] Alfredo Goldman and Yanik Ngoko, “A milp approach to schedule parallel independent tasks,” *2008 International Symposium on Parallel and Distributed Computing*, pp. 115–122, 2008.

[49] Weichen Liu, Zonghua Gu, Jiang Xu, Xiaowen Wu, and Yaoyao Ye, “Satisfiability modulo graph theory for task mapping and scheduling on multiprocessor systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 8, pp. 1382–1389, 2010.

[50] Avinash Malik, Cameron Walker, Michael O’Sullivan, and Oliver Sinnen, “Satisfiability modulo theory (smt) formulation for optimal scheduling of task graphs with communication delay,” *Computers & Operations Research*, vol. 89, pp. 113–126, 2018.

[51] Ridvan Gedik, Darshan Kalathia, Gokhan Egilmez, and Emre Kirac, “A constraint programming approach for solving unrelated parallel machine scheduling problem,” *Computers & Industrial Engineering*, vol. 121, pp. 139–149, 2018.

[52] Irvin J Lustig and Jean-François Puget, “Program does not equal program: Constraint programming and its relationship to mathematical programming,” *Interfaces*, vol. 31, no. 6, pp. 29–53, 2001.

[53] Philippe Laborie, "Ibm ilog cp optimizer for detailed scheduling illustrated on three problems," *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pp. 148–162, 2009.

[54] Laurent Perron, Paul Shaw, and Vincent Furnon, "Propagation guided large neighborhood search," *International Conference on Principles and Practice of Constraint Programming*, pp. 468–481, 2004.

[55] Philippe Refalo, "Impact-based search strategies for constraint programming," *International Conference on Principles and Practice of Constraint Programming*, pp. 557–571, 2004.

[56] Robert P Dick, David L Rhodes, and Wayne Wolf, "Tgff: task graphs for free," *Proceedings of the Sixth International Workshop on Hardware/Software Codesign.(CODES/CASHE'98)*, pp. 97–101, 1998.

[57] Takao Tobita and Hironori Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *Journal of Scheduling*, vol. 5, no. 5, pp. 379–394, 2002.

[58] Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Moldable fork-join tasks scheduling techniques with inter-and intra-task communications," *Accepted for publication in International Journal of Embedded Systems*.

[59] Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Scheduling of moldable fork-join tasks with inter-and intra-task communications," *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, pp. 7–12, 2020.

[60] Dror G Feitelson and Larry Rudolph, "Toward convergence in job schedulers for parallel supercomputers," *Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 1–26, 1996.

[61] Yu-Kwong Kwok and Ishfaq Ahmad, "On multiprocessor task scheduling using efficient state space search approaches," *Journal of Parallel and Distributed Computing*, vol. 65, no. 12, pp. 1515–1532, 2005.

[62] Tatjana Davidović and Teodor Gabriel Crainic, "Benchmark-problem instances for static scheduling of task graphs with communication delays on homogeneous multiprocessor systems," *Computers & operations research*, vol. 33, no. 8, pp. 2155–2177, 2006.

[63] Ahmed Zaki Semar Shahul and Oliver Sinnen, "Scheduling task graphs optimally with a," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 310–332, 2010.

[64] Sarad Venugopalan and Oliver Sinnen, "Ilp formulations for optimal task scheduling with communication delays on parallel systems," *Ieee transactions on parallel and distributed systems*, vol. 26, no. 1, pp. 142–151, 2014.

[65] Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Communication-aware scheduling of data-parallel tasks on multicore architectures," *IPSJ Transactions on System LSI Design Methodology*, vol. 12, pp. 65–73, 2019.

[66] Yu-Kwong Kwok and Ishfaq Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[67] Oliver Sinnen, *Task scheduling for parallel systems*, vol. 60, John Wiley & Sons, 2007.

[68] Shankar Ramaswamy, Sachin Sapatnekar, and Prithviraj Banerjee, "A framework for exploiting task and data parallelism on distributed memory multicomputers," *IEEE transactions on parallel and distributed systems*, vol. 8, no. 11, pp. 1098–1116, 1997.

[69] Saniya Ben Hassen, Henri E Bal, and Ceriel JH Jacobs, "A task-and data-parallel programming language based on shared objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 6, pp. 1131–1170, 1998.

[70] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic, "Techniques optimizing the number of processors to schedule multi-threaded tasks," *2012 24th Euromicro Conference on Real-Time Systems*, pp. 321–330, 2012.

[71] Yang Liu, Lin Meng, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "A dual-mode scheduling approach for task graphs with data parallelism," *International Journal of Embedded Systems*, vol. 9, no. 2, pp. 147–156, 2017.

[72] Yang Liu, Lin Meng, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "A branch-and-bound approach to scheduling of data-parallel tasks on multi-core architectures," *International Journal of Embedded Systems*, vol. 12, no. 1, pp. 125–135, 2020.

[73] Klaus Jansen and Felix Land, "Scheduling monotone moldable jobs in linear time," *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 172–181, 2018.

[74] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal on Computing*, vol. 18, no. 2, pp. 244–257, 1989.

[75] Tao Yang and Apostolos Gerasoulis, "List scheduling with and without communication delays," *Parallel Computing*, vol. 19, no. 12, pp. 1321–1344, 1993.

[76] E Ilavarasan, P Thambidurai, and R Mahilmannan, "Performance effective task scheduling algorithm for heterogeneous computing system," *The 4th international symposium on parallel and distributed computing (ISPDC'05)*, pp. 28–38, 2005.

[77] Rashid Morady and Deniz Dal, "A multi-population based parallel genetic algorithm for multiprocessor task scheduling with communication costs," *2016 IEEE Symposium on Computers and Communication (ISCC)*, pp. 766–772, 2016.

[78] Sanjit Kumar Roy, Rajesh Devaraj, Arnab Sarkar, Sayani Sinha, and Kankana Maji, "Optimal scheduling of precedence-constrained task graphs on heterogeneous distributed systems with shared buses," *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 185–192, 2019.

[79] Min-You Wu and Daniel D Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE transactions on parallel and distributed systems*, vol. 1, no. 3, pp. 330–343, 1990.

[80] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi, "Characterizing and profiling scientific workflows," *Future generation computer systems*, vol. 29, no. 3, pp. 682–692, 2013.

[81] Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Simultaneous scheduling and core-type optimization for moldable fork-join tasks on heterogeneous multicores," *Accepted for publication in IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E105-A, no. 3, 2022.

[82] Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Energy-aware scheduling of malleable fork-join tasks under a deadline constraint on heterogeneous multicores," *ACM SIGBED Review*, vol. 16, no. 3, pp. 57–62, 2019.

[83] Jorge Barbosa, Celeste Morais, Ruben Nobrega, and António P Monteiro, “Static scheduling of dependent parallel tasks on heterogeneous clusters,” *2005 IEEE international conference on cluster computing*, pp. 1–8, 2005.

[84] Yang Qin, Gang Zeng, Ryo Kurachi, Yutaka Matsubara, and Hiroaki Takada, “Energy-aware task allocation for heterogeneous multiprocessor systems by using integer linear programming,” *Journal of Information Processing*, vol. 27, pp. 136–148, 2019.

[85] Hiroki Nishikawa, Kenta Shirane, Ryohei Nozaki, Ittetsu Taniguchi, and Hiroyuki Tomiyama, “Function-level module sharing techniques in high-level synthesis,” *ETRI Journal*, vol. 42, no. 4, pp. 527–533, 2020.

[86] Daniel D Gajski, Nikil D Dutt, Allen CH Wu, and Steve YL Lin, *High—Level Synthesis: Introduction to Chip and System Design*, Springer Science & Business Media, 2012.

[87] Michael C McFarland, Alice C Parker, and Raul Camposano, “The high-level synthesis of digital systems,” *Proceedings of the IEEE*, vol. 78, no. 2, pp. 301–318, 1990.

[88] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii, “Behavioral partitioning with exploiting function-level parallelism,” *2008 International SoC Design Conference*, vol. 1, pp. I–121, 2008.

[89] Frank Vahid, “Partitioning sequential programs for cad using a three-step approach,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 7, no. 3, pp. 413–429, 2002.

[90] Welson Sun, Michael J Wirthlin, and Stephen Neuendorffer, “Fpga pipeline synthesis design exploration using module selection and resource sharing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[91] Ritchie Zhao, Mingxing Tan, Steve Dai, and Zhiru Zhang, “Area-efficient pipelining for fpga-targeted high-level synthesis,” *Proceedings of the 52nd Annual Design Automation Conference*, pp. 1–6, 2015.

[92] Mythri Alle, Antoine Morvan, and Steven Derrien, “Runtime dependency analysis for loop pipelining in high-level synthesis,” *Proceedings of the 50th Annual Design Automation Conference*, pp. 1–10, 2013.

[93] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada, "Function call optimization for efficient behavioral synthesis," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 90, no. 9, pp. 2032–2036, 2007.

[94] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada, "Partitioning of behavioral descriptions with exploiting function-level parallelism," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 93, no. 2, pp. 488–499, 2010.

[95] Salil Raje and Reinaldo A Bergamaschi, "Generalized resource sharing," *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pp. 326–332, 1997.

[96] João MP Cardoso, "Novel algorithm combining temporal partitioning and sharing of functional units," *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, pp. 31–40, 2001.

[97] Jason Cong and Wei Jiang, "Pattern-based behavior synthesis for fpga resource reduction," *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pp. 107–116, 2008.

[98] Marco Minutoli, Vito Giovanni Castellana, Antonino Tumeo, and Fabrizio Ferrandi, "Inter-procedural resource sharing in high level synthesis through function proxies," *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, 2015.

[99] Ryohei Nozaki, Hiroki Nishikawa, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Function-level module sharing in high-level synthesis," *2019 International SoC Design Conference (ISOCC)*, pp. 50–51, 2019.

[100] John Hauser, "Softfloat," *http://HTTP. CS. Berkeley. EDU/˜ jhauser/arithmetic/softfloat. html*, 1997.

Publications

Journal Publications

Peer-Reviewed

1. Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "A constraint programming approach to scheduling of malleable tasks," *International Journal of Networking and Computing*, vol. 9, no. 2, pp. 131-146, July 2019.
2. Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Energy-aware scheduling of malleable fork-join tasks under a deadline constraint on heterogeneous multicores," *ACM SIGBED Review*, vol. 16, no. 3, pp. 57-62, October 2019.
3. Hiroki Nishikawa, Kenta Shirane, Ryohei Nozaki, Ittetsu Taniguchi, Hiroyuki Tomiyama, "Function-level module sharing techniques in high-level synthesis,“ *ETRI Journal*, vol. 42, no. 4, pp. 527-533, August 2020.
4. Satoshi Ito, Hiroki Nishikawa, Xiangbo Kong, Yusuke Funabashi, Atsuya Shibata, Shunsuke Negoro, Ittetsu Taniguchi and Hiroyuki Tomiyama, "Energy-aware routing of delivery drones under windy conditions," *IPSJ Transactions on System LSI Design Methodology*, vol. 14, pp. 30-39, August 2021.
5. Takuma Hikida, Hiroki Nishikawa, Hiroyuki Tomiyama, "Heuristic algorithms for dynamic scheduling of moldable tasks in multicore embedded systems," Accepted for publication in *International Journal of Reconfigurable and Embedded Systems*, IAES, vol. 10, no. 3, November 2021.
6. Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, Hiroyuki Tomiyama, "Moldable fork-join task scheduling techniques with inter- and intra-task communications," Accepted for publication in *International Journal of Embedded Systems*, Inderscience Publishers.
7. Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, Hiroyuki Tomiyama, "Simultaneous scheduling and core-type optimization for moldable fork-

join tasks on heterogeneous multicores," Accepted for publication in *IEICE Transactions on Fundamentals*, Vol. E105-A, no.3, March 2022.

International Conference Publications

Peer-Reviewed

8. Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Energy-aware scheduling of malleable fork-join tasks under a deadline constraint on heterogeneous multicores," In *Proc. of Embedded Operating System Workshop (EWiLi)*, Torino, Italy, October 2018.
9. Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Scheduling of malleable tasks based on constraint programming," In *Proc. of IEEE Region 10 Conference (TENCON)*, pp. 1499-1504, Jeju, Korea, October 2018.
10. Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Scheduling of malleable fork-join tasks with constraint programming," In *Proc. of International Symposium on Computing and Networking (CANDAR)*, pp. 133-138, Hida-Takayama, November 2018.
11. Ryohei Nozaki, Hiroki Nishikawa, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Function-level module sharing in high-level synthesis," In *Proc. of International SoC Design Conference (ISOCC)*, pp. 50-51, Jeju, Korea, October 2019.
12. Kana Shimada, Takuma Hikida, Hiroki Nishikawa, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Scheduling of malleable tasks with dma-based communication," In *Proc. of International SoC Design Conference (ISOCC)*, pp. 48-49, Jeju, Korea, October 2019.
13. Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Scheduling of moldable fork-join tasks with inter- and intra-Task communications," In *Proc. of International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pp. 7-12, Sankt Goar, Germany (Online), May 2020.
14. Takuma Hikida, Hiroki Nishikawa, Hiroyuki Tomiyama, "Heuristic algorithms for dynamic scheduling of moldable tasks," In *Proc. of International SoC Design Conference (ISOCC)*, pp. 55-56, Yeosu, Korea, October 2020.

15. Mayu Ida, Hiroki Nishikawa, Xiangbo Kong, Ittetsu Taniguchi, and Hiroyuki Tomiyama, “A quadcopters flight simulation considering the influence of wind,” In *Proc. of International SoC Design Conference (ISOCC)*, pp. 334-335, Yeosu, Korea, October 2020.
16. Takaya Watanabe, Hiroki Nishikawa, and Hiroyuki Tomiyama, “Scheduling of rigid tasks on heterogeneous multicores,” In *Proc. of International SoC Design Conference (ISOCC)*, pp. 330-331, Yeosu, Korea, October 2020.
17. Takuya Egashira, Hiroki Nishikawa, Xiangbo Kong, and Hiroyuki Tomiyama, “A home security camera system with container-based resource allocation on raspberry pi,” In *Proc. of International Conference on Electronics, Information, and Communication (ICEIC)*, Jeju, Korea, January-February 2021.
18. Yuho Toku, Satoshi Ito, Tomoyasu Shimada, Hiroki Nishikawa, Xiangbo Kong, and Hiroyuki Tomiyama, “Accuracy and speed evaluation of eye blink detection algorithms via openCV and dlib libraries,” In *Proc. of International Workshop on Nonlinear Circuits, Communications and Signal Processing (NCSP)*, pp. 57-60, Online, March 2021.
19. Eiji Sugahara, Hiroki Nishikawa, Takuya Egashira, Xiangbo Kong, and Hiroyuki Tomiyama, “A low-power security camera system using opencv and yolo,” In *Proc. of International Workshop on Nonlinear Circuits, Communications and Signal Processing (NCSP)*, pp. 298-300, Online, March 2021.
20. Wanyin Shi, Hiroki Nishikawa, Xiangbo Kong, and Hiroyuki Tomiyama, “Improvement of detection rate using preprocessed infrared images,” In *Proc. of International Workshop on Nonlinear Circuits, Communications and Signal Processing (NCSP)*, pp. 337-340, Online, March 2021.
21. Tomoyasu Shimada, Hiroki Nishikawa, Xiangbo Kong, and Hiroyuki Tomiyama, “A dynamic path planning method for multirotor using depth images in airsim,” In *Proc. of International Workshop on Nonlinear Circuits, Communications and Signal Processing (NCSP)*, pp. 341-344, Online, March 2021.
22. Satoshi Ito, Keishi Akaiwa, Yusuke Funabashi, Hiroki Nishikawa, Xiangbo Kong, Ittetsu Taniguchi and Hiroyuki Tomiyama, “Routing of delivery drones considering load and wind effects,” Accepted for publication at *International Symposium on Advanced Technologies and Applications in the Internet of Things (ATAIT)*, August 2021.
23. Tomoyasu Shimada, Hiroki Nishikawa, Xiangbo Kong and Hiroyuki Tomiyama, “Pix2Pix-based depth estimation from monocular images for dynamic path planning of multirotor on airSim,” Accepted for publication at *International*

Symposium on Advanced Technologies and Applications in the Internet of Things (ATAIT), August 2021.

24. Chiharu Shiro, Hiroki Nishikawa, Xiangbo Kong, Hiroyuki Tomiyama, Shigeru Yamashita, “Minimization of routing area in meda biochips,” Accepted for publication at *Biomedical Circuits and Systems Conference (BioCAS)*, Online, October 2021.
25. Kenta Shirane, Hiroki Nishikawa, Xiangbo Kong, Hiroyuki Tomiyama, “High-level synthesis of approximate computing circuits with dual accuracy modes,” Accepted for publication at *International SoC Design Conference (ISOCC)*, Jeju, Korea, October 2021.
26. Koyu Ohata, Kenta Shirane, Hiroki Nishikawa, Xiangbo Kong, Hiroyuki Tomiyama, “Scheduling with variable-cycle approximate functional units in high-level synthesis,” Accepted for publication at *International SoC Design Conference (ISOCC)*, Jeju, Korea, October 2021.
27. Masaki Sano, Kenta Shirane, Hiroki Nishikawa, Xiangbo Kong, Hiroyuki Tomiyama, Tongxin Yang, Tomoaki Ukezono, “Design of a 32-bit accuracy-controllable approximate multiplier for fpgas,” Accepted for publication at *International SoC Design Conference (ISOCC)*, Jeju, Korea, October 2021.
28. Yilin Zhao, Qidi Zhang, Hiroki Nishikawa, Xiangbo Kong, Hiroyuki Tomiyama, “Power side-channel analysis for different adders on fpga,” Accepted for publication at *International SoC Design Conference (ISOCC)*, Jeju, Korea, October 2021.
29. Tomoyasu Shimada, Hiroki Nishikawa, Xiangbo Kong, and Hiroyuki Tomiyama, “Depth Estimation from Monocular Infrared Images for Autonomous Flight of Drones,” Accepted for publication at *International Conference on Electronics, Information, and Communication (ICEIC)*, Jeju, Korea, February 2022.
30. Mao Nishira, Satoshi Ito, Hiroki Nishikawa, Xiangbo Kong, and Hiroyuki Tomiyama, “An ILP-based Approach to Delivery Drone Routing under Load-dependent Flight Speed,” Accepted for publication at *International Conference on Electronics, Information, and Communication (ICEIC)*, Jeju, Korea, February 2022.

Domestic Conference Publications

Peer-Reviewed

30. 西川広記, 島田佳奈, 谷口一徹, 冨山宏之, “Malleable Task Scheduling with Constraint Programming,” 回路とシステムワークショップ論文集, 北九州, pp. 203-207, 2018年5月.
31. 嶋田知泰, 西川広記, 孔祥博, 冨山宏之, “ドローンの衝突回避のための単眼画像を用いた深度推定画像の生成,” 回路とシステムワークショップ, オンライン, 2021年8月.
32. 大幡孝融, 白根健太, 西川広記, 孔祥博, 冨山宏之, “高位合成における可変サイクル近似演算のスケジューリング,” 回路とシステムワークショップ, オンライン, 2021年8月.
33. 佐野正樹, 白根健太, 西川広記, 孔祥博, 冨山宏之, ヨウドンキン, 請園智玲, “FPGA向け32ビット可変精度近似乗算器の設計と解析,” 回路とシステムワークショップ, オンライン, 2021年8月.
34. 菅原英治, 江頭拓也, 西川広記, 孔祥博, 冨山宏之, “セキュリティカメラシステムの低電力化と高速化,” 回路とシステムワークショップ, オンライン, 2021年8月.
35. 城千春, 西川広記, 孔祥博, 冨山宏之, 山下茂, “MEDAバイオチップにおける使用面積の最小化,” 回路とシステムワークショップ, オンライン, 2021年8月.

Non Peer-Reviewed

36. 西川広記, 島田佳奈, 谷口一徹, 冨山宏之, “非均質マルチコアにおける可変並列度タスクの低消費エネルギー化スケジューリング,” 電子情報通信学会*VLD/DC/*情報処理学会*SLDM/EMB*研究会, 広島, 2018年12月.
37. 西川広記, 島田佳奈, 谷口一徹, 冨山宏之, “タスク間とスレッド間の通信を考慮した可変並列度Fork-Joinタスクのスケジューリング,” 情報処理学会*SLDM/EMB/*電子情報通信学会*VLD/DC*研究会, 松山, 2019年11月.
38. 疋田拓万, 西川広記, 冨山宏之, “可変並列度タスクの動的スケジューリングアルゴリズム,” 電子情報通信学会*VLD/HWS*研究会, オンライン, 2021年3月.
39. 白根健太, 西川広記, 孔祥博, 冨山宏之, “精度可変な近似計算回路の高位合成,” 電子情報通信学会*VLD/HWS*研究会, オンライン, 2021年3月.
40. 伊藤哲, 根來俊介, 孔祥博, 谷口一徹, 冨山宏之, “配送用ドローンの消費エネルギーモデリングの改良,” 情報処理学会組込みシステム研究会, 横浜, 2021年11月.

41. 嶋田知泰, 西川広記, 孔祥博, 冨山宏之, “ドローンの自律飛行のための赤外線画像から深度画像の生成,” 情報処理学会組込みシステム研究会, 横浜, 2021年11月.
42. 西羅真央, 伊藤哲, 西川広記, 孔祥博, 冨山宏之, “荷重により速度変化する荷物配送ドローンの経路計画に対する近似解法,” 情報処理学会組込みシステム研究会, 横浜, 2021年11月.
43. 徳雄帆, 伊藤哲, 嶋田知泰, 西川広記, 孔祥博, 冨山宏之, “OpenCVとDlibライブラリを用いた瞬き検出プログラムの開発,” 情報処理学会組込みシステム研究会, 横浜, 2021年11月.
44. 趙意琳, 張啓迪, 西川広記, 孔祥博, 冨山宏之, “FPGAにおける加算器の電力解析攻撃耐性の評価,” 情報処理学会組込みシステム研究会, 横浜, 2021年11月.
45. 水野拓己, 張啓迪, 西川広記, 孔祥博, 冨山宏之, “高位合成における最適化のサイドチャネル攻撃耐性への影響,” 情報処理学会組込みシステム研究会, 横浜, 2021年11月.
46. 伊藤哲, 赤岩慧士, 舟橋勇佑, 西川広記, 孔祥博, 谷口一徹, 冨山宏之, “荷重と風による飛行速度の変化を考慮したドローン配送計画,” 電子情報通信学会VLD/DC/RECONF/ICD/情報処理学会SLDM研究会, オンライン, 2021年12月.