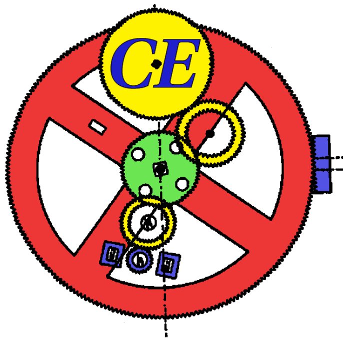

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Task Scheduling Methods for Composable and Predictable MPSoCs

Ba Thang Nguyen

CE-MS-2010-29

Abstract

Multiprocessor Systems on a Chip (MPSoCs) are suitable platforms for executing complex embedded applications. To reduce the cost of the hardware platform, applications share resources, which may result in inter-application timing interference due to resource request conflicts. Bounding or prohibiting this interference is crucial, as the timing of real-time applications has to be predicted in each possible case. Resources that allow sharing without application interference are denoted as composable. Composability is a desired platform property, as it enables the design and analysis of applications in isolation, and their integration with linear effort. Previous work demonstrates composability for different resources, i.e., processor, interconnect, memory. Processor composability is achieved by utilizing an Operating System (OS) that schedules fixed duration task slots, using a two-level, hierarchical approach. First, the OS determines which application owns the next slot following a strict, preemptive Time Division Multiplexing (TDM) policy, and then it picks and schedules a task of that application. As scheduling decisions are taken exclusively at slots borders, when a task finishes before its slot depletes, the time left is wasted. This may result in low processor utilization for streaming applications for which the execution of a task may start after its predecessor tasks have finished. In this work we propose a new task scheduling strategy, namely application-space task scheduling that eliminates wasted slot time. We make use of the fixed duration slots and the application TDM, to preserve composability, but the application invokes the task scheduler immediately after each task finish, inside its slot. As the application-space task scheduling strategy alone may not support all types of task scheduling, e.g., preemptive, we propose to combine OS-space and application-space scheduling on the same processor. To experimentally investigate the composability and performance of our scheme we survey existing benchmarks for the embedded domain, and build a workload consisting of two streaming applications and a synthetic application. We executed these applications on an MPSoC with two processor tiles, a monitor tile, all connected by a Æthereal NoC. Our experiments indicate that mixing application-space and OS-space task schedulers is composable. Furthermore, the application-space task scheduling achieves 17% to 40% better performance than the OS-space task scheduling for the streaming applications exercised.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Task Scheduling Methods for Composable and Predictable MPSoCs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Ba Thang Nguyen
born in Hai Phong, Vietnam

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Task Scheduling Methods for Composable and Predictable MPSoCs

by Ba Thang Nguyen

Abstract

Multiprocessor Systems on a Chip (MPSoCs) are suitable platforms for executing complex embedded applications. To reduce the cost of the hardware platform, applications share resources, which may result in inter-application timing interference due to resource request conflicts. Bounding or prohibiting this interference is crucial, as the timing of real-time applications has to be predicted in each possible case. Resources that allow sharing without application interference are denoted as composable. Composability is a desired platform property, as it enables the design and analysis of applications in isolation, and their integration with linear effort. Previous work demonstrates composability for different resources, i.e., processor, interconnect, memory. Processor composability is achieved by utilizing an Operating System (OS) that schedules fixed duration task slots, using a two-level, hierarchical approach. First, the OS determines which application owns the next slot following a strict, preemptive Time Division Multiplexing (TDM) policy, and then it picks and schedules a task of that application. As scheduling decisions are taken exclusively at slots borders, when a task finishes before its slot depletes, the time left is wasted. This may result in low processor utilization for streaming applications for which the execution of a task may start after its predecessor tasks have finished. In this work we propose a new task scheduling strategy, namely application-space task scheduling that eliminates wasted slot time. We make use of the fixed duration slots and the application TDM, to preserve composability, but the application invokes the task scheduler immediately after each task finish, inside its slot. As the application-space task scheduling strategy alone may not support all types of task scheduling, e.g., preemptive, we propose to combine OS-space and application-space scheduling on the same processor. To experimentally investigate the composability and performance of our scheme we survey existing benchmarks for the embedded domain, and build a workload consisting of two streaming applications and a synthetic application. We executed these applications on an MPSoC with two processor tiles, a monitor tile, all connected by a Æthereal NoC. Our experiments indicate that mixing application-space and OS-space task schedulers is composable. Furthermore, the application-space task scheduling achieves 17% to 40% better performance than the OS-space task scheduling for the streaming applications exercised.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2010-29

Committee Members :

Advisor: Prof. Sorin Cotofana, CE, TU Delft

Advisor: Dr. Anca Monos, CE, TU Delft

Member: Prof. Sorin Cotofana, CE, TU Delft

Member: Prof. Kees Goossens, TU Eindhoven

Member: Prof. Georgi Kuzmanov, CE, TU Delft

Member: Dr. Anca Molnos, CE, TU Delft

Dedicated to my parents and dearest wife

Contents

List of Figures viii

List of Tables ix

Acknowledgements xi

1 Introduction **1**
1.1 Problem Statement 3
1.2 Contributions 3
1.3 Organization 5

2 Background **7**
2.1 Hardware Platform 7
2.1.1 Processor Tile 7
2.1.2 Monitor Tile 9
2.2 Streaming Application Model 9
2.2.1 Task Communication 10
2.2.2 Task Construction 11
2.3 CompOSe 11
2.3.1 Functionality 11
2.3.2 Data Structures 13
2.3.3 Initialization 15
2.4 Summary 16

3 Composable Task Scheduling **17**
3.1 Related Work 17
3.2 OS-Space and Application-Space Task Scheduling 18
3.3 Task Scheduler Implementation 19
3.3.1 Data Structure 19
3.3.2 Task Slot Functionality 20
3.3.3 OS Slot Functionality 23
3.4 Task Scheduler Algorithms 24
3.5 Summary 26

4 Embedded Benchmark Applications **27**
4.1 Portability of Existing Benchmarks 27
4.2 Embedded Application Benchmark Suite Implementation 29
4.2.1 H.264 Decoder Application 29
4.2.2 JPEG Decoder Application 31
4.2.3 Synthetic Application 33

4.3 Summary 34

5 Experiments and Results **37**
5.1 Composability Experiments 37
5.1.1 Task Scheduler Composability 38
5.1.2 Interconnect, Memory Composability 39
5.2 Application Performance Experiments 43
5.2.1 Application Performance over Multiple Iterations 43
5.2.2 Task Slot Duration Variations 44
5.3 Summary 47

6 Conclusions and Future Work **49**
6.1 Conclusions 49
6.2 Future Work 50
6.2.1 Embedded Application Benchmark Suite 50
6.2.2 Hierarchical Slot 50

Bibliography **53**

List of Figures

1.1 Architecture of an MPSoC 1
1.2 Data-Driven Application Model 2
1.3 Task Scheduling Strategies 4

2.1 Hardware Platform Architecture 7
2.2 Processor tile architecture 8
2.3 Voltage Frequency Control Unit 9
2.4 Monitor Tile Architecture 9
2.5 C-HEAP Protocol 10
2.6 System Slot Time 11
2.7 Functional Flowchart 12
2.8 Task State Transition 13
2.9 Data Structure Organization 14

3.1 Hierarchical Scheduler Framework 18
3.2 OS-Space Task Scheduling 19
3.3 Application-Space Task Scheduling 19
3.4 New ACB Data Structure 20
3.5 New Task Slot Functional Flowchart 21
3.6 New Task State Transition 23
3.7 New OS Slot Functional Flowchart 25

4.1 Mapping of H.264 Decoder 30
4.2 Execution Time of Tasks in H.264 Decoder 32
4.3 Mapping of JPEG Decoder 33
4.4 Execution Time of Tasks in JPEG Decoder 34
4.5 Synthetic Application Graph 34

5.1 Application TDM Order Experiment 38
5.2 Differences in H.264 Start Time - Scheduler Composability Experiment . 38
5.3 Differences in H.264 Response Time - Scheduler Composability Experiment 39
5.4 Differences in JPEG Start Time - Scheduler Composability Experiment . 40
5.5 Differences in JPEG Response Time - Scheduler Composability Experiment 40
5.6 Differences in H.264 Start Time - Memory Composability Experiment . . 41
5.7 Differences in H.264 Response Time - Memory Composability Experiment 41
5.8 Differences in JPEG Start Time - Memory Composability Experiment . . 42
5.9 Differences in JPEG Response Time - Memory Composability Experiment 42
5.10 H.264 Scheduler Experiment 43
5.11 JPEG Scheduler Experiment 44
5.12 H.264 Application Performance Improvement 45
5.13 JPEG Application Performance Improvement 45
5.14 H.264 Task Slot Duration Experiment 46
5.15 JPEG Task Slot Duration Experiment 46

6.1 Hierarchical Slot 50

List of Tables

4.1 Difficulties to Port the Existing Benchmarks on our platform 29
4.2 H.264 Communication FIFOs 30
4.3 Workload Percentage of H.264 Decoder Tasks 31
4.4 JPEG Communication FIFOs 31
4.5 Workload Percentage of JPEG Decoder Tasks 33

5.1 Processor Tile Configuration 37

Acknowledgements

First and foremost, I would like to thank my advisors, Prof. Kees Goossens and Dr. Anca Molnos, for showing me the path of research with their enlightening advise and expert guidance, and for their continuous support and constant encouragement throughout this research work. I will also take this opportunity to thank all my fellow graduate students for their help and friendship. Last, but not the least, I would like to appreciate my parents, sister and my dearest wife for their endless love, encouragement and support that permitted me to complete this work.

Ba Thang Nguyen
Delft, The Netherlands
December 21, 2010

1 Introduction

In the recent years, the number and complexity of the applications executed on an embedded platform have drastically increased [8]. Multiprocessors system-on-chip (MPSoC) are the platforms employed to execute these complex applications. Such a platform consists of multiple processing tiles (PT), memory tiles, peripheral tiles, etc. Figure 1.1 presents a typical architecture of an MPSoC. The tiles can communicate with each other via an interconnection infrastructure. A processing tile (PT) consists of a programmable processor core, typically a CPU or DSP, and local resources, e.g. local memory, that are tightly coupled with the core.

A typical MPSoC executes a set of applications, that shared the MPSoC resources, in order to reduce the cost of the hardware platform. Each application consists of one or more tasks, each of which executing sequentially on a processor core. Based on their timing constraints, there are two main classes of applications: real-time and non real-time. An application has real-time constrains if the correctness of its operation depends not only upon its functionality, but also upon its timing behaviour, e.g. software radio decoding, video decoding. On the other hand, non real-time applications are applications for which no deadlines or no timing constrains exists, e.g., file download, email clients, etc.

Real-time applications can be divided into three categories based on the criticality of their requirements, namely hard real-time (HRT), firm real-time (FRT) and soft real-time (SRT). In hard real-time applications, missing a deadline can have disastrous consequences. An example of such an application is the airbag control system of a vehicle. Firm real-time applications have similar real-time requirements to hard real-time applications. However, missing a deadline results in unacceptable output quality degradation,

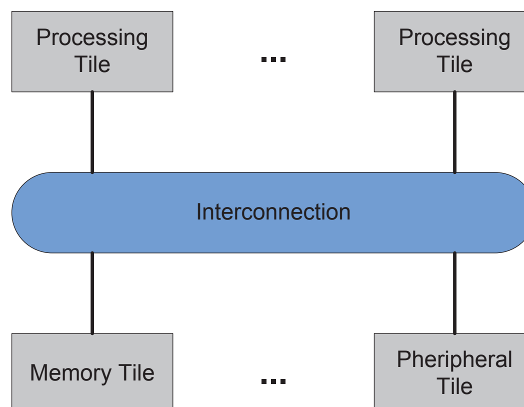

Figure 1.1: Architecture of an MPSoC

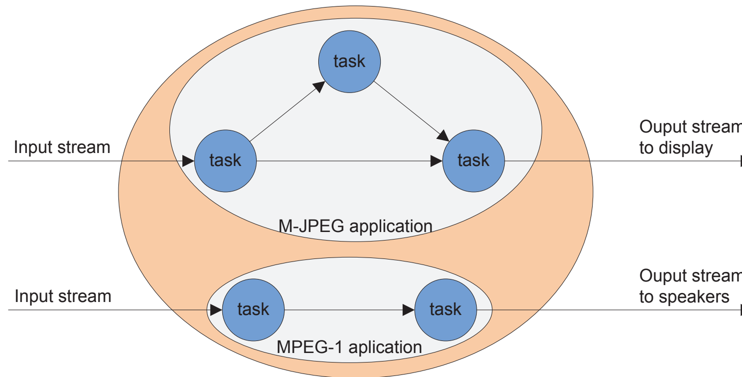

Figure 1.2: Data-Driven Application Model

but has no catastrophic implications. Example of such applications are audio decoders and software radios. Soft real-time applications are applications which can accommodate occasional deadline misses. Examples of such applications are some video encoders and decoders, for which the quality degradation due to missing a deadline can be alleviated by repeating a previous picture. To guarantee by design that real-time applications met their deadlines, their timing behavior should be **predictable** when they execute on a platform. In general, a system is predictable if we can forecast the timing behavior of its part and the interaction between them. Thus, in order to build such a system, the hardware, software and the arbitration at shared resources have to be predictable.

Furthermore, applications can also be divided into three broad classes based on the invocation of their tasks over time. These classes are: time-driven, event-driven and data-driven. Time-driven applications are applications whose tasks are triggered at precise periodic moments in time. Example of such applications are sensor monitoring and video capturing. Event-driven applications are applications whose tasks are triggered sporadically, at the occurrence of a given set of events. Examples of such applications are human-computer interaction applications. Data-driven, i.e. streaming, applications are applications that consist of communicating tasks which are triggered based on data availability. Examples of such applications are MPEG encoder and decoder. Typically, time-driven applications have real-time constrains while event-driven and data-driven applications may be real-time or non real-time. In this work, we mainly focus on data-driven applications. An example of two applications belonging to this class is presented in Figure 1.2.

As mentioned above, to decrease the cost of a hardware platform, applications share its resources, e.g., memory, peripherals, etc. The applications might interfere at shared resources due to resource request conflicts. Due to this interference, when the behavior of one application changes, e.g. an application is updated, the timing behavior of all other applications mapped on the same platform is affected. Hence sharing of resources makes timing verification difficult and integration effort exponential in the number of applications. This is especially problematic for real-time applications, for which meeting the time constrains should be guaranteed by construction.

Furthermore, timing verification becomes even more difficult when **energy and/or**

power management are employed in the system. Energy and power management involves scaling the resources operating points which has a direct impact on the application time behavior. A change in one application leads to the need to reverify all other applications.

To avoid large integration and verification effort, a **composable** design approach [13], [16], [21] is required. A system is composable if the functionality and the timing behavior of each application is independent of other applications mapped on the same platform. The composable platform ensures that the applications do not interfere each other, thus the verification and integration effort is linear in the number of applications.

1.1 Problem Statement

Prior work [25], [12] demonstrated a composable and predictable MPSoC platform. In this thesis we build upon this existing platform and we further investigate task scheduling. [25], [12] employ a preemptive two level hierarchical scheduling framework which is able to execute a mix of real-time and non real-time applications, each scheduled according to its suitable policy. The application scheduler uses Time Division Multiplexing (TDM) scheduling algorithm in order to guarantee the composability between applications running on the shared processor. TDM is implemented using precise timer interrupts. The tasks of an application can be scheduled by any scheduling algorithm. The application and task scheduler on a processor takes local decisions, and it does not need to be aligned or synchronized with any other scheduler in the system. In the existing platform [12], the processor time is divided into system time slots of equal length. One part of the system time slot, i.e. the Operating System (OS) slot, is utilized by the OS, and the rest is used as task slot where applications' tasks are executed.

Currently [25], [12] implement an OS which performs task scheduling inside the OS slot. We denote this strategy as OS-space task scheduling, and we graphically present it in Figure 1.3(A). The advantage of this strategy is that it can potentially schedule all types of applications, e.g., time-driven, event-driven and data-driven. In OS-space task scheduling, the task scheduler is invoked at the OS slot. It means that task scheduling is performed periodically, between two consecutive task slots. This implies that, even if a task finishes before the end of its slot, the OS will not schedule another task in that slot. We denote this phenomenon as task slot fragmentation. This fragmentation may results in processor under-utilization, thus potential application performance degradation.

In this work, we propose a novel application-space task scheduling strategy (Figure 1.3(B)). We propose to invoke the task scheduler in the task time slot (application time) aiming to reduce the performance loss due to slot fragmentation.

1.2 Contributions

The main contribution of this thesis is the application-space task scheduler. In this strategy task scheduling is performed immediately after a task iteration is finished, removing fragmentation. We expect that the application-space task scheduling strategy gives better application performance than OS-space task scheduling strategy, especially,

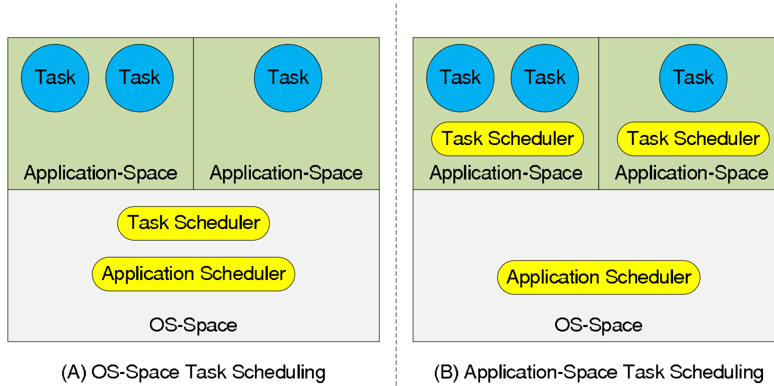

Figure 1.3: Task Scheduling Strategies

in case of data-driven applications and non real-time applications.

Although application-space task scheduling can give better application performance, this strategy alone cannot support all applications. Specifically, it cannot schedule time-driven applications if the application does not have exclusive access to a timer and an interrupt, i.e. the platform does not support virtualized timers and interrupts. Moreover, the application-space task scheduling strategy may not support all types of task scheduling, e.g., preemptive task scheduling algorithms.

We tackle this limitation by proposing a new composable strategy that supports both OS-space and application-space task scheduling. In other words, at initialization an application can specify in which manner its tasks should be scheduled. Thus the resulting platform can service different types of applications. For example, time-driven applications can use OS-space task scheduling while data-driven applications can use application-space task scheduling.

Moreover, application-space task scheduling has implications in platform energy management. Real-time applications demand a conservative energy management approach that requires the Energy Manager to closely monitor application progress [22]. In the application-space scheme the moments when a task finishes an iteration are not by default visible to the OS, thus the OS cannot monitor application progress. As a result, OS-space energy management is not possible in conjunction application-space task scheduling. Future work will address the OS-application interaction, and the energy management interfaces that alleviated this limitation.

In summary, we achieved the following:

- Design and implement a new composable task scheduling strategy which gives better performance for the applications running on the platform, namely application-space task scheduling.

- Integrate the OS-space and application-space task scheduling to offer composable and flexible task scheduling options.

- Build a set of applications to investigate the composability and compare the performance of application-space vs. OS-space task scheduling.

1.3 Organization

The remainder of the thesis is as follows. We start by giving an overview of the hardware, application model and Operating System in Chapter 2. Next, Chapter 3 details advantages and limitations of OS-space and application-space task scheduling strategies together with the implementation of new proposed task scheduler. Then Chapter 4 describes the set of applications which we built for our experiments. Finally, experimental results are presented in Chapter 5, followed by conclusion and future works in Chapter 6.

Background

2

In this chapter, we present an overview of the existing composable, predictable hardware platform and its software architecture [12]. In Section 2.1 we describe of the existing hardware platform is presented. The application model is described in Section 2.2, followed by the details of the Operating System in Section 2.3. Finally, Section 2.4 summarizes this chapter.

2.1 Hardware Platform

The hardware platform architecture is shown in Figure 2.1. Our platform consists of several processing tiles, a monitor tile and an external memory tile, all connected by a Æthereal NoC [9]. A platform is composable if each and every shared resource of the platform is composable. A composable platform is also predictable if the time to serve an application request on a resource is bounded. We utilize a composable and predictable NoC (Æthereal offers composability and predictability for every logical connection between pairs of memory-mapped initiator and target ports) and memory tile [1]. In the following we present the details of the processor tiles relevant for our task scheduling work.

2.1.1 Processor Tile

A processor tile has several main components: a MicroBlaze processor core, a set of local memories assisted by a set of Remote Direct Memory Access (RDMA) units, and a Voltage Frequency Control Unit (VFCU).

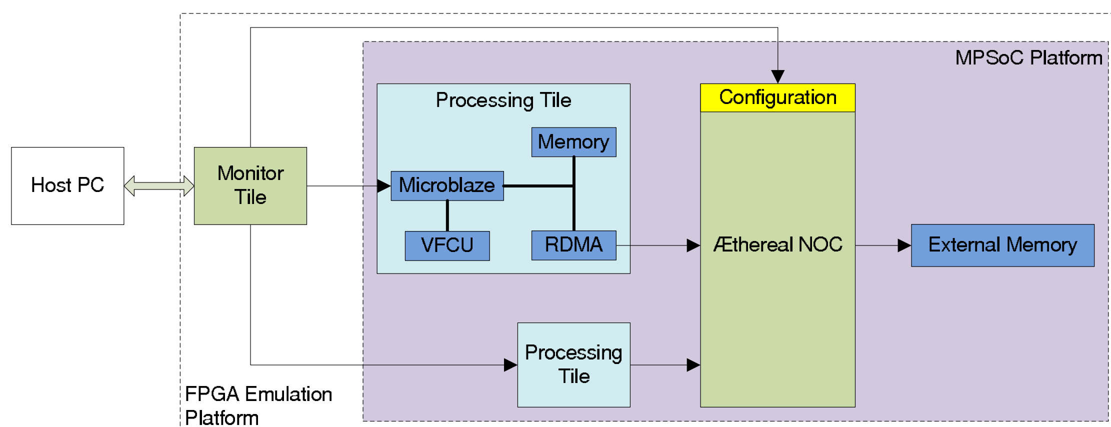

Figure 2.1: Hardware Platform Architecture

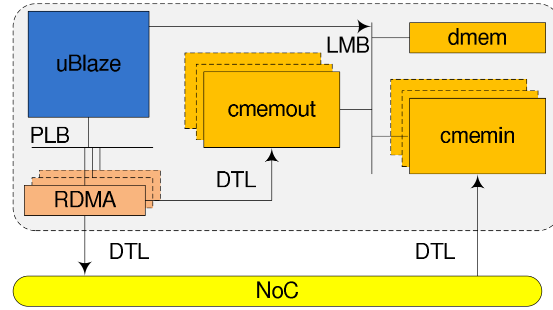

Figure 2.2: Processor tile architecture

The MicroBlaze processor core is responsible for executing the code of applications. The MicroBlaze is a 32-bit processor soft-core provided by Xilinx [29]. The instruction set architecture (ISA) of the MicroBlaze core is a Reduced Instruction set computer (RISC) ISA.

Local memories are connected with the processor via a Local Memory Bus (LMB) that has one cycle access time. The local memory is split into a data memory, which is accessible only by the local processor, and a communication memory (CMEM), as illustrated in Figure 2.2. The communication memory is further split in outgoing communication memory (cmem_out) and incoming communication memory (cmem_in). We assume that the entire task's private data and code fits in the data memory on the processor executing that task. A FIFO is mapped either in its consumer's cmem_in or in a separate memory tile. The RDMA units are used to post data from the local cmem_out of a processor to the cmem_in of another processor tile or to a remote memory tile by initiating a set of NoC transactions. The advantage of a separate communication memory is that the arbitration is avoided, resulting in shorter memory access delay and smaller chip area. To ensure composability each application that has tasks mapped on a processor has associated an RDMA unit and a pair of cmem_in and cmem_out memories.

The Voltage Frequency Control Unit (VFCU) is used to support composability and power management on a processor. The processor is programming the VFCU via a Fast Simplex Link (FSL) [28]. The structure of VFCU is shown in Figure 2.3. VFCU contains two sub modules: the system timer and the frequency generator. The system timer is responsible for providing time information for the frequency generator so that the frequency generator can function correctly. In addition to, it also provides the time reference to the processor. The programmable frequency generator provides two basic functionalities:

- Setting output frequency based on parameters given by the processor. This feature is utilized for energy and power management.
- Gating the output clock, i.e., set output frequency to 0, and un-gate it at a given future moment in time. This feature is utilized to ensure the fixed duration OS slots required by composability.

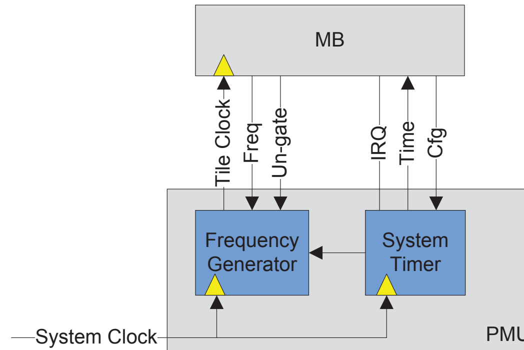

Figure 2.3: Voltage Frequency Control Unit

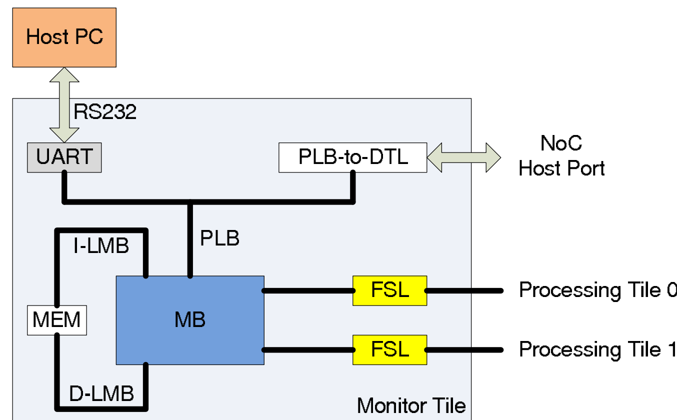

Figure 2.4: Monitor Tile Architecture

2.1.2 Monitor Tile

The monitor tile consists of a MicroBlaze core, its local memory, and peripheral interfaces to communicate with the host PC. The monitor has two basic functions. First, it configures the system i.e. NoC, UART before the start of application execution. The FSL channel from the monitor to each processor tile are used to synchronize the initialization of the processor tiles. Second, the monitor tile can be used to gather log data from each processor tiles after the application has started. When events of interest happen, the processing tiles send messages to the monitor tile through the monitor FSL. The monitor tile collects this information and sends it to the host PC. The architecture of the monitor tile is presented in Figure 2.4.

2.2 Streaming Application Model

A suitable model for a streaming application is a task graph in which tasks communicate tokens through blocking FIFOs. A task performs an infinite number of iterations, in each iteration consumes its input tokens, executes some computation, then

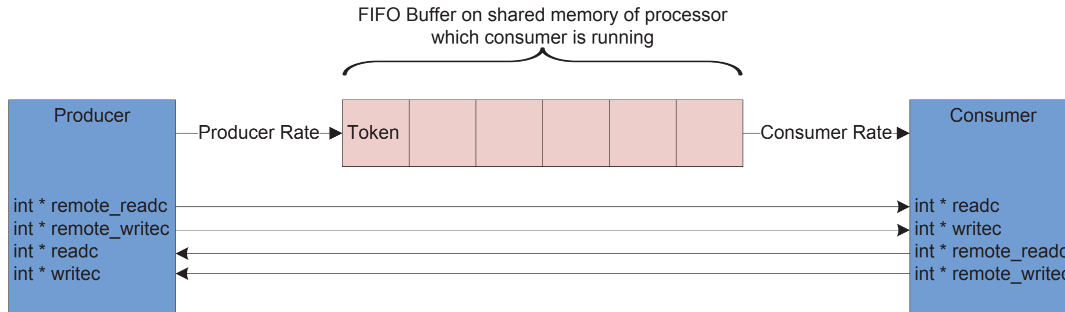

Figure 2.5: C-HEAP Protocol

produces its output tokens, which in turn may be processed by other tasks. Each FIFO is attached to one producer task and one consumer task, which block if the FIFO is full and empty, respectively. These task graphs can be modeled as Cyclo-Static Data-Flow(CSDF) graphs [7], [23] when real-time analysis is demanded. The details on how the task is constructed and communicate each other, is described as follows.

2.2.1 Task Communication

The inter-task communication is implemented using the C-HEAP [24] protocol and FIFOs. A C-HEAP FIFO is implemented as a circular buffer in shared memory, where each buffer has one producer and one consumer, as shown in Figure 2.5. The unit element in the communication is called token. The size of token is fixed per FIFO. The number of tokens, which are written/read by producer/consumer in each transaction, are denoted as token rates. The producer and consumer of the same FIFO can have different token rates. In the current implementation, we do not support communication between two tasks which belong to different applications.

If the producer and consumer tasks are mapped on the same processor, the FIFO administration and data buffer are mapped in the local memory. If the data buffer is too large for the local memory, it can be mapped onto the a remote memory tile. If the producer and consumer tasks are mapped on different processors, the FIFO data buffer may be mapped in the consumer's local memory. In this case, the communicated data is only written across the NoC. This eliminates the round trip delay that would be incurred if the consumer had to read the data across the NoC.

Regardless of the location of the FIFO data buffer, the FIFO administration, e.g. read and write pointers, are kept in the local memories of both the producer and consumer in order to avoid polling remote memory locations. Because the OS often checks the task eligibility, remote access that should be avoided, would cause a large time penalties. Moreover, the FIFO administration does not occupy a large memory space, thus, the storage overhead is negligible. The FIFO administration is updated and synchronized after data is produced into or consumed from the FIFO buffer.

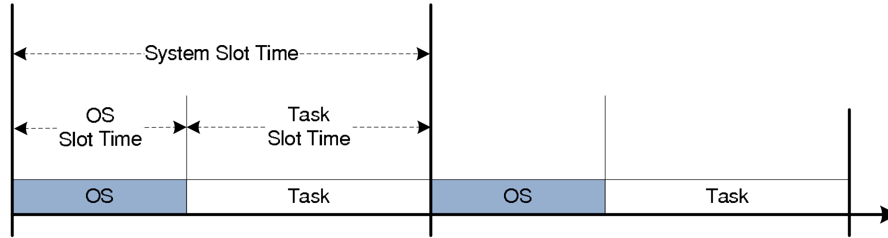

Figure 2.6: System Slot Time

2.2.2 Task Construction

Typically, a task is implemented as a never-ending loop that reads input data, performs computation and produces output. In our case, the input and output operations are left for operating system, which is discussed later in next section. We implemented a task as a function that executes and returns (for each invocation), as exemplified in Listing 2.1. This function consists of two arguments which are arrays of pointers. The first argument points to input tokens and the second argument points to output tokens.

Listing 2.1: Task Code

```
void Task1(int **in, int **out){
  out = function(in);
}
```

2.3 CompOSe

CompOSe[6] is a composable, lightweight operating system (OS). CompOSe utilizes a preemptive two-level hierarchical scheduler to enable different task schedulers per application. In our platform, each processor runs an independent instance of CompOSe, without any knowledge of the other processors. Thus each processor is completely decoupled from other processors and memories and has its own clock. We start by introducing the CompOSe functionality in section 2.3.1. Next, the data structure of CompOSe is presented in section 2.3.2. Finally, we present how to initialize CompOSe on each processor tile.

2.3.1 Functionality

In this section, we present the functionality of CompOSe. The processor time is divided into system time slots of equal length. A part of the system time slot, the OS slot, is used to execute CompOSe. The rest is used as a task slot, as presented in Figure 2.6. The operating system slot is responsible for saving context of the previous task on its stack, to schedule a new application along with its new task and restore the context of the selected task while the task slot is responsible for executing the task. Both the OS and the task slot have a fixed duration to ensure composability.

The core of CompOSe is the functional loop presented in Figure 2.7. An interrupt from the system timer marks the beginning of a new system slot that starts with an

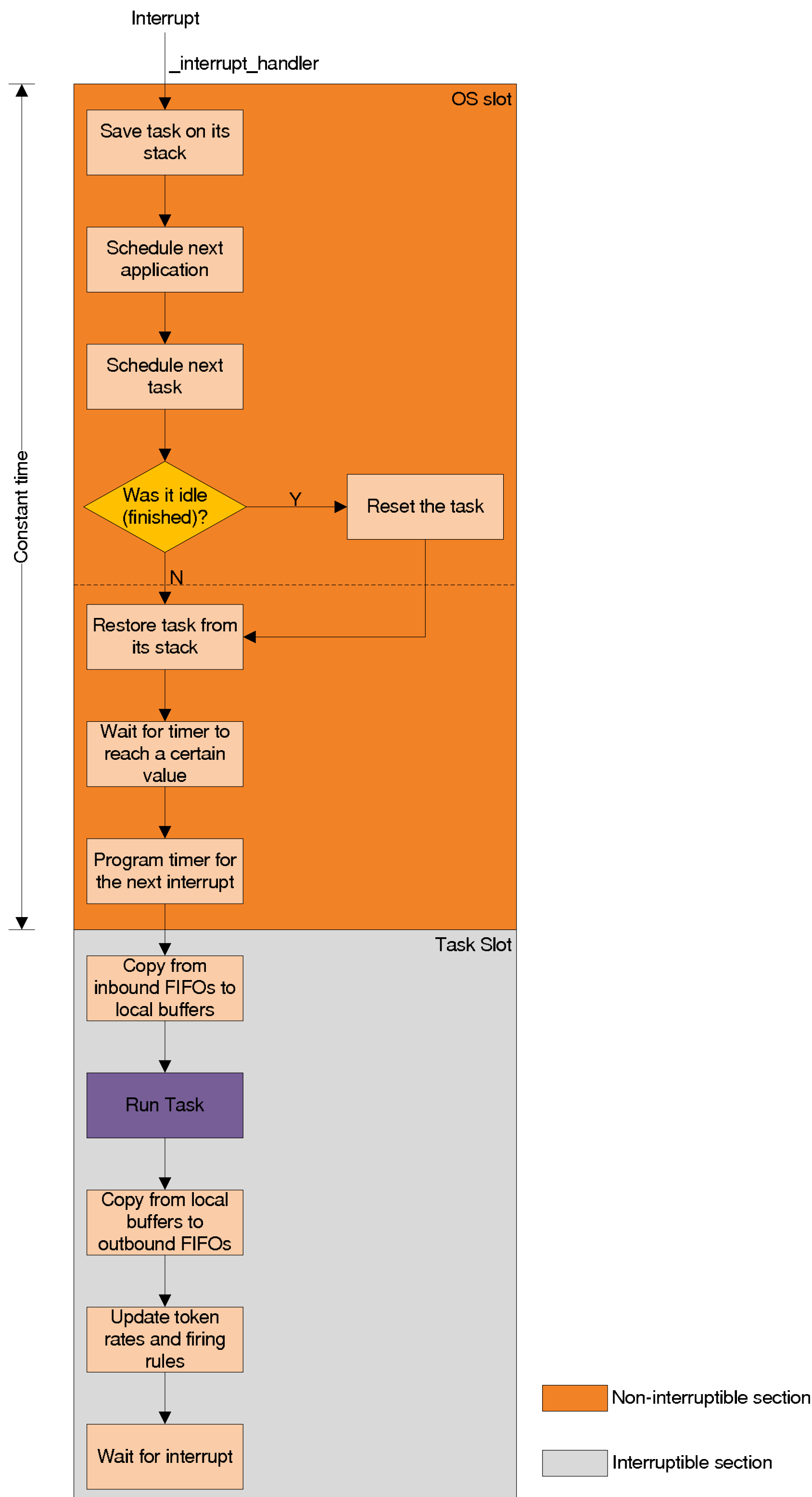

Figure 2.7: Functional Flowchart

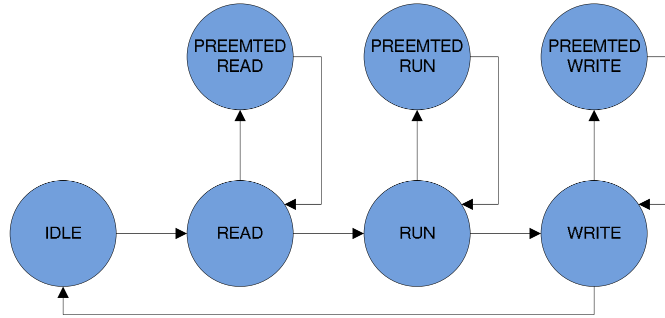

Figure 2.8: Task State Transition

OS slot. In _interrupt_handler(), the context of the interrupted task is saved on its stack. Next, the application-scheduler selects an application using the defined scheduler. Currently, there is a built-in TDM application scheduler in our system.

In the next step, the task scheduler selects the task. Each application may use any task scheduling algorithm. Then the task state is checked. If the selected task is marked as idle (finished), it is reset to its original state, meaning that the task's register values are reseted. The original state is defined as the state in which the task was in after the application was initialized, but not yet started. Finally, the task context is restored to the system from its stack and the task state changes to running.

If the timer interrupt comes while the task is executing, the task state changes to preempted and the context of the preempted task is saved on its stack. Otherwise, if the task returns before the timer interrupt, its state changes to idle (iteration finished). The details of task state transition is illustrated in Figure 2.8. The initial state of a task is IDLE. If the task is selected to run, its state changes to READ. During this time, the task reads the tokens from input FIFOs. Next, the task is executed and its state changes to RUN. After the task finishes an iteration, its state changes to WRITE where the task write tokes to output FIFOs. Finally, the task comes back to IDLE state. If the interrupt comes during the task in READ, RUN or WRITE state, its state changes to PREEMPTED READ, PREEMTED RUN and PREEMMTED WRITE state, respectively.

In order to achieve the composability for our system, we need to have a constant operating system execution time. We remove the variation in duration by halting the processor after the operating system execution, up to its worst-case duration. The worse-case operating system duration must accommodate all time required to service the interrupt, reset the task, run the application scheduler, run the task scheduler.

Moreover, in the OS slot task information is sent to monitor tile. CompOSe offers slack and power management to exploit potential unused processor capacity – idle task slots.

2.3.2 Data Structures

The key data structure elements of CompOSe are shown in Figure 2.9. At the top, we have the Core Control Block (CCB), followed by the Application Control Block (ACB),

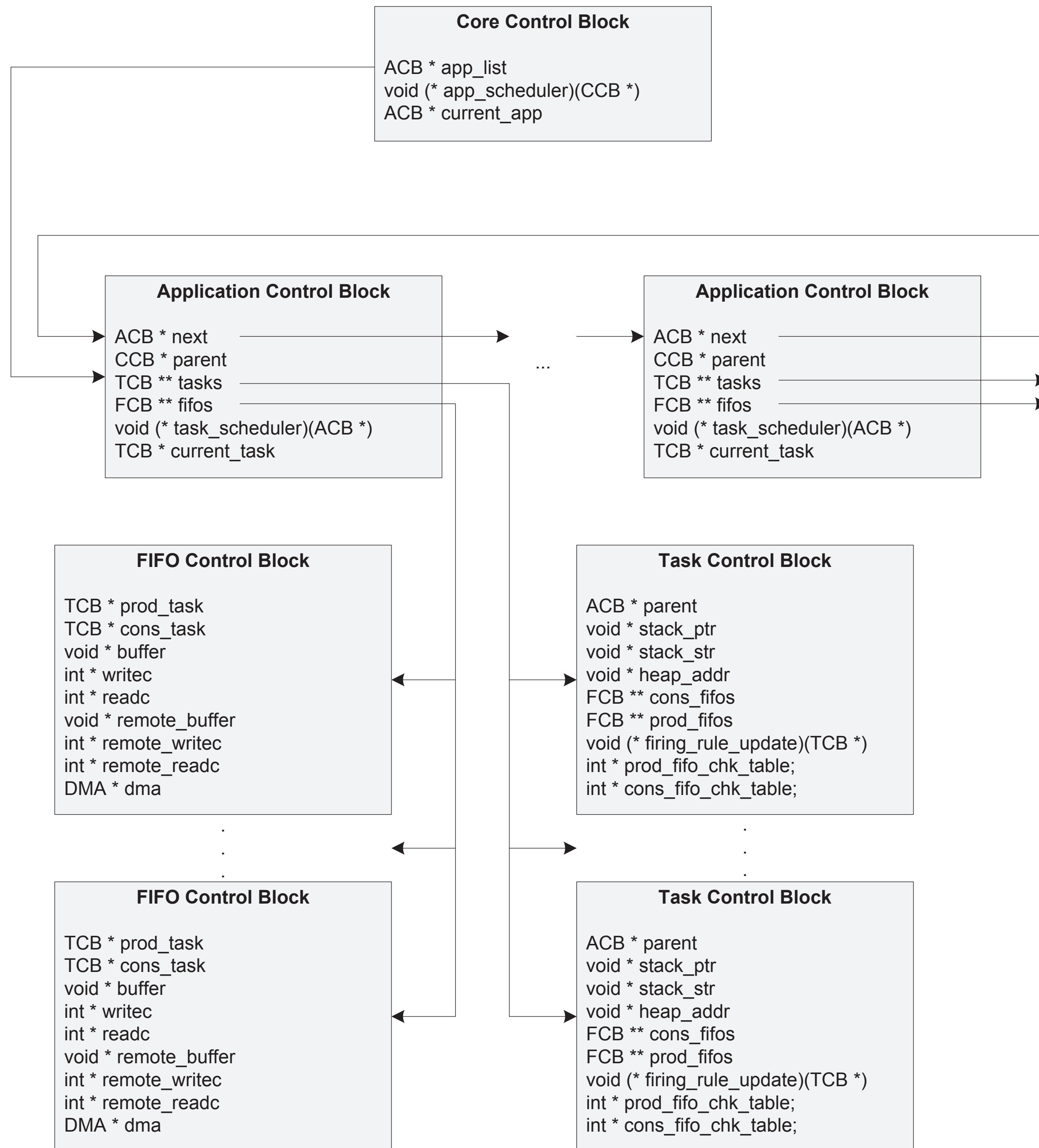

Figure 2.9: Data Structure Organization

Task Control Block (TCB) and FIFO Control Block (FCB). The data structure of each processor is dynamically allocated in the local memory of the processing unit during the system initialization. In our platform, each processor is unaware of tasks or applications on other processors, even tasks belonging to the same application.

As shown in Figure 2.9, the CCB holds the list of applications on the processor by pointing to a linked list of applications. Besides, the CCB has a function pointer to select the application scheduler for the processor in question. On the application level,

we see that each ACB holds information about all the tasks and FIFOs that belongs to the application and resides on the processor in question. Moreover, the ACB also has a function pointer, thus allowing to define different task schedulers for different applications.

The TCB consists of pointers to stack, heap start and instructions of task. Each TCB also holds the firing rules tables for input and output FIFOs. The firing rules tables hold information about which input FIFOs and output FIFOs a task reads and writes each iteration. In this manner, CompOSe knows what conditions must be satisfied for the task to run. In addition, the TCB has a pointer to a function which is called at the end of each iteration to update the firing tables. Note that, from the CSDF graphs, we can get these information to create firing rules tables.

As seen in Figure 2.9, the FCB contains pointers to the local buffer memory and remote buffer memory. Each FCB also points to its producer and consumer task. If two tasks belong to different processors, a pointer is used to select the RDMA for the communication between these two tasks.

2.3.3 Initialization

Before executing applications and tasks, we need to initialize CompOSe on each processor tile. Listing 2.2 presents a simple example of CompOSe initialization. First, this code initializes a system in which application scheduler uses TDM scheduling algorithm. Next, os_add_application and os_add_task functions are used to add applications and tasks to the system. In os_add_application function, we can specify which task scheduling algorithm is used for each applcation, e.g, round-robin task scheduling algorithm. After creating applications and tasks, we create FIFOs and link them to producer and consumer tasks. Then, we set the duration of task slot. Finally, we call os_start function to start CompOSe.

Listing 2.2: Initialization Pseudo-Code Example

```
int main(){

  //Set application schedule
  os_create_compose(system_id, &app_scheduler_tdm, ...);
  ...

  //Create and add applications to system
  os_add_application(app_id, &task_scheduler_rr, ...);
  ...

  //Create and add tasks to application
  os_add_task(task_id, app_id, &task_function, &task_firing_rule, ...);
  ...

  //Create FIFOs and link to producer and consumer task
  os_add_fifo(fifo_id, app_id, producer_task_id, consumer_task_id, ...);
  ...

  //Set the duration of task slot
  os_set_task_slot_time(task_slot_duration);
```

```
  //Start executing applications and tasks
  os_start();
}
```

2.4 Summary

In this chapter, we present an overview of the hardware and software utilized as a basis for the contributions of this thesis. The hardware platform consists of multiple tiles that are connected by a Æthereal NoC. The processor tile is the central element in the system. The main responsibility of a processor tile is executing applications. Each processor tile runs an instance of CompOSe, independently of the other processor tiles in the system. All task communication and synchronization is using C-HEAP protocol. The core of CompOSe is a functional loop which consists of two major parts, the Operating System slot and the task slot. The Operating System slot is responsible for saving the context of the previous task, to schedule a new application and along with its new task. In the following chapter, we discuss about advantages and limitations of different task scheduling strategies and propose a new task scheduler strategy.

Composable Task Scheduling

3

In this chapter, we address the problem regarding the location at which the scheduling algorithm is executed. Two potential solutions are operating system(OS)-space task scheduling and application-space task scheduling, as presented in Figure 1.3.

The structure of this chapter is as follows. We start by introducing the related work in Section 3.1. Next, Section 3.2 discusses the advantages and limitations of both task scheduling strategies. Based on these advantages and limitations, we propose a new task scheduler which is a mixed of OS-space and application-space task scheduling. Section 3.3 details the implementation of our proposed task scheduler. Finally, different task scheduling algorithms, which are currently employed in our task scheduler, are described in Section 3.4 and continue the summary is given in Section 3.5.

3.1 Related Work

Goyal et al. [10] first proposed a hierarchical scheduling framework to support different scheduling algorithms for different application classes in a multimedia system. In hierarchical scheduling framework, multiple applications can run at the same time on the system while guaranteeing independent execution of these applications. The hierarchical scheduling framework can be presented as a tree of nodes. Each node represents an application with its own scheduler for its internal workloads (e.g., tasks). Resource (e.g., processor usage) is allocated from the parent node to its children nodes, as shown in Figure 3.1.

Over the years, there has been a growing attention to hierarchical scheduling framework of real-time systems. A number of researches have used hierarchical scheduling techniques to create flexible real-time systems. Deng and Liu [3] presented a two-level real-time scheduling framework in scheduling real-time applications and non real-time applications for open systems, where the schedulability of real-time applications can be validated independently of other applications. In the open systems, during the runtime, the user may request the start of some applications whose behavior has not been analyzed together with current executing application. Kuo and Li [17] proposed exact schedulability analysis techniques for such a two-level framework with the fixed priority scheduler as global OS scheduler. Lipari and Baruah [20] presented exact schedulability analysis techniques for two-level framework with EDF based global OS scheduler.

Most of researches in hierarchical scheduling framework focus on the application-level scheduling or the schedulability when a new application enters the system. In this thesis, we discuss about the design and implementation of task-level scheduling in hierarchical scheduling framework.

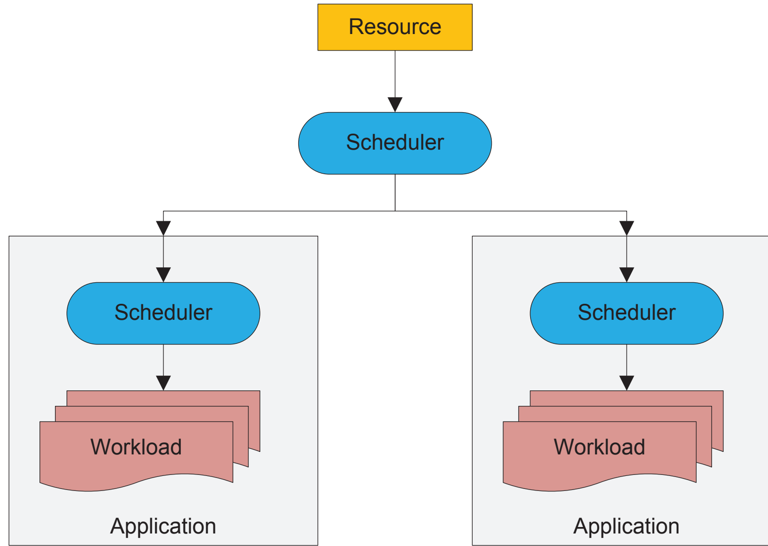

Figure 3.1: Hierarchical Scheduler Framework

3.2 OS-Space and Application-Space Task Scheduling

In the previous work [25], the platform employed a preemptive two level hierarchical scheduling framework which is able to execute a mix of real-time and non real-time applications, each scheduled according to its suitable policy. In order to guarantee the composability and predictability of the platform, the application scheduler uses TDM scheduling algorithm. The task of application can be scheduled by any scheduling algorithm. The application and task scheduler takes local decisions, and is not aligned or synchronized with any other scheduler in the system. Thus, we do not need any communication between schedulers on different processors.

Previous work [25] [12] implements the OS-space task scheduling strategy. The advantage of this strategy is that it can schedule different types of applications, e.g., time-driven, event-driven and data-driven. In OS task scheduling, the task scheduling decisions are made at specific time instance, OS periods, independent of events such as task finish. A hardware timer is used to generate the interrupt at specific time instance to invoke the scheduler. This implies that, even if a task finishes before the end of its slot, the OS will not schedule another task in that slot. As a result, the OS task scheduling might cause lots of idle CPU processing time which leads to potential performance degradation, as presented in Figure 3.2.

On the other hand, in application-space task scheduling strategy, the task scheduler is invoked immediately after the task finishes an iteration, as presented in Figure 3.3. It leads to a potentially better task slots utilization. Thus, we can achieve better application performance, finish time and throughput.

Although, application-space task scheduling strategy can give better application performance, this strategy alone still cannot suffice to all applications. Compared to OS-space task scheduling, application-space task scheduling has following limitations:

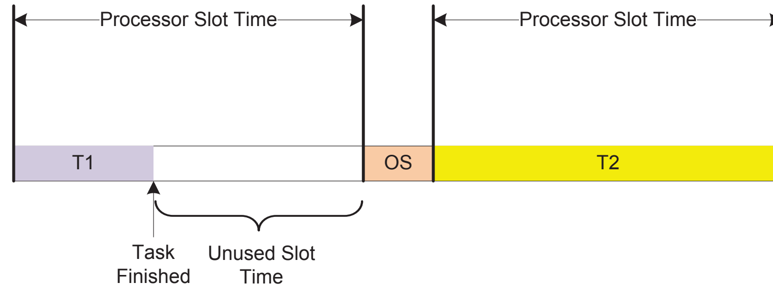

Figure 3.2: OS-Space Task Scheduling

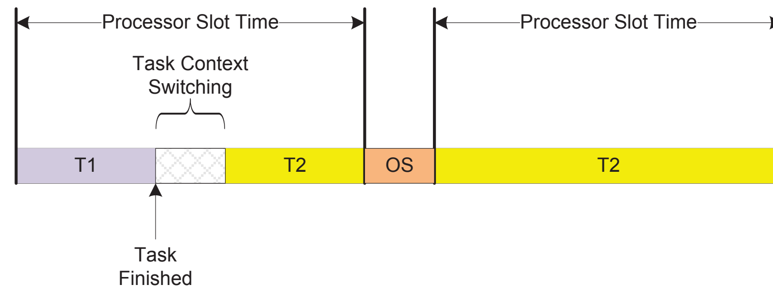

Figure 3.3: Application-Space Task Scheduling

- Unable to schedule time-driven applications because the application does not have exclusive access to a timer and an interrupt.
- Unable to support all types of task scheduling, e.g., preemptive task scheduling algorithms.

Thus, we propose a new task scheduler which allows both OS-space and application-space task scheduling in the same system. In this manner, our new task scheduler can service different types of applications. For example, time-driven applications can use OS task scheduling strategy while data-driven applications can use application-space task scheduling strategy.

3.3 Task Scheduler Implementation

In this section, we describe the implementation of our new task scheduler in details. The structure of this chapter is as follows. We start by introducing the data structure in Section 3.3.1. Next, the functionality of task slot is described in Section 3.3.2. Then, Section 3.3.3 presents the functionality of OS slot.

3.3.1 Data Structure

In order to implement the new task scheduler, we first need to modify the structure of ACB (Application Control Block). We add one more variable, namely *scheduling_strategy*, to the ACB to store which task scheduling strategy the application wants

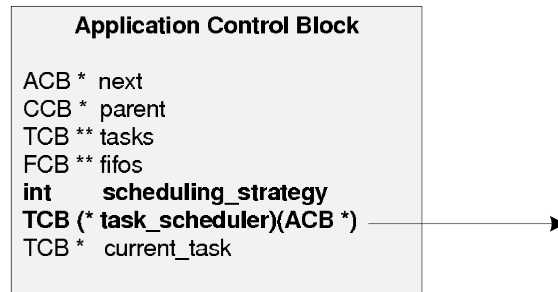

Figure 3.4: New ACB Data Structure

to use. If the value of this variable is 0, application uses OS-space task scheduling strategy. Otherwise, application uses application-space task scheduling strategy if the value of this variable is 1. New ACB data structure is presented in Figure 3.4. In order to initialize the value for *scheduling_strategy* variable, we add one more parameter to the function os_add_application(...,...,int scheduling_strategy, ..., ...). Thus, user can decide which task scheduling strategy for each application during the initialization time before executing applications together with their tasks.

In the prior work, ACB has a function pointer thus allowing to define different task schedulers for different applications. The functionality of this function pointer is used to call the task scheduling algorithm function to select next executable task. In this work, we still keep this function pointer. However, we change the implementation of task scheduling algorithm functions, e.g., round-robin task scheduling function, TDM task scheduling function. Instead of changing the current_task variable inside these functions, new functions will send next executable task as a return value. The differences between old and new task scheduling functions are presented in Listing 3.1. The reason why we need to modify the structure of this function will be discussed later.

Listing 3.1: Differences between Old and New Task Scheduling Function

```
void old_task_scheduling_function(ACB * app){
  app->current_task = find_next_executable_task();
}


TCB * new_task_scheduling_function(ACB * app){
  return find_next_executable_task();
}
```

3.3.2 Task Slot Functionality

In this section, we describe new functionality of task slot. The new functional loop of task slot is illustrated in Figure 3.5.

As the previous implementation, at the starting of the task slot, it checks the input FIFOS and copies tokens from input FIFOs to local buffer. Next, the selected task is executed. If the interrupt comes while the task is executing, the task state changes to preempted and the context of the interrupted task is saved on its stack.

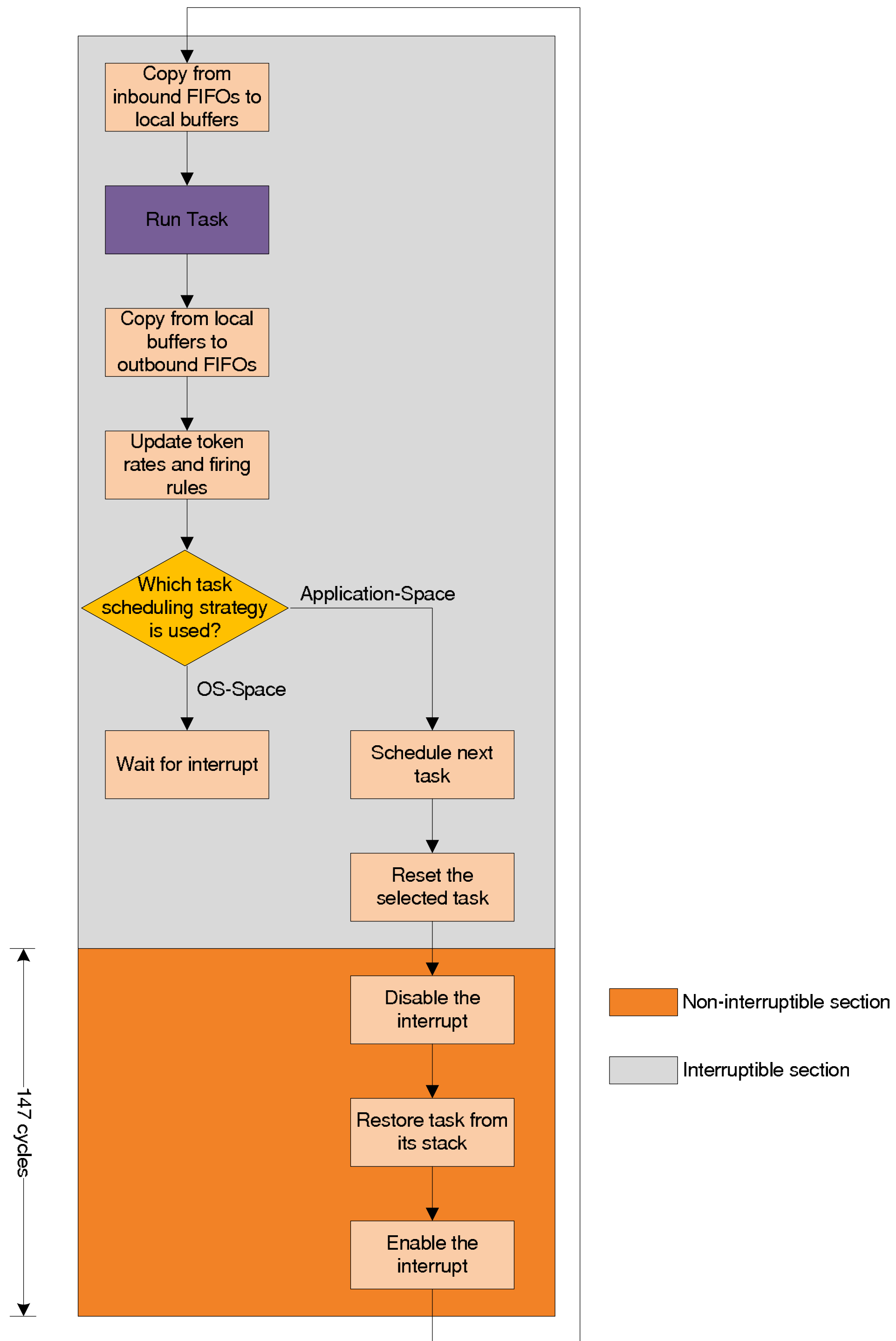

Figure 3.5: New Task Slot Functional Flowchart

Otherwise, if the task finishes an iteration before the timer interrupt, it checks which task scheduling strategy the current running application uses. If the current running application uses the OS-space task scheduling strategy, a wait loop is used to wait for the next timer interrupt.

Otherwise, if the application-space task scheduling strategy is used, the task scheduler is called to select the next task to run. Next, the selected task is reset to its original state and the task context switch is executed to restore task context from its stack. The interrupt is enabled at the end of task context switch. The pseudo-code that demonstrates the functionality of task slot is presented in Listing 3.2. The function os_execute_task is called at the beginning of task slot to start application time. Note that, we present task_contxtsw function as C code in Listing 3.2 in order to understand easily. In our platform, this function is implemented using assembly to change the content of registers.

Listing 3.2: Task Slot Functionality Pseudo-code

```
void os_execute_task(TCB * task){
  //Not execute the idle task, only infinite loop
  if (task->id != 0)
  {
    //read input tokens
    ...
    //execute task
    ...
    //write output tokens
    ...
  }

  curr_app = task->parent; //get current application

  if (curr_app->scheduling_strategy == 1)
  {
    //Application uses application-space task scheduling
    ...
    //schedule next task
    do {
      selected_task = curr_app->task_scheduler(curr_app);
    } while (selected_task->id == 0); //do not execute idle task

    disable_interrupts(); //disable interrupt
    os_reset_task(selected_task);
    //task context switch
    curr_app->current_task = selected_task;
    task_contxtsw();
  } else {
    //Application uses os-space task scheduling

    //infinite while loop to wait for next interrupt
    while(1);
  }
}

void task_contxtsw()
{
  //load context of new task
```

```
    ...
    enable_interrupts(); //enable interrupt
    os_execute_task(curr_app->current_task);
}
```

As seen in Listing 3.2, if the application-space task scheduling strategy is used, we need to disable interrupts before doing task context switch. In the task context switch, we need to update the variable current_task of ACB data structure. Thus, if we use the old task scheduling function where the variable current_task is modified inside the function, we need to disable interrupt before running task scheduler. In order to keep the minimum time when interrupt is disabled, we need to use new task scheduling function.

Note that, the task context switch happening at task slot takes 147 cycles to swap out the current task and load the selected task to the system. It leads to the delay on the OS unit because the timer interrupt can come at the time when task context switch is happening at task slot. In order to guarantee the composability of our platform, we added this time to the worst case execution time of the OS slot. And the processor after the operating system execution is halt up to its worst-case duration.

Similar to the prior work, if the timer interrupt comes while the task is executing, the task state changes to preempted and the context of the preempted task is saved on its stack. Otherwise, if the task returns before the timer interrupt, its state changes to idle (iteration finished). The details of task state transition is illustrated in Figure 3.6. The initial state of a task is IDLE. If the task is selected to run, its state changes to READ. During this time, the task reads the tokens from input FIFOs. Next, the task is executed and its state changes to RUN. After the task finishes an iteration, its state changes to WRITE where the task write tokes to output FIFOs. In the new task state transition, we introduce a new state, namely FINISH. This state is only used when application employs application-space task scheduling algorithm. In this state, task finishes its iteration execution and calls task scheduler to select next executable task. Finally, the task comes back to IDLE state. If the interrupt comes during the task in READ, RUN or WRITE state, its state changes to PREEMPTED READ, PREEMTED RUN and PREEMMTED WRITE state, respectively.

3.3.3 OS Slot Functionality

The new functional loop of operating system unit is illustrated in Figure 3.7. Similar to previous implementation, an interrupt from the system timer marks the start of the new

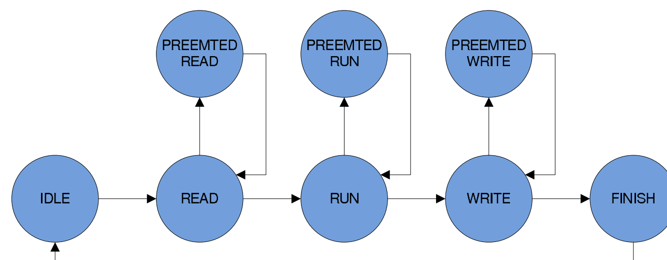

Figure 3.6: New Task State Transition

cycle. In _interrupt_handler(), the context of the interrupted task is saved on its stack. Next, the application-scheduler selects an application using the defined scheduler in CCB. The current_app pointer in CCB is updated to point out the scheduled application.

In the next step, the task scheduling strategy of the current running application is checked. If the current running application uses the OS-space task scheduling, the OS task scheduler is run to select the next task. Otherwise, if the application-space task scheduling strategy is used, the OS task scheduler selects the current preempted task of current running application. Then, the task context switch happens to restore the context of selected task from its stack. In our platform, function _interrup_handler is implemented using Assembly. In order to make its functionality understand easily, we present this function in C pseudo-code, as presented in Listing 3.3.

Listing 3.3: OS Slot Functionality Pseudo-code

```
void _interrupt_handler(){
  //Save state of current running task to its stack
  ...

  //Call application scheduler
  //Variable current_app is modified at the end of this function
  system->application_scheduler(system);

  curr_app = system->current_app;

  //Check scheduling strategy
  if (curr_app->scheduling_strategy == 0)
  {
    //OS-space task scheduling strategy is used
    //Call task scheduler to select next task
    curr_app->current_task = curr_app->task_scheduler(curr_app);
  } else {
    //Application-space task scheduling strategy is used
    //Select current preempted task
    curr_app->current_task = curr_app->current_task;
  }

  task_contxtsw(); //task context switch
}
```

3.4 Task Scheduler Algorithms

In our platform, an application can potentially use any type of scheduling techniques to schedule its task. Currently, there are three built-in scheduling algorithms for task scheduler: TDM, round-robin and static-order.

In TDM task scheduling algorithm, there is a list which stores the order of task. The corresponding task is selected to run in each time slot. If the corresponding task cannot run in the given time slot, the time slot is left to be empty.

Round-robin scheduling algorithm is a fair scheduling algorithm which schedules the tasks according to the task order which is stored in the system. The different between

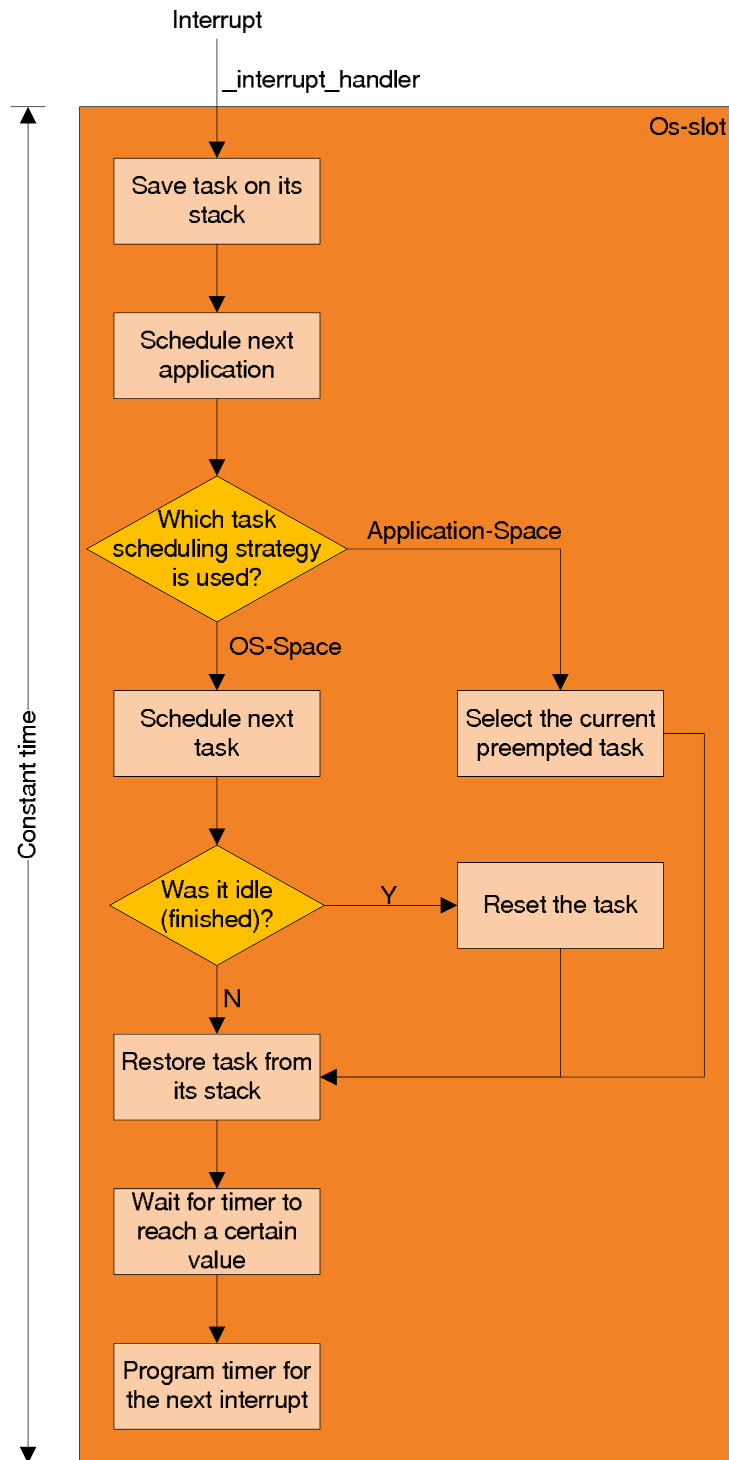

Figure 3.7: New OS Slot Functional Flowchart

the round-robin and TDM scheduling algorithm is that if the task cannot execute, round-robin scheduling will try to find other executable task.

Static-order scheduling algorithm is an algorithm in which we can specify the order of executable task. The main different of this scheduling compared to TDM scheduling algorithm is that tasks cannot switch to others until they finish an iteration. Compared to round-robin scheduling algorithm, the task order in static-order scheduling algorithm is fixed and this algorithm reduces the overhead for task scheduler to schedule the next task because the task order is known.

Round-robin scheduling algorithm and Static-order scheduling algorithm can be used in both OS-space and application-space task scheduling strategy. TDM scheduling algorithm can only be used in OS-space task scheduling because it needs fixed-size time slots. Thus, without timer interrupt, the application-space task scheduling strategy cannot schedule tasks.

3.5 Summary

In this chapter, we first discuss related work. Then, the advantages and limitations of two task scheduling strategies are presented. Next, the implementation of our new task scheduler is described in details. Based on these advantages and limitations, we propose a new task scheduler which is a mixed of OS-space and application-space task scheduling. By combining two scheduling strategies, our task scheduler can service different types of applications. For example, time-driven applications can use OS task scheduling while data-driven applications can use application-space task scheduling. Finally, three different scheduling algorithms for task scheduler were discussed: TDM, round-robin and static-order scheduling algorithms. TDM scheduling algorithm can only be used in OS task scheduling, while round-robin and static-order scheduling algorithm can be used in both OS-space and application-space task scheduling.

Continuously, the quantitative investigation of our task scheduler is presented in Chapter 5. Before that, in Chapter 4, we introduce the application benchmark which is used to design experiments.

4 Embedded Benchmark Applications

In this work, we focus on data-driven, e.g. streaming, applications. Thus, we first examine a set of existing streaming benchmark suites consisting of streaming applications and evaluate the difficulties in order to port existing benchmarks on our platform. Next, we implement a embedded benchmark suite consisting of two streaming applications and a synthetic application. Two streaming applications are H.264 decoder and JPEG decoder. Finally, we map these applications on our platform which consists of two processor tiles.

The structure of this chapter is as follows. Next section discusses the compatibility of existing embedded benchmark suites with our platform. Then the implementation of our embedded benchmark applications is presented in the section 4.2. Finally, the summary is given in section 4.3.

4.1 Portability of Existing Benchmarks

Embedded system often has tight constraint on resources, e.g., memory. In addition to, the performance is not the most important factor in the embedded domain. Other factors such as power, parallelism ,i.e. also have to be considered. Thus, the general-purpose computer benchmark such as SPEC [26], Drystone, Whetstone cannot be used in embedded benchmark domain. In this section, we examine a set of existing embedded benchmark suites for the streaming embedded system, and discuss their compatibility with our platform.

EDN Embedded Microprocessor Benchmark Consortium (EEMBC) [5] The EEMBC consortium made some efforts to characterize embedded domain. EEMBC benchmarks produced a set of suite targeting in different embedded market such as automotive, consumer, networking, office automation, telecoms, and multi-core. The limitation of the EEMBC benchmarks is that it requires high cost of joining the consortium in order to use these benchmarks.

The EEMBC multi-core benchmark applications are parallelized using functions which are similar to pthread functions. EEMBC benchmark applications are parallelized using data-parallel approach to make them easily port on different platforms. The EEMBC multi-core benchmark applications are paralleled using the pthread library, which has the API task creation functions similar to our API functions. However, it also uses some other pthread library functions, e.g., mutex, condition, which are currently unsupported in our platform.

In this work, we made initial porting efforts. The result shows that we can remove these pthread functions, e.g., mutex, condition, by creating FIFOs to synchronize between tasks.

MediaBench [19] The purpose of MediaBench is to measure the performance of multimedia and communications systems. MediaBench contains 8 individual benchmark suites. The image and video media benchmark suite is composed of six image/video compression standards: JPEG, JPEG-2000, H.263, H.264, MPEG-2, and MPEG-4. The benchmark applications are: cjpeg, djpeg, h263dec, h263enc, h264dec, h264enc, jpg2000dec, jpg2000enc, mpeg2dec, mpeg2enc, mpeg4dec, and mpeg4enc. MediaBench benchmark applications are not parallelized, thus, MediaBench is not suitable for our multi-core platform where each processor runs an independent instance of operating system, without any knowledge of other processors.

Embedded System Synthesis Benchmarks Suite (E3S) [4] E3S follows the organization of the EEMBC benchmarks. There are five application suites: automotive/industrial, consumer, networking, office automation, and telecommunications. This benchmark was designed for use in automated system-level allocation, assignment, and scheduling research. Based on the description of processors which is provided by the vendors, the benchmark will find the scheduling and resource allocation. Then, it will estimate execution times and power consumption. Because E3S benchmark does not run on the real platform, the result might not be accurate.

MiBench [11] MiBench has many similarities to the EEMBC benchmark suite. However, MiBench is composed of freely available source code. MiBench consists of six categories including: Automotive and Industrial, Control, Network, Security, Consumer Devices, Office Automation, and Telecommunications. MiBench Benchmarks provide a small and large data set. The small data set represents a light-weight, useful embedded application, while the large data set provides a more stressful, real-world application. MiBench Benchmarks evaluate the instruction level parallelism of the system by measuring the instruction per cycle (IPC). Similarly to MediaBench, MiBench applications are not parallelized. Thus, MiBench is not suitable for our multi-core platform

ALPBench [18] ALPBench is a new multimedia benchmark suite. ALPBench consists of parallelized complex media applications gathered from various sources, and modified to expose thread-level and data-level parallelism using POSIX threads and Intel SSE2 instructions, respectively. The applications are: Speech recognition, Face recognition, Ray Tracer, MPEG-2 encoder and MPEG-2 decoder. With the thread-level parallelism, ALPBench applications are useful to exploit the performance on the multi-core system. However, our platform does not support the data-level parallelism instructions (Intel SSE2 instruction). Therefore, it is difficult to port the ALPBench to run on our platform.

PARSEC [2] PARSEC is a benchmark suite for chip multiprocessors (CMPs) composed of multithreaded programs. The current version of the suite consists of 9 applications and 3 kernels which were chosen from a wide range of application domains such as computer vision, video encoding, financial analytics, animation physics and media processing. Each application has been parallelized by using data parallel or pipeline model. The PARSEC benchmark applications are paralleled using the pthread library

Benchmark	Parallelization Model	Difficulties
EEMBC	data-parallel	mutex and condition pthread functions
MediaBench	No	Non multi-threads application
E3S	No	Non multi-threads application
MiBench	No	Non multi-threads application
ALPBench	data-parallel/pipeline	Need data-level parallelism instructions (Intel SSE2 instruction) support
PARSEC	data-parallel/pipeline	Need large memory for applications and libraries

Table 4.1: Difficulties to Port the Existing Benchmarks on our platform

and OpenMP library, which has the API task creation functions similar to our API functions. However, the PARSEC benchmark applications were implemented based on external libraries, e.g., GLib, GNU Scientific Library (GSL) ,i.e., it requires lots of memory to port the benchmark applications together with these libraries. Because the instruction memory and data memory on each processing unit is only at 128KB, it is impossible to port the PARSEC benchmark applications on our platform.

4.2 Embedded Application Benchmark Suite Implementation

In the previous section, existing embedded benchmarks were investigated. From the summary of existing benchmarks, as shown in Table 4.1, we can see that there are some difficulties to port the current existing embedded benchmarks on our platform. In this section, we describe about our embedded application benchmark suite. This work focuses on data-driven, e.g. streaming, applications. Thus, the proposed embedded application benchmark suite consisting of two streaming applications and a synthetic application. Two streaming applications are H.264 decoder and JPEG decoder. We map these applications on our platform which consists of two processor tiles. In order to balance workload between two processors, we first execute these application on one processor and measure the execution time of each task iteration. Then we calculate the workload percentage of each task by dividing the total task execution time by the total application execution time. Application execution time is defined as the sum of all its tasks' execution time. The details on implementation and mapping these applications on our platform are presented as follows.

4.2.1 H.264 Decoder Application

H.264/AVC [27] is a standard for video compression/decompression. H.264/AVC codec standard is developed by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC Moving Picture Experts Group (MPEG). The main goal of H.264/AVC standardization is to enhance compression video coding performance and to provide efficient transmission of video through low bandwidth channels such as xDSL or UTMS.

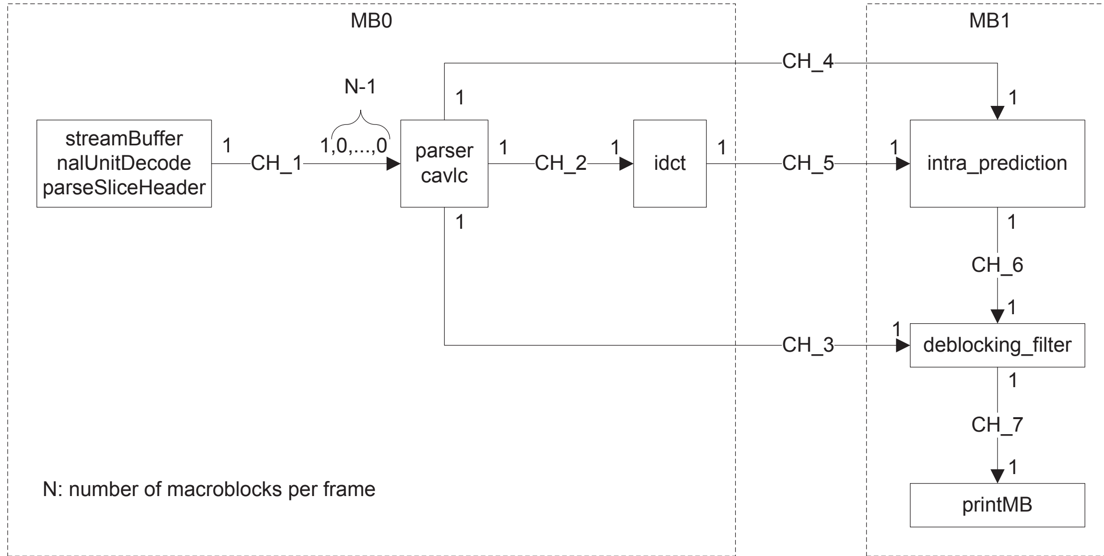

Figure 4.1: Mapping of H.264 Decoder

FIFO	Data Type	Token Size (bytes)
CH_1	CH_1_Struct	5053
CH_2	CH_2_Struct	1644
CH_3	dbParams	40
CH_4	intraParams	88
CH_5	MacroBlock	1536
CH_6	MacroBlock	1536
CH_7	MacroBlock	1536

Table 4.2: H.264 Communication FIFOs

There are several existing H.264 decoder implementations such as ffmpeg, x264, etc. We used the sequential H.264 decoder which was designed by Leiden Embedded Research Center [14] to be easily parallelized using Kahn Process Network [15]. This H.264 decoder implementation only supports intra prediction. Thus, the video input must only consists of I-Macroblocks.

We divided H.264 decoder into six tasks, as presented in Figure 4.1. The first task is responsible for retrieving frame header information. The cavlc and parser task decode various syntax elements from the NAL unit and perform context adaptive variable length decoding. The idct task applies inverse quantization and inverse transform on the residual data. The intra prediction task performs spatial prediction and produces reconstructed picture samples. The deblocking filter task removes blocking artifacts from the reconstructed picture. Finally, the printMB task checks the correctness of output by comparing with the correct result which was stored in memory or using CRC algorithm.

As presented in Figure 4.1, the H.264 decoder consists of seven communication FIFOs. The size of token on each communication FIFO is shown in Table 4.2.

Task	Average Execution Time (cycltes)	Iterations	Workload Percentage
parseSliceHeader	154470	10	0.2%
cavlc + parser	228300	990	32%
idct	173140	990	24%
intra prediction	108480	990	15%
deblocking filter	189470	990	27%
printMB	10250	990	1.8%

Table 4.3: Workload Percentage of H.264 Decoder Tasks

FIFO	Data Type	Token Size (bytes)
CH_01	FBlock	256
CH_02	PBlock	64
CH_03	IMG_Header	128

Table 4.4: JPEG Communication FIFOs

In order to balance the workload on two processor, we first measure the execution time of each task iteration on one processor. Based on the execution time information, we derive the workload percentage of each task, as shown in Table 4.3. From this table, we see that the total workload percentage of the first three tasks is 56.2%. Thus, we decide to map the first three tasks on one processor and the rest on the other processor.

4.2.2 JPEG Decoder Application

JPEG standard is defined by the Joint Photographic Experts Group. This standard defines the encoding and decoding of continues-tone still image. JPEG typically uses lossy compression method. JPEG can achieve 10:1 compression with little perceptible loss in image quality. We used the JPEG decoder implementation, which is originally written by Pierre Guerrier, as a starting point. Firstly, we divided the JPEG decoder into three tasks, as shown in Figure 4.3. The variable length decoding (VLD) task parses the input bit stream and performs the Huffman decoding. The inverse discrete cosine transformation (IDCT) task performs zigzag scan and dequantization operation. The color conversion (CC) task is used to convert from the YCbCr color space to the RGB color space.

As presented in Figure 4.3, JPEG decoder consists of three communication FIFOs. The size of token on each communication FIFO is shown in Table 4.4. At the first iteration, the VLD task parses the input image to retrieve the image header information and send it to CC task through channel CH_3. Based on the received image header information, CC task changes the read token rate of channel CH_2 to N, where N is the number of pixel-space values block (PBlock) for one MCU. After that, VLD task writes one block of frequency-space values (FBlock) to channel CH_1 task per iteration. At each iteration, IDCT task reads one FBlock from channel CH_1 and writes one PBlock to channel CH_2. Finally, CC task reads N PBlocks from channel CH_3 per iteration.

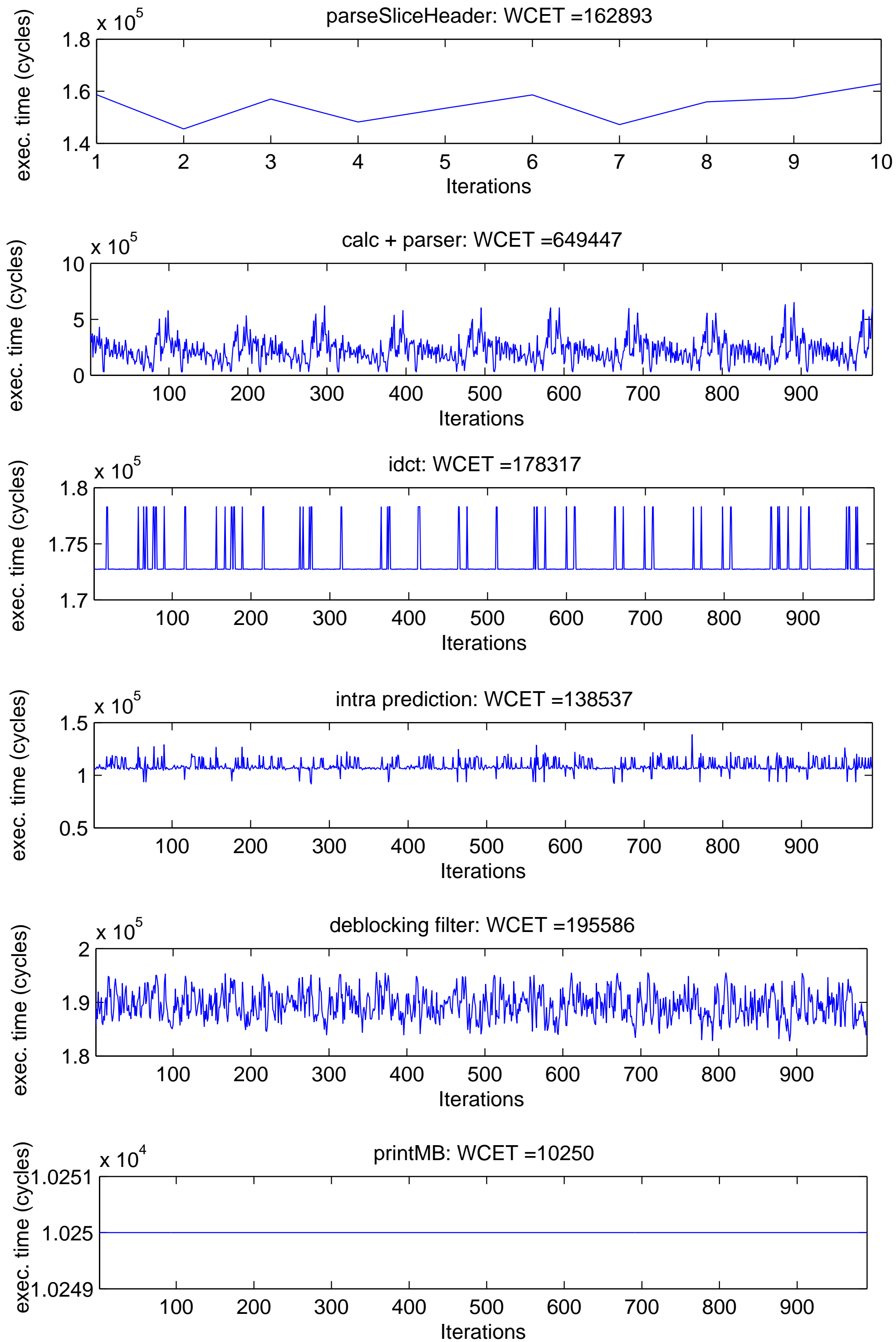

Figure 4.2: Execution Time of Tasks in H.264 Decoder

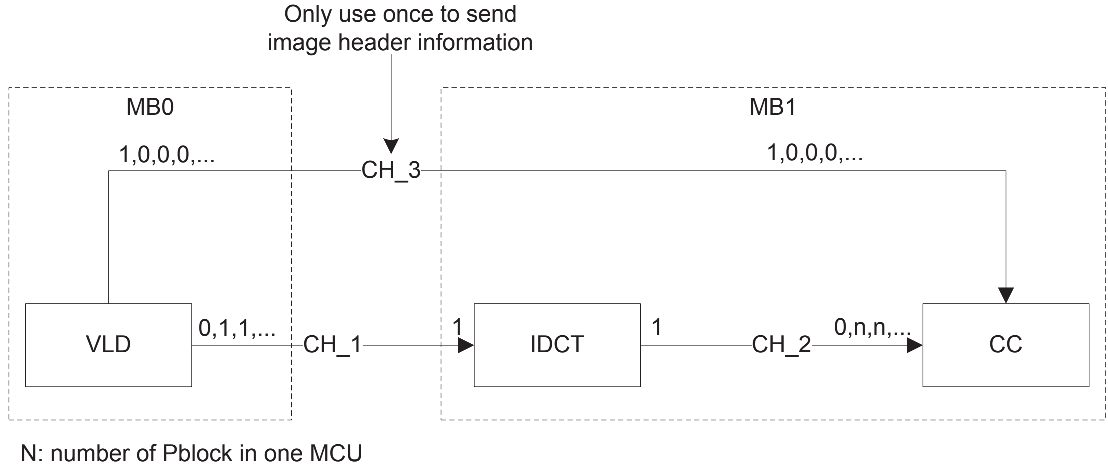

Figure 4.3: Mapping of JPEG Decoder

Task	Average Execution Time (cycltes)	Iterations	Workload Percentage
VLD	13501	480	35%
IDCT	16111	480	41%
CC	28734	160	24%

Table 4.5: Workload Percentage of JPEG Decoder Tasks

Figure 4.4 shows the execution time of each tasks. The workload percentage of each task is shown in Table 4.5. Similar to H.264 decoder mapping, we map VLD task on the first processor while other is on the second processors in order to balance the computation and minimize the communication between two processors.

4.2.3 Synthetic Application

Synthetic applications are typically used to emulate the behavior of different applications. In our implementation, synthetic application consists of five tasks, as shown in Figure 4.5. In latter experiments, we want to investigate the composability of our interconnect and memory. Thus, we want all task communication happen between two different processors. In this manner, producer and consumer task of all communication FIFOs need to be at different processors.

The implementation of a synthetic task is fairly simple. Each task reads one token from the input channel, consume a certain number of cycles, then writes one token to output channel per iteration. The behavior of different kinds of FIFOs is simulated by setting different token size and FIFO size, and changing the token rate of the FIFOs.

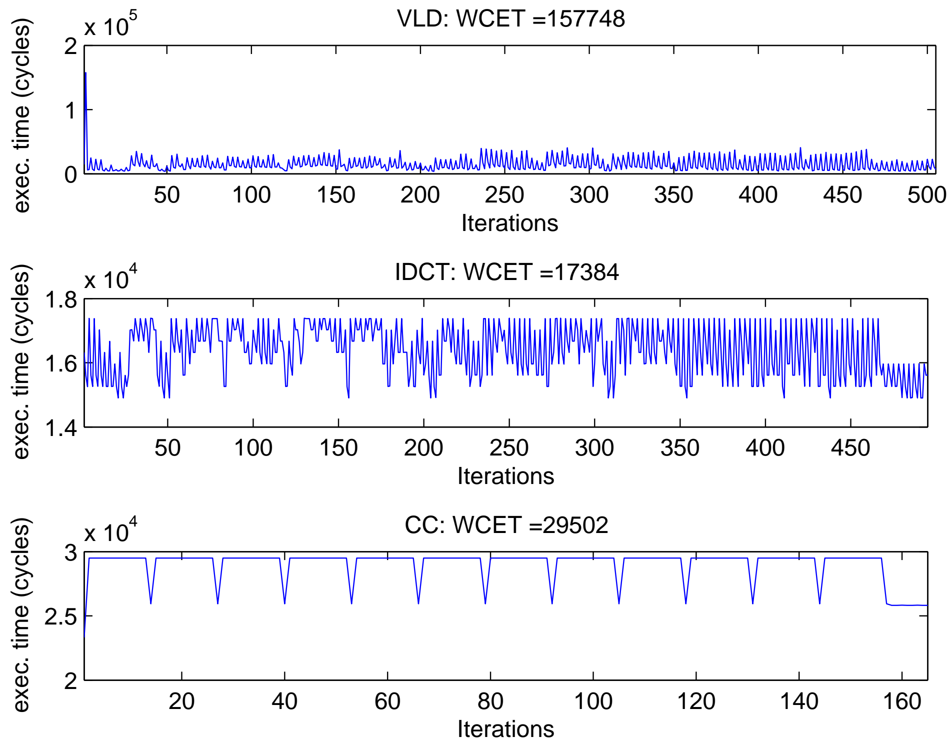

Figure 4.4: Execution Time of Tasks in JPEG Decoder

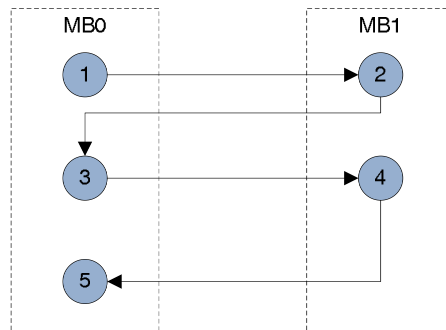

Figure 4.5: Synthetic Application Graph

4.3 Summary

In this chapter, we discussed potential applications which are suitable to investigate composability and application performance of our proposed task scheduler. For this, we first examine existing embedded streaming benchmarks. E3S benchmark applications does not run on the real platform while MiBench and MediaBench benchmark applications are not parallelized, thus, they are not suitable for our multi-core platform. ALPBench

benchmark applications need to use Intel SSE2 instruction which are currently unsupported in our platform. PARSEC benchmark applications use external libraries, thus, it requires large memory to port these libraries on our platform. EEMBC multi-core benchmark applications are parallelized using pthread library. However, the current implementation of our platform does not support some pthread library functions, e.g., mutex_init, cond_signal. Thus, we need to make some significant changes on EEMBC applications or our platform in order to port these application on our platform.

Based on the difficulties to port existing benchmark applications on our platform, we developed an application embedded benchmark suite, which consists of two streaming applications and one synthetic application. Two streaming applications are H.264 decoder and JPEG decoder. Based on these applications, we designed experiments to investigate the composability and the effect on application performance of our task scheduler. The details on these experiments and results are presented in Chapter 5.

5 Experiments and Results

We performed the experiments on an MPSoC with two processor tiles, a monitor tile, all connected by a Æthereal NoC. The detail processor tile configuration is presented in Table 5.1.

In this work, we focus on the composability of our task scheduler and on the application performance when using different task scheduling strategies and algorithms. For all experiments, we ran a workload consisting of a media application and the synthetic application. Media application can be H.264 decoder or JPEG decoder application. The application TDM order is presented in Figure 5.1. The mappings of each application on two processors are presented in Figure 4.1, Figure 4.3 and Figure 4.5.

The structure of this chapter is as follows. We start by investigating the composability of our task scheduler. The result of this experiment is presented in Section 5.1. Next, Section 5.2 compares the performance of media applications when using application-space task scheduling and OS-space task scheduling. Finally, the summary is given in section 5.3.

5.1 Composability Experiments

Two experiments are used to investigate the composability of our platform. In the first experiment, we use different task scheduling strategies for synthetic application. Then, we verify whether the start time and response time of a media application remains constant when synthetic application uses different task scheduling strategies. If the start time and response time of two media applications remain constant, it means that our task scheduler is composable. Note that, start/finish time is the absolute time when a task iteration starts/ends while response time is difference between finish time and start time in each task iteration.

In order to have a complete investigation on the composability of our platform, we do the second experiments to investigate composability for interconnect and memory. For this experiment, we change to token size of the inter-processor FIFO of synthetic

Parameter	Configuration
D-MEM Size	128KB
I-MEM Size	128KB
CMEMIN Size	32KB
CMEMOUT Size	32KB

Table 5.1: Processor Tile Configuration

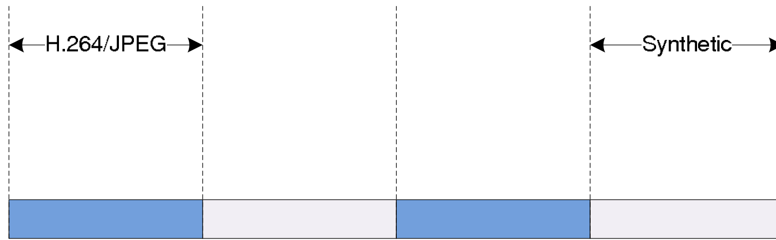

Figure 5.1: Application TDM Order Experiment

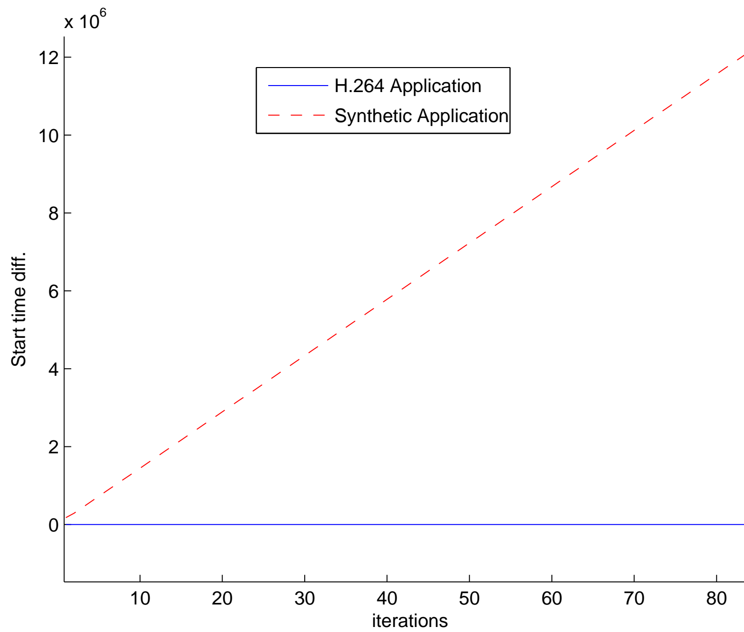

Figure 5.2: Differences in H.264 Start Time - Scheduler Composability Experiment

application. Similarly, we verify whether the start time and response time of a media application remains constant when synthetic application uses different FIFO token size.

For all experiment, we use a task slot duration of 20000. Media applications, e.g, H.264 video decoder and JPEG decoder, use round-robin scheduling algorithm with the application-space task scheduling strategy. The details on these experiments and result are as follows.

5.1.1 Task Scheduler Composability

In the first experiment, we investigate the composability of our task scheduler. This experiment is performed through two steps. We start by running synthetic application using OS-space task scheduling strategy. Then, we run synthetic application using application-space task scheduling strategy.

Figure 5.2 and Figure 5.3 presented the start and response time differences for H.264

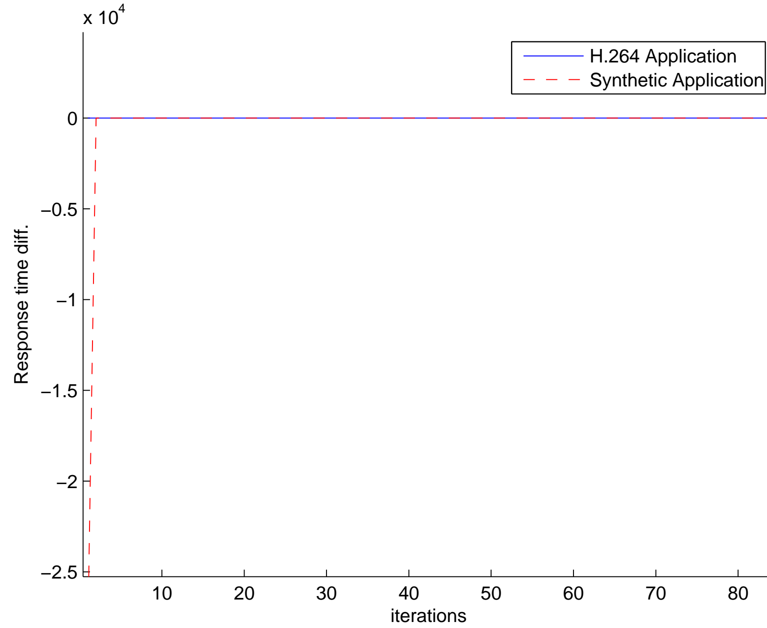

Figure 5.3: Differences in H.264 Response Time - Scheduler Composability Experiment

video decoder application's last task between two cases: synthetic application using OS-space task scheduling strategy and synthetic application using application-space task scheduling strategy. Similarly, the start and response time differences for JPEG decoder application are presented in Figure 5.4 and Figure 5.5. From these figures, we see that the when we change the task scheduling strategy of synthetic application, the behavior of synthetic application changes, e.g., start time, response time. However, differences in start time and response time of our media applications stay at zero, showing no interference. Thus, we can conclude that our task scheduler is composable.

5.1.2 Interconnect, Memory Composability

In the second experiment, we investigate the composability of our interconnect and memory by modyfing the token size of the inter-processor FIFO of synthetic application. We also perform two runs in this experiments. First, we run synthetic application using a token size of 4B. Then, in the second run, synthetic application uses a token size of 16K. Finally, we verify whether the start time and response time of a media application remains constant when synthetic application uses different FIFO token size.

Figure 5.6 and Figure 5.7 presented the start and response time differences for H.264 video decoder application's last task between two cases: token size of 4B and token size of 16K. Similarly, the start and response time differences for JPEG decoder application is presented in Figure 5.8 and Figure 5.9. From these figures, we see that the when we change the token size of inter-processor FIFO of synthetic application, the behavior of synthetic application changes, e.g., start time, response time. However, differences

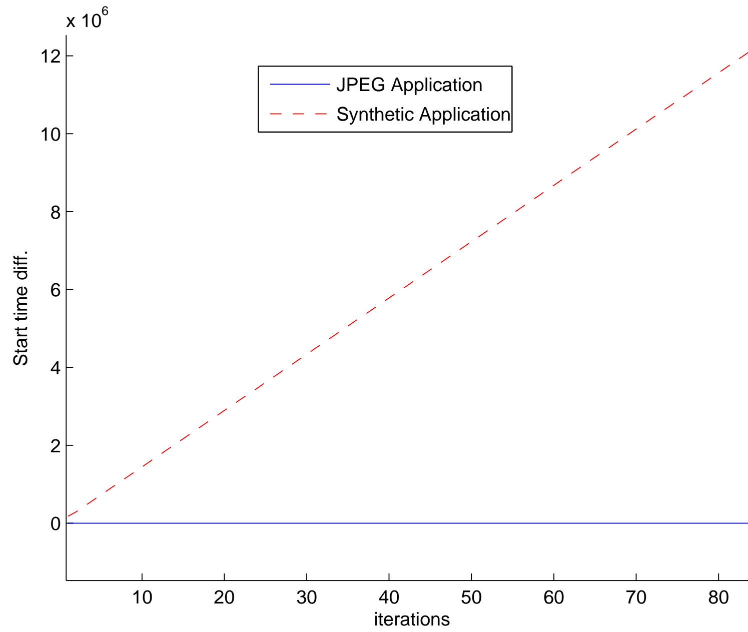

Figure 5.4: Differences in JPEG Start Time - Scheduler Composability Experiment

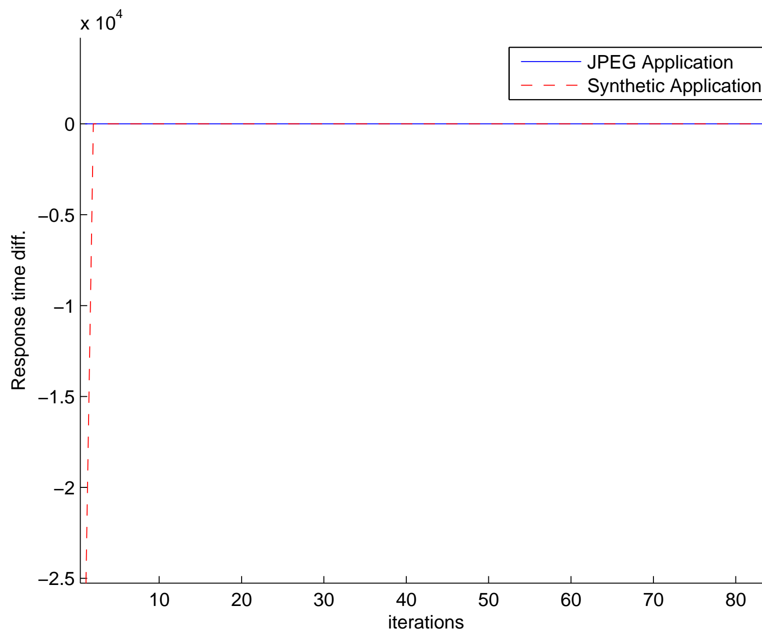

Figure 5.5: Differences in JPEG Response Time - Scheduler Composability Experiment

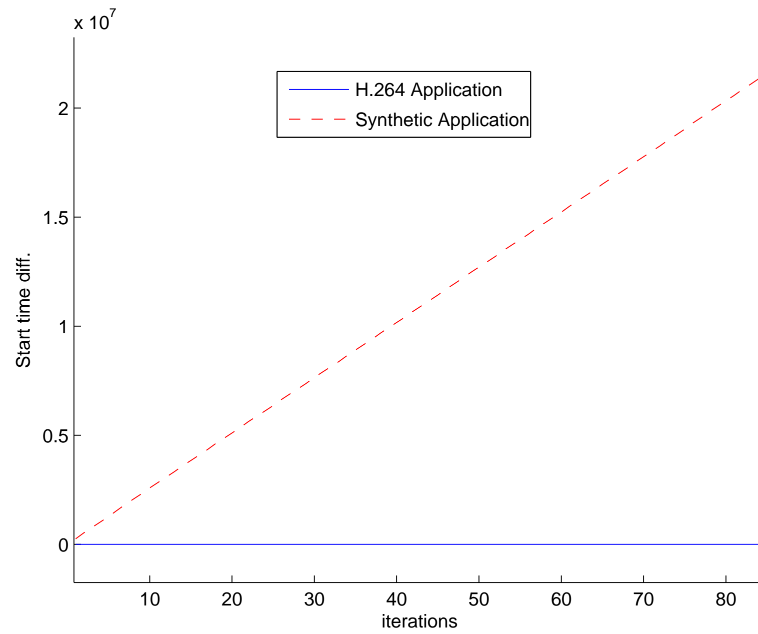

Figure 5.6: Differences in H.264 Start Time - Memory Composability Experiment

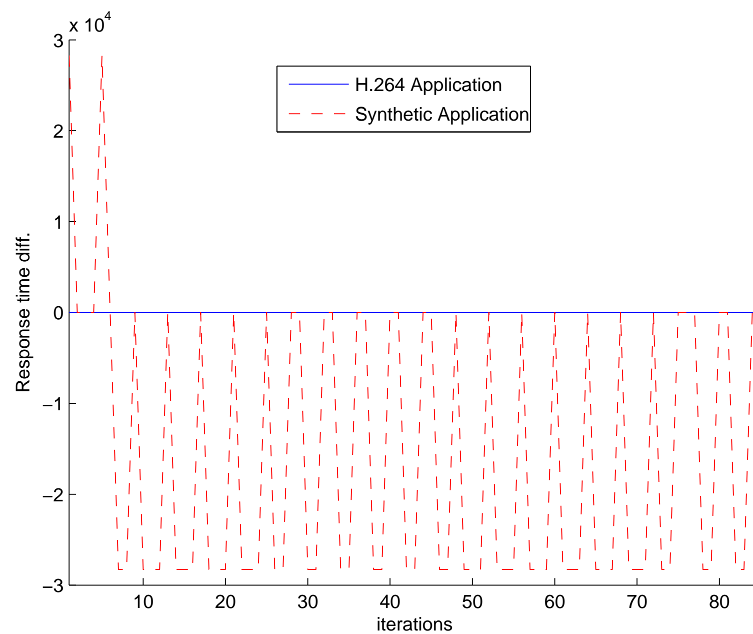

Figure 5.7: Differences in H.264 Response Time - Memory Composability Experiment

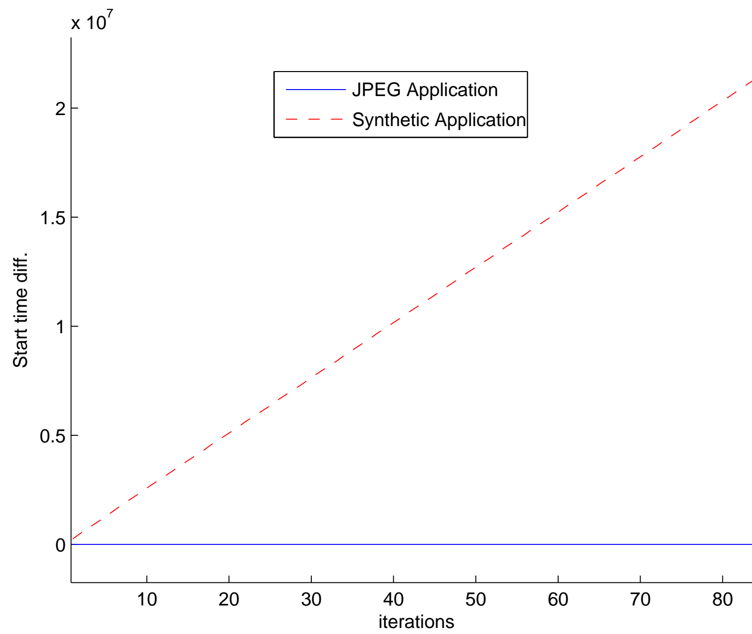

Figure 5.8: Differences in JPEG Start Time - Memory Composability Experiment

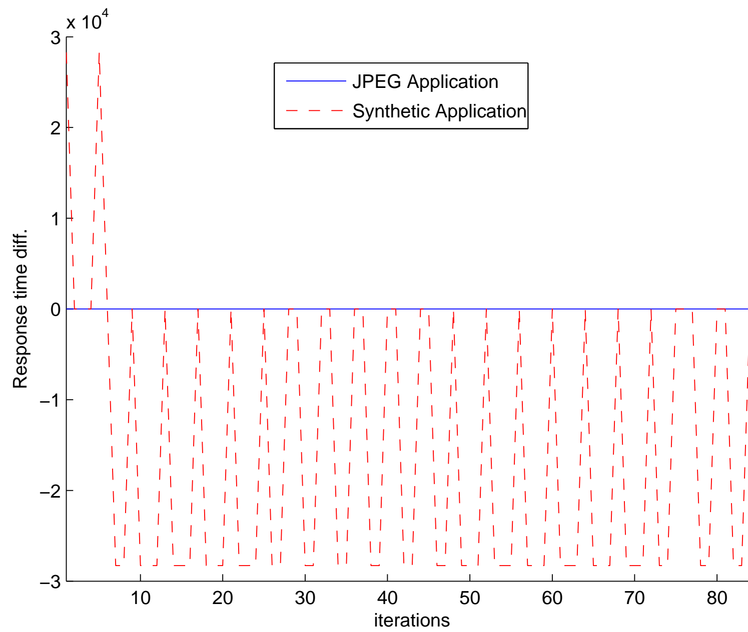

Figure 5.9: Differences in JPEG Response Time - Memory Composability Experiment

in start time and response time of our media applications stay at zero, showing no interference. Thus, we can conclude that our interconnect and memory is composable.

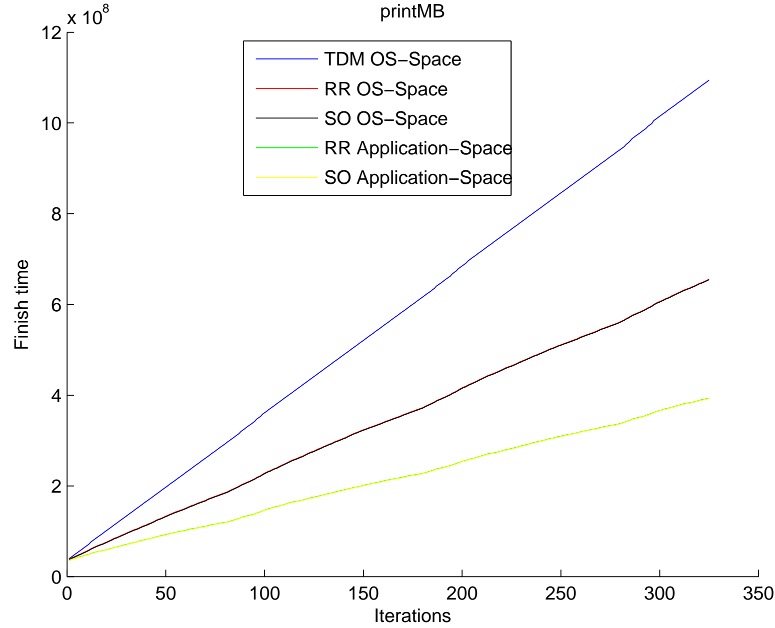

Figure 5.10: H.264 Scheduler Experiment

5.2 Application Performance Experiments

In this section, we analyze the effect of task scheduling on streaming application performance in two experiments. In the first experiment, we investigate the application performance when using different task scheduling algorithms and strategies. In the second experiment, we study the effect of variations of task slot duration on each task scheduling algorithms and strategies.

5.2.1 Application Performance over Multiple Iterations

In this experiment, we look into the effect of using OS-space and application-space task scheduling strategies on streaming application by observing the application finish time for some iterations. We measure the last task finish time in each iterations when using different scheduling model. We used the task slot duration of 200000 for H.264 application and the task slot duration of 20000 is used for JPEG application. The result is presented in Figure 5.10 and Figure 5.11 respectively.

Figure 5.10 and Figure 5.11 show that the application-space task scheduling strategy reduces the finish time of application for all task scheduling algorithms, thus, it gives better performance than OS-space task scheduling strategy.

Figure 5.12 and Figure 5.13 present the improvement in per scheduling policy, i.e., for round-robin and static-order task scheduling algorithms. The result indicates that application-space task scheduling achieves 17% to 40% better performance than using OS-space task scheduling for the JPEG decoder and H.264 decoder, respectively. These graphs show that static-order scheduling algorithm gives nearly same result as round-

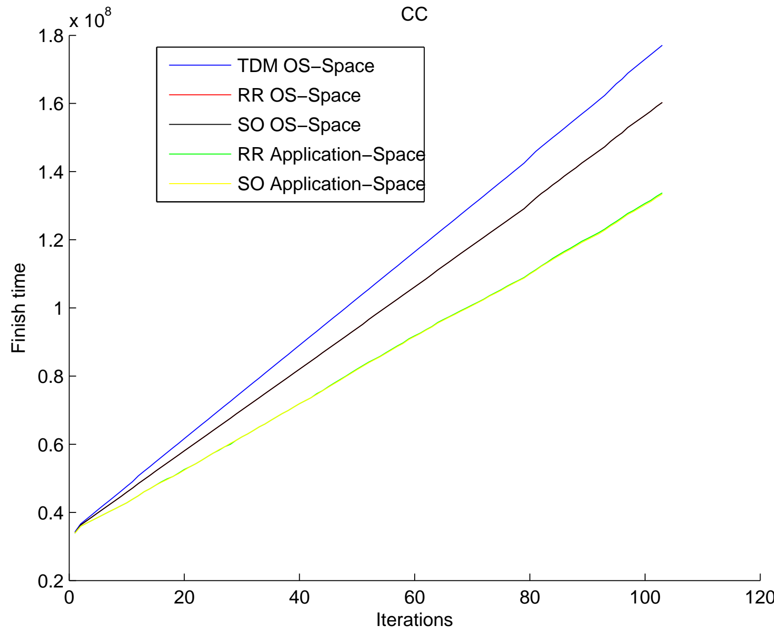

Figure 5.11: JPEG Scheduler Experiment

robin scheduling algorithm. As seen in Figure 4.1 and Figure 4.3, the order of executable tasks in our media applications is always fixed. Thus, even we use round-robin scheduling algorithm, it still schedules tasks with order as as static-order scheduling algorithm.

Moreover, we observe that the application performance improves much more in later iterations. Thus, it means that by using the application-space task scheduling, the application performance can improve, especially when application needs to run for a long period of time. We have this improvement because application-space task scheduling strategy reduces not only internal fragmentation of task slot but also overhead which is caused by OS slots and task slots of other applications.

5.2.2 Task Slot Duration Variations

In this experiment, we study the effect of variations of task slot duration on each task scheduling algorithms and strategies. We run each streaming application with different task slot duration and measure the finish time of last task at a given iteration. H.264 application was run with different task slot duration starting from 20000 to 200000. As seen in Figure 4.2 and Figure 4.4, most of H.264 decoder tasks will finish inside the task slot duration of 200000. Thus, if the task slot duration is larger than 200000, lots of time will be wasted due to internal fragmentation. Similarly, JPEG application was run with different task slot duration starting from 20000 to 50000. The result is presented in Figure 5.14 and Figure 5.15 respectively.

Figure 5.14 and Figure 5.15 show that by choosing the appropriate task slot duration, we can improve the performance of applications. Furthermore, the experiments also show

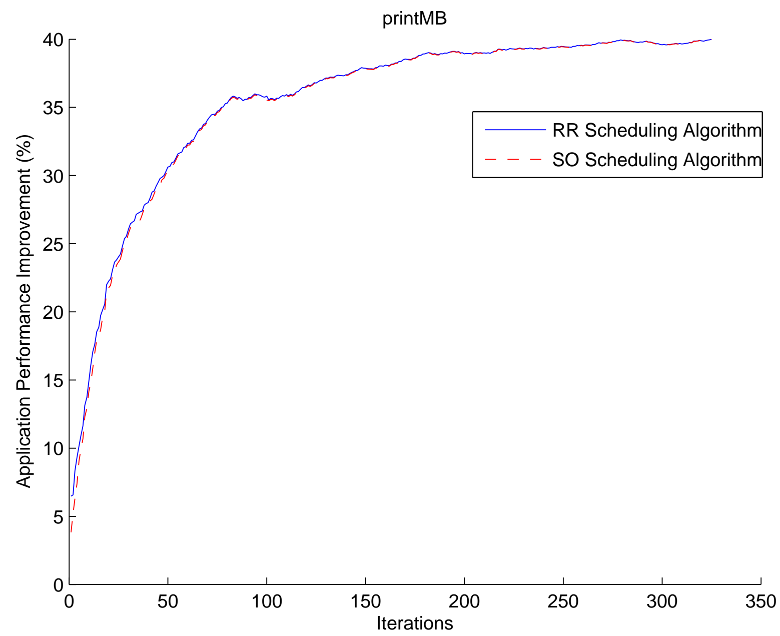

Figure 5.12: H.264 Application Performance Improvement

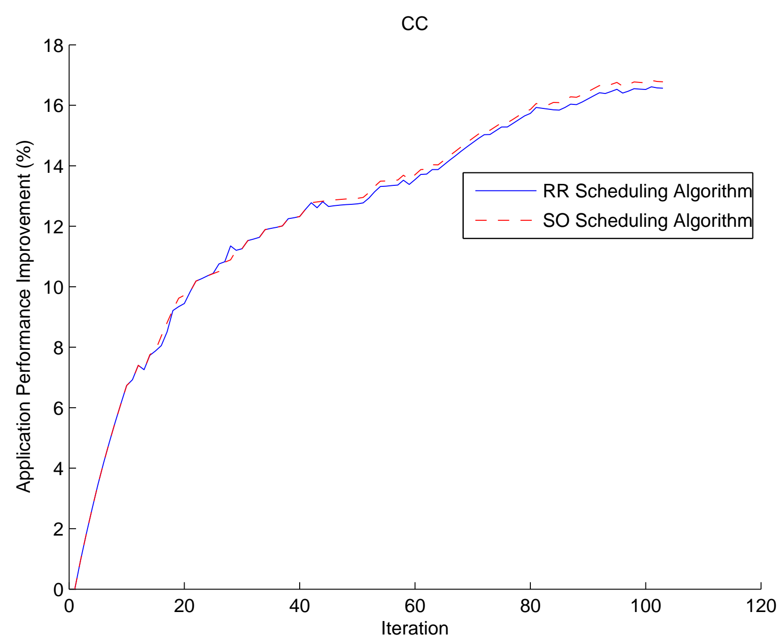

Figure 5.13: JPEG Application Performance Improvement

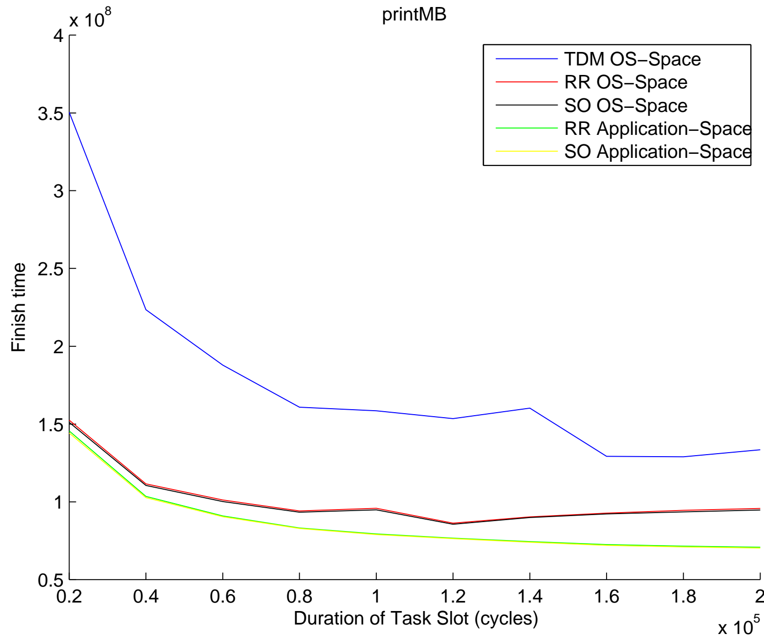

Figure 5.14: H.264 Task Slot Duration Experiment

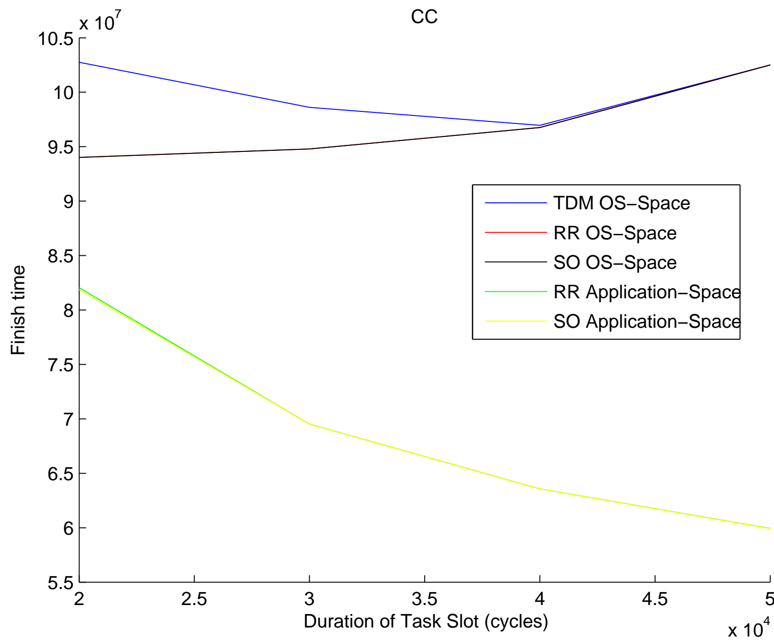

Figure 5.15: JPEG Task Slot Duration Experiment

the different behavior of each task scheduling strategies. For OS-space task scheduling, the performance improves as long as the tasks iterations do not fully fit inside a task slot. After a task slot duration that accommodate entire task iterations, if we increase task slot duration more, the performance reduces because of internal fragmentation. This is not the case of application-space task scheduling where the internal fragmentation is eliminated. Thus, application-space task scheduling increases application performance when using larger task slot duration. Thus, using OS-space task scheduling is more difficult to predict which task slot duration gives best performance than application-space task scheduling, especially when application consists of tasks having a large variation in execution time between each iterations, e.g, H.264 decoder application as presented in Figure 4.2.

5.3 Summary

In this chapter, we discuss about experiments and result. Our experiments indicate that our task scheduler and our platform are composable. They also indicate that application-space task scheduling strategy achieves better application performance than OS-space task scheduling strategy, as expected. Moreover, applications using application-space task scheduling experience more improvement on application performance when they execute more iterations. Finally, these experiments demonstrate the effect of task slot duration on performance of application when using with different task scheduling algorithms and strategies. For OS-space task scheduling, the performance improves as long as the tasks iterations do not fully fit inside a task slot. After a task slot duration that accommodate entire task iterations, if we increase task slot duration more, the performance reduces because of internal fragmentation. This is not the case of application-space task scheduling where the internal fragmentation is eliminated. Thus, application-space task scheduling increases application performance when using larger task slot duration.

6 Conclusions and Future Work

This chapter concludes the thesis and presents a view on future work. The structure of this chapter is as follows. The conclusion is given in Section 6.1, while future work is given in Section 6.2.

6.1 Conclusions

In this thesis, we build upon the existing platform [25], [12] and we further investigate task scheduling. Prior work employs a preemptive two level hierarchical scheduling framework which is able to execute a mix of real-time and non real-time applications, each scheduled according to its suitable policy. The Operating System (OS) schedules applications and task at fixed duration task slots. First, the OS determines which application owns the next slot following a strict, preemptive Time Division Multiplexing (TDM) policy, and then it picks and schedules a task of that application. As scheduling decisions are taken exclusively at slots borders, when a task finishes before its slot depletes, the time left is wasted. This may result in low processor utilization for streaming applications for which the execution of a task may start after its predecessor tasks have finished.

In this work we propose a new task scheduling strategy, namely application-space task scheduling that eliminates wasted slot time. We leave the fixed duration slots and the application TDM scheduler unaltered, to preserve composability, but the application invokes the task scheduler immediately after each task finish, inside its slot. As the application-space task scheduling strategy alone may not support all types of task scheduling, e.g., preemptive, we propose to combine OS-space and application-space scheduling on the same processor.

To experimentally investigate the composability and performance of our scheme we survey existing benchmarks for the embedded domain, and build a workload consisting of two meda applications and a synthetic application. Two media applications are H.264 video decoder and JPEG image decoder. We executed these applications on an MPSoC with two processor tiles, a monitor tile, all connected by a Æthereal NoC. Our experiments indicate that mixing application-space and OS-space task schedulers is composable. The application-space task scheduling achieve 40% and 17% better performance than using OS-space task scheduling for H.264 decoder and JPEG decoder application respectively. Furthermore, applications using application-space task scheduling get more improvement on application performance when they execute more iterations. Finally, these experiments demonstrate the effect of task slot duration on performance of application when using with different task scheduling algorithms and strategies. For OS-space task scheduling, the performance improves as long as the tasks iterations do not fully fit inside a task slot. After a task slot duration that accommodate entire task iterations, if

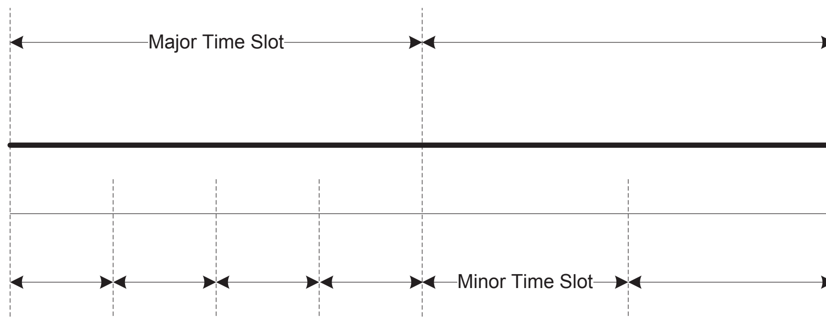

Figure 6.1: Hierarchical Slot

we increase task slot duration more, the performance reduces because of internal fragmentation. This is not the case of application-space task scheduling where the internal fragmentation is eliminated. Thus, application-space task scheduling increases application performance when using larger task slot duration.

6.2 Future Work

In this section, a number of improvements and useful features are presented as the possible future research areas.

6.2.1 Embedded Application Benchmark Suite

It is important to have a strong benchmark suite that exhibits typical embedded system application behavior. The current embedded application benchmark suite only contains of three applications. In the future, we can extend our benchmark applications by writing a tool to port all EEMBC multi-core applications on our platform. EEMBC multi-core benchmark applications use some pthread library functions which are currently not supported on our platform, e.g, mutex, cond. In other to solve this problem, there are two possible solutions: implement these functions in our platform or remove these functions by rewriting the applications.

6.2.2 Hierarchical Slot

Currently, task slot of all applications running on the platform has to be at the same duration. As presented in Figure 5.14 and Figure 5.15, applications might behave better with different task slot duration. An idea to show this problem is to create a hierarchical slot in our platform. The system time will be divided into major time slot. In each major time slot, an application is selected to run. The major time slot is then divided into minor time slot. And the application will select a task or some tasks to run in minor time slot. The duration of minor time slot can be different for each application. The structure of hierarchical slot is presented in Figure 6.1.

Bibliography

[1] Benny Akesson, Andreas Hansson, and Kees Goossens, *Composable resource sharing based on latency-rate servers*, Proc. Euromicro Symposium on Digital System Design (DSD), aug 2009.

[2] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li, *The parsec benchmark suite: Characterization and architectural implications*, Tech. report, Princeton University, 2008.

[3] Z. Deng and J. W.-S. Liu, *Scheduling real-time applications in an open environment*, RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (Washington, DC, USA), IEEE Computer Society, 1997, p. 308.

[4] R.P. Dick, *The embedded system synthesis benchmarks suite (e3s)*, 2002.

[5] EEMBC, *Edn embedded microprocessor benchmark consortium*, http://www.eembc.org/.

[6] Marcus Ekerhult, *Compose: Design and implementation of a composable and slack-aware operationg system targeting a multi-processor system-on-chip in the signal processing domain*, Master's thesis, Technical University of Lund, 2008.

[7] G. Bilsen et al., *Cyclo-static dataflow*, IEEE Transactions on Signal Processing (1996), 44.

[8] Shay Gal-On, *Multicore benchmarks help match programming to processor architecture*, MultiCore Expo, April 2008.

[9] Kees Goossens, John Dielissen, and Andrei Rădulescu, *The Æthereal network on chip: Concepts, architectures, and implementations*, IEEE Design and Test of Computers (2005), no. 5, 414–421.

[10] Pawan Goyal, Xingang Guo, and Harrick M. Vin, *A hierarchical cpu scheduler for multimedia operating systems*, Usenix Association Second Symposium on Operating Systems Design and Implementation(OSDI) (1996), 107–121.

[11] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown, *Mibench: A free, commercially representative embedded benchmark suite*, Workload Characterization, Annual IEEE International Workshop (2001), 3–14.

[12] Andreas Hansson, Marcus Ekerhult, Anca Molnos, Aleksandar Milutinovic, Andrew Nelson, Jude Ambrose, and Kees Goossens, *Design and implementation of an operating system for composable processorsharing*, Microprocessors and Microsystems (2010), –.

[13] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken, *Compsoc: A template for composable and predictable multi-processor system on chips*, ACM Trans. Des. Autom. Electron. Syst. (2009), no. 1, 1–24.

[14] Adarsha Rao Hristo Nikolov, S.K. Nandy Ed Deprettere, and Ranjani Narayan, *A h.264 decoder: A design style comparison case study*, Int. Asilomar Conference on Signals, Systems, and Computers (Nov. 1-4, 2009), 100.

[15] Gilles Kahn, *The semantics of a simple language for parallel programming*, in Proc. of the IFIP Congress (1974), 74.

[16] H. Kopetz, *Real-time systems: Design principles for distibuted embedded applications*, Kluwer Academic Publishers, 1997.

[17] Tei-Wei Kuo and Ching-Hui Li, *A fixed-priority-driven open environment for real-time applications*, RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium (Washington, DC, USA), IEEE Computer Society, 1999, p. 256.

[18] Man lap Li, Ruchira Sasanka, Sarita V. Adve, Yen kuang Chen, and Eric Debes, *The alpbench benchmark suite for complex multimedia applications*, In Proc. of the IEEE Int. Symp. on Workload Characterization, 2005, pp. 34–45.

[19] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-smith, *Mediabench: A tool for evaluating and synthesizing multimedia and communications systems*.

[20] Giuseppe Lipari and Sanjoy K. Baruah, *Efficient scheduling of real-time multi-task applications in dynamic systems*, RTAS '00: Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000) (Washington, DC, USA), IEEE Computer Society, 2000, p. 166.

[21] Reinhard Maier, Georg Stöger, Stefan Poledna, and Günther Bauer, *Ieee micro: Electronics architecture - time-triggered architecture: A consistent computing platform*, IEEE Distributed Systems Online (2002), no. 9, 36–45.

[22] Anca Molnos and Kees Goossens, *Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors*, DSD, 2009, pp. 409–418.

[23] Orlando Moreira, Frederico Valente, and Marco Bekooij, *Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor*, EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software (New York, NY, USA), ACM, 2007, pp. 57–66.

[24] André Nieuwland, Jeffrey Kang, Om Prakash Gangwal, Ramanathan Sethuraman, Natalino Busá, Kees Goossens, Rafael Peset Llopis, and Paul Lippens, *C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems*, Design Automation for Embedded Systems (2002), no. 3, 233–270.

[25] Dongrui SHE, *Fpga platform for emulation of composable and predictable mpsoc power management*, Master's thesis, Eindhoven University of Technology, 2009.

[26] SPEC, *Standard performance evaluation corporation benchmark suite*, http://www.spec.org/.

[27] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra, *Overview of the h.264/avc video coding standard*, Circuits and Systems for Video Technology, IEEE Transactions on (2003), no. 7, 560 –576.

[28] Xilinx, Inc., *Logicore ip fast simplex link (fsl) v20 bus.*

[29] Xilinx, Inc., *Microblaze processor reference guide.*