TIMING ANALYSIS OF THE ABORT-AND-RESTART PARADIGM ON A SCRATCHPAD MEMORY-BASED EXECUTION PLATFORM

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Zeinab Kazemi

August 2015

TIMING ANALYSIS OF THE ABORT-AND-RESTART PARADIGM ON A SCRATCHPAD MEMORY-BASED EXECUTION PLATFORM

Zeinab Kazemi

APPROVED:

Dr. Albert M.K. Cheng
Dept. of Computer Science

Dr. Edgar Gabriel
Dept. of Computer Science

Dr. Yuhua Chen
Dept. of Electrical and Computer Engineering

Dean, College of Natural Sciences and Mathematics

Acknowledgement

The completion of this thesis would not have been possible without the contribution of many individuals who have provided me with their valuable assistance. My special thanks go to my advisor, Professor Albert M.K. Cheng, for his guidance and encouragement throughout my endeavour as his student. This research is the result of his dedication and motivation towards novel research.

I would also like to thank my committee members professor Edgar Gabriel and professor Yuhua Chen for their brilliant comments and suggestions on the thesis and research. I also would love to thank my beloved family, for all the sacrifices they made on my behalf. Last but certainly not least; my deepest gratitude goes to my boyfriend, Ebrahim, who always supported me throughout my studies.

TIMING ANALYSIS OF THE ABORT-AND-RESTART PARADIGM ON A SCRATCHPAD MEMORY-BASED EXECUTION PLATFORM

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Zeinab Kazemi

August 2015

Abstract

Priority-based Functional Reactive Programming (P-FRP) is a new variant of FRP to model reactive applications in real-time systems. In P-FRP, when the currently running task is preempted by an arriving higher-priority task, the lower-priority running task is aborted and the higher-priority task will execute. The lower-priority task restarts when the higher-priority one completes. However, unlike the preemptive model, when a task aborts, all the changes made by this task are discarded. That is to say, when an aborted task restarts, it should execute from the beginning. In order to provide a realistic Worst-Case Response Time (WCRT) of the tasks in P-FRP, it is therefore mandatory to derive a realistic Worst-Case Execution Time (WCET) of each task. Previous studies have ignored memory latency in the derivation of the WCRT, making the resulting estimate inaccurate and unrealistic. Furthermore, these studies have also assumed that the WCET of each task is known a priori. In this thesis, a scratchpad memory (SPM)-based platform for executing P-FRP tasks and an approach to determine the WCET of the tasks by considering the memory cost of the aborted tasks is introduced. The WCET of a task in a P-FRP system is first computed, and then the memory penalty caused by preemption is derived. In the next step, the WCRT of the task sets in P-FRP is calculated by considering memory latency in the proposed platform. Experimental results from the derivations of the WCET and WCRT using task sets from the SNU real-time benchmarks and randomly generated tasks are presented to validate this approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 An Introduction to P-FRP

Functional Reactive Programming (FRP) is a framework for constructing reactive applications. FRP was originally developed with the Haskell language [21, 42] as the basis. It declares the notion of events and behaviors which are discrete and time-varying, respectively. FRP exploits many of the characteristics of high-level languages such as abstraction, maintainability, and comprehensibility. Nowadays, FRP is used in many applications such as robotics, animation, and visions [34, 19, 35]. Extensive research results have also been presented on the semantics of FRP [15, 2, 41, 44]. Applicability of functional programming for real-time systems is explored by the research community [41].

The characteristics of FRP have encouraged the academics to exploit it in real-time systems. Unfortunately, Haskell does not provide real-time guarantees and so the same applies to FRP. Only a small amount of research has been devoted towards making FRP more adaptive to reactive applications and real-time systems [45, 22]. In [45], Real-Time FRP (RT-FRP) is proposed as a statically bounded language in which the execution time of each program is bounded. A new variation of FRP called Priority-Based FRP (P-FRP) is also presented [22]. In this model, while keeping the semantics of the language, a priority is assigned to each task (event). P-FRP is quite different from preemptive and non-preemptive systems. In P-FRP systems, a higher-priority task always preempts the lower-priority one and in case of preemption, the latter will be aborted and all the changes made by the task will be discarded.

To optimize resources in a real-time system, analysis of the worst-case response time (WCRT) is a must. Belwal and Cheng have developed a method for determining the actual response time using an enumeration for the idle periods in a P-FRP system [9]. From the same authors [10], a new technique using a game-board for calculating the response time has been presented. This model is easier to implement due to its use of native data structures. In both works, the value of the worst-case execution time (WCET) is assumed to be known a priori and the effect of memory latency has not been considered for computing the WCRT. However, the result of WCRT analysis will not be reliable unless an accurate WCET of tasks and their memory access latency are derived.

In order to estimate a tight WCET, prior understanding of the system on which a task executes is a must. To the best of our knowledge, there is no description of

such a system based on the P-FRP model. To address this challenge, a P-FRP based system is introduced and denoted as a *functional reactive system*. In such systems, the tasks follow the abort-and-restart paradigm, which is neither preemptive nor non-preemptive. The behavior of the tasks is quite similar to memory transactions—a task is either committed or discarded. Namely, when a lower-priority task is preempted by a higher-priority one, all the changes made by the former task are discarded and the task is returned to the scheduler to be executed again. Thus, there is an overhead in recomputing all the preempted tasks. However, there is no need of logging and validation in P-FRP systems, which can be time-consuming. Therefore, functional reactive systems benefit from the lack of task side effects, no checkpointing, rollback cost, and type safety. In addition, the Abort-and-restart model prevents the priority inversion problem [7]. Priority inversion occurs when a higher-priority task is blocked on a lower-priority one. Assume a scenario when there are two tasks in the system, task τ_L with priority P_L and task τ_H with priority P_H such that $P_L < P_H$. An exclusive region R is shared between the two tasks. τ_L acquires R first, so until τ_L relinquishes R, τ_H is blocked. On the other hand, an unrelated medium-priority task τ_M arrives, where $P_L < P_M < P_H$, and preempts τ_L, so, τ_H is unable to run and is preempted by a lower-priority task. In this situation, it is told that the priority of the higher-priority task has been inverted to the lower priority level. Hence, a functional reactive system could be a potential alternative to the current preemptive and non-preemptive models, especially in hard real-time systems.

Another challenge to estimate a more accurate WCET is to find micro-architectural specifications suitable for functional reactive systems. Since no P-FRP based system has been fully implemented nor even described, one needs to specify a certain platform for them. In the abort-and-restart model, only a fully-executed task's changes will be committed to memory. In order to guarantee this characteristic, the preempted tasks' write operations should be discarded. In current real-time systems, discarding written changes is prohibitively expensive, e.g., rollback or logging. To meet this challenge, a platform applicable for P-FRP is assumed, based on ScratchPad Memory (SPM) which will be described in Section 4. But first, the effect of memory-write operations of a P-FRP task in a system without a discard mechanism is explained using the motivational example in Section 1.2.

In partial fulfillment of the thesis, the approaches and methods discussed in this thesis, have been published in two papers. [24] has focused on the static WCET analysis of P-FRP systems, and an event-based approach to model the behaviour of P-FRP tasks. [25] focuses on the timing analysis of P-FRP on an SPM-based platform. The latter paper discusses the advantages of using the SPM in functional reactive systems and provides WCET and WCRT analysis of P-FRP, considering memory latency.

1.2 Motivational Example

Consider a P-FRP system with two tasks τ_1 and τ_2 under rate monotonic (RM) scheduling [32]. Figure 1.1 shows the period and WCET of the tasks. Assume τ_2 has

a memory store instruction in the first unit of its execution. At time zero, both τ_1 and τ_2 arrive, and since the scheduling policy is RM, τ_1 starts executing (see Figure 1.1). After τ_1 finishes at time 2, τ_2 starts its execution until it completes. However, during the next period of τ_2, τ_1 arrives (time 10) and preempts τ_2. At this point, τ_2 has already stored some changes to the memory without being fully executed, causing unwanted changes to the system state.

Table 1.1: The characteristics of tasks in Figure 1.1.

ID	Period	WCET
τ_1	5	2
τ_2	8	3

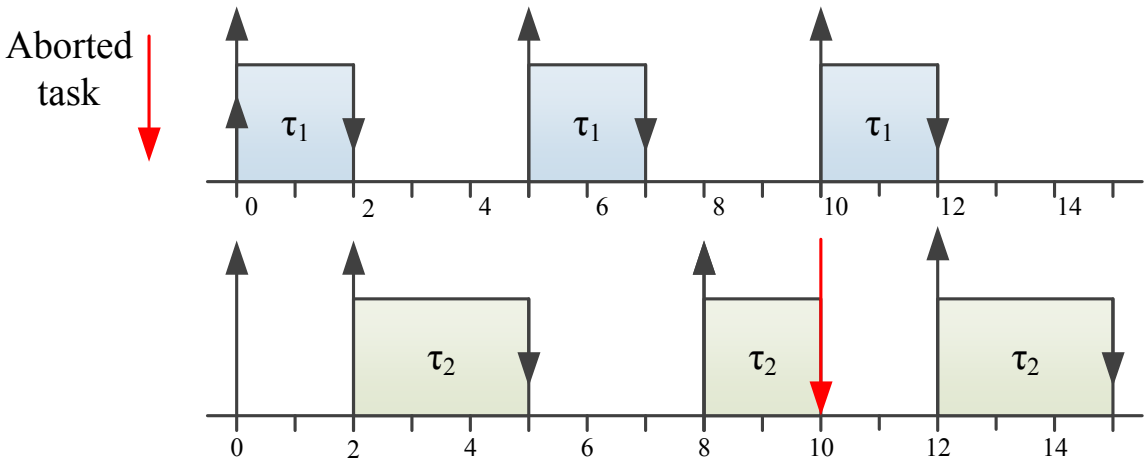

Figure 1.1: The depiction of the abort-and-restart paradigm for the motivational example.

1.3 A Simple Example of FRP

In this section, in order to compare FRP with an imperative language, a simple example of a program written in FRP and Java from [14] is presented. Figure 1.2 shows a simple code in FRP. This program displays a red circle on the screen that tracks the current mouse position.

```
1  ball = stretch 0.3 (withColor red circle)
2  anim = (lift2 move) (p2v mouseB) (constB ball)
3  main = animate anim
```

Figure 1.2: A simple program in FRP.

```
1  Drawable circle = new ShapeDrawable(new
       Ellipse2D.Double(-1,-1,2,2));
2  Drawable ball = circle.withColor(Color.red).stretch(0.3);
3  Behavior mouse = FRPUtilities.makeBehavior(sched, frame,
       "mouse");
4  Behavior anim = FRPUtilities.liftMethod(sched, new
       ConstB(ball), "move", new Behavior[] { mouse });
5  franimator.setImageB(anim);
```

Figure 1.3: The equivalent code of Figure 1.2 in Java.

At line 1, a red ball is created with the size of 0.3. In the second line, a lifting combinator (lift2) has been applied to function *move*. *lift2* converts functions over

static values to functions over behaviors. An equivelant code of Figure 1.2 has been implemented in Java in Figure 1.3.

1.4 Contribution

The primary contributions of this work are as follows:

• As a first step towards addressing the WCET derivation of tasks in P-FRP, using SPM as an on-chip memory is discussed, in order to calculate a reliable and tight upper bound on the WCET.

• The memory latency of P-FRP tasks is measured in the abort-and-restart model to improve the accuracy of the derived upper bound on WCET.

• By estimating a realistic WCET, and designing an SPM-based execution platform, the accuracy of the WCRT derivation in a P-FRP task set is improved.

• Extensive experiments with the SNU real-time benchmark and synthetic tasks are conducted to evaluate and validate the approach.

1.5 Organization

In Chapter 2, background information about the Control Flow Graph (CFG) and SPM micro-architecture is provided. In Chapter 4, the benefits of SPM, how it is relevant to P-FRP systems, and how it is exploited in order to have a tight bound on WCET are discussed. Characteristics of P-FRP tasks have been discussed in

Chapter 5. In Chapter 6, an introduction to existing approaches to WCET analysis and how to modify them to adapt them for the system is described. The response time analysis of P-FRP tasks is explored in detail in Chapter 7. Chapter 3 reviews the prior algorithms to calculate response time, WCET, and related work. This approach is validated by experimenting on different task sets in Chapter 9. The thesis is concluded in Chapter 10.

Chapter 2

Preliminary

In this section, a brief introduction to the concepts used in the approach is provided. In Section 2.1 the definition of notations and concepts used in this research are explained. In Section 2.2, SPM, and the benefits of having SPM in a P-FRP system are discussed. In Section 2.3, the structure of a control flow graph (CFG) is briefly described.

2.1 Notations and Basics

The notation and definitions for important concepts used in this research are as follows:

- A *periodic task set* is a set of tasks that their job's arrival time periods are fixed.

• A *sporadic task set* is a set of tasks that their job's arrival time periods are variable during the execution of the tasks.

• *WCET* of a task is the maximum time it takes for a task to finish its execution under different inputs.

• *Response time* of a task, is the total amount of time it takes to respond to the task.

• Let *task set* $\Gamma_n = \{\tau_1, \tau_2, ..., \tau_n\}$ be a set of n periodic or sporadic tasks.

• The *priority* of $\tau_i \in \Gamma_n$, is P_i a positive integer, where a lower number implies higher-priority.

• C_i is the worst-case execution time for τ_i.

• A *job* is an instance of a task. A task can have multiple jobs during a period of time.

• T_i is the arrival time period between two successive jobs of τ_i.

• The total utilization factor (U) of a task set is the sum of ratios of the tasks' execution times to their arrival periods:

$$U = \sum_{i=1}^{N} C_i/T_i \tag{2.1}$$

• *Interference* on τ_i is the action where the execution of τ_i is interrupted by the release of a higher-priority task τ_j.

• *Abort cost* is the amount of time spent in the aborted execution of a task.

• A *preemptive system* is a system that allows a higher-priority task τ_j to interrupt

a lower-priority task τ_i with the intention of resuming the task τ_i at a later time.

• A *non-preemptive system* is a system that does not allow a higher-priority task interrupt the running task.

• *Rate monotonic scheduling* is a scheduling algorithm used in real-time systems. The task's priority is assigned based on the duration of execution of a job. The shortest task has the highest priority.

2.2 Scratchpad Memory

Due to the increasing gap between the performance of processors and off-chip memory, using an on-chip memory for real-time systems is obvious. While caches are controlled by hardware, SPM can be managed at the software level. SPM lets programs request for space allocation, loads their data from main memory, and commits the changes back to the memory. Although hardware management of a cache makes it a desirable choice for general systems, it leads to unpredictable timing behavior. Using software manageable SPM provides predictable memory access latency, making it a better alternative for real-time systems [39]. Since preempted P-FRP tasks should not have any side effect and only the completed tasks can write back the changes to memory, SPM is used which allows commit after each task completion and also de-allocation of the changes made by the preempted task. Furthermore, SPM exhibits efficient power consumption for real-time embedded systems. The benefits of SPM have motivated many commercial systems to include it in their architecture e.g., the *Sony PS1's R3000*, the *Cyrix 6x86*, and the *Sony's PS2 Emotion Engine*.

2.3 Control Flow Graph

CFG is a representation of the control flow of all the paths that might be traversed in a piece of program using a graph notation. Each node in this graph represents a basic block which is a straight-line program with only one entry and no branching or jumping until at the end of the block. At the end of each block, the jump instruction will determine which node (basic block) the control flow of the program will traverse next. Each edge in this graph represents possible control flow from one block to the other. Each block can have multiple incoming and outgoing edges [1]. The CFG of a program can be used in order to derive the static analysis of WCET of the program. The CFG can be derived using the input C code and binary representation of the program. Further information on how CFG of a program is exploited, is given in Section 6.1.

Chapter 3

Related Work

In this chapter the recent work in the literature has been explored. The chapter is organized into three sections: (1) Related Work on P-FRP, (2) Related Work on WCET, and (3) Related Work on SPM.

3.1 Related Work on P-FRP

Extensive research has been presented on the semantics of FRP. Dai et al., propose an implementation of major features of FRP in C++ by exploiting the template class and operator overloading built in C++ library to benefit from both FRP and C++ advantages [15]. Besides, Taha et al., review the basics of functional programming, the techniques relevant to implementing a real-time software, and how functional programming is applicable to real-time systems [41]. Amsden has surveyed the literature on FRP implementation and its applications in [2]. New ideas for future

research on FRP, along with examples are also proposed. Wan and Hudak review the semantics of FRP and how it can be implemented based on streams which represent continuous behaviors [44]. They show both the power and limitations of their design.

Since FRP has shown good promises in the reactive systems domain, researchers have been encouraged to use it in real-time systems in several previous studies. In spite of the potentials of FRP, there should be a few modifications to this language to make it promising for real-time systems. FRP itself cannot be a good programming language for real-time systems. Although in real-time systems, the concept of bounding the time of a task is crucial, this issue was not considered in the FRP itself. Wan et al., present a real-time FRP (RT-FRP) that bounds time and space of a program in [45]. In their work, they have focused on the high-level aspect of the language and how to modify it in order to make it adaptive to a real-time system. Another variant of FRP which is adaptive to real-time systems is called P-FRP [22]. In this model, a priority is assigned to each task, and the higher-priority task can preempt the lower-priority one. However, the latter should restart its execution from the beginning when executing for the next time.

In the paper proposed by Wen et al., they prove an optimal priority assignment for task sets containing more than two tasks when the task periods are multiple of each other [46]. First, they describe the priority assignment characteristics of tasks in P-FRP, and then, they discuss the priority assignment for a task set of two tasks. Eventually, they prove an optimal solution for a P-FRP system where the tasks priorities are multiple of each other.

Belwal and Cheng have exploited the P-FRP paradigm for determining the actual response time in such systems. In their model, P-FRP is similar to a transactional memory. That is to say, the execution of a task is "all or nothing", meaning P-FRP uses a copy of program state; if a task aborts, all the changes will be discarded, otherwise all the changes made by the task will be committed to memory. They present an exhaustive enumeration for idle periods in a tree-like structure [9]. However, due to the implementation complexity of the gap-enumeration method, another technique for calculating the actual response time is presented using game-boards [10]. This model is easier to implement due to the use of native data structures. Although the new method is simpler to implement, the gap-enumeration algorithm is more efficient. In both methods, the assumption is that the WCET of each task is known a priori. Besides, they do not consider the abort cost of memory operations. In this research, the accuracy of calculating the response time in the gap-enumeration method is improved by computing a tight bound on the WCET of P-FRP systems and estimating memory cost of each preemption. In another study, Belwal et al., present a polynomial-time algorithm to calculate the bounds on the response time of P-FRP tasks [11]. Nonetheless, the new algorithm guarantees a result for all task sets. In their method, they compute the upper bound of the response time based on the lower bound using fixed point iteration algorithm developed in [5]. As explained in their paper, there is a trade-off between a safe upper-bound on the response time and a tight bound on that value.

Most of the studies on FRP and P-FRP focused on the semantics of the language

and to the best of our knowledge, this research is the only work presenting an execution platform considering both architecture and the behavior of a system under P-FRP.

3.2 Related Work on WCET

Unlike FRP, the WCET domain is very well researched. Hansen et al., present statistical estimation of the WCET [20]. They model the estimation by using Extreme Value Theory (EVT). Although this approach is simple and easy, the main contribution of their work is to improve the prediction ability of the exceedance of estimated the WCET. While this method might be a good approach for soft real-time systems, hard real-time systems cannot afford any deadline misses that can result from the EVT estimation.

Stappert et al., present a tight bound on the WCET estimate of hard real-time and straight-line programs [37]. They consider both low-level and high-level aspects of such systems. Cache, pipelining and instruction parallelism are considered in the low-level analysis. Eventually, they use the low-level analysis to find the longest executable path of the program (high-level analysis) and at the end calculate the WCET. Li et al., present an open source timing analyzer for generating the WCET of a given C program using static analysis [28]. Chronos models detailed micro-architectures of a system, including cache, pipelining, out-of-order execution [31, 30], and branch predictions.

As mentioned in earlier sections, the presence of cache will cause lack of predictability and uncertainty on the estimate of the WCET, which may lead to deadline misses in hard-real time systems. Chattopadhyay and Roychoudhury model a generic cache architecture with separate instruction and data cache, and a unified cache in the second level to model timing effects of caches for WCET estimation [13]. Li et al., [29] investigate the effects of speculative execution and caching on the WCET of a program. Speculative execution can indirectly affect the cache performance, e.g., increasing cache misses resulting from speculative execution of blocks. They use Integer Linear Programming (ILP) to estimate the WCET of a program and evaluate their techniques on realistic benchmarks. [33] models both local and global branch predictions and estimates their effects on the WCET of a program. Local branch prediction predicts the branch result based only on its execution history, whereas global prediction considers the execution history of other branches. In their model, they present linear inequalities for bounding branch mis-predictions for all possible inputs. Finally, they evaluate the resulting WCET on different benchmarks. [30] models out-of-order processor pipelines for WCET analysis. They estimate the WCET by modeling the interactions between consecutive basic blocks and the effects of instruction cache. [31] models out-of-order superscalar processor pipelines for WCET estimation of a program. They analyze the timing by using a fixed-point time intervals at which the instructions enter/leave a stage of the pipeline. They use ILP to combine the timing analysis of different basic blocks and calculate the WCET estimate in a platform containing instruction cache and branch prediction. Yan and Zhang [47] bound the WCET for threads running on multi-core processors

with shared L2 instruction caches by considering inter-thread instruction conflicts.

3.3 Related Work on SPM

Because of the advantages of SPM in estimating the WCET, several techniques have been proposed for SPM management. Li et al., present a general-purpose compiler approach called memory coloring to efficiently allocate arrays in main memory by partitioning the SPM into a register file [27]. A compiler-controlled dynamic SPM management framework is proposed which uses loop and data transformations [23]. Egger et al., propose an automatic dynamic SPM management method for instructions [17]. This method loads the code segment to SPM during run time. The mapping is based on ILP formulation that approximate their demand paging method. Their approach is based on postpass analysis and optimization techniques. The postpass technique enables demand paging by analyzing the object files of the program and converting them to binary images. Their technique reduced energy consumption and increases performance by around 24% and 23%. In another work by Egger et al., [18], a dynamic SPM allocation strategy has been presented that targets the horizontally partitioned memory subsystems. Using Memory Management Unit's page-fault exception mechanism, they find the frequently accessed code segments and load the pages to SPM before execution. They evaluate their method on a few embedded applications and compare it to a fully-cached system. On average, a 12% improvement in runtime performance and a 33% reduction in energy consumption are achieved.

Kim et al., introduce dynamic code-management techniques for loading program code from main memory to SPM [26]. In such architectures, any access to main memory is done through DMA, making dynamic code management critical for such systems. On the other hand, previous techniques for dynamic code management for Software Managed Multicore architectures are designed for average-case execution time. As it is indispensable for real-time systems to have a tight-bounded WCET, they have presented two new WCET-aware techniques, one optimal solution based on ILP and the other is a heuristic. In order to enhance the predictability of memory accesses latencies in embedded systems, SPM is used as an alternative to cache [16]. Using a dynamic allocation technique during compile time will result in data placement to the SPM during run time. Previous literature has focused on reducing average-case execution time and energy consumption in embedded systems. However, in real-time systems, the main concern is to decrease the WCET. The authors present a WCET-directed method to dynamically allocate static data and stack data of a program to SPM. They also discuss the trade-offs of different granularities of memory transfers.

[8] uses SPM as an alternative of cache to address the problem of on-chip memory selections for computationally intensive applications. The energy consumption of SPM and cache are evaluated and the result shows that SPM is a low power alternative of cache in most situations. Scratchpad allocation techniques for data memory to minimize a task's WCET is presented in [39]. Because of the efficiency of SPM in energy consumption, die area, and performance, SPM can be used in embedded applications. [3] provides an optimal algorithm to solve the challenge of mapping

memory locations to scratchpad locations. The algorithm maps the segments of external memory to physically partitioned SPM, in a polynomial time. [43] provides an allocation technique analyzing the application and inserts instructions to copy both code segments and variables onto the scratchpad during runtime. The experimental results show a reduction of energy consumption and application runtime. Suhendra et al., provide an integrated task mapping, scheduling, data allocation technique and SPM partitioning based on ILP [40]. Because of SPM's increasing application in embedded systems, the authors present a technique to improve the performance of SPM in such systems. They experiment their technique on embedded applications and show that SPM optimization and integrated task-scheduling lead to performance improvement by up to 80%. An outstanding percentage of developed embedded systems belongs to mobile applications [38]. Because of the limitation of battery capacities, the use of less power hungry memories is essential. Steinke et al., present an algorithm integrated with a compiler to analyze and select data pieces to be placed in the SPM. They evaluate their algorithm by comparing them to a cache solution and show significant improvements for energy consumptions. [4] provides an efficient algorithm to solve the memory-to-SPM mapping problem by using dynamic programming applied to a synthesizable hardware architecture. Their algorithm maps segments of external memory to physically partitioned banks of the SPM. The partitioning overhead has been considered in the algorithm and the number of SPM segments is unlimited. Although in previous methods, the running times are exponential to this number, this solution's execution time is polynomial in the number of segments.

Chapter 4

Scratchpad Memory in P-FRP Systems

One crucial matter to consider for the abort-and-restart model is that, tasks follow the "all or nothing" behavior of memory transactions. That is to say, after preemption of each task, all the changes made by the preempted task should be discarded. For this end, a task should not commit anything to memory while executing. Instead, it updates the changes to a temporary location or volatile registers. Furthermore, all the memory-write operations need to be at the end of each task. If otherwise, the task can be preempted after the memory-store operation and this would violate the P-FRP requirements. Assume a scenario where a task τ_i with priority P_i is executing. A higher-priority task τ_j with priority P_j, such that $P_j > P_i$, arrives in the middle of τ_i's store operation. Since there should be no side effect of the aborted task, the system has to either explore all the log files and change the program state

to the way it was before executing the aborted task or it can keep checkpoints of different program states. Both cases are very time-consuming. Especially for real-time systems that bounding the cost of each step's execution is mandatory, other alternatives should be considered. To address this problem, an on-chip SPM is used to store the in-progress work, and then commit all the changes made by a task to off-chip memory. SPM benefits from predictable timing behavior and is a suitable fast on-chip memory to facilitate the derivation of the WCET. Since the proposed system is hard real-time, the predictability of SPM can be exploited in order to tighten the bound of WCET. Moreover, since in this study, P-FRP systems are generally embedded real-time systems, and in such systems, lower-priority tasks may frequently get preempted by higher-priority ones, these tasks should be as small as possible for the system to have a minimal overhead.

4.1 Read and Write Latency in P-FRP

The idea of using SPM in embedded real-time systems is to ensure that prior knowledge of the task's code and data location is available. Before execution of a task, its data and code should be located in the SPM. To do so, a table is used which lists the tasks and information on whether they are located in the SPM or in memory. A function is defined to determine the validity of the information for a specific task in the SPM:

$$map(\tau) = \begin{cases} 1, & \text{if task } \tau \text{ is already in SPM.} \\ 0, & \text{otherwise,} \end{cases} \tag{4.1}$$

where τ is a task scheduled for execution.

In the SPM, read and write from/to main memory are handled by Direct Memory Access (DMA) to minimize the memory latency. For each P-FRP task τ, the memory read latency L_τ^R is computed as follows:

$$L_\tau^R = \begin{cases} dma_\tau^C + dma_\tau^I & \text{if map } (\tau) = 0, \\ dma_\tau^I & \text{if map } (\tau) = 1, \end{cases} \tag{4.2}$$

where dma_τ^C is the cost of reading the task's code and dma_τ^I is the cost of reading the input data from memory. This equation means that when a new task arrives, by checking function $map(\cdot)$, it is known whether the task is already in the SPM. If $map(\cdot)$ returns 1, then there is no need to read the task's code again. However, it is required to load the input data from the SPM since the data may have changed during the execution of the higher-priority tasks. Yet, the code is valid until it is de-allocated. On the other hand, if function $map(\cdot)$ returns zero meaning the task is not available in the SPM, then both data and code of the task should be read into the SPM.

For each P-FRP task τ, the memory write latency L_τ^W can also be computed as follows:

$$L_\tau^W = dma_\tau^O, \tag{4.3}$$

where dma_τ^O is the cost of writing the task's changes (output) to memory.

The cost of a DMA operation consists of DMA request and transferring data. In order to read the data and code of a task, multiple DMA requests may be required

depending on the location of the data in memory. The cost of DMA requests of a task is computed during the WCET computation of the initial basic block. The same applies for DMA write operations. To calculate the latency of transferring data, dma_τ^C, dma_τ^I, and dma_τ^O are computed based on the size of code and data, and the DMA bandwidth:

$$dma_\tau^I = size(Data_\tau^I)/BW_{DMA} \tag{4.4}$$

$$dma_\tau^C = size(Code_\tau)/BW_{DMA} \tag{4.5}$$

$$dma_\tau^O = size(Data_\tau^O)/BW_{DMA} \tag{4.6}$$

where $size(Data_\tau^I)$, $size(Code_\tau)$, and $size(Data_\tau^O)$ are the size of the input data, code, output data in bytes, respectively, and BW_{DMA} is the minimum bandwidth of DMA in clock cycles per bytes. The minimum bandwidth is used to ensure a safe upper bound on the WCET.

Chapter 5

Handling Tasks in SPM

In order to increase the applicability of this design for real-time systems, the model should consider different types of task sets. The tasks can be small enough to fit into the SPM, or they can be larger than the SPM's size. When the scheduler schedules a new task to start execution, the task and its data will be read into the SPM depending on its state. If the task already exists, only its data should be read. Otherwise, both task and data will be read from memory. Therefore, before starting a task, a lookup function will search through the SPM state to determine whether the task is in the SPM. This process is handled at the software level, and by optimizing the search, the amount of extra time to find the location of the task can be kept at a minimum. The lookup function consists of a hash table that keeps the pointers to the tasks' location in the SPM. Hence, the time complexity of the lookup function is constant. In case the task should be read into the SPM, a new space will be created and the pointer to the task will be added to the hash table. If the task size exceeds

the available space in the SPM, a replacement policy will be used to replace an old task with the new one. The records of the old task are also deleted from the hash table.

If the new task size is larger than the SPM's size, all the previous information on the SPM will be discarded. The task code is read into SPM, and task data is divided into multiple partitions. The partitions will be brought into the SPM using DMA read during the execution of the task. The new part of data also replaces the old data using Least Recently Used (LRU) replacement policy.

Depending on the location of the task data in memory, the number of DMA operations varies. If the data is scattered in the main memory, data will be read using multiple DMA reads, hence taking more time for a task to start its execution and therefore, this leads to a longer response time. This model calculates the DMA read time by computing the data transfer latency and DMA request latency as explained in Chapter 4.

5.1 Types of Tasks in a P-FRP System

The P-FRP system can handle both periodic and sporadic tasks. If tasks are sporadic, in addition to each task's data, the task itself should also be read every time so the execution time will be more than that for periodic task sets with similar execution steps. On the other hand, on average, the execution time will decrease for periodic task sets.

5.2 DMA Write in the Abort-and-Restart Paradigm

At the end of a task execution, all the changes will be committed to memory. The write operations are done using DMA write. To assure the lack of task side effects in P-FRP, committing data to memory should be atomic. There are multiple solutions available. All the interrupts in the system can be disabled, or the highest priority can be temporarily assigned to the task during DMA writes so it will not be preempted. To turn off the interrupts, it is necessary to save the current state of the system and to restore it after the interrupts are enabled again. To avoid extra time to save and restore the program state, it is best to use the second option. In this model, the highest priority is assigned to DMA write and the priority of the executed task is also increased to the highest so that it can commit the changes to memory atomically without any preemption. The priority of the task will be changed to its previous value after the commit step.

Chapter 6

WCET Analysis

6.1 Static WCET Analysis Using Control Flow Graph

In this approach, the CFG of a program is built by disassembling the input binary code. When reaching a function call, a new CFG is created for it, unless the CFG for that function already has been built. Reading from the top of the function code, each instruction belongs to a basic block until a branch instruction is countered. While tracking the code, the CFG of the functions will be built. In the end, all the CFGs are merged into one global CFG which has all the control flow possibilities of the given program. To describe the CFG with an example, assume global CFG, GCFG = (B, E, entryB, FinalB), in which B is the set of all the basic blocks, E is the set of all edges created to connect these blocks, entryB and FinalB are the entry

and final basic blocks, respectively. As an example, the CFG of the a C program in Figure 7.2, is shown in Figure 7.3.

6.2 Safe Upper-bound for WCET

There have been a lot of research on finding a tight bound on the WCET of systems under different micro-architectures. However, for a P-FRP system, the problem of calculating the WCET is different since in such systems, the memory will not be updated until a task is fully executed. At the end of the task execution, all the changes to SPM will be committed to the main memory.

For this research, the current WCET solutions are changed to adapt them to a P-FRP system. A C program and hardware parameters are taken as inputs and the WCET is returned as the final output. For this end, the upper bound on the execution of each basic block is calculated. In the next stage by using ILP, the WCET estimate of the program is found. To do so, the WCET of the P-FRP system is computed based on the extension of the ILP objective function introduced in Chronos, an open source static timing analyzer [28]. Chronos calculates a safe upper bound on the WCET of a program by modeling micro-architectural details. Equations 6.1 and 6.3 from [28] have been exploited to derive the ILP objective function of P-FRP:

$$N_B = \sum_{B_i \to B} E_{B_i \to B} = \sum_{B \to B_i} E_{B \to B_i}, \tag{6.1}$$

where $E_{B_i \to B_j}$ is an ILP variable showing the number of transitions on basic blocks,

$B_i \rightarrow B_j$ on the control flow graph.

Li et al., [31] model super scalar processors with pipeline and out-of-order execution for WCET analysis. They estimate the execution time of each basic block and at the end, they use ILP to determine the program's WCET. In their work, instruction cache and branch prediction units are also considered. The objective function of ILP is shown in the equation below:

$$ET = \sum_{i=1}^{N} \sum_{j \rightarrow i} \sum_{\omega \in \Omega_i} t_{j \rightarrow i}^{c,\omega} * E_{j \rightarrow i}^{c,\omega} + t_{j \rightarrow i}^{m,\omega} * E_{j \rightarrow i}^{m,\omega}, \tag{6.2}$$

where $t_{j \rightarrow i}^{c,\omega}$ is the WCET of block B_i. In this equation, B_j is a predecessor of B_i. The cache state under which the block B_i has been executed is $\omega \in \Omega_i$. Ω_i is all the cache scenarios of B_i which is a subset of the loop levels where B_i is contained. Furthermore, B_i is the result of a correct branch prediction. $E_{j \rightarrow i}^{c,\omega}$ is the number of transitions from the preceding block B_j to B_i. $E_{j \rightarrow i}^{c,\omega}$ is an ILP variable and will be used to maximize the ILP objective function to obtain WCET. The number of transitions needs to be bounded by the set of all possible states of the cache. So $t_{j \rightarrow i}^{c,\omega}$ is the WCET of the block B_i when the prediction was correct at the end of B_j. The rest is also similar for $t_{j \rightarrow i}^{m,\omega}$ and $E_{j \rightarrow i}^{m,\omega}$. By knowing the set of cache states, using out-of-order execution, and branch prediction, Li et al., prove that $t_{j \rightarrow i}^{c,\omega}$ and $t_{j \rightarrow i}^{m,\omega}$ are constant, so only $E_{j \rightarrow i}^{c,\omega}$ and $E_{j \rightarrow i}^{m,\omega}$ need to be bounded. $E_{j \rightarrow i}^{c,\omega}$ and $E_{j \rightarrow i}^{m,\omega}$ are modified forms of $E_{j \rightarrow i}^{c}$ and $E_{j \rightarrow i}^{m}$ under cache state. Therefore:

$$E_{j \rightarrow i}^{q} = \sum_{\omega \in \Omega_i} E_{j \rightarrow i}^{q,\omega} q \in \{c, m\} \tag{6.3}$$

If there is no branch at the end of B_j, $E_{j \rightarrow i}^{m} = 0$. Eventually, WCET problem will

be solved using ILP to maximize ET.

$$WCET = maximize(ET) \tag{6.4}$$

This formula is modified for the P-FRP system in order to derive the WCET. For this purpose, some changes need to be proposed. In the proposed FRP system, instead of cache, SPM is used for the reasons discussed before. Besides, at the time of executing a task, it is needed to determine if the task has already been copied into SPM. After modifying the ILP objective function defined in [31], the following adapted objective function is derived:

$$\begin{aligned} WCET = max(L_\tau^R + \\ \sum_{i=1}^{N} \sum_{j \to i} t_{j \to i}^c * E_{j \to i}^c + t_{j \to i}^m * E_{j \to i}^m + \\ L_\tau^W), \end{aligned} \tag{6.5}$$

where L_τ^R and L_τ^W are the costs of reading/writing from/to memory using DMA. The costs of DMA read and write requests have been added to the first and last basic block of the task, respectively. The latency of read and write by DMA is constant. $t_{j \to i}^c$ and $t_{j \to i}^m$ are the execution times of each block which have been calculated using out-of-order execution and branch prediction. Hence, these are also constants. Since an SPM with predictable latency has been used in the proposed P-FRP system instead of the cache, the number of transitions from one block to another,$E_{j \to i}^c$ and $E_{j \to i}^m$, are bounded only by loops and user constraints. Therefore, based on Equation 6.5, the total WCET of a task in P-FRP is bounded.

Chapter 7

Response Time Analysis of P-FRP

In order to analyze the timing of tasks in a P-FRP system, it is necessary to evaluate the WCRT of such tasks in addition to calculating the WCET. WCRT specifies the schedulability of a task set, and the response time of the lowest-priority task in the set. Resolving the actual value of WCRT in P-FRP demands one to evaluate all the release scenarios of a task set which is exponential as the number of tasks in a task set increases. However, in order to analyze the timing of P-FRP systems, it is a necessity to see the effects of the abort-and-restart paradigm on WCRT. A couple of researches have been developed for calculating the value of WCRT. Belwal et al., [9] have presented an exhaustive gap-enumeration method for calculating the response time for P-FRP tasks. Their approach is extended to compute the WCRT under all release scenarios of a task set. Although the aforementioned method can compute an accurate response time based on only CPU costs, it does not consider the extra memory costs caused by aborting a task. In the above approach, the abort

and preemption cost of lower-priority tasks are both only the execution penalty of preemption by a higher-priority task. To determine the WCRT in P-FRP, considering the memory is vital. When preempted by a higher-priority task, the lower-priority task will be aborted, and so depending on the state of SPM, the task may require to be read from the memory again.

7.1 WCRT Derivation Technique

in order to calculate an accurate value of WCRT, Belwal et al's gap-enumeration approach is exploited for calculating the response time of P-FRP task and extend it to consider memory latency in an SPM-based platform. Under each release scenario, the WCRT of each task set is calculated from the execution time of the tasks, the time the task code and its data are read from main memory to SPM, and finally, the time it takes for DMA operations to write the changed data back to memory. Despite WCRT, to compute the WCET of a task, the release scenario of tasks in the task set does not affect the results. Therefore, after computing the WCET of an isolated task, the CPU execution time of the task along with the memory operation times will be sent to the WCRT algorithm as input arguments. To further demonstrate, the method for computing the WCRT in Algorithm 1 along with an example is provided.

Algorithm 1 Algorithm to calculate the WCRT of a task τ in an SPM-based P-FRP system.

1: Inputs: $taskset$, $SPMSIZE$, $Lasttime$
2: $WCRT \leftarrow 0$
3: **for** $every\ release$ **do**
4: $temps \leftarrow 0$
5: $\tau_{low} = GetLowestpriority(taskset)$
6: **for** $time\ unit\ from\ 0\ to\ Lasttime$ **do**
7: $\tau = GetReadyHighestpriority(taskset)$
8: **if** $map(\tau)$ **then**
9: $\tau_{WCET} \leftarrow \tau_{Exe} + \tau_{Data} + \tau_{Write}$
10: **else**
11: $\tau_{WCET} \leftarrow \tau_{Exe} + \tau_{Code} + \tau_{Data} + \tau_{Write}$
12: **end if**
13: $\tau_{exesteps} + 1$
14: $update\ temp_{\tau}$
15: **if** $\tau_{exestep} == \tau_{WCET}$ **then**
16: $Remove\ \tau\ from\ queue$
17: **end if**
18: **end for**
19: **end for**
20: $WCRT = max(temps)$
21: return $WCRT$

7.2 An Example of WCRT under the Abort-and-Restart Model

Assume a task set of size 3, and an SPM that can fit only two tasks. The characteristics of the tasks in the set are shown in Figure 7.1. The highest-priority task is τ_3 and the lowest-priority one is τ_1. Assume τ_1 is released at time 0. At the beginning of the analysis, the SPM is empty. Therefore, the WCET of τ_1 will be 3 and it will be inserted into SPM. This value is the sum of τ_1's execution time and memory operations. τ_1 will execute until time 2 and is preempted by the higher-priority task τ_3. The current state of the SPM only contains the information of τ_1. Thus, τ_3 will be read into SPM with the WCET of 4. At time 6, τ_3 has finished its execution, and τ_1 starts with WCET of 2 since it is already located in SPM, and the task code will not be read again. τ_1 executes until time 7. τ_2 is not located in SPM, SPM is full so it will replace the least recently used task τ_3. The execution continues until 12, at this time τ_1 will begin the execution of 2 and finishes at time 14. Thus, the response time in this example is 12. The timing diagram of the example is shown in Table 7.1 and Figure 7.1.

To calculate the WCRT of the lowest-priority task in the task set, the highest effective value for SPM size is computed in 7.1.

Theorem 1 (Effective SPM size)**.** *In order to calculate the WCRT of a task set in an SPM-based platform, the maximum effective value for SPM size is:*

$$SPMSize = TasksetSize - 1. \tag{7.1}$$

Table 7.1: The characteristics of the tasks in Figure 7.1.

TID	Priority	Period	Release offset	CPU Execution time	Code Read time	Data Read time	Data Write time
τ_1	3	17	0	1	1	1	0
τ_2	2	8	7	2	1	1	1
τ_3	1	20	2	1	1	0	2

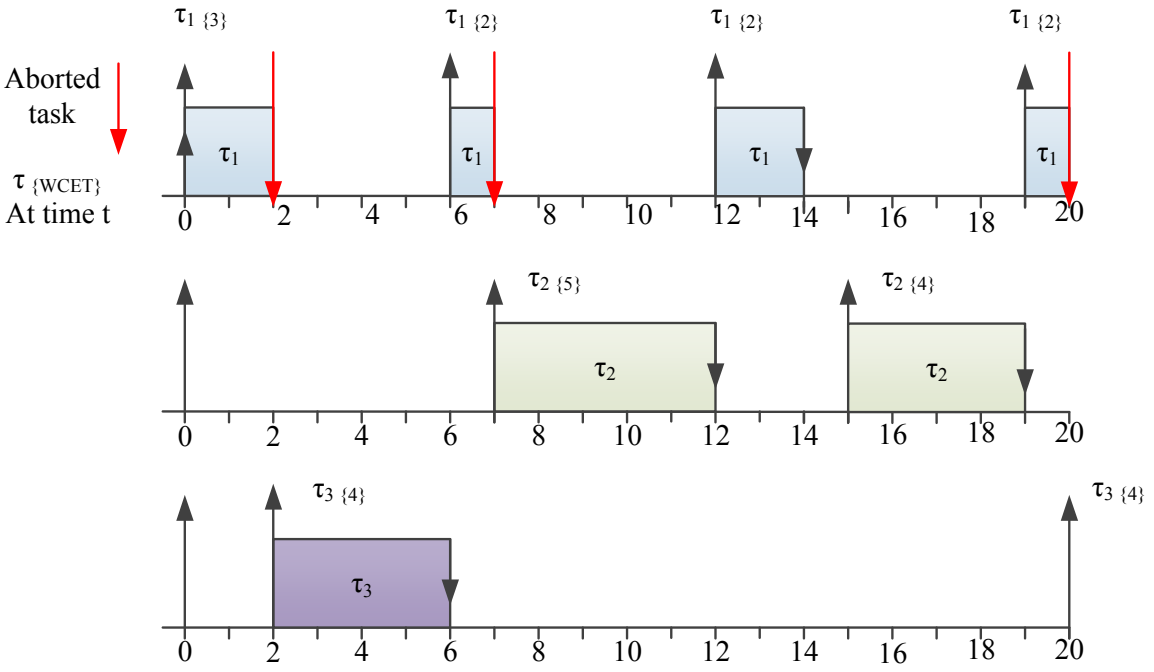

Figure 7.1: An example of the response time of the lowest-priority task in a task set of size three in the abort-and-restart model.

Proof. Assume there are n tasks in the task set. To compute the WCRT of the lowest-priority task τ_n, the $n-1$ highest-priority tasks need to be executed. Therefore, the

effective size of SPM is at least $n-1$. Since the WCRT of τ_n is the total amount of time to respond to the lowest-priority task, after execution of $n-1$ tasks, the state of SPM does not affect the result and thus for evaluating the WCRT, there is no need to increase the SPM's size more than the size of the first executed $n-1$ tasks. □

7.3 Blocking Latency in P-FRP

As discussed in Section 5, at the end of each task execution, all the changes should be committed to memory using atomic DMA writes. The DMA write and running task will be temporarily assigned the highest priority to ensure the write-back is atomic. If during the task's commit, a new task with a priority higher than the original priority of the running task, arrives, it needs to be delayed until DMA commit is completed. Therefore, this blocking time will increase the response time of the new task. Thus, the blocking latency for the newly arriving task τ_j, waiting for task τ_i to commit, will be:

$$\begin{aligned} Blocking_{\tau_j} = & L^W_{\tau_i} + \\ & DMA_{request} + \\ & Committime_i - arrival_j, \end{aligned} \tag{7.2}$$

where $L^W_{\tau_i}$ is the latency of the committed data, $DMA_{request}$ is the time it takes for the DMA to respond to a write request, $Committime_i$ is the start of the data commit of task τ_i, and $arrival_j$ is the time τ_j arrives.

```
1 int binary_search
2   (int x, int * data, int n){
3   int fvalue, mid, up, low ;
4   low = 0; up = n - 1; fvalue = -1;
5   while(low <= up){
6     mid = (low + up) >> 1;
7     if(data[mid].key == x){
8       up = low - 1;
9       fvalue = data[mid].value;
10    }
11    else{
12      if(data[mid].key > x){
13        up = mid - 1;
14      }
15      else{
16        low = mid + 1;
17      }
18    }
19  }
20  return fvalue;
21 }
```

BinarySearch.c

Figure 7.2: A simple binary search program implemented in C.

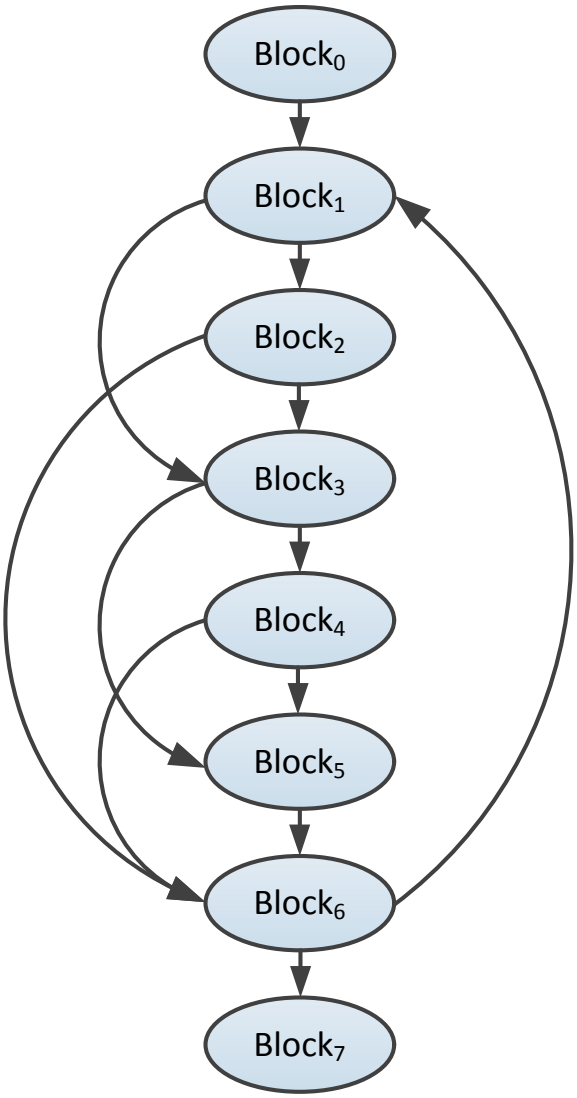

Figure 7.3: The CFG of binary search program implemented in C. As shown in the CFG, there are forward and backward edges in the graph which shows the control flow of different paths in the program. $Block_0$ is the entry block and $Block_7$ is the final block of the CFG.

Chapter 8

Upper-bound on Response Time

Resolving the actual value of WCRT in P-FRP demands one to evaluate all the release scenarios of a task set which is exponential as number of tasks in a task set increases. Therefore, in addition to using brute-force evaluation, an approximation of the upper bound on the response time presented by Belwal et al., [11], has been extended in this thesis. Belwal et al., propose a new algorithm (UBRT) for determining the upper bound for response time of P-FRP tasks. They first define an algorithm to calculate the lower bound on the response time and then introduce the improved upper bound algorithm based on the lower bound analysis. Minimum value of response time of task τ_i is calculated using equation below:

$$(LBRT_i)^{n+1} = C_i + \sum_{\forall j \in hp_i} \lceil \frac{LBRT_i^n}{T_j} \rceil . (C_j + 1), \tag{8.1}$$

where, $LBRT_i^0$ is set to C_i. hp_i is a set of all tasks that have higher priority than τ_i and T_j is the arrival time of task τ_j. You can find the proof of Equation 8.1 in [11].

The maximum abort cost a higher-priority task may cause to task τ_i is $C_i - 1$. Since for task τ_i to be aborted, it is required that τ_i finishes at least one unit of time from its execution time. Thus, the maximum cost of aborting the task is $C_i - 1$. Maximum preemption cost for τ_i is the sum of its abort cost and the computation time of the higher-priority task τ_j, therefore $C_i - 1 + C_j$.

The pessimistic assumption made in the computation of the upper bound is that each job of the higher priority tasks will add maximum preemption cost to the response time of τ_i. Therefore, by computing the abort cost a higher priority task τ_j can cause to all the lower priority tasks, the maximum preemption cost by higher-priority task τ_j is sum of computation time of τ_j and abort cost of all lower-priority tasks: $C_j + (max \sum_{m=1}^{k-1} C_m - 1)$.

To better understand upper bound algorithm, the pseudo-code of UBRT algorithm, is presented in Algorithm 2. In the given algorithm, T_a is the arrival time period of two successive jobs of τ_a.

Although the aforementioned algorithm can compute an efficient upper bound for the response time analysis, it does not consider the extra memory costs caused by aborting a task. In the above algorithm, the abort and preemption cost of lower-priority tasks are both only the execution penalty of preemption by a higher-priority task. Though to determine the upper bound on response time in P-FRP, considering the memory is vital. Therefore, the memory read and write cost, and the DMA request latency are added to the value of WCET (C_i) in the UBRT algorithm to make the upper-bound more realistic.

Algorithm 2 Algorithm to calculate the upper bound on response time of a task τ in P-FRP systems.

Inputs: $taskset_n$, $LBRT_j$

$UBRT_j \leftarrow LBRT_j$

for $a \leftarrow j+1$ *to* n **do**

  $Cost_{newjobs} \leftarrow 0$

  for $b \leftarrow j+1$ *to* $a-1$ **do**

    $jobs_b \leftarrow \lceil (UBRT_j - OldUBRT_j)/T_b \rceil$

    $Cost_{newjobs} \leftarrow Cost_{newjobs} + jobs_b.(\max(\sum_{k=1}^{b} C_k - 1))$

  end for

  $UBRT_j \leftarrow UBRT_j + Cost_{newjobs}$

  if $(a < n+1)$ **then**

    $jobs_a \leftarrow \lceil UBRT_j/T_a \rceil$

    $OldUBRT_j \leftarrow UBRT_j$

    $UBRT_j \leftarrow UBRT_j + jobs_a.(\max(\sum_{k=1}^{a} C_k - 1))$

  end if

end for

return $UBRT_j$

Chapter 9

Evaluation

The approach is evaluated by comparing the maximum simulated value of the execution time of each task with the estimated WCET using the modified ILP objective function in Equation 6.5 and then, the calculated WCRT under different SPM sizes is experimented. In order to compute the maximum execution time of each task, the SimpleScalar/PISA is used which is a computer architecture simulator developed by Austin [12], [6]. Simplesim3.0e version with the PISA instruction set is used. This simulator receives the P-FRP programs written in C and a configuration file. In the configuration file, the micro-architecture specifications are described. The results are evaluated in a system including SPM, in-order execution, branch prediction unit, and pipelining. The final value of the simulation for a task execution is the maximum execution time of 100 simulations of the task. The memory latency and the cost of DMA operations caused by preemption are also added to the simlescalar source code to adapt it to the P-FRP abort-and-restart model. Since the waiting time of a

higher-priority task during data commit was short in the evaluated benchmarks, to calculate the WCRT, the blocking time computed in Equation 7.2 has been assigned to 0 for all the tasks in the task sets.

In this work, the tick-based approach has been followed; only the scheduling decisions on every time unit (tick) are allowed. To analyze the WCRT of P-FRP tasks, it is essential to calculate every possible release scenario. The numbers of release scenarios will increase prohibitively if a non-tick-based scheduling is allowed. Therefore, to prevent additional WCRT derivation complexities, the tick-based policy is followed in this thesis.

9.1 Experimental Results for the WCET

Table 9.1: The comparison of simulation and estimation of benchmarks under perfect branch prediction.

Benchmarks	Simulation (cc)	Estimation (cc)	Read data (B)	Committed data (B)	Memory Latency (cc)	Ratio
fib	145	155	164	4	71	1.07
cnt	5362	6546	840	416	570	1.22
insertsort	1142	1518	496	44	245	1.33
crc	22213	23666	1108	4	504	1.07
matmul	3241	5152	752	144	406	1.59
qurt	653	1780	1072	32	501	2.73
bs	366	553	380	4	173	1.51
jfdctint	4560	4580	2952	256	1457	1.00

Although the characteristics of P-FRP systems are described in the real-time domain in this thesis, tasks in such systems are similar to the ones in conventional

Table 9.2: The comparison of simulation and estimation of benchmarks under combination branch prediction.

Benchmarks	Simulation (cc)	Estimation (cc)	Read data (B)	Committed data (B)	Memory Latency (cc)	Ratio
fib	152	170	164	4	71	1.12
cnt	5454	6833	840	416	570	1.25
insertsort	1188	1548	496	44	245	1.30
crc	25754	29339	1108	4	504	1.14
matmul	3390	5473	752	144	406	1.61
qurt	657	1914	1072	32	501	2.91
bs	386	623	380	4	173	1.61
jfdctint	4578	4616	2952	256	1457	1.01

Table 9.3: The comparison of simulation and estimation of benchmarks under instruction fetch queue size of 1.

Benchmarks	Simulation (cc)	Estimation (cc)	Read data (B)	Committed data (B)	Memory Latency (cc)	Ratio
fib	144	226	164	4	71	1.57
cnt	5362	11175	840	416	570	2.08
insertsort	1142	2352	496	44	245	2.06
crc	22212	44723	1108	4	504	2.01
matmul	3240	7225	752	144	406	2.23
qurt	653	1951	1072	32	501	2.99
bs	366	834	380	4	173	1.28
jfdctint	4560	6950	2952	256	1457	1.52

real-time systems. For instance, in an autonomous car, assume there is a task that warns an alarm signal when the car is too close to an obstacle. To design an autonomous car in a P-FRP system, the same task needs to be implemented again. However, as mentioned in the prior sections, the memory latency of each interference and preemption should be considered. Therefore, to evaluate this approach,

real-time tasks are used and their WCET and the WCRT in an abort-and-restart paradigm are analyzed. 8 benchmarks of different sizes are selected from the SNU real-time benchmarks [36] to compare the values of maximum execution time and the estimated WCET.

Tables 9.1, 9.2, and 9.3 compare the simulation results with WCET estimation of the selected benchmarks (tasks) under three different micro-architectures. Table 9.1 shows the comparison of simulation and estimation under a system with a perfect branch prediction unit. Table 9.2 shows the same comparison under a system with a prediction unit that combines a bi-modal and a 2-level predictor. Both systems execute with instruction fetch queue of size 4. Table 9.3 shows the same comparison under a system with an instruction fetch queue of size 1. These tasks are fib, insertsort, crc, cnt, matmul, qurt, bs, jfdctint selected from the SNU library. The data read from memory to SPM, and data committed back to memory in Bytes can be observed in Tables 9.1, 9.2, and 9.3. To evaluate the accuracy of the estimated WCET, the metric *AccuracyRatio* is defined for each task τ as follows:

$$AccuracyRatio = Estimation_{\tau}/Simulation_{\tau} \tag{9.1}$$

The cost of reading/writing N bytes from/to SPM via DMA are modeled as described in Equations 4.2 and 4.3. Every time task τ arrives, its data and code should be loaded into the SPM. Since computing the WCET of a task is not affected by the preemptions and the state of the SPM, the code read time is included in computing the WCET. On the other hand, the value of the WCRT is affected by the SPM state

which will be explained in section 9.2.

As anticipated, the estimated WCET's of each task are greater than the corresponding simulation results. Therefore, the ratio is always greater than 1. The average AccuracyRatios for the evaluated benchmarks in all the architectures are 1.23, 1.50, and 2.09. The maximum AccuracyRatios are 2.73, 2.91, and 2.99 and the minimums are almost 1, 1.01, and 1.52. In the evaluated benchmarks, qurt has the highest ratio. The reason for this jump in the ratio is that the maximum value of the WCET has been derived from the simulation of qurt by 100 times, and this value may not be the actual value of the WCET. In addition, the estimation and simulation of tasks under a combination branch prediction are higher than a perfect prediction as predicted. The average AccuracyRatio for Table 9.3 is higher than those of the other two architectures and the reason is due to the pessimism in selecting the constraints to estimate the WCET. Thus, there is a trade-off between deriving a tight upper bound on the WCET and a safe estimate for different architectures.

9.2 Experimental Results for the WCRT Evaluation

In order to evaluate the timing of a P-FRP task in an SPM-based platform, the enhanced WCRT have been analyzed using both synthetic tasks and real-time tasks from the SNU benchmark suite. Since the Code Read time of the SNU benchmarks compared to their execution time is insignificant, minor changes in the value of

WCRT are expected under different SPM sizes. Therefore, synthetic task sets are generated to evaluate the model further. To analyze the synthetic tasks, two randomly generated groups of task sets of size 3 and 4 have been evaluated. The WCRT computation is trivial for task sets of size 2; therefore, they are not included in this work. Each task has been given a random period, CPU execution time, data read time, data write time, Code Read time. The periods of both task sets are between 40 and 110. The range of CPU execution time, and CodeRead, data read and write is from 1 to 20 time units. For each task size, a sample of 100 task sets are evaluated under different SPM sizes. For task sets of size 3, the evaluated sizes are 1 and 2 tasks. The SPM sizes for task sets of size 4 are 1, 2, and 3. The effective SPM size is already discussed in Theorem 1.

To show the impact of SPM size on real tasks, two groups of task sets of size 3 and 4 from the SNU real-time benchmarks are also created under different SPM sizes. Each group contains 100 task sets. The derived estimated WCET, read data, code, and write data time are used and 100 tasks for each task set with variable release times and priorities have been generated. Since the Code Read time of the evaluated benchmarks are very small compared to their execution time, different SPM sizes do not impact significantly on their WCRT results. Δt is the difference of WCRT of a task set under SPM sizes of 1 and 2 or 2 and 3. In the task sets of size 3, the value of WCRT under SPM size of 1 and 2 decreased for 10% of the task sets. The average change in the WCRT is around 2%. The maximum change is around 8%. For task sets of size 4, the Δt for SPM sizes 1 and 2, decreased for 78% of the task sets. The average change is around 2%, the maximum change is 6%. For SPM sizes of 2 and

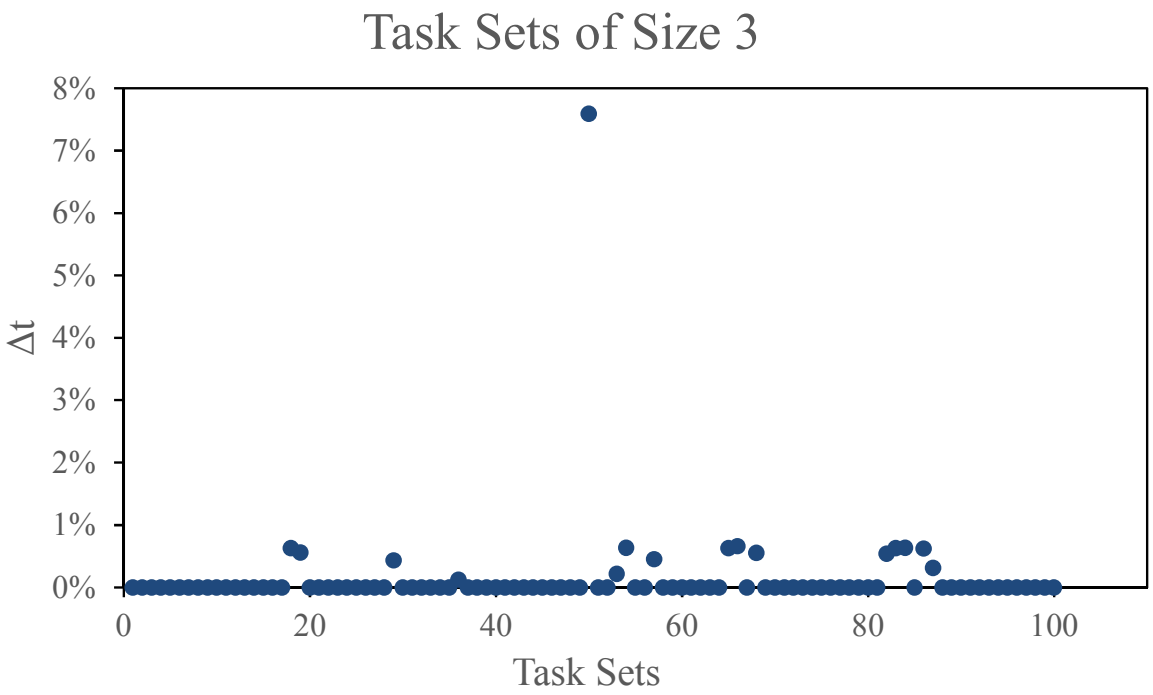

Figure 9.1: The difference of WCRT of task sets from SNU benchmark with size 3 under SPM size of 1 and 2. $\Delta t = WCRT_1 - WCRT_2$.

3, 76% of the task sets have smaller WCRT. The average Δt is around 2% and the maximum is also around 2%. Figures 9.1, 9.2, and 9.3, show the distribution of Δt for different task sets.

Figures 9.4, 9.5, and 9.6 show the comparison of WCRT derivation of 100 randomly generated task sets under different SPM sizes. As shown in Figure 9.4, the value of WCRT has decreased for 87% of the tasks. The average change in the value of WCRT is around 7%. The maximum change is 46%. As the size of the SPM increases, fewer task codes will be read into memory, resulting in smaller WCET and therefore WCRT. In addition, since the value of the WCET of the tasks decreases during the calculation of the WCRT, several previously unschedulable task

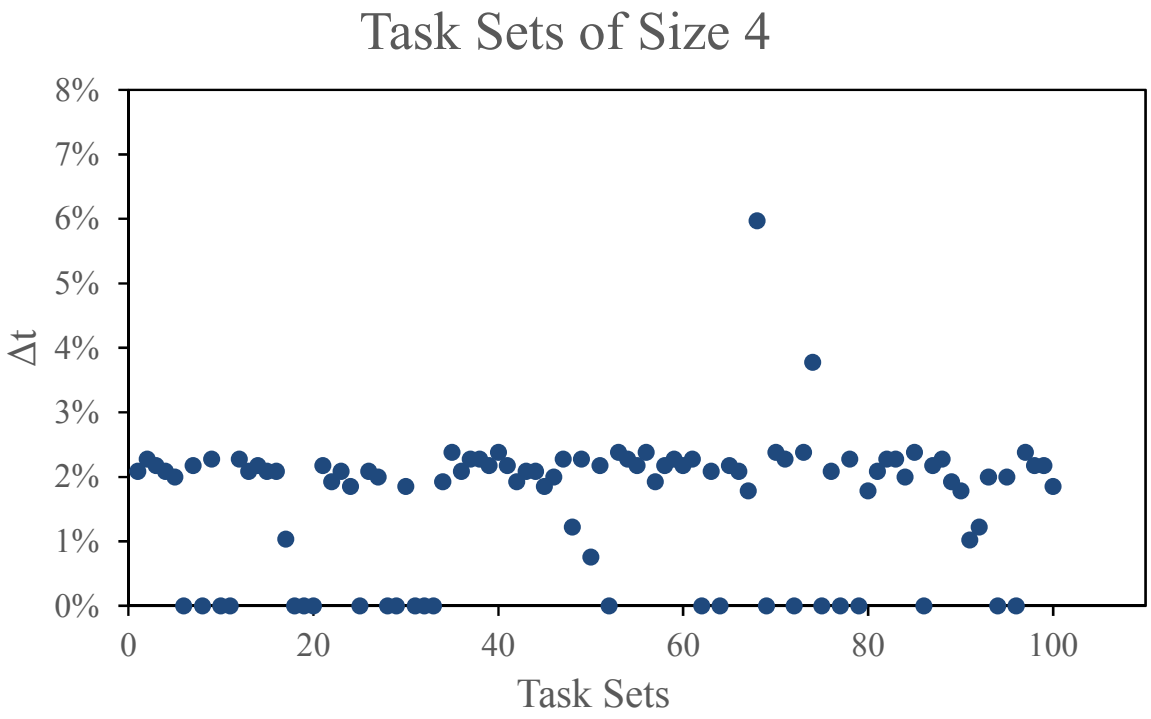

Figure 9.2: The difference of WCRT of task sets from SNU benchmark with size 4 under SPM size of 1 and 2. $\Delta t = WCRT_1 - WCRT_2$.

sets become schedulable. The number of unschedulable task sets has been reduced by 1.7%. Figure 9.5 and 9.6 show the change in WCRT of task sets of size 4 by increasing the SPM size from 1 to 3. By increasing the SPM size from 1 to 2, the value of WCRT decreases for 51% of the task sets. The average Δt is around 3% and the maximum change is 33%. The number of unschedulable task sets has been reduced by 2%. After increasing the size of the SPM to 3, the WCRT of 100% of the task sets decremented. The average Δt is around 10% and the maximum change is 46%. The number of unschedulable task sets has been reduced by 1%. As expected, the value of WCRT will decrease more for the task sets with larger Code Read times because the cost of reading the task's code will be avoided if it is already located in

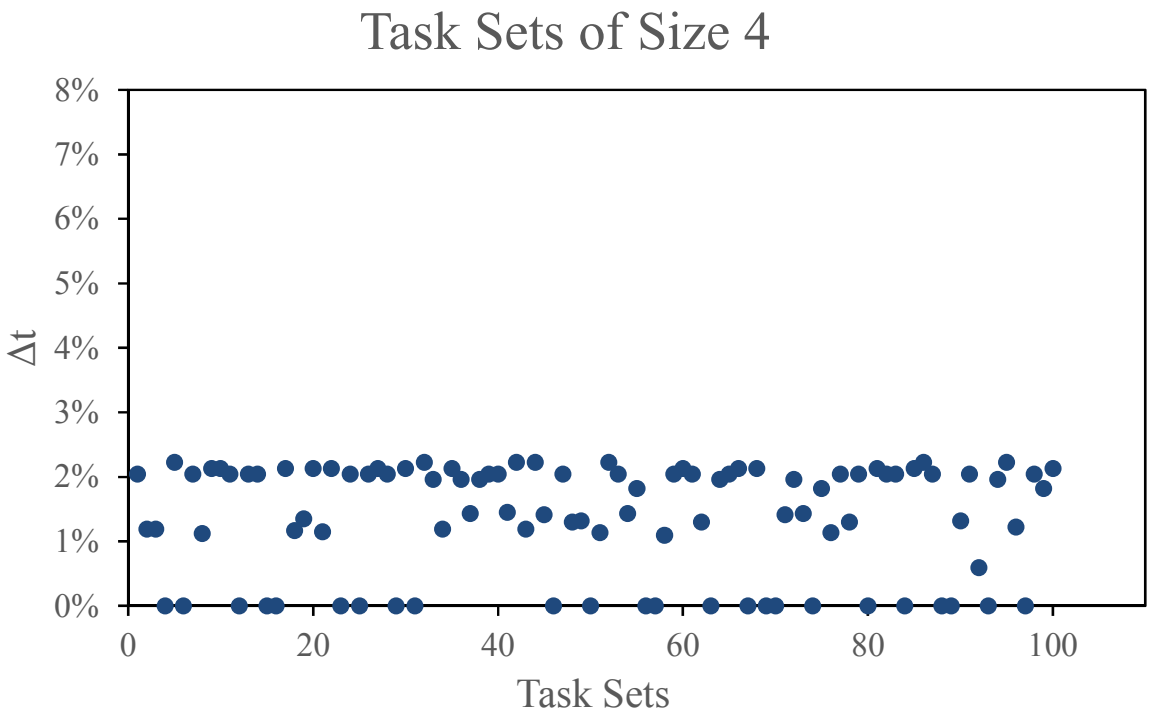

Figure 9.3: The difference of WCRT of task sets from SNU benchmark with size 4 under SPM size of 2 and 3. $\Delta t = WCRT_2 - WCRT_3$.

the SPM.

9.3 Experimental Results for the UBRT Algorithm

After evaluating the WCRT of each task, the upper bound on the response time of P-FRP tasks is also evaluated using the computed WCET with the memory cost by using the algorithm explained in Chapter 8. In order to evaluate the UBRT algorithm, the result is compared with the value of WCRT derived in Section 9.2. The

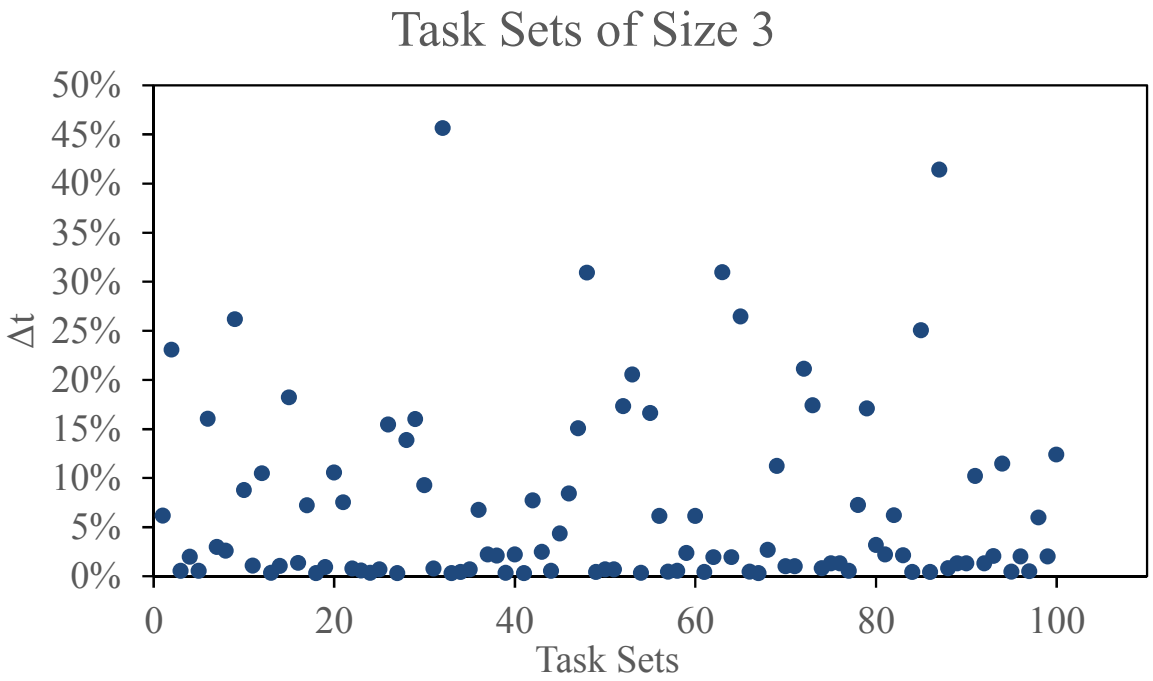

Figure 9.4: The difference of WCRT of randomly generated task sets with size 3 under SPM size of 1 and 2. $\Delta t = WCRT_1 - WCRT_2$.

correctness of estimation on the response time upper bound is verified by running selected benchmarks on the enhanced UBRT. The enhanced UBRT has been evaluated using two groups of tasks sets of size 3 and 4 from the evaluated benchmarks from the previous section. Each task set is unique, since the arrival time and priority of each task are different than others and selected from [10,70]. The computation time of each task is the WCET calculated in section 6.2. An accuracy factor for each task set, is defined as follows:

$$Accuracy\ Factor = UBRT/WCRT.$$

Figures 9.7, 9.8, 9.9, and 9.10 show the Accuracy Factor calculated for both groups of low-utilization and high-utilization tasks in 3 and 4 tasks sets. Every task set,

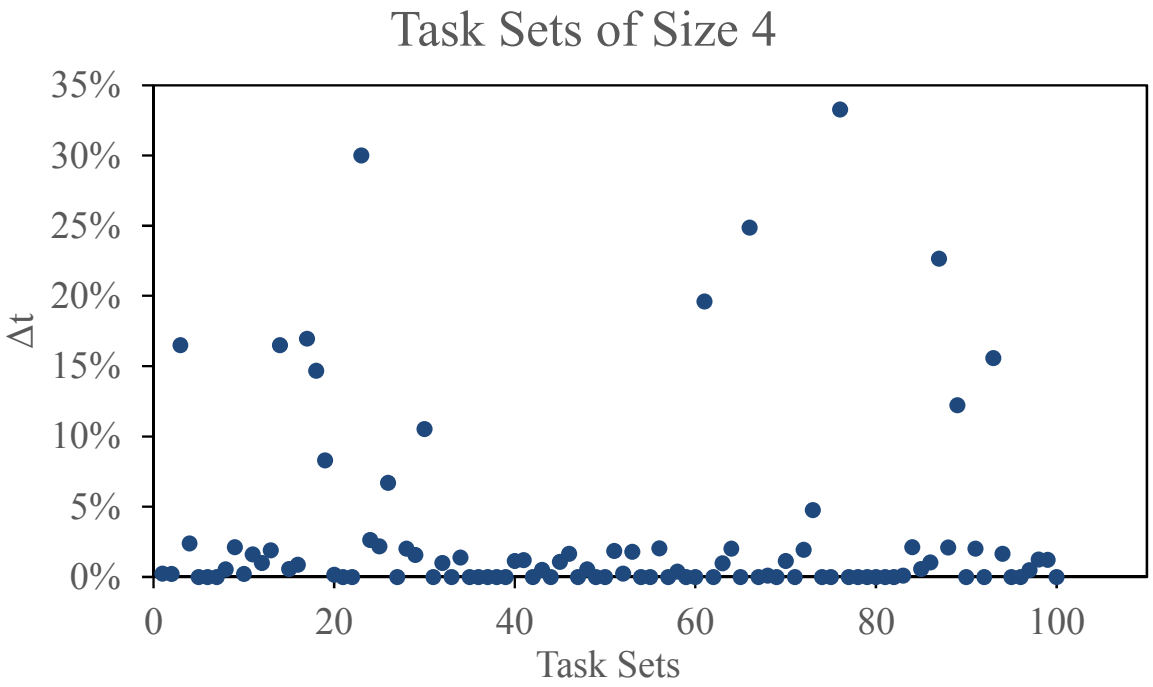

Figure 9.5: The difference of WCRT of randomly generated task sets with size 4 under SPM size of 1 and 2. $\Delta t = WCRT_1 - WCRT_2$.

contains two groups of data of 100 task release scenarios, one group with a utilization greater than 0.5 and the other with utilization less than 0.5.

The average Accuracy Factor for Figures 9.7, 9.8, 9.9, and 9.10 is around 1.41, 3.10, 1.10, and 2.55 respectively. In general, for the task set with a higher utilization, and fewer tasks, the Accuracy Factor is very close to 1. This result is expected since, when the number of tasks increases, the abort cost, and cost of preemption will increase. Furthermore, for the group of tasks with higher utilization, the interference between tasks increases and thus preemption costs are much closer to the upper bound considered in UBRT algorithm. Hence, by comparing 9.9 and 9.10 with 9.7 and 9.8, it can be realized that the accuracy ratio will be closer to 1 when the

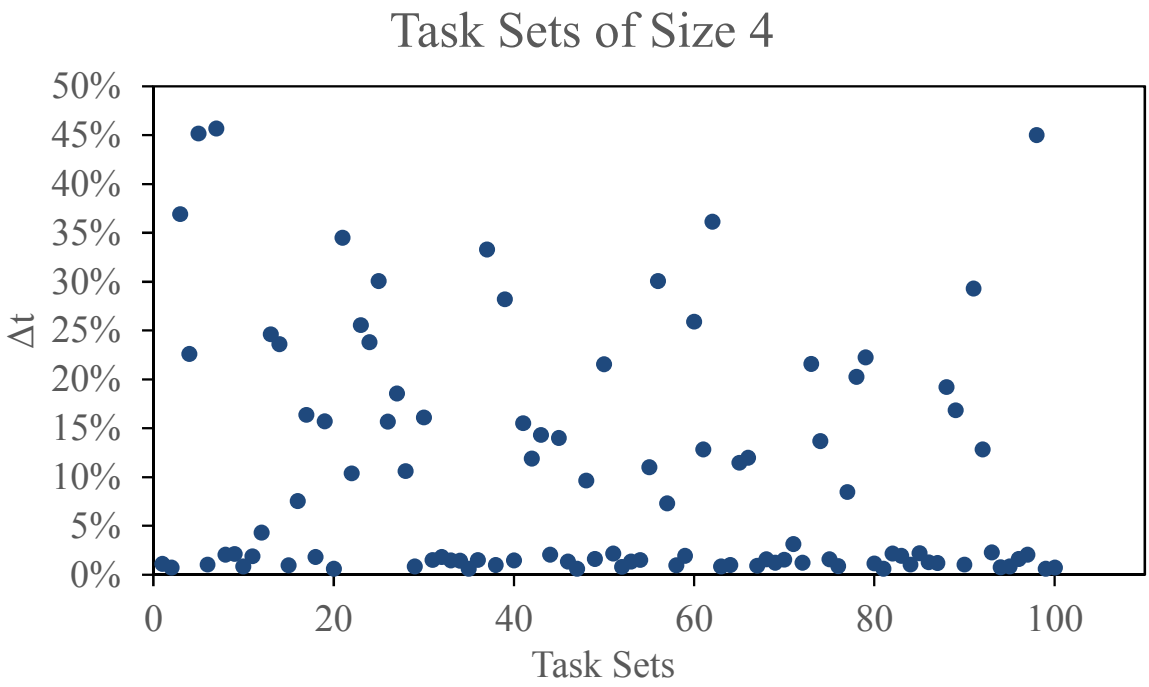

Figure 9.6: The difference of WCRT of randomly generated task sets with size 4 under SPM size of 2 and 3. $\Delta t = WCRT_2 - WCRT_3$.

average utilization of tasks in a task set is higher because with a higher utilization, the interference between lower-priority and higher-priority tasks will be more. Thus, the WCRT will increase and get closer to the upper bound calculated by enhanced UBRT algorithm.

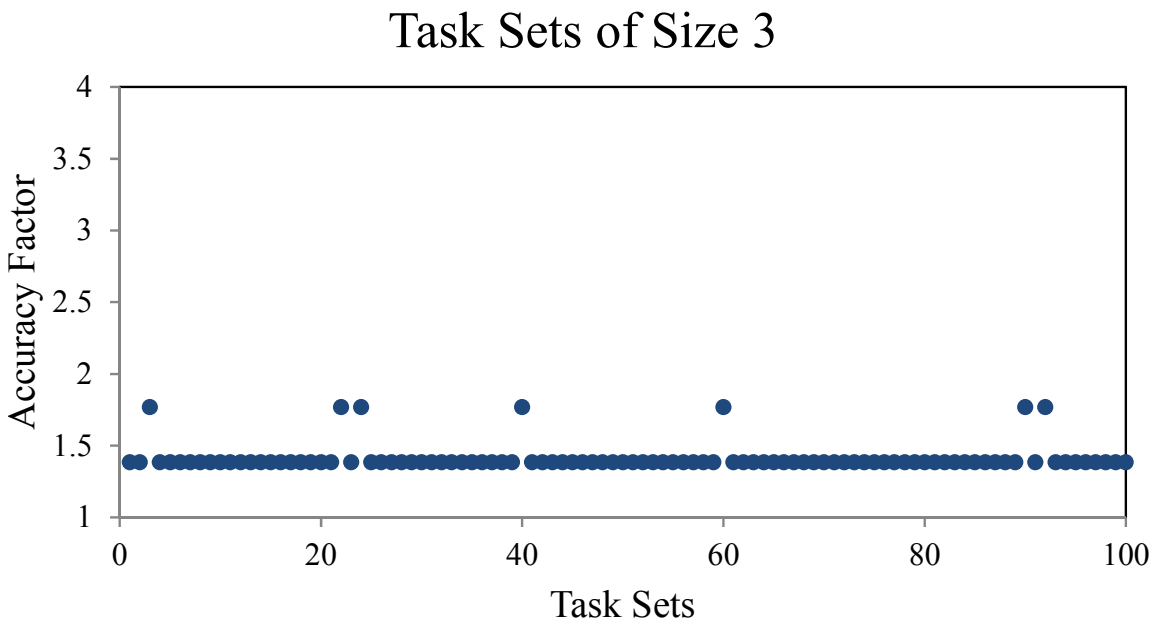

Figure 9.7: The Accuracy Factor for task sets with utilization <= 0.5. This figure shows the Accuracy Factor for three-sized task sets, where the utilization of each set is less than 0.5. The tasks have been selected from the SNU real-time benchmark suite, and 100 task sets have been evaluated. Each task set has three tasks.

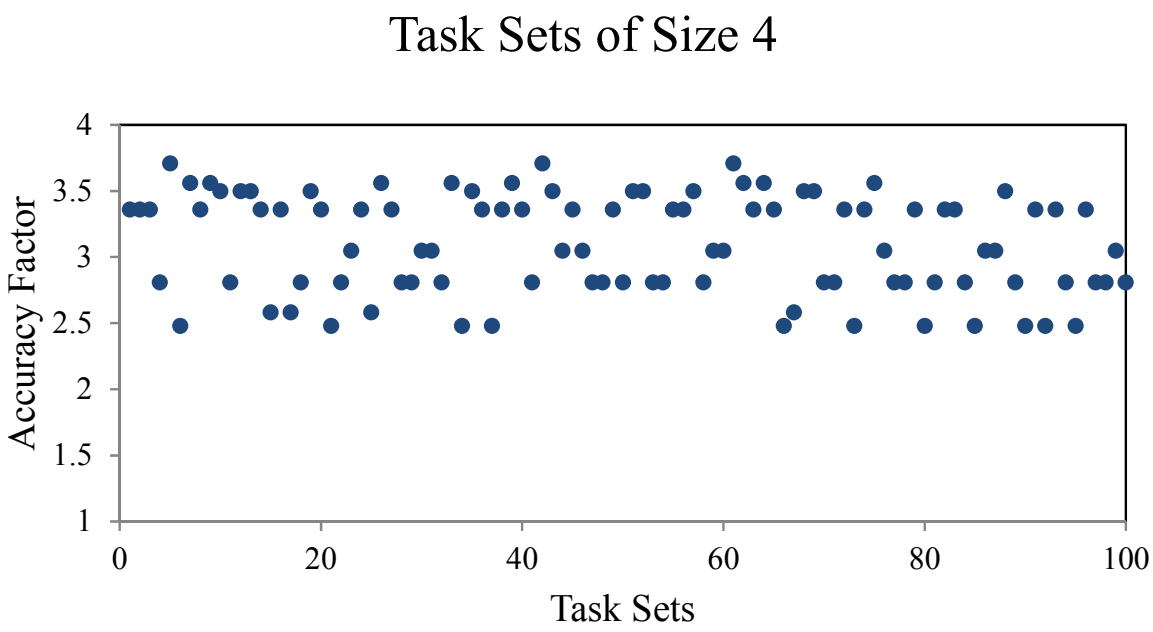

Figure 9.8: The Accuracy Factor for task sets with utilization <= 0.5. This figure shows the Accuracy Factor for four-sized task sets, where the utilization of each set is less than 0.5. The tasks have been selected from the SNU real-time benchmark suite, and 100 task sets have been evaluated. Each task set has four tasks.

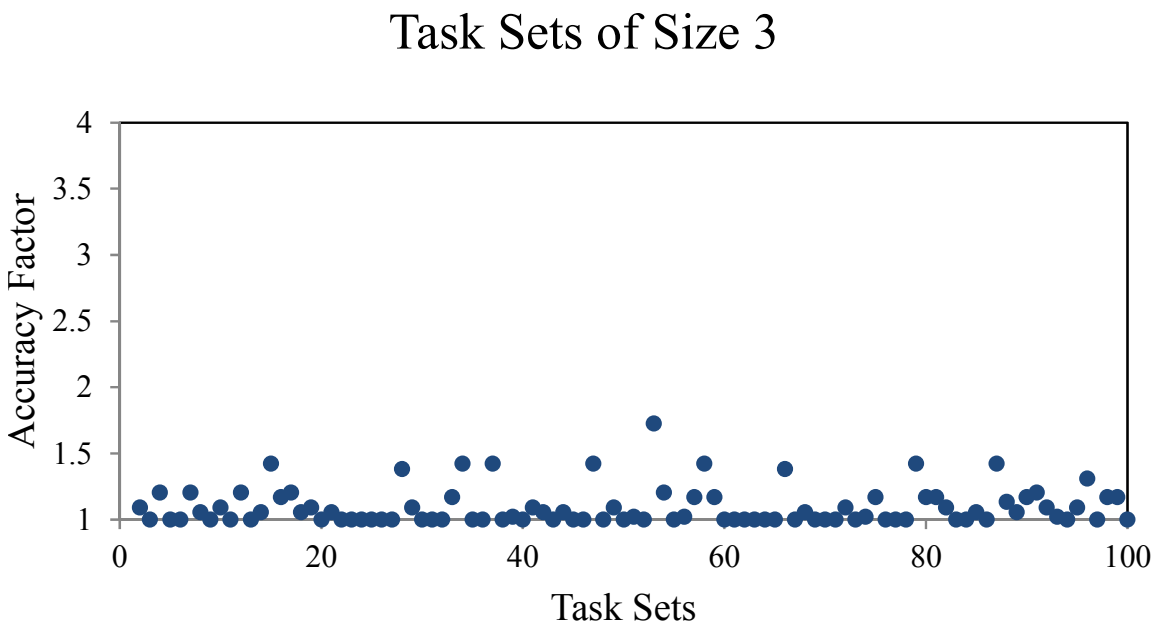

Figure 9.9: The Accuracy Factor for task sets with utilization >= 0.5. This figure shows the Accuracy Factor for three-sized task sets, where the utilization of each set is greater than 0.5. The tasks have been selected from the SNU real-time benchmark suite, and 100 task sets have been evaluated. Each task set has three tasks. As shown in this figure, the average Accuracy Factor for these highly-utilized tasks compared to the 3-sized task sets with low utilization is lower. The lower Accuracy Factor shows that the UBRT algorithm generated a tighter bound on the WCRT compared to Figure 9.7.

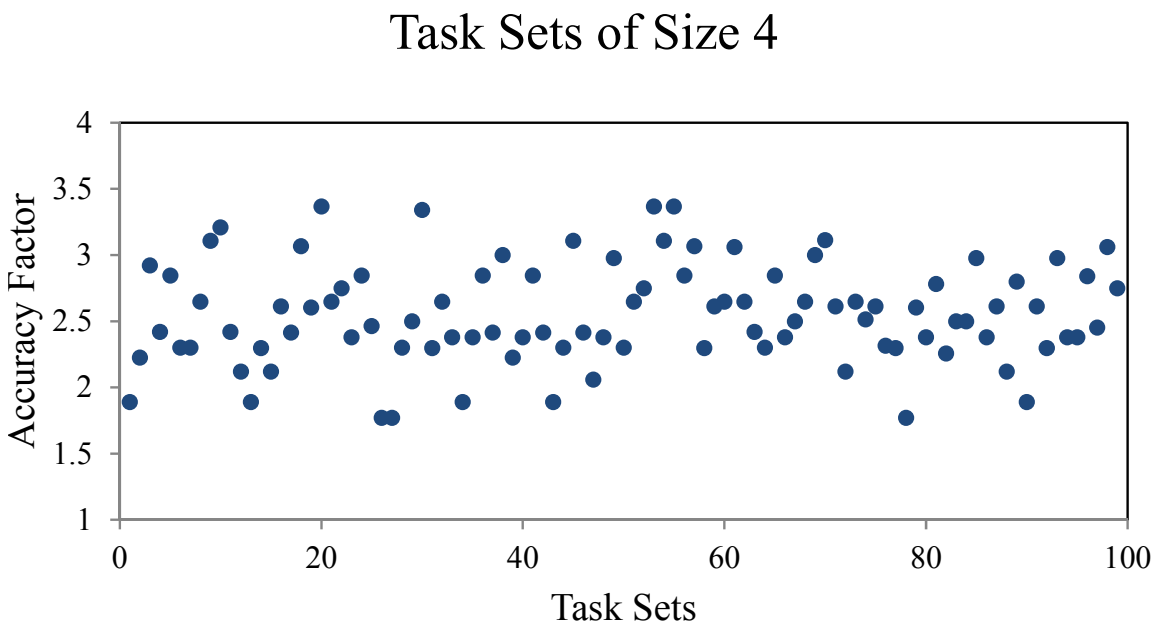

Figure 9.10: The Accuracy Factor for task sets with utilization >= 0.5. This figure shows the Accuracy Factor for four-sized task sets, where the utilization of each set is greater than 0.5. The tasks have been selected from the SNU real-time benchmark suite, and 100 task sets have been evaluated. Each task set has four tasks. As shown in this figure, the average Accuracy Factor for these highly-utilized tasks compared to the 4-sized task sets with low utilization is lower. The lower Accuracy Factor shows that the UBRT algorithm generated a tighter bound on the WCRT compared to Figure 9.8.

Chapter 10

Conclusion

10.1 Conclusion

In this thesis, an SPM-based execution platform for functional reactive systems is introduced. The properties of the SPM, its benefits for solving the WCET analysis problem, and SPM management are discussed. In the next step, a tight bound on the WCET of P-FRP tasks using static analysis is calculated. The WCET is derived by modifying the ILP objective function computed for the WCET of tasks in conventional systems to make it applicable to P-FRP. Using the derived function, a tight bound on the WCET for P-FRP tasks is estimated under the abort-and-restart model. The SPM-based abort-and-restart paradigm has been applied to calculate the WCRT of tasks under variable SPM sizes by considering memory latency resulting from the preempting higher-priority tasks.

In order to test the approach, both synthetic tasks and tasks from the SNU real-time benchmark set are experimented. By analyzing the results for the WCET derivation, a very tight upper bound on the WCET is reached. After checking the WCET estimation, the WCRT algorithm is evaluated with memory penalty consideration. it is shown that when the tasks code size increases, under an increasing SPM size, the value of WCRT decreases compared to similar tasks with smaller code sizes. As shown in the experiments, the availability of tasks in the SPM can result in shorter memory read latency, and less pessimism, thus, smaller WCET and WCRT. Although in some task sets the abort-and-restart model may have more overhead than preemptive or non-preemptive systems, this model provides advantages like type safety, lack of side effects and rollback costs. In addition, by introducing an SPM-based execution model, the overhead of preemption is reduced to a minimum cost.

10.2 Future Work

For the future work, one of the research directions is to extend the described WCET/WCRT analysis approach to P-FRP systems running on multi-core and multiprocessor platforms. Moreover, the abort-and-restart model will be extended to consider I/O latency in the WCET/WCRT derivation on the SPM-based platform.

Bibliography

[1] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[2] E. Amsden. A survey of functional reactive programming. *Unpublished*, 2011.

[3] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 318–326. ACM, 2003.

[4] F. Angiolini, L. Benini, and A. Caprara. An efficient profile-based algorithm for scratchpad memory partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(11):1660–1676, 2005.

[5] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[6] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[7] Ö. Babaoğlu, K. Marzullo, and F. B. Schneider. A formalization of priority inversion. *Real-Time Systems*, 5(4):285–303, 1993.

[8] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM, 2002.

[9] C. Belwal and A. M. Cheng. Determining actual response time in p-frp. In *Practical Aspects of Declarative Languages*, pages 250–264. Springer, 2011.

[10] C. Belwal and A. M. Cheng. Determining actual response time in p-frp using idle-period game board. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, pages 136–143. IEEE, 2011.

[11] C. Belwal, A. M. Cheng, and Y. Wen. Response time bounds for event handlers in the priority based functional reactive programming (p-frp) paradigm. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, pages 282–287. ACM, 2012.

[12] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[13] S. Chattopadhyay and A. Roychoudhury. Unified cache modeling for wcet analysis and layout optimizations. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 47–56. IEEE, 2009.

[14] A. Courtney. Frappé: Functional reactive programming in java. In *Practical Aspects of Declarative Languages*, pages 29–44. Springer, 2001.

[15] X. Dai, G. D. Hager, and J. Peterson. Frp in C+. 2010.

[16] J.-F. Deverge and I. Puaut. Wcet-directed dynamic scratchpad memory allocation of data. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 179–190. IEEE, 2007.

[17] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 223–233. ACM, 2006.

[18] B. Egger, J. Lee, and H. Shin. Scratchpad memory management for portable systems with a memory management unit. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 321–330. ACM, 2006.

[19] C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[20] J. Hansen, S. A. Hissam, and G. A. Moreno. Statistical-based WCET estimation and validation. In *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[21] P. Hudak and J. H. Fasel. A gentle introduction to haskell. *ACM Sigplan Notices*, 27(5):1–52, 1992.

[22] R. Kaiabachev, W. Taha, and A. Zhu. E-frp with priorities. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 221–230. ACM, 2007.

[23] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th annual Design Automation Conference*, pages 690–695. ACM, 2001.

[24] Z. Kazemi and A. M. Cheng. Static worst case execution time analysis of functional reactive systems. In *11th IEEE International Conference on Embedded Software and Systems (ICESS)*, 2014.

[25] Z. Kazemi and A. M. Cheng. A scratchpad memory-based execution platform for functional reactive systems and its static timing analysis. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, volume 8, page 13, 2015.

[26] Y. Kim, D. Broman, J. Cai, and A. Shrivastaval. Wcet-aware dynamic code management on scratchpads for software-managed multicores. In *Proc. of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[27] L. Li, L. Gao, and J. Xue. Memory coloring: A compiler approach for scratchpad memory management. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 329–338. IEEE, 2005.

[28] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56–67, 2007.

[29] X. Li, T. Mitra, and A. Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *Design Automation Conference, 2003. Proceedings*, pages 466–471. IEEE, 2003.

[30] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 92–103. IEEE, 2004.

[31] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34(3):195–227, 2006.

[32] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[33] T. Mitra and A. Roychoudhury. A framework to model branch prediction for worst case execution time analysis. In *Proc. 2nd Intl. Workshop on Worst-Case Execution Time Analysis*, 2002.

[34] J. Peterson, G. D. Hager, and P. Hudak. A language for declarative robotic programming. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1144–1151. IEEE, 1999.

[35] A. Reid, J. Peterson, G. Hager, and P. Hudak. Prototyping real-time vision systems: An experiment in dsl design. In *Proceedings of the 21st international conference on Software engineering*, pages 484–493. ACM, 1999.

[36] SNU. Homepage of snu real-time benchmark suite, 2014.

[37] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[38] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 409–415. IEEE, 2002.

[39] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Wcet centric data allocation to scratchpad memory. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10–pp. IEEE, 2005.

[40] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 401–410. ACM, 2006.

[41] W. Taha, P. Hudak, and Z. Wan. Directions in functional programming for real (-time) applications. In *Embedded Software*, pages 185–203. Springer, 2001.

[42] S. Thompson. *Haskell: the craft of functional programming*, volume 2. Addison-Wesley, 1999.

[43] M. Verma, L. Wehmeyer, and P. Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109. ACM, 2004.

[44] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Notices*, volume 35, pages 242–252. ACM, 2000.

[45] Z. Wan, W. Taha, and P. Hudak. Real-time frp. In *ACM SIGPLAN Notices*, volume 36, pages 146–156. ACM, 2001.

[46] Y. Wen, C. Belwal, and A. M. Cheng. Towards optimal priority assignments for the transactional event handlers of p-frp. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, pages 428–433. ACM, 2013.

[47] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS'08. IEEE*, pages 80–89. IEEE, 2008.