January 2019

Effective And Efficient Preemption Placement For Cache Overhead Minimization In Hard Real-Time Systems

John Cavicchio
Wayne State University, cavicchio@comcast.net

EFFECTIVE AND EFFICIENT PREEMPTION PLACEMENT FOR CACHE OVERHEAD MINIMIZATION IN HARD REAL-TIME SYSTEMS

by

JOHN C. CAVICCHIO

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2019

MAJOR: COMPUTER SCIENCE

Approved By:

Nathan Fisher, Ph.D.

Daniel Grosu, Ph.D.

Loren Schwiebert, Ph.D.

Hongwei Zhang, Ph.D.

Song Jiang, Ph.D.

ACKNOWLEDGEMENTS

First and foremost, Dr. Fisher is everything a professor should be and so much more. Dr. Fisher's consummate professional demeanor and enduring patience are just a few of the many fine qualities that I will try to take with me as I hope to transition from industry to academia over the next several years. Dr. Fisher is exceptional at providing students with the space to explore and grow technically while respectfully reeling us back in when needed with insightful feedback that fosters continual learning. He has been quite generous in sharing his experiences, frequently offering glimpses into the life of a professor with the right touch of humor that still brings a smile to my face. I have always dreamed of being a computer science professor, ever since that first high school data processing class with punch cards some years ago. I find myself nearing graduation with a doctoral degree that I hardly thought possible, largely in part due to Dr. Fisher's gentle guidance that helped set me on the path to success. Words cannot express the impact you have had on my academic achievements and future career.

I also wish to express my profound gratitude to the other members of my dissertation committee, Dr. Loren Schwiebert, and Dr. Daniel Grosu of the Computer Science Department at Wayne State University, along with Dr. Hongwei Zhang of the Department of Electrical and Computer Engineering at Iowa State University and Dr. Song Jiang of the Department of Computer Science and Engineering at the University of Texas Arlington for their example and continued support during my doctoral journey. I want to also thank the remaining professors and staff of the Wayne State University Computer Science Department for their hard work and dedication, and for their many contributions to my Computer Science career preparation.

I would like to especially give a most heartfelt thank you to my extended family for their love, support, and encouragement; whose sacrifices during this long and arduous endeavor will always be remembered for their part in this great achievement.

Lastly, the research contained in this dissertation was supported in part by the US National Science Foundation (CNS Grant Nos. 0953585 and 1618185).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

Motivation

Real-time Systems

Real-time systems are hardware and software systems where the utility of computational results exists within well defined time deadlines [43]. For example, in automotive engine control applications, it is imperative that the controller provide the engine with the proper air/fuel/spark mixture at the proper instant in time, otherwise engine and emissions performance will degrade significantly. Another example in automotive adaptive cruise control (ACC) applications where the controller must dynamically adjust the vehicle's speed to avoid a potential collision using hard-real-time data generated by several distinct sources each processed by a distinct hard-real-time software task. Figure 1 illustrates a high-level block diagram of an adaptive cruise control system [32]. In the ACC system, the wheel sensors provide vehicle

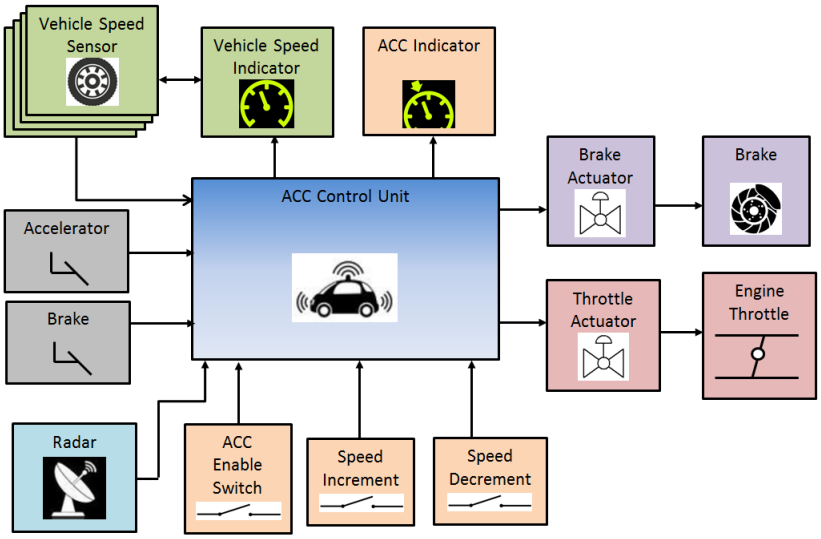

Figure 1: Adaptive Cruise Control (ACC) High Level Block Diagram in "Efficient Real-Time Support for Automotive Applications: A Case Study", by Goud et al., Proc. of the Embedded Real-Time Computing Systems and Applications, 2006.

speed data while the radar software component detects and tracks other vehicles and/or objects that exist along the current direction of travel. Once the raw data has been obtained, the control unit must execute sophisticated software algorithms in a timely fashion each partitioned into disparate real-time software tasks. The tasks subsequently perform the necessary computations to implement the complex adaptive cruise control feature in real-time. Figure 2 illustrates the diverse task set and the corresponding data set used in the complex computations involved in adaptive cruise control systems [32]. The ACC tasks are shown as circles with the corresponding input and output data for each task. The diagram illustrates the

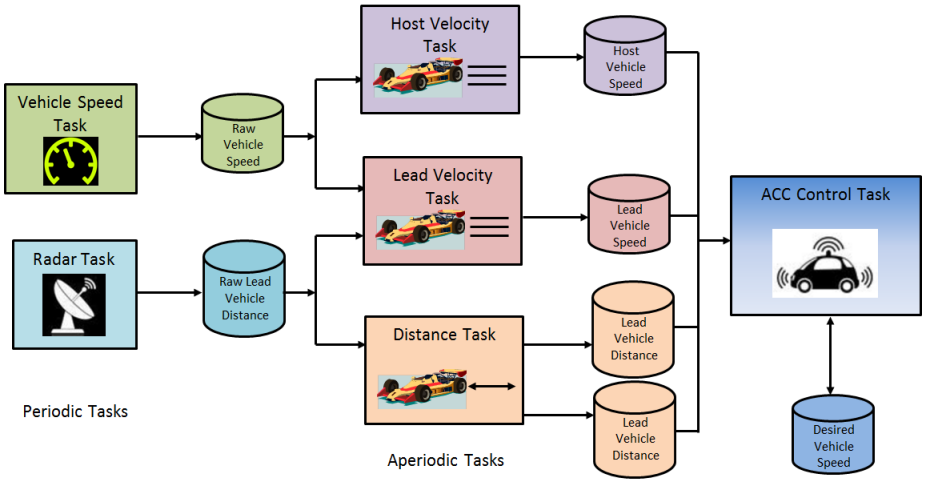

Figure 2: Adaptive Cruise Control (ACC) Taskset in "Efficient Real-Time Support for Automotive Applications: A Case Study", by Goud et al., Proc. of the Embedded Real-Time Computing Systems and Applications, 2006.

dependencies of each task on the correct and timely output of predecessor tasks. Thus, it is clear that ACC tasks must meet their respective deadlines in order for the vehicle to promptly and automatically adjust the vehicle speed to avoid a potential collision. Thus, the real-time systems computation is dependent on both the generation of correct logical results along with the time when these results are available. Three prominent categories or classifications of real-time systems are hard-real-time systems, firm-real-time systems, and soft-real-time systems. Soft-real-time systems are systems where an occasional deadline miss results in diminished utility or reduced quality of service (QoS). Firm-real-time systems are systems that can tolerate or are resilient to an occasional deadline miss. Here, each deadline miss results in a useless or unusable computation thereby adversely impacting system performance. In hard-real-time systems, however, deadline misses result in catastrophic operational effects, adversely impacting critical system functions. Thus, these systems are constrained to meet stringent deadlines regardless of computational load on the system, otherwise the system is considered to have completely failed. Today, real-time computing systems have become ubiquitous building blocks in many engineering applications comprising the most critical element in achieving required system performance. For example, in airplane flight applications, the modern fly by wire flight control system is categorized as a hard-real-time system. The system is comprised of a high-performance microprocessor and associated electro-mechanical actuators, where the timeliness and accuracy of the frequent in-flight control actions in response to various environmental and weather-related conditions is paramount for the safety of the pilot, crew, and passengers. Due to the complexity of real-time application requirements, the requisite functionality is implemented via a set of tasks, each responsible for performing some suitable subset of system computation. Figure 3 illustrates a

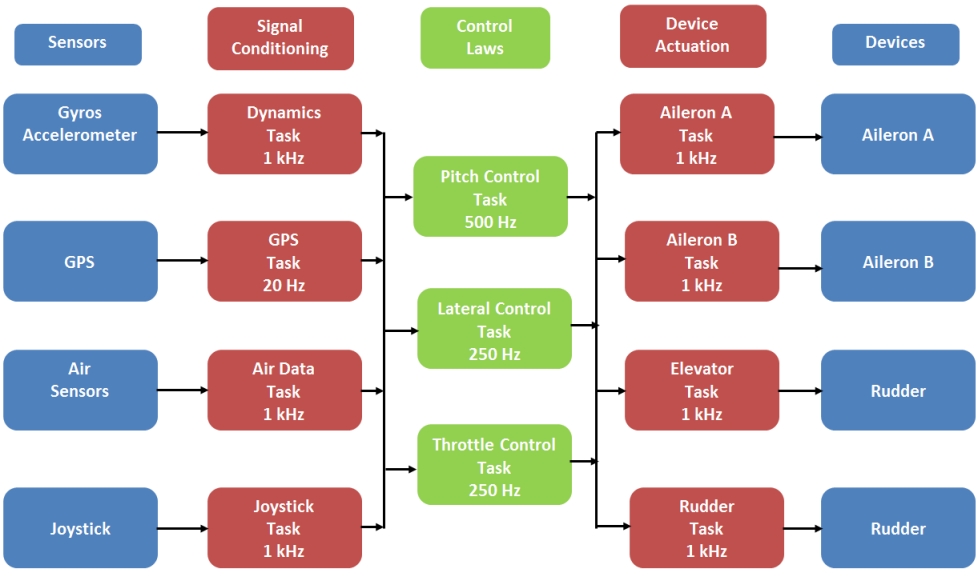

Figure 3: Fly by Wire Hard-Real-Time Taskset in "Real-Time Systems", by Jane W.S. Liu, 2000.

sample hard-real-time taskset utilized by a fly-by-wire avionics flight control system [21, 44]. Raw data from each sensor shown on the left is filtered and processed using signal conditioning algorithms running at the proscribed rate. The conditioned data is used by several motion control algorithms each running at the required rates for proper motion control. Lastly, the motion control decisions result in the actuation of the individual ailerons to carry out proper flight control. Like the ACC system example, it can be readily seen that the output of each real-time task in the fly-by-wire flight control system is critical to correct system operation. Therefore, the real-time system designer must assure the correct and timely functionality of each real-time computing task and the associated electro-mechanical parts for all the operating conditions of the system. Verifying that all tasks in hard-real-time system tasksets meet their respective deadlines is no doubt a non-trivial computational task.

Worst Case Execution Time (WCET)

Worst Case Execution Time (WCET) is the maximum execution time of a real-time computing task to produce correct computational results [59]. One of the notable challenges of computing WCET is the set of inputs that result in the maximum execution time is not always known [71]. WCET is used primarily by real-time system designers to assess or characterize the worst-case timing behavior of tasks in a partitioned task scheduled implementation for various input and output scenarios. The resulting WCET is then compared to pre-established timing budgets or used as input to real-time schedulability analysis. Real-time schedulability analysis is the mathematical and functional verification of the scheduling software component along with the various computing algorithms comprising each distinct task. A real-time system

implemented via multiple tasks consists of key software components such as a task scheduler, a system clock or tick, and the processing algorithms contained in software tasks. One of the characteristics that real-time systems possess over non-real-time systems is the utility of computations heavily depends on the schedulability of its processes or tasks. In other words, the ability of each task to meet is specified timing deadline. A deadline is defined as the maximum time allotted for each task to complete its computation or processing for a given set of inputs. In a real-time system, tasks are accepted by the scheduler software component, then scheduled to use the CPU in accordance with their assigned priority as dictated by the chosen scheduling algorithm, subject to the constraint that its processing must be completed as specified by the task deadline. The ability of a given task to meet its deadline is highly dependent on the performance characteristics of the chosen scheduling algorithm in conjunction with the performance characteristics of the task specific algorithms. The precise modeling and evaluation of real-time system scheduling behavior is the rigorous mathematical analysis of the scheduling algorithm's ability to ensure that all real-time system tasks meet their specified deadlines is the subject of schedulability analysis. A real-time system meets the definition of schedulable if the scheduling algorithm can guarantee that all tasks can meet their assigned deadlines, otherwise the system is said to be un-schedulable. The accuracy of the schedulability assessment of real-time system performance is critically dependent on the accuracy of the various task WCET values.

Task Preemption

Real-time scheduling algorithms are responsible for selecting tasks to use the CPU at any given moment from a list of tasks that are ready to execute. Most scheduling algorithms accomplish this via a process known as task preemption. Task preemption results in the task currently using the CPU to be momentarily suspended or interrupted by the real-time system scheduler, itself a privileged task. The scheduler then selects the highest priority task to run, restores its execution context, thereby allowing the new task to continue executing. This type of task scheduling change in real-time systems is known as a context switch. A context switch is the process of storing and restoring the execution context of a running task (typically a thread) so that execution can be resumed from the point at which the task was interrupted some time later. In effect, the preemptive scheduler is a key real-time system component enabling multiple tasks to share the CPU. The rules governing when task preemption occurs is a function of the task preemption model and scheduling algorithm. The scheduling algorithms used in scheduling real-time system tasks are categorized as fully preemptive, limited preemption or non-preemptive. A fully preemptive scheduling algorithm allows tasks to be preempted at any code location during their execution. Tasks run to completion in a non-preemptive scheduling algorithm before another task may use the CPU. Hence

non-preemptive scheduling does not permit tasks to be preempted, introducing undesirable blocking on higher-priority tasks. In a limited preemption scheduling algorithm, tasks may be preempted only at a limited number of locations. Limited preemption scheduling is also known as deferred preemption scheduling. Preemptions caused by the execution of higher priority tasks are deferred until a later point in time. The amount of preemption deferment is based on taskset execution characteristics, and the amount of available execution slack during system operation. The task preemption model along with the chosen scheduling algorithm utilized in any given real-time system can facilitate or inhibit the ability of tasks to meet their required deadlines.

Cache Related Preemption Delay (CRPD)

Cache memory is basically high-speed memory inserted between the CPU and the much slower main memory (RAM) acting much like a buffer. The concept of cache memories relies on two fundamental characteristics exhibited by most software programs, namely spatial locality and temporal locality. Spatial locality refers to the property where software programs repeatedly executes the same instructions and/or manipulates data stored in the same location. Temporal locality is the property where software programs repetitively access the same memory locations around the same time frame as opposed to being spread evenly over the execution life of the program. The objective of cache memory designs is to limit the number of accesses to the slower main memory by storing frequently accessed data in the faster cache memory, thereby increasing the effective memory speed and thus allowing the CPU to operate at a faster rate. For real-time systems that employ instruction and/or data caches, task preemption has the effect of introducing additional delays in task execution resulting from cache related preemption delay. Cache related preemption delay (CRPD) occurs when a task is preempted by higher priority tasks, whose subsequent execution causes memory blocks stored in cache memory to be evicted [39]. When the interrupted task resumes execution, the evicted cache memory blocks must be reloaded, introducing additional delay to the overall task execution time. The additional delay resulting from CRPD can significantly impact the ability of tasks in a real-time system to meet their deadlines thereby degrading system operation or causing total system failure. Therefore, it is imperative that CRPD effects be accounted for when determining whether hard-real-time tasksets can reliably meet their deadlines.

Schedulability Analysis

Today, high-performance microprocessors with advanced capabilities are commonplace in real-time systems. These processors operate at higher clock frequency and often contain multiple cores in a single processor [45]. Also, many processor manufacturers offer complete embedded computers, which include a powerful processor, several I/O interfaces, and serial communication interfaces in a single compact printed

circuit board (PCB) [46]. These modern embedded computers are ideal for contemporary hard-real-time system development as they have higher computing capabilities, exhibit higher reliability, and are more affordable than their previous counterparts. To realize the necessary performance gains, cache memory designs are employed to enable the CPU to operate at the fastest rate possible. Cache memory, despite its advantages, introduces schedulability challenges that real-time system designers must address during system design to ensure that real-time tasks consistently meet their deadlines during runtime. Schedulability analysis of the chosen scheduling algorithm and task preemption model in conjunction with the real-time taskset relies heavily on accurate WCET and CRPD estimates. It is imperative that the WCET and CRPD estimates be safe, meaning greater than or equal to the actual task performance in order for the schedulability analysis results to be meaningful to real-time system designers. The fundamental problem with existing CRPD calculation techniques is a significant level of pessimism or inaccuracy. The effect of this inaccuracy in the context of schedulability analysis is that some tasksets may be declared un-schedulable when in fact they are indeed schedulable, leading to over provisioning of CPU resources.

Until the last decade, real-time systems were mainly constrained by the available CPU processing power, where the CPU processing power was the scarcest resource. Earlier, the cost of real-time hardware systems was substantially high, compared to the combined cost of software design and testing [62]. In recent times, the average cost per tera flop has been reduced exponentially [38]. Today, modern real-time system manufacturers face different challenges. For real-time computing applications, the software design, development, and testing costs now represent the highest portion of product development costs due to increasing size and complexity of software programs as witnessed in a majority of engineering application domains. Lack of proper preemption point placement and schedulability analysis frameworks along with the difficultly involved in streamlining the real-time design process are prominent contributors to the higher development costs. It is clear that real-time system designers need to focus more attention on the software design and associated implementation issues. With the ever-increasing complexity exhibited by the real-time system physical domain along with variable hardware constraints, the number, size, and complexity of the real-time tasks developed to meet these diverse requirements is rapidly increasing. Thus, designing a reliable system with efficient resource provisioning where real-time tasks are guaranteed to meet their deadlines has become a central issue in achieving guaranteed robust real-time system performance. With the increasing size and complexity of real-time software tasks, accurate WCET and CRPD estimates are paramount to realizing efficient provisioning of CPU resources.

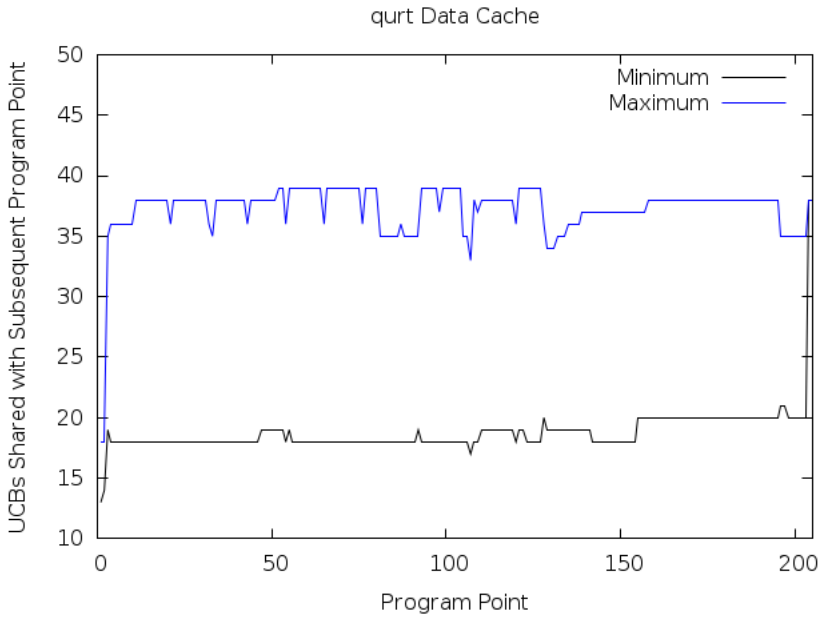

Figure 4: qurt Data Cache.

Motivation Summary

There are many important previous research results on cache related preemption delay (CRPD), schedulability analysis, and limited preemption point placement in real-time system designs (see Chapter 2). However, from a CRPD perspective, previous work has characterized CRPD as a single valued function dependent only on task preemption at each program location during execution. Figures 4 and 5 illustrate that actual CRPD values are inherently an interdependent function based on the previous and next preemption point within the context of a limited preemption model. The horizontal axis captures program locations corresponding to execution in a linear control flow graph. The vertical axis measures the CRPD at each program location. Two curves are shown, one denoting the minimum CRPD, and the other denoting the maximum CRPD assuming preemption occurs at that program location. What is readily apparent from the two figures is the CRPD values are program location dependent. To see the potential benefits of the interdependent CRPD function, the CRPD values computed in current start of the art methods would correspond to the maximum lines shown in the two figures. It is clear that no state of the art methods have incorporated this inherent interdependence of CRPD on the actual locations of preemptions in the task code. The benefits of integrating an interdependent CRPD metric into automated preemption point placement in a limited preemption model are increased accuracy, increased taskset schedulability, and improved CPU resource provisioning. The resulting accuracy improvements our newly proposed CRPD metric as integrated into our proposed preemption point placement algorithms have resulted in substantial gains in real-time taskset schedulability as expected. Therefore, in this thesis, we fill this vacancy by suggesting a

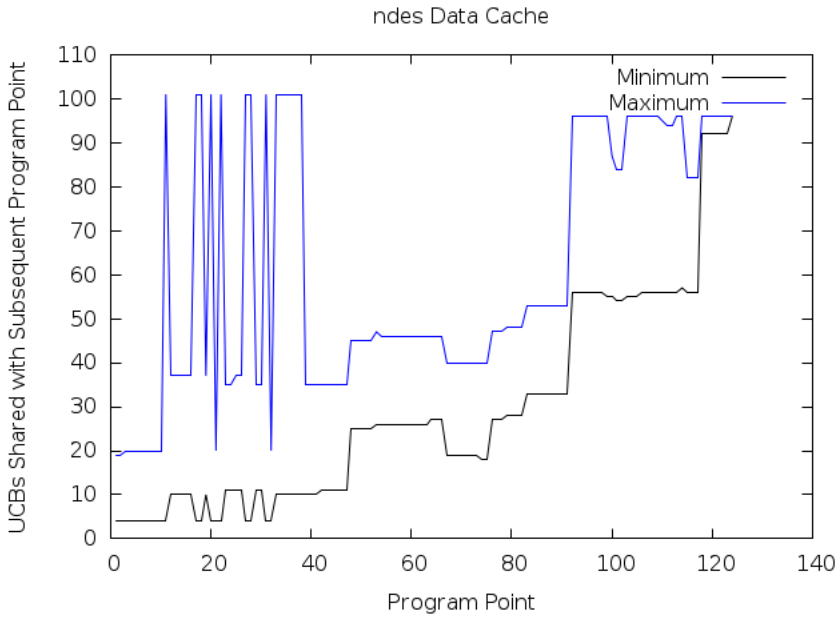

Figure 5: ndes Data Cache.

suitable schedulability analysis general framework for limited preemption scheduling.

In the rest of this chapter, we first discuss the objectives as well as the contributions of our work. Then we will outline the rest of the thesis in the end of this chapter.

Objectives

In this thesis, we introduce a new metric called *loaded cache blocks (LCB)* which more accurately characterizes the CRPD a real-time task may be subjected to due to the preemptive execution of higher priority tasks. Our LCB metric is integrated into our newly developed algorithms that automatically place preemption points supporting both linear and arbitrary control flow graphs (CFGs). Our proposed preemption point placement algorithms are shown to achieve optimal runtime performance for a given taskset. Our schedulability framework offers a reliable and formal process, thus saving time and costs incurred during the real-time system design cycle.

Thesis

The thesis of this document is:

> Cache related preemption delay cost is inherently dependent on task preemption locations in contrast to existing research. The CRPD interdependence can be leveraged to optimize preemption point placement supporting limited preemption scheduling in real time systems. The benefit of this approach is increased schedulability of hard-real-time tasksets by effectively minimizing the preemption overhead, and can be evaluated and verified empirically over real-time benchmarks and synthetic test sets.

Summary of Contributions

The main contributions of this thesis are listed as follows:

1. We propose a new CRPD metric, called *loaded cache blocks (LCB)* which accurately characterizes the CRPD a real-time task may be subjected to due to the preemptive execution of higher priority tasks (see Chapter 4).
2. We show how to integrate our new LCB metric into our newly developed algorithms that automatically place preemption points supporting linear control flow graphs (CFGs) for limited preemption scheduling applications (see Chapter 5).
3. We extend the derivation of *loaded cache blocks (LCB)*, that was proposed for linear control flow graphs (CFGs) to conditional CFGs for structured programs. (see Chapter 6).
4. We demonstrate how to integrate our revised LCB metric into our newly developed algorithms that automatically place preemption points supporting conditional control flow graphs (CFGs) for limited preemption scheduling applications. (see Chapter 6).
5. We show how to integrate the revised LCB metric into a revised algorithm that automatically place preemption points supporting linear and conditional control flow graphs (CFGs) for limited preemption scheduling applications. (see Chapter 6).
6. We show how to integrate preemption placement with preemption threshold scheduling supporting linear and conditional control flow graphs (CFGs) for limited preemption scheduling applications. (see Chapter 7).

Organization

The following table gives the details of each chapter:

Table 1: Chapter Contribution Summary

Chapter #	Contribution
Chapter 2	Related work
Chapter 3	Models and definitions used in the dissertation
Chapter 4	Loaded cache block calculation for linear and conditional control flow graphs
Chapter 5	Preemption point placement details for linear control flow graphs
Chapter 6	Preemption point placement details for conditional control flow graphs
Chapter 7	Integrating Preemption Thresholds with Limited Preemption Scheduling
Chapter 8	Future work
Chapter 9	Conclusion of this dissertation
Chapter 10	List of Publications

CHAPTER 2 RELATED WORK

In this chapter, we present prior research on cache related preemption delay and limited preemption point placement techniques. Our proposed linear and conditional PPP algorithms leverage elements from these two prominent areas of real-time theory. We now briefly summarize the prior work in each of these areas and describe how it relates to our proposed algorithm. For clarity, the descriptions of the related works are divided into two separate subsections, namely, CRPD calculation, and limited preemption scheduling.

CRPD Calculation

The concept and algorithm for computing the set of useful cache blocks (UCBs) was proposed by Lee et al. [39] analyzing the preempted task. Similarly, the set of evicting cache blocks (ECBs) was realized by Tomiyamay and Dutt [69] analyzing the preempting task. More formal definitions of UCBs and ECBs by Altmeyer and Burguiere [5] were subsequently refined and presented. By convention, the preempting task's memory accesses (ECBs) will evict the preempted task's UCBs thereby resulting in non-negligible CRPD.

Tighter bounds on the CRPD computation via the intersection of the ECB and UCB sets were achieved by Negi et al. [52] and Tan and Mooney [67]. A cache state reduction technique by Staschulat and Ernst [66] was employed trading off CRPD accuracy or tightness for a reduction in computational complexity. CRPD analysis using memory access patterns was proposed by Ramaprasad and Mueller [60].

The over-approximation of cache misses in WCET analysis tools by introducing definitely-cached useful cache block (DC-UCB) as proposed by Altmeyer and Burguiere [5]. DC-UCBs are cache blocks that must reside in cache memory.

The ECB Union approach and a combined UCB Union and ECB Union approach by Altmeyer et al. [6] were later introduced and shown to dominate earlier CRPD methods. The UCB Union and ECB Union multiset approaches were subsequently introduced to reduce the pessimism in CRPD analysis also proposed by Altmeyer et al. [7].

The above approaches assume that the CRPD cost of a preemption is computed independently to obtain a safe, conservative bound. Conversely, an interdependent CRPD model using loaded cache blocks (LCBs) was proposed. Recent work to improve the CRPD accuracy in PPP algorithms by Cavicchio et al. [25] leading to tangible schedulability gains. Our research leverages the interdependent CRPD model and extends preemption placement to conditional real-time code structures.

Limited Preemption Scheduling

Research in limited preemption scheduling attempts to address well known limitations of the non-preemptive and fully preemptive scheduling paradigms. Two limited preemption scheduling models are the deferred preemption scheduling model and the preemption threshold scheduling model.

First, the fixed preemption point model proposed by Burns [20] and the floating-point preemption model proposed by Baruah [9] are two distinct sub-categories of deferred preemption scheduling. In the floating preemption point model [9], the currently executing task continues executing for a minimum of Q_i time units or runs to completion if the remaining execution time is less than Q_i. The parameter Q_i is computed via task set schedulability analysis [9] representing the maximum amount of blocking time that task τ_i may impose on higher priority tasks. Since the start of the non-preemptive execution region coincides with the arrival of a higher priority task, region locations are nondeterministic or floating. In contrast, the fixed-point preemption model [20] differs from the floating preemption point model in that the task code preemptions are confined to pre-defined fixed locations and computed using an offline PPP algorithm.

Second, preemption threshold scheduling proposed by Wang and Saksena [70] employs a modified priority-based scheme to determine when tasks may preempted. Each task in the preemption threshold scheduling [70] approach is assigned two distinct priority values, each for a different purpose. These priorities are known as the nominal static priority p_i and a preemption threshold Π_i priority. The preempting task τ_k may only preempt the currently executing task τ_i if τ_k has a nominal priority p_k exceeding the assigned preemption threshold Π_i.

Fixed PPP algorithms sought to minimize preemption overhead in fixed priority task sets in early work by Simonson and Patel [64] and Lee et al. [39]. For tasks executing via preemption triggered floating non-preemptive regions, Marinho et al. [48] successfully computed an upper-bound on task CRPD. Lastly, work in fixed priority (FP) preemption threshold schedulability (PTS) analysis by Bril et al [19], supporting task sets with arbitrary deadlines, successfully assigned optimal priority thresholds by minimizing independent CRPD costs. This work was later complimented by combining the optimal threshold assignment algorithm with a simulated annealing algorithm that optimizes task layout [18]. The effectiveness of these heuristic solutions is limited by the employed cost model.

An optimal PPP algorithm was realized by Bertogna et al. [13, 14] with linear time complexity for strictly linear CFG structures. A pseudo-polynomial preemption point placement algorithm was realized [58] supporting well-structured series/parallel conditional CFG structures. The limitations of these soluti-

ons stem from utilizing the less accurate independent CRPD cost model as compared to the interdependent cost model accounting for the dependency between selected preemption points. Recently, a quadratic PPP algorithm was proposed by Cavicchio et al. [25] using the interdependent CRPD cost model for linear CFG structures.

In our work, we extend the dynamic programming algorithm proposed by Peng et al. [58] to incorporate the more accurate interdependent CRPD cost model supporting separate instruction and data direct-mapped caches. Our conditional PPP algorithm realizes an enhanced minimized safe upper bound preemption point placement solution resulting in substantial improvements over previous PPP methods commensurate with the accuracy improvements using interdependent CRPD costs. Furthermore, the handling of inline function calls is refined into function definition and function invocation components providing the basis for future non-inline function support. Our comprehensive simulation approach utilizes real-time code benchmark examples as a basis for comparison against several current state of the art scheduling methods. Lastly, our evaluation demonstrates the superiority of the interdependent CRPD model coupled with the limited preemption scheduling model resulting in a new state of the art.

CHAPTER 3 MODELS AND DEFINITIONS

Our work is cross-disciplinary research that leverages the concepts from both computer science and electrical engineering. Therefore, in this chapter, we will introduce a basic overview on notations, concepts, and models, to make readers more comfortable with the main concepts of this document. Also, we only provide a very concise yet complete overview on the concepts and theories, details can be found in relevant textbooks and research papers.

Real-Time Systems Model

In next couple of sub sections, real-time models, notations, and definitions are presented.

Jobs and Tasks

Any real-time system can be considered as a set of concurrent *tasks* [44]. Each task generates an infinite number of jobs. The jobs from a same task obey sequential order of execution.

Definition 1 (Job)**.** *A real-time job $j = (A, E, D)$ is characterized by three parameters, an arrival time A, an execution time E, and a deadline D, with the requirement that this job must receive E units of execution over the interval $[A, D)$.*

Figure 6 illustrates the definition of a real time job.

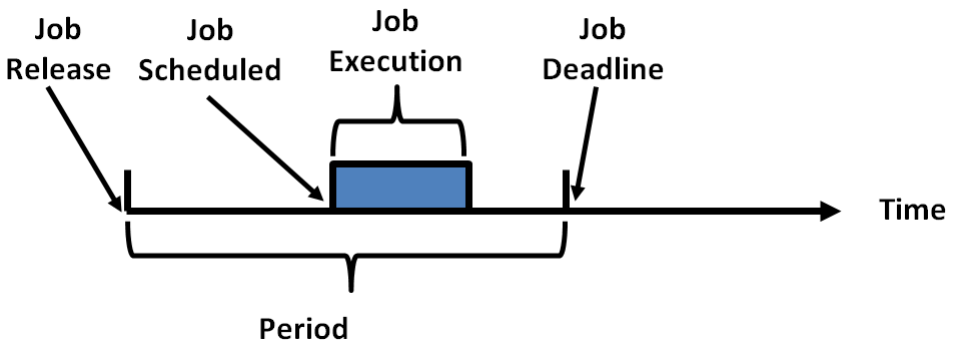

Figure 6: Real-Time Job Execution.

Periodic Task Model

In the periodic task model [10], a task T_i is completely characterized by a 4-tuple (a_i, e_i, d_i, p_i), where

1. the *offset* a_i denotes the instant at which the first job generated by this task becomes available for execution.
2. the *execution requirement* e_i specifies an upper limit on the execution requirement of each job generated by this task.
3. the *relative deadline* d_i denotes the temporal separation between each job's arrival time and deadline. A job generated by this task arriving at time-instant t has a deadline at time-instant $(t + d_i)$.

4. the *period* p_i specifies the separation of or the inter-arrival times of successive jobs generated by the task.

That is, $T_i = (a_i, e_i, d_i, p_i)$ generates a potentially infinite succession of jobs, each with execution-requirement e_i, at each instant $(a_i + k \cdot p_i)$ for all integer $k \geq 0$, and the job generated at instant $(a_i + k \cdot p_i)$ has a deadline at instant $(a_i + k \cdot p_i + d_i)$ [10].

Sporadic Task Model

The sporadic task model differs from the periodic task model because of its inability to express job arrival time until the run-time moment. A sporadic task system contains a set of sporadic tasks [10].

Definition 2 (Sporadic Task)**.** *A* **sporadic task** τ_i *is completely described by a* 3*-tuple* $\tau_i = (e_i, d_i, p_i)$ *where* e_i *is the worst-case execution time,* d_i *is the relative deadline, and* p_i *is the minimum inter-arrival time between successive jobs. By definition, sporadic tasks generate an infinite sequence of jobs. Successive job arrivals will be separated by a minimum time denoted by* p_i*. A sporadic task system* $\tau \stackrel{def}{=} \{\tau_1, \ldots, \tau_n\}$ *is a collection of* n *sporadic tasks.*

The utilization indicates the amount of time that system becomes busy due to tasks in a task model. Formally, the utilization U_i of a periodic or sporadic task τ_i is defined to be the ratio of its execution requirement to its period: $U_i = \frac{e_i}{p_i}$. The utilization $U(\tau)$ of a periodic or sporadic task system τ is defined to be the sum of the utilizations of all tasks in τ : $U = \sum_{T_i \in \tau} U_i$. For any task system under any known uniprocessor scheduling algorithm, the utilization should not exceed 1 and cannot schedulable otherwise; however, this does not necessarily say that a system is schedulable if the utilization is under 1.

Control Flow Graphs

Each task τ_i contains real-time code represented by a series/parallel control flow graph (CFG), denoted G_i. We introduce the basic block notation δ_i^j where i is the task identifier and j is the basic block identifier. A dummy basic block δ_i^0 with zero WCET is added at the beginning of each task to capture the preemption that occurs prior to task execution. In our model, a basic block is a set of one or more instructions that execute non-preemptively. Basic blocks are essentially the vertices V_i of a conditional control flow graph (CFG) connected by directed edges E_i representing the execution sequence of one or more job instructions. Figure 7 shown below illustrates the conditional basic block connection structure.

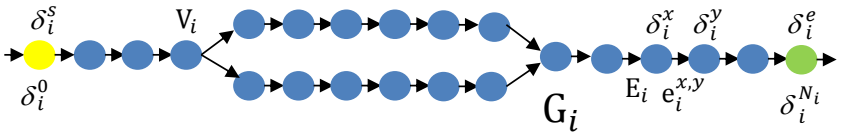

Figure 7: Conditional Control Flow Graph.

Linear CFGs represents a special case or subset of the problem for conditional CFGs as shown in Figure 8.

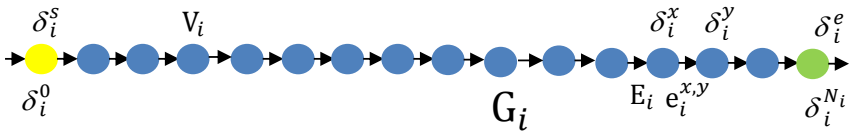

Figure 8: Linear CFG Structure.

The CFG for task τ_i denoted G_i is a tuple (V_i,E_i,δ_i^s,δ_i^e) representing the real-time task code structure. Flow graphs are directed graphs describing the task execution sequence from a starting basic block δ_i^s to an ending basic block δ_i^e through one or more execution paths $p \in P_i(G_i, \delta_i^s, \delta_i^e)$, where $P_i(\cdot)$ denotes all possible execution paths starting with δ_i^s and ending at δ_i^e in G_i. An execution path p through G_i is an ordered sequence of basic blocks from some starting instruction δ_i^s to an ending instruction δ_i^e. A directed edge $e_i^{x,y} = (\delta_i^x, \delta_i^y)$ where $e_i^{x,y} \in E_i$ connecting basic blocks δ_i^x and δ_i^y represents the execution of basic block δ_i^x immediately preceding the execution of basic block δ_i^y. We introduce an operator $\preceq_p$ describing a more general execution precedence relation, where $\delta_i^x \preceq_p \delta_i^y$ represents the execution of basic block δ_i^x preceding the execution of basic block δ_i^y by zero or more instructions along some path p. Similarly, operator $\prec_p$ represents the execution of basic block δ_i^x preceding the execution of basic block δ_i^y by one or more instructions along some path p. We further introduce notation describing a subgraph in task τ_i as G_i^a with the tuple (V_i^a,E_i^a,$\delta_i^{s^a}$,$\delta_i^{e^a}$) where a is the subgraph identifier, $\delta_i^{s^a}$ is the starting basic block, and $\delta_i^{e^a}$ is the ending basic block. Similar notation used to describe the paths in subgraph a is given as $P_i^a(G_i^a, \delta_i^{s^a}, \delta_i^{e^a})$. Preemptions are permitted at the edges $e_i^{x,y}$ between basic blocks. We introduce the non-preemptive basic block execution time notation b_i^j where i is the task identifier and j is the basic block identifier, hence using this convention we have:

$$C_i^{NP} = \max_{p \in P_i(G_i, \delta_i^s, \delta_i^e)} [\Sigma_{\delta_i^j \in p}\, b_i^j]. \tag{1}$$

Tasks may be preempted by multiple higher-priority tasks as determined by the employed scheduling algorithm. In this paper, we support fixed priority (FP), Deadline-Monotonic (DM) and Earliest-Deadline-First (EDF) scheduling. The processor utilization U_i of task τ_i is given by:

$$U_i = C_i^{NP}/T_i. \tag{2}$$

In a limited preemption approach, each task is permitted to execute non-preemptively for a maximum amount of time denoted by Q_i. Previous research on limited preemption scheduling by Baruah et al. [9] has used the above information to determine the value of Q_i for each task. The determination of Q_i is dependent on the placement of preemption points. Preemptions occur after the last basic block instruction

and before the first instruction of the next basic block. The details supporting the computation of Q_i are presented in the schedulability analysis section of Chapter 4.

Online and Offline Algorithms

In *offline* scheduling algorithms, all scheduling decisions are made before the system begins executing. These scheduling algorithms select jobs to execute by referencing a table describing the pre-determined schedule. Usually, offline schedules are repeated after a least common multiple (LCM) period. The offline schedulers require the full knowledge of the job before the they execute.

In *online* scheduling algorithms, scheduling decisions are made without specific knowledge of jobs that have not yet arrived. These scheduling algorithms select jobs to execute by examining properties of active jobs. Online algorithms can be more flexible than offline algorithms since they can schedule jobs whose behavior cannot be predicted ahead of time.

Scheduling Algorithms

Real-time jobs are selected for execution by the scheduler component. The scheduler uses a scheduling algorithm to perform job selection. Typical examples of scheduling algorithms are earliest deadline first (EDF), least laxity first (LLF), rate monotonic (RM), deadline monotonic (DM), and fixed priority (FP) to name a few [22]. At each time instance t scheduling algorithms select an active job j whose priority is the highest. In essence, scheduling algorithms tend to differ in the manner and/or task metric used to determine the highest priority job. The EDF algorithm selects the job with the earliest deadline [10]. Similarly, the LLF algorithm selects the job with the smallest laxity as the highest priority job. One very well-known fixed priority scheduling algorithm is the RM algorithm. The RM algorithm, selects jobs with shorter periods as the highest priority job [44]. Finally, the DM algorithm selects the job with the smallest or shorter deadline as the highest priority job [44].

Depending on the scheduling requirements, a hard-real-time system might employ *online* or *off-line* scheduling algorithms. For example, if a hard-real-time system has a well-defined taskset whose characteristics are known a priori, and there are no runtime task modifications expected, then off-line scheduling algorithms are often deemed suitable. The off-line scheduling algorithms are more suitable for systems with sufficient resources to handle the known taskset. However, an off-line scheduling algorithm is generally unsuitable to manage continuously varying and somewhat dynamic task/job characteristics that are unknown ahead of time. Online scheduling techniques which account for dynamic task characteristics are often employed in these instances [44]. It is customary that online scheduling algorithms used in dynamic environments may use heuristic methods optimized for the specific taskset.

The primary intention of this section is to provide a review of previous work that summarizes the

real-time system model used in our work. As such, research on scheduling algorithms and associated techniques are in scope, as our algorithms must be compatible with the underlying scheduling algorithm. For convenience, Table 2 summarizes the terminology presented in this section.

Term	Description
b_i^j	Basic block j WCET in task τ_i CFG
C_i^{NP}	Non-preemptive task τ_i execution time
D_i	Relative deadline of task τ_i
δ_i^j	Basic block j in task τ_i CFG
δ_i^0	Dummy basic block with zero WCET added to each task CFG
$\delta_i^{e^a}$	Ending basic block in task τ_i subgraph a CFG
$\delta_i^{s^a}$	Starting basic block in task τ_i subgraph a CFG
E_i	The set of directed edges in task τ_i CFG
E_i^a	The set of directed edges in task τ_i subgraph
$e_i^{x,y}$	A directed edge connecting basic blocks δ_i^x and δ_i^y
G_i	Task τ_i control flow graph (CFG)
G_i^a	Task τ_i subgraph a CFG
i	Index variable denoting the preempted task
$N_i + 1$	Number of basic blocks in task τ_i CFG
P_i	Unique set of paths through task τ_i CFG
P_i^a	Unique set of paths through task τ_i subgraph a CFG
P_i^a	$P_i^a(G_i^a, \delta_i^{s^a}, \delta_i^{e^a})$ using graph boundaries $\delta_i^{s^a}$ and $\delta_i^{e^a}$
p	Current path through task τ_i CFG
p_i	Fixed priority of task τ_i
Q_i	Maximum non-preemptive execution region for task τ_i
T_i	Task τ_i inter-arrival time or period
τ	Taskset containing n tasks $\tau_1, \tau_2, \ldots \tau_n$
τ_i	Task τ_i in taskset τ
U_i	The processor utilization of task τ_i
V_i	The set of vertices in task τ_i CFG
V_i^a	The set of vertices in task τ_i subgraph a CFG

Table 2: System Model Terminology

In the next subsections, we give details on the system hardware cache models employed on uniprocessor systems.

Cache Memory Models

Cache Memory

Cache memory is basically a high-speed memory array which essentially buffers the slower main memory from the CPU. The goal of cache memory is to minimize the number of occurrences where instructions and data must be fetched from main memory. With the ever-increasing gap between CPU processing speed and main memory access speed, real time system designers are attempting to maximize CPU throughput via cache memory. Cache memory designs consist of cache data memory, cache directory,

cache controller, address buffers, and data buffers. Figure 9 illustrates the high-level block diagram of key cache memory design components [33].

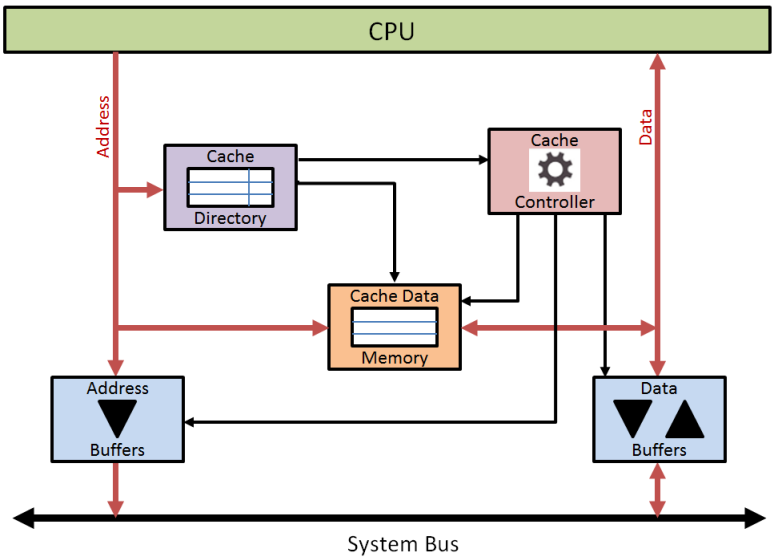

Figure 9: Key Cache Memory Design Components in "The Cache Memory Book: The Authoritative Reference on Cache Design", Second Edition, by J. Handy, 1998.

The cache data memory is the high-speed memory array where replicas of recently used main memory data are stored. The cache directory acts as storage for the main memory addresses whose data is currently stored in the cache data memory. The cache controller contains hardware logic responsible for controlling the cache memory operation. The operational characteristics of hardware logic are generally referred to as cache policies. The cache directory is examined by the cache controller when main memory is accessed to determine whether the data is already held in the cache data memory. The cache directory stores the address tag bits along with a valid bit indicating if that cache memory location has valid data. Alternative terms used for cache directory are cache-tag and content addressable memory (CAM). In the electronics domain, the data buffer is a hardware device used to store data temporarily while it is being moved from main memory to the cache data memory. The address buffer stores the most recently memory address accessed and is used by the cache controller. The data and address buffers also serve to amplify the current in order to facilitate the fan out needed by the cache memory design.

Direct Mapped Caches

Initially, we consider a single processor system with direct mapped instruction and data caches. Direct mapped cache memory possesses the property where only one cache memory location exists for each main memory data entry. Hence there is no need for a cache replacement policy as only a single unique cache memory location is replaced with each subsequent main memory access. Let a be the number of address

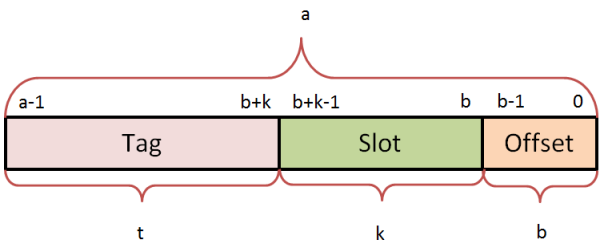

Figure 10: Cache Memory Address Decomposition.

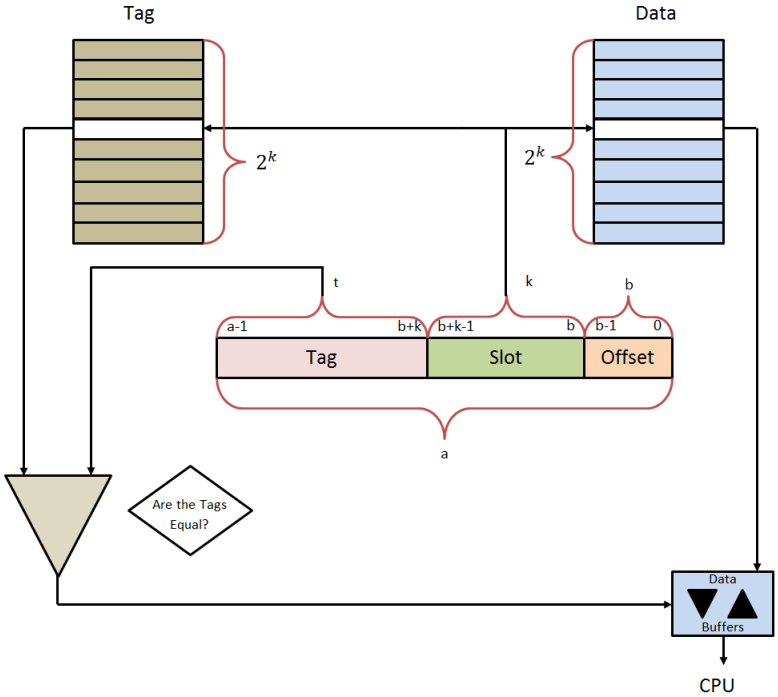

Figure 11: Direct Mapped Cache Hit Determination Logic.

bits used in a CPU that employs a direct mapped cache memory design. Further, let b the number of bits representing the cache block size and let k be the number of bits representing the number of cache lines in the cache data memory. The number of bits used for the cache directory or cache-tag is given by $t = a - k - b$. Figure 10 illustrates a typical address decomposition for direct mapped caches [33].

Direct mapped caches have a distinct advantage of simplicity, as the tag bits only have to be checked for a single slot to determine a match or hit. One of the primary disadvantages of direct mapped caches is that the scheme suffers from many addresses, 2^{t-k} specifically, that end up colliding to the same cache data memory slot. This causes each cache line to be potentially frequently evicted, thereby imposing a cache reload delay or penalty on a running task. Figure 11 illustrates the cache hit determination logic for direct mapped caches [33].

The k bits comprising the slot field of the address are used to index or select the cache data memory and cache tag memory location. On a CPU memory access request, the t bits comprising the tag field of

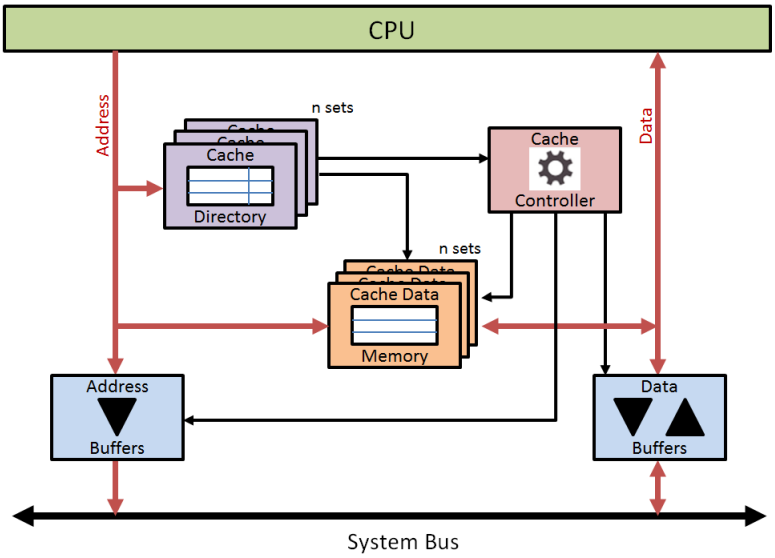

Figure 12: Set Associative Cache Memory Design Components.

the address are compared to the tag bits stored in the tag memory. If they are equal, then a cache hit occurs, and the data is buffered for use by the CPU.

Set Associative Caches

To overcome the frequent eviction of cache lines, cache designers may employ the use of multiple sets for each cache data memory location. Let the parameter n denote the number of cache sets. Cache memory designs that have multiple sets for each cache data memory location are known as n-way set associative caches. Figure 12 illustrates a high-level block diagram of key set associative cache memory design components [33].

Due to having multiple sets for each cache data memory location, a mechanism to determine which specific set to replace is required. This mechanism is known as a replacement policy or replacement algorithm. Figure 13 illustrates the cache hit determination logic for set associative caches [33].

Similar to direct mapped caches, the k bits comprising the slot field of the address are used to index or select the cache data memory and cache tag memory location. On a CPU memory access request, the t bits comprising the tag field of the address are compared to the tag bits stored in the tag memory. The primary difference for set associative caches is the tag memory has to be examined simultaneously for all n sets. If one set contains a match, then a cache hit occurs, and the data is buffered for use by the CPU.

Fully Associative Caches

A fully associative cache implements a cache policy allowing any main memory block to be mapped to any cache line. The hardware for determining whether the desired memory block is in cache memory requires comparing the memory block address tag bits to the stored tag bits of every cache block (line,slot)

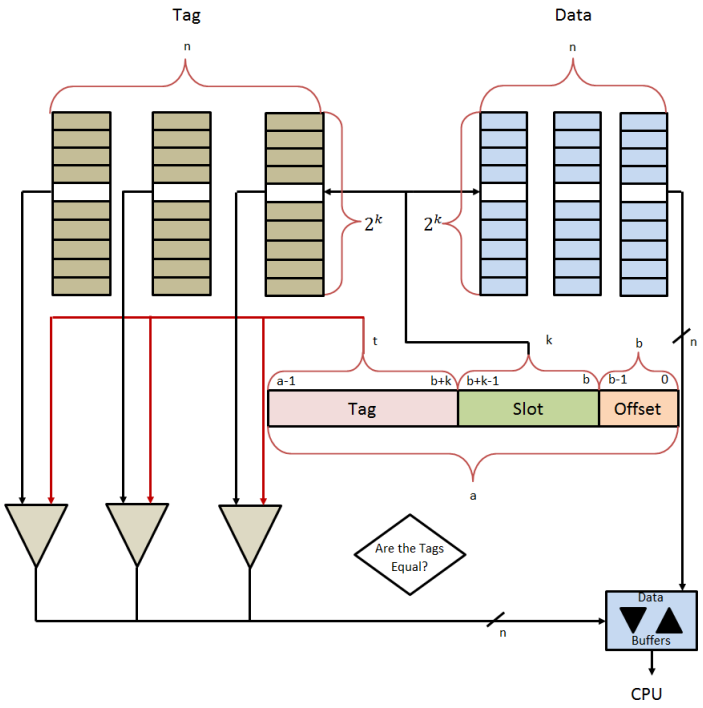

Figure 13: Set Associative Cache Hit Determination Logic.

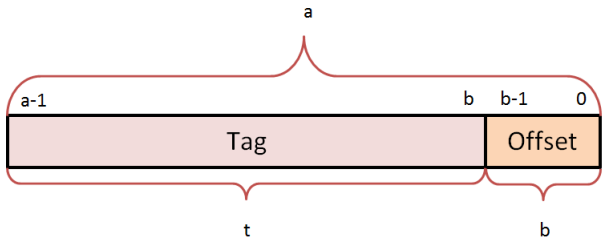

Figure 14: Fully Associative Cache Memory Address Decomposition.

in parallel. A fully associative cache must use content addressable memory to translate processor addresses into cache data RAM addresses.

Let a be the number of address bits used in a CPU that employs a fully associative cache memory design. Further, let b the number of bits representing the cache block size and let k be the number of bits representing the number of cache lines in the cache data memory. The number of bits used for the cache directory or cache-tag is given by $t = a - b$. Since the memory block can occupy any location in cache data memory, there is no need for including the k cache slot bits. Figure 14 shown below illustrates a typical address decomposition for fully associative caches [33].

Due to having all cache blocks available for each cache data memory access, a mechanism to determine which specific set to replace is required. This mechanism is known as a replacement policy or replacement algorithm. Figure 15 illustrates the cache hit determination logic for fully associative caches [33]. When a cache miss occurs, the cache controller must select a cache block that is empty or not valid to place

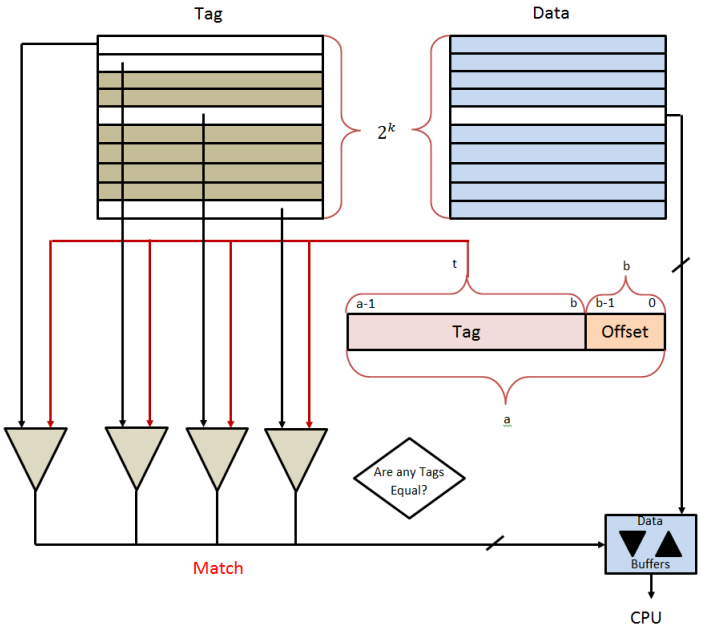

Figure 15: Fully Associative Cache Hit Determination Logic.

the main memory data currently being accessed. If no empty cache blocks exist, the cache controller must choose a cache block to evict based on the cache eviction policy design. If the evicted cache block contains dirty data (i.e. the data is different from main memory contents), then the data in the cache block must be copied back to main memory before the new data may be loaded. One of the primary advantages of fully associative caches is their excellent temporal locality properties. In this model, the flexible memory block placement allows the design and implementation of replacement policies to minimize the replacement of cached memory blocks that will be referenced again. One of the disadvantages is its high cost for large cache memory designs due to the need for a tag comparator for each cache block.

In the following subsections, we give details on the various cache replacement policies used on uni-processor systems. Since our future work will likely implement the CRPD metric and limited preemption point placement for various cache replacement policies or algorithms, the supporting summaries of popular algorithms are provided for reference.

Cache Replacement Policies

Cache replacement policies [65], also known as cache replacement algorithms, are responsible for the selection of the cache line to place the newly referenced data in when a read miss occurs. If empty cache lines are available, then they are prioritized over occupied or valid cache lines. If there are no empty cache lines, then an occupied cache line must be chosen for removal. The rules or algorithms that choose an occupied cache line for removal are designed to select cache lines that are least likely to be reused in the future. In other words, cache lines that are most likely to be reused should be avoided where possible.

Lastly, before the new data may be loaded, the data in the selected cache line must be written back to main memory if it has been modified.

Belady's Algorithm

Belady's cache replacement algorithm [11,49], also known as the optimal algorithm or the clairvoyant algorithm, always selects the cache block whose information is no longer needed or will not be needed for the longest period of time. In order to accomplish this, the system must examine all future memory references. For obvious reasons, it isn't practical to determine how far in the future cached data will be required using online replacement algorithms. As such, this replacement algorithm is strictly used as a comparator to determine the relative effectiveness as compared to more practical approaches.

Least Recently Used (LRU)

The least recently used (LRU) replacement algorithm selects the cache block that has the largest age. In other words, the cache block that was used least recently. Figure 16 illustrates the operation of the algorithm. It is often convenient to visualize LRU algorithm operation by using a FIFO list data structure. The

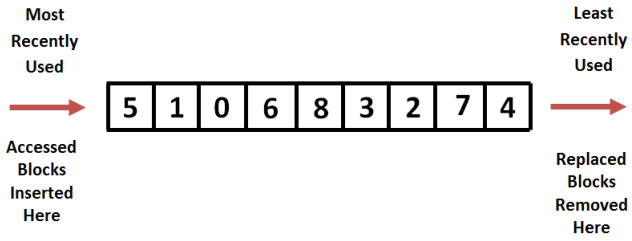

Figure 16: LRU Cache Replacement Algorithm.

LRU algorithm requires aging information to be maintained in order to determine which block is the least recently used. For set associative and fully associative caches, aging information is maintained for each cache set. For example, an 8-way set associative cache has 8! or 40320 different block orderings requiring sixteen bits to represent aging information. For fully associative caches, its customary to represent the aging information by using counters or lists. When a cache hit occurs, the ages of all other cache blocks are incremented. Memory access workloads where LRU performs well have two important characteristics: (1) a preponderance of memory references with smaller reuse distances [29]; and (2) the vast majority of these memory references have reuse distances that are smaller than the number of available cache memory blocks. Workloads with these two characteristics exhibit strong locality properties, resulting in a higher number of cache hits. On the other hand, LRU is known to perform poorly with workloads exhibiting weak locality properties. The primary reason for this is that LRU inherently operates under the assumption that cache blocks that haven't been accessed for the longest time are expected to be next accessed in the longest time frame. Common applications of the LRU algorithm include virtual memory management,

file buffering, and database buffering. Cost wise, LRU is infeasible and unaffordable, as it must maintain the LRU ordering for each memory access. In practice, more practical cost-effective implementations of LRU are used whose algorithms are discussed in subsequent sections.

Least Recently Used K (LRU-K)

The least recently used K (LRU-K) replacement algorithm [53, 54], a more practical implementation of the LRU algorithm, measures the time intervals using counters tracking successive memory references whose primary goal is to retain cache blocks with smaller reference or inter-arrival counts. Figure 17 illustrates the operation of the algorithm [53, 54]. The inter-arrival counts are known as the backward

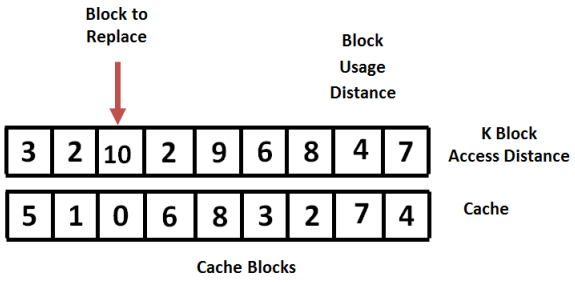

Figure 17: LRU-K Cache Replacement Algorithm.

K-distance metric, denoted $b_t(p, K)$, where $b_t(p, K)$ is the backward distance to the K^{th} most recent reference to cache block p, otherwise, if there have been less than K references, $b_t(p, K)$ is assigned the value of ∞. LRU-K selects the cache block whose backward K-distance metric is the maximum value from the last K memory references. In other words, cache blocks selected for replacement are those with the largest K reference length. In cases where multiple cache blocks have the same maximum backward K-distance value, a secondary selection algorithm must be employed to break ties. In practice, the LRU-K replacement algorithm is widely used in database buffering applications.

Most Recently Used (MRU)

The most recently used (MRU) replacement algorithm selects the cache block that was used most recently. The MRU algorithm has been found useful for applications using repeated sequential scanning of file data. Figure 18 illustrates the operation of the algorithm [28]. It is most suitable for memory access

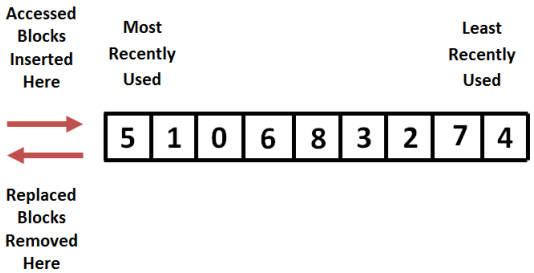

Figure 18: MRU Cache Replacement Algorithm.

patterns where the older data items are more likely to be accessed [28]. MRU is not widely used as several new cache replacement algorithms have been recently developed that have successfully addressed cache performance during sequential scans of file data.

Random Replacement (RR)

The random replacement (RR) algorithm randomly selects a cache blocks discarding the contents to make room as needed. The RR algorithm requires no bookkeeping information about past memory access history. The primary advantage of the random replacement approach is its simplicity. This replacement policy has been employed on ARM Cortex-R series processors.

First-In First-Out (FIFO)

The first-in first-out (FIFO) algorithm, known as one of the simplest cache replacement algorithms, stores cached memory blocks in a queue data structure. The most recently accessed memory blocks are inserted at the back end and the least recently used or oldest memory blocks are located at the front end of the queue. FIFO selects the oldest memory block at the front end for replacement. Figure 19 illustrates the operation of the algorithm [63]. Advantages of the FIFO approach is its simplicity and

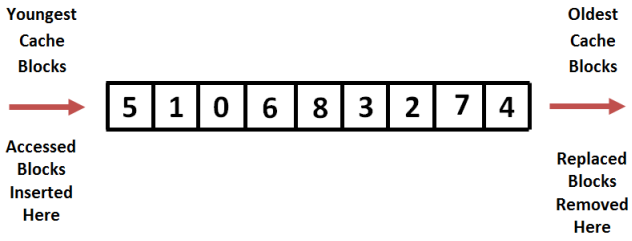

Figure 19: FIFO Cache Replacement Algorithm.

associated low overhead. By itself, FIFO performs poorly in practical applications, hence it is rarely used unmodified. Historically, the FIFO algorithm has been used in the VAX/VMS operating system, with modifications [63]. The Second Chance (SC) algorithm [68], is widely known as one of the more commonly used FIFO algorithm variants.

Least Frequently Used (LFU)

The least frequently used (LFU) replacement algorithm selects the cache block that has been referenced the least amount. The LFU algorithm maintains usage counters that increment every time a cache hit occurs for a particular cache block. Figure 20 illustrates the operation of the algorithm [4, 26]. A frequency-based memory access workload characterization known as the Independent Reference Model (IRM) has been extensively studied. Under IRM, memory access patterns occur in a randomly independent manner according to some fixed probability distribution over all blocks in main memory. In essence, the IRM model favors frequency-based memory access patterns or workloads. In accordance with the IRM

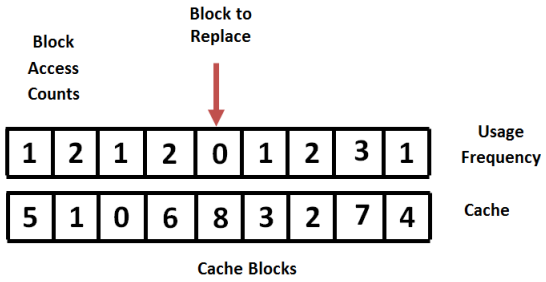

Figure 20: LFU Cache Replacement Algorithm.

model, the LFU cache replacement algorithm has been shown to be optimal [4, 26]. On the other hand, disadvantages of LFU include logarithmic complexity, ignoring recent memory accesses, and performing poorly as memory access patterns change, as its frequency bias can often result in stale cache contents that may not be referenced again.

Least Recently/Frequently Used (LRFU)

The Least Recently/Frequently Used (LRFU) replacement algorithm [40, 41], utilizes a weighted recency and frequency (CRF) metric that effectively combines and subsumes the LRU and LFU replacement algorithms. A self-adaptive tuning parameter λ acts as the weighting factor. When λ approaches a value of 1, the CRF value will favor access recency causing LRFU to behave like LRU. When λ approaches a value of 0, the CRF value will favor access frequency causing LRFU to behave like LFU. One of the major disadvantages of the LRFU approach is its high computational overhead.

Clock/ClockPro

The Clock replacement algorithm [2, 27], was designed to approximate the LRU algorithm. Advantages include implementation simplicity, minimal hardware support required, and it outperforms the FIFO algorithm. Disadvantages include a relatively weak estimation of LRU algorithm, poor performance for memory access patterns with weak locality properties, and a failure to account for frequency of access. Clock is appropriately named after the means by which pages are selected for replacement. Figure 21 illustrates the operation of the algorithm [2, 27]. Each cache block maintains a bit known as the clock bit representing recent usage. Each cache set requires a pointer called the clock hand responsible for tracking the next cache block to examine. As each cache block is examined, its usage bit is extracted and inserted into the MSB of a k-bit shift register as the previous register contents are shifted right by a single bit position. If the usage bit is set, the cache block remains cached, otherwise, it is evicted. The usage bit is subsequently cleared with the clock hand pointing to the current circular list entry is advanced. A revised version of the algorithm, ClockPro [35], extended the basic Clock algorithm to consider both access recency and access frequency. Figure 22 illustrates the operation of the algorithm [35]. Access frequency

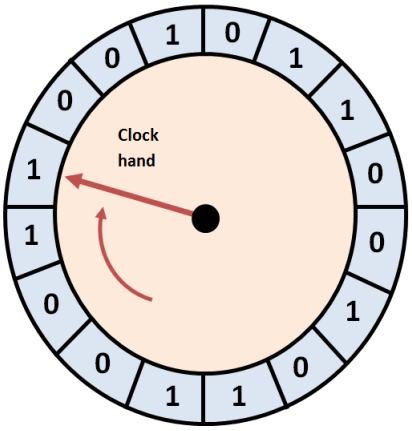

Figure 21: Clock Cache Replacement Algorithm.

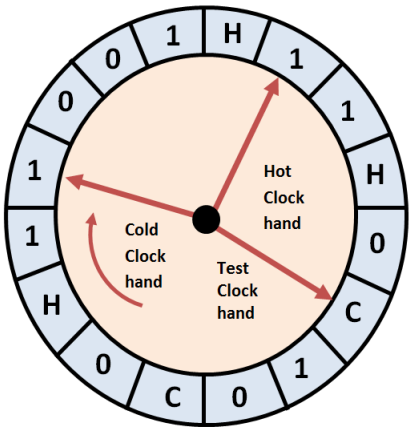

Figure 22: ClockPro Cache Replacement Algorithm.

information is added where each cache block is categorized as a hot block (frequently accessed) or a cold block (not frequently accessed). This is implemented by three clock hands called hothand, coldhand, and testhand. The coldhand pointer points to the least recently used cold cache block and is used to identify candidate cold cache blocks to replace. The hothand pointer points to the hot cache block least recently used and is used to modify its hot status to cold if not recently accessed. The testhand pointer is used to test cold cache blocks as a means to identify resident cold blocks that are replacement candidates. Each cache block maintains three status bits, namely, clock bit, test, and hot. The hot bit categorizes the block as either hot or cold. The test bit is used for cold cache blocks to indicate its usage is being evaluated or tested for replacement during the test period. If the cold cache block is not accessed during the test period once terminated by the hothand, then it is evicted from cache memory. Advantages include consideration of both frequency and recency data, implementation simplicity compared to LRU, minimal additional hardware support in the form of three bits per cache block, three address pointers for each cache set, and tangible performance improvements over the Clock algorithm. Disadvantages include increased

implementation complexity compared to Clock. Clock and ClockPro are primarily used in virtual memory (VM) applications.

Segmented LRU (SLRU)

The segmented least recently used (SLRU) replacement algorithm uses an approach whereby the cache is divided into two distinct segments. These segments are called the probationary segment and the protected segment. The cache contents of each segment are stored in order of most recently accessed to least recently accessed. Data from cache hits are moved to the protected segment and placed in the cache block or slot corresponding to the most recently accessed information. The data movement to the protected segment may displace a cache block already stored which is moved to the most recently used location of the probationary segment. If the probationary segment is full, data must be discarded from the cache, where cache blocks stored in the least recently used location are chosen for replacement. Figure 23 illustrates the operation of the algorithm. Likewise, data from cache misses are moved to the probationary segment

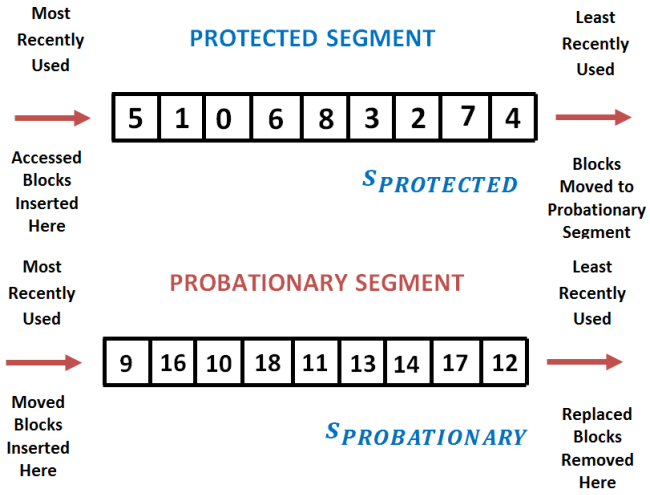

Figure 23: Segmented LRU Cache Replacement Algorithm.

and placed in the cache block or slot corresponding to the most recently accessed information. One can readily see that cache blocks stored in the protected cache segment have been accessed a minimum of two times. The sizes of the probationary and the protected segments is a key SLRU algorithm parameter set in accordance with expected memory access patterns.

Low Inter-Reference Recency Set (LIRS)

The Low Inter-Reference Recency Set (LIRS) replacement algorithm [36], like the SLRU approach, divides the cache into two distinct partitions or sets. These sets are called the High Inter-Reference Recency (HIR) block set and Low Inter-Reference Recency (LIR) block set. The LIR set comprises the majority of the cache (size L_{LIRS}) while the HIR set comprises a minority (size L_{HIRS}). LIRS records history

information on each block called the recent Inter-Reference Recency (IRR) metric. In the LIRS approach, the IRR metric measures the number of other memory blocks that are accessed between two consecutive references to a specific cache block. In cache literature, the term reuse distance [29] is frequently used. LIRS operates under the assumption that cache blocks with large IRR values are likely to continue to be large on the next IRR metric update. In accordance with this assumption, cache blocks with large IRR values are selected for replacement. The reasoning behind this assumption is cache blocks with large IRR values are more likely to be later evicted before being re-referenced again. In other words, LIRS attempts to keep cache blocks in the LIR set in the cache and selects blocks from the HIR set for replacement. Cache blocks move between the LIR and HIR sets when the updated IRR value of a cached HIR block is smaller than the maximum recency values of all LIR cache blocks. In this scenario, the LIR/HIR statuses of the HIR block and the LIR block with the maximum recency value are then switched. In practice, a maximum recency threshold R_{MAX} is used to trigger LIR/HIR status switching. When an HIR cache block has an updated IRR value smaller than the threshold, the associated cache block re-categorization occurs. In essence, the threshold controls the relative difficulty or ease that an HIR cache block becomes an LIR cache block.

Adaptive Replacement Cache (ARC)

The Adaptive Replacement Cache (ARC) replacement [50], is an adaptive online algorithm that analyzes recent memory access patterns to dynamically prioritize between recency favorable workloads and frequency favorable workloads. Like the SLRU and LIRS approaches, ARC divides the cache into two distinct partitions or sets, denoted as L_1 and L_2. Set L_1 contains cache blocks that have been accessed once, whereas set L_2 contains cache blocks that have been accessed at least twice. L_1 emphasizes access recency whereas L_2 emphasizes access frequency. Figure 24 illustrates the operation of the ARC algorithm. The ARC sets L_1 and L_2 can be thought of as a cache directory with twice the number of cache blocks as the cache memory contains. If we let c denote the number of memory blocks that are resident in the cache memory, then the size of the cache directory sets L_1 and L_2 is $2c$. ARC has been shown to be what is known as scan-resistant, where "one access only memory" requests remain in the L_1 partition of the ARC cache without evicting cache blocks with higher temporal locality stored in the L_2 partition. L_1 and L_2 are further subdivided into top and bottom subsets denoted as T_1, B_1, T_2, and B_2. Memory blocks resident in the cache are stored in subsets T_1 and T_2 whose aggregate size is always c. The size of set T_2, denoted by p, is the set with higher locality memory blocks. The parameter p is known as the ARC algorithms adaptive parameter. The size of set T_1 is therefore $c-p$. Sets B_1 and B_2 contain memory blocks not stored in the cache whose memory accesses conform to membership in sets L_1 and L_2 respecti-

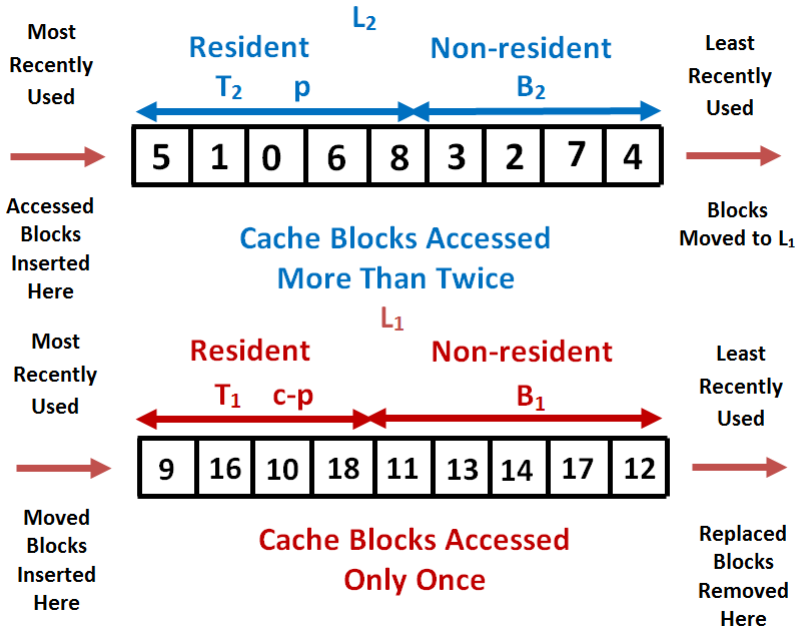

Figure 24: ARC Cache Replacement Algorithm.

vely. The online nature of ARC results in continuous updates to the p parameter according to memory access patterns. At a high level, pseudo cache hits in the B_1 subset cause p to increase, whereas pseudo cache hits in the B_2 subset cause p to decrease. The size of the increase or decrease, denoted by learning rate variables δ_1 and δ_2, is a function of the relative sizes of subsets B_1 and B_2. Similar in some respects to the SLRU approach, ARC differs from SLRU in its dynamic adjustment of the sizes of the protected segment (L_2) and the probationary segment (L_1). ARC attempts to predict the best possible partition of cache memory by analyzing recently evicted cache blocks. In fiduciary terms, ARC invests in the cache directory subset exhibiting the best performance in the recent past. Basically, ARC attempts to strike a balance between the LRU and LFU approaches.

Clock with Adaptive Replacement (CAR)

The Clock with Adaptive Replacement (CAR) replacement algorithm [8], combines the adaptive replacement (ARC) and the Clock replacement algorithms. Like ARC, CAR maintains two ordered sets: L_1 stores recently accessed memory blocks and L_2 stores frequently accessed memory blocks. L_1 and L_2 are cache directories partitioned into subsets of resident cache blocks (T_1 and T_2) and non-resident cache blocks (B_1 and B_2). The sizes of the resident cache block sets T_1 and T_2 are dynamically adjusted in accordance with memory access patterns as per the parameter p. Further, as in ARC, the parameter p is updated as pseudo cache hits occur to blocks in the non-resident cache block sets B_1 and B_2. The difference with the CAR approach is the subsets T_1 and T_2 are managed using the Clock algorithm. The benefit of using the Clock algorithm to manage T_1 and T_2 is a simpler hardware implementation. As the

sizes of T_1 and T_2 vary, the new entries are added to the list tail.

Clock with Adaptive Replacement with Temporal Filtering (CART)

The Clock with Adaptive Replacement with Temporal filtering (CART) replacement algorithm [8], was designed to prevent correlated memory references from polluting the T_2 cache subset. Correlated memory references [61] are basically successive short-term cache hits that may not correspond to the long-term utility that T_2 embodies. The result of such cache pollution is a reduction in cache hit performance. This phenomenon results from the fact that two successive memory references form the criteria by which ARC and CAR use to characterize potential long-term utility of cache blocks. The improvement offered by CART to address this limitation is the creation of what is termed a temporal filter whose objective is to incorporate a temporal characteristic into the long-term utility test. The mechanism behind this temporary filter is the use of a temporal locality window to distinguish between short term and long-term successive memory references. Short term successive memory references fall inside the temporal locality window whereas long term successive memory references fall outside. In CART, the temporal locality window is an adaptive parameter updated in accordance with memory access patterns. Application of the temporal filter assigns a utility status to each cache block, S for short term utility and L for long term utility. The utility status is updated when cache misses occur. With respect to the ARC sets, the following conventions are used: 1) cache blocks in T_2 and B_2 are assigned a long term utility status L; 2) cache blocks in B_1 are assigned a short term utility status L; and 3) cache blocks in T_1 may be assigned either short term utility status L or long term utility status L. Cache blocks at the head of T_1 are replaced if the reference bit is unset and the filter status is short term L, otherwise if the filter status is long term L, the cache block is moved to the tail of T_2. Cache blocks at the head of T_1 are moved to the tail of T_1 if the reference bit is set and the filter status is short term L. Cache blocks at the head of T_2 are replaced if the reference bit is unset. Cache blocks at the head of T_2 are moved to the tail of T_1 if the reference bit is set. All cache block movement results in the reference bit being cleared. CAR/CART outperforms LRU and CLOCK, while offering comparable performance to ARC.

Summary

In this chapter, we briefly built the essential technical background required by the other chapters. Since our work is cross-disciplinary, we carefully introduced the basic overview on notations, concepts, and models, to make readers more comfortable with the main concepts covered in this document. While we only provided a concise yet complete overview on related concepts and theory, additional details can be found in relevant textbooks and research papers. In the next chapter, we will extend our discussion to present a new approach for minimizing cache overhead in a limited preemption model. We achieve this via

an interdependent approach for computing CRPD and integrating it into a new algorithm for preemption point placement in a linear control flow graph executing on a uniprocessor system.

CHAPTER 4 CRPD COMPUTATION

In the previous chapter, we introduced various cache models for uniprocessor systems to provide the reader with the necessary background information. In this chapter, we present our research regarding a new approach for computing CRPD that emphasizes the interdependence of preemption points along with accounting for the cache memory blocks that are reloaded thereby realizing significant accuracy gains.

This chapter presents a methodology for analyzing cache related preemption delay in uniprocessor hard-real-time systems. The first section presents brief introduction and overview of this research. The second section provides an overview of the hardware, real-time, and cache models used throughout the chapter. The enhanced CRPD computation approach supporting both linear and conditional control flow graphs is detailed in the third section. The fourth section illustrates method of computing loaded cache blocks using a concrete example. The concept of LCB interdependence is discussed in the fifth section. Finally, the sixth section gives a chapter summary.

Introduction

The importance of CRPD in schedulability analysis stems from it comprising the majority of preemption overhead. CRPD occurs when a task denoted τ_i is preempted by one or more higher priority tasks denoted τ_k. The execution of high priority tasks results in the eviction of cache memory blocks that must be subsequently reloaded when task τ_i resumes execution. Two primary models of CRPD computation exist, 1) the independent CRPD cost model, and 2) the interdependent CRPD cost model. The vast majority of CRPD research falls under the independent CRPD model. Here, costs are solely a function of the preemption location under consideration. Since the next preemption may occur at any forward point in the task code, independent CRPD methods must conservatively utilize the next code location corresponding to the maximum CRPD cost. The interdependent CRPD cost model, however, overcomes this limitation by considering and computing costs between each pair of task code locations thereby achieving more accuracy. A key factor in preemption location decisions, CRPD cost accuracy is of paramount importance to schedulability and preemption placement algorithms.

This chapter discusses the following important contribution:

- We propose a new CRPD metric, called *loaded cache blocks (LCB)* which accurately characterizes the CRPD a real-time task may be subjected to due to the preemptive execution of higher priority tasks. Our objective is to account for the cache blocks that are reloaded between each set of potential preemption points.

Interdependent CRPD Computation

While existing research has focused on computing the upper bounds on cache related preemption delay (CRPD), our approach achieves higher accuracy by computing the *loaded cache blocks* (LCBs), as defined below, due to higher priority task preemption by using the set of potential/chosen preemption locations. The sets of various cache blocks are represented as sets of integers. We employ standard definitions of CRPD terms UCB and ECB as described by [5, 39].

Definition 1. ***Useful Cache Block (UCB)****: A memory block m is called a useful cache block at program point $\delta_i^{j_1}$, if*

(a) m may be cached at $\delta_i^{j_1}$ and

(b) m may be reused at program point $\delta_i^{j_2}$ that may be reached from $\delta_i^{j_1}$ without eviction of m on this path.

More formally, cache block $m \in UCB(\tau_i)$ if and only if τ_i has m as a useful cache block in some cache-set s. Note this definition of *UCB* embodies a task level view. Cache block $m \in UCB_{\text{out}}(\delta_i^j)$ if and only if δ_i^j has m as a useful cache block in some cache-set s where

$$UCB(\tau_i) = \bigcup_{\delta_i^j \in \tau_i} UCB_{\text{out}}(\delta_i^j). \tag{3}$$

The notation $UCB_{\text{out}}(\delta_i^j)$ is the set of useful cache blocks cached in task τ_i post-execution of basic block δ_i^j. Similarly, the notation $UCB_{in}(\delta_i^j)$ is the set of useful cache blocks cached in task τ_i pre-execution of basic block δ_i^j.

Definition 2. ***Evicting cache block (ECB)****: A memory block m of the preempting task is called an evicting cache block, if it may be accessed during the execution of the preempting task.*

Cache block $m \in ECB(\tau_k)$ if and only if τ_k may evict m in some cache-set s. Note this definition of *ECB* also embodies a task level view. Cache block $m \in ECB(\delta_i^j)$ if and only if δ_k^j may evict m in some cache-set s where

$$ECB(\tau_k) = \bigcup_{\delta_k^j \in \tau_k} ECB(\delta_k^j). \tag{4}$$

The notation $ECB(\delta_k^j)$ is the set of evicting cache blocks accessed in task τ_k during execution of basic block δ_k^j. In order to determine which cache blocks may be reloaded once preemption occurs, we introduce the notion of an accessed useful cache block (AUCB).

Definition 3. ***Accessed useful cache block (AUCB)****: A memory block of the preempted task is called an accessed useful cache block if it may be accessed during the execution of a basic block δ_i^j for task τ_i.*

The term $AUCB_{\text{out}}(\delta_i^j)$ represents the useful cache blocks (UCBs) accessed by task τ_i during execution of basic block at location δ_i^j. The definition of *AUCB* is introduced to capture the set of task accessed memory at a specific basic block location subsequently used in the calculation of blocks that must be reloaded when task preemptions occur. We compute the set of accessed useful cache blocks (AUCBs) as follows:

$$AUCB_{\text{out}}(\delta_i^j) = UCB_{\text{out}}(\delta_i^j) \cap ECB(\delta_i^j). \tag{5}$$

where $UCB_{\text{out}}(\delta_i^j)$ is the set of useful cache blocks for task τ_i post basic block δ_i^j execution; and $ECB(\delta_i^j)$ denotes the set of cache blocks accessed in task τ_i during execution of basic block δ_i^j. It is important to note that only cache block evictions due to preemption are considered, as intrinsic cache misses are captured as part of WCET analysis in the term C_i^{NP}. Using the previously defined terms, we may now define and explicitly compute the cache blocks that are re-loaded due to preemption which are called loaded cache blocks (LCBs).

Definition 4. ***Loaded cache block (LCB)****: A memory block of the preempted task is called a loaded cache block, if it may be re-loaded during the non-preemptive region (i.e., within a series of basic blocks with no preemptions) immediately following a preemption.*

$LCB(\delta_i^{curr}, \delta_i^{next})$ denotes the set of cache blocks re-loaded during execution of the non-preemptive region between basic block δ_i^{curr} and basic block δ_i^{next}, resulting from preemption of task τ_i, where basic block location δ_i^{curr} and δ_i^{next} are the potential/selected preemption point and next potential/selected preemption point respectively. In our model, a preemption point located at basic block δ_i^{curr} occurs at the edge between δ_i^{curr} and δ_i^{curr+1}.

The definition of *LCB* is introduced to capture the set of reloaded cache memory at a specific basic block as a function of the current and next selected preemption points. Here, we account for the overhead within a non-preemptive region for reloading UCBs that could have potentially been evicted by the preemption occurring immediately after δ_i^{curr} and used by some basic block prior to the preemption occurring immediately after δ_i^{next}.

$$LCB(\delta_i^{curr}, \delta_i^{next}) = [UCB(\delta_i^{curr}) \cap [\cup_{\nu \in \lambda} AUCB(\delta_i^{\nu})]] \cap [\cup_{\tau_k \in hp(i)} ECB(\tau_k)] \tag{6}$$

$$\text{where } \lambda \stackrel{\text{def}}{=} \{\nu | \nu \in p; p \in P_i(G_i, \delta_i^{curr}, \delta_i^{next})\}$$

The linear flowgraph structure is a special case of the conditional flowgraph structure as shown in Figure 8 where the equation for computing loaded cache blocks *(LCBs)* exhibits a simpler form as shown in

Equation 7:

$$\mathit{LCB}(\delta_i^{curr}, \delta_i^{next}) = [\mathit{UCB}_{\text{out}}(\delta_i^{curr}) \cap [\cup_{\nu \in \lambda} \mathit{AUCB}_{\text{out}}(\delta_i^{\nu})]] \cap [\cup_{\tau_k \in hp(i)} \mathit{ECB}(\tau_k)] \tag{7}$$

where $\lambda \stackrel{\text{def}}{=} \{\nu | \nu \in [\mathit{curr}{+}1, \mathit{curr}{+}2, \ldots, \mathit{next}]\}$ and δ_i^{curr} represents the current selected preemption point where $\delta_i^{curr} \in \rho_i$ and δ_i^{next} represents the next selected preemption point where $\delta_i^{next} \in \rho_i$, and $\rho_i \subseteq \{\delta_i^0, \delta_i^1, \ldots, \delta_i^{N_i}\}$ is an ordered set by ascending index of selected preemption points for task τ_i:

$$\rho_i \stackrel{\text{def}}{=} \{\delta_i^m | \delta_i^m \text{ is a selected preemption point of task } \tau_i \ \wedge m \in [0, 1, 2, \ldots, N_i]\}$$

This formula for $\mathit{LCB}(\delta_i^{curr}, \delta_i^{next})$ results in the accounting of loaded cache blocks where the preemption occurs. Once we have the set of cache blocks that must be re-loaded due to preemption, the CRPD related preemption overhead may be computed as shown below.

$$\gamma_i(\delta_i^{curr}, \delta_i^{next}) = |\mathit{LCB}(\delta_i^{curr}, \delta_i^{next})| \cdot BRT. \tag{8}$$

where BRT is the cache block reload time; and $\mathit{LCB}(\delta_i^{curr}, \delta_i^{next})$ represents the loaded cache blocks or memory accessed by the preempted task τ_i at basic block δ_i^{curr} caused by higher priority preempting tasks. The modified preemption cost as a function of the current and next preemption points is given by:

$$\xi_i(\delta_i^{curr}, \delta_i^{next}) \ = \ \gamma_i(\delta_i^{curr}, \delta_i^{next}) + \pi + \sigma + \eta(\gamma_i(\delta_i^{curr}, \delta_i^{next})). \tag{9}$$

where π is the processor pipeline cost, σ is the scheduler processing cost, and $\eta()$ is the front side bus contention resulting from the cache reload interference as described in [55–57]. For convenience, Table 3 summarizes the terminology presented in this chapter.

To illustrate our approach for computing LCBs, two supporting examples are presented. The first example demonstrates how LCBs are computed. The second example illustrates the interdependence of LCBs.

Example LCB Calculation

To illustrate our approach for computing LCBs, consider the following example. Assume we have two tasks, τ_1 and τ_2 with UCBs and ECBs for each listed in Figure 25. Please note that we have not shown the ECBs, UCBs, or AUCBs in the following figures for δ_i^0 as it is a dummy basic block with no elements for each of the aforementioned sets. The computation for LCBs uses the accessed useful cache blocks (AUCBs) since the cache blocks that are re-loaded during execution of the non-preemptive region between preemption points is a function of the memory that is explicitly accessed by the preempted

Term	Description
$AUCB_{\text{out}}(\delta_i^j)$	Accessed UCBs post basic block j execution
BRT	Cache memory block reload time
δ_i^{curr}	Basic block $curr$ denoting the current preemption
δ_i^{next}	Basic block $next$ denoting the next preemption
$ECB(\delta_k^j)$	Accessed cache memory blocks during task τ_k basic block j execution
$ECB(\tau_k)$	Accessed cache memory blocks during task τ_k execution
$\xi_i(\delta_i^{curr}, \delta_i^{next})$	Preemption cost for preemptions at basic blocks δ_i^{curr} and δ_i^{next}
$LCB(\delta_i^{curr}, \delta_i^{next})$	Loaded Cache Blocks for preemptions at δ_i^{curr} and δ_i^{next}
$\eta(\gamma_i)$	The front side bus contention resulting from the cache reload interference for preemptions at δ_i^{curr} and δ_i^{next}
ρ_i	The set of preemption points selected for task τ_i
π	The processor pipeline cost
σ	The scheduler processing cost
$UCB_{\text{in}}(\delta_i^j)$	The set of Useful Cache Blocks before task τ_i basic block j execution
$UCB_{\text{out}}(\delta_i^j)$	The set of Useful Cache Blocks after task τ_i basic block j execution
$UCB(\tau_i)$	The set of Useful Cache Blocks during task τ_i execution
$\gamma_i(\delta_i^{curr}, \delta_i^{next})$	LCB cardinality for preemptions at basic blocks δ_i^{curr} and δ_i^{next}

Table 3: CRPD Terminology

task. The computed AUCBs for each task is shown in Figure 26. In accordance with Equation 7 one can readily see that the AUCBs are simply the intersection of the UCBs and ECBs for each basic block. In our example, assume preemptions are taken at basic blocks δ_1^2 and δ_1^4 for task τ_1. For simplicity, we

Task ID	Evicting Cache Blocks ECB(δ_i^j)				
	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5
τ_1	{1,2}	{3,4,8}	{4,5,6,8}	{1,2,7,8}	{1,2,7,8}
τ_2	{1,9}	{3,10}	{11,12}	{5,7,13}	{1,3,7,8}

Task ID	Useful Cache Blocks $UCB_{out}(\delta_i^j)$				
	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5
τ_1	{1,2}	{1,2,4,8}	{1,2,8}	{1,2,7,8}	{1,2,7,8}
τ_2	{1}	{1,3}	{1,3}	{1,3,7}	{1,3,7,8}

Figure 25: Taskset ECBs and UCBs.

Task ID	Accessed Useful Cache Blocks $AUCB_{out}(\delta_i^j)$				
	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5
τ_1	{1,2}	{4,8}	{8}	{1,2,7,8}	{1,2,7,8}
τ_2	{1}	{3}	{1,3}	{7}	{1,3,7,8}

Figure 26: Taskset AUCBs.

calculate the LCBs associated with these two preemption points. For $LCB(\delta_1^2, \delta_1^4)$, we have $UCB(\delta_1^2) = \{1, 2, 4, 8\}$. The second expression is the set of memory that is accessed in basic blocks δ_1^3 and δ_1^4, namely $\{8\} \cup \{1, 2, 7, 8\} = \{1, 2, 7, 8\}$ comprising the set of AUCBs. The third expression is the set of ECBs for task τ_2 where $ECB(\tau_2) = \{1, 3, 5, 7, 8, 9, 10, 11, 12, 13\}$. Thus, $LCB(\delta_1^2, \delta_1^4)$ is given by the

intersection of the three sets:

$$LCB(\delta_1^2, \delta_1^4) = \{1,2,4,8\} \cap \{1,2,7,8\} \cap \{1,3,5,7,8,9,10,11,12,13\} = \{1,8\}$$

The preemption cost $\gamma(\delta_1^2, \delta_1^4)$ for a BRT = 390μs is given by:

$$\gamma(\delta_1^2, \delta_1^4) = |\{1,8\}| \cdot 390 = 780\mu s$$

Using the same method, $LCB(\delta_1^4, \delta_1^5)$ is given by:

$$LCB(\delta_1^4, \delta_1^5) = \{1,2,7,8\} \cap \{1,2,7,8\} \cap \{1,3,5,7,8,9,10,11,12,13\} = \{1,7,8\}$$

The preemption cost $\gamma(\delta_1^4, \delta_1^5)$ for a BRT = 390μs is given by:

$$\gamma(\delta_1^4, \delta_1^5) = |\{1,7,8\}| \cdot 390 = 1170\mu s$$

Using the method illustrated here, the preemption cost matrix entries for each pair of basic blocks are computed in a similar fashion and used as input to our preemption point placement algorithm.

Example of LCB Interdependence

To further exemplify the interdependence of preemption points, consider the example shown below. In order to account for all re-loaded cache blocks (LCBs), preemptions are always included at the first basic block δ_i^0 and the last basic block $\delta_i^{N_i}$ as shown in Figure 27. This is commensurate with the preemptions that occur before and after the task executes. Assume we have two tasks where τ_2 contains four basic blocks which may be preempted by task τ_1. For simplification, assume that the ECBs of task τ_1 evicts all UCBs of task τ_2. Let us further assume that we have $\rho_2 = \{\delta_2^0, \delta_2^1, \delta_2^2, \delta_2^4\}$. Using our LCB computation approach, only the re-loaded lines as captured in the terms $LCB(\delta_2^1, \delta_2^2)$ and $LCB(\delta_2^2, \delta_2^4)$ are included in the C_2 computation. $LCB(\delta_2^0, \delta_2^1) = 0$ as no LCBs have been cached until after execution of basic block δ_2^1.

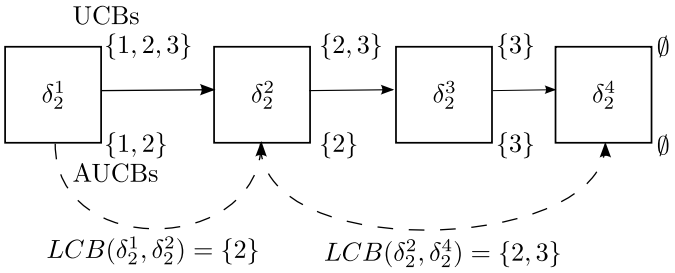

Figure 27: LCB Interdependence

Summary

In this chapter, we presented a new method for computing a substantially improved interdependent CRPD metric. Our novel new CRPD metric is utilized in the following chapters supporting our work in preemption placement algorithms for linear and conditional CFGs.

CHAPTER 5 MINIMIZING CACHE OVERHEAD VIA LOADED CACHE BLOCKS AND PREEMPTION PLACEMENT

In the previous chapter, we introduced various cache models for uniprocessor systems to provide the reader with the necessary background information. In this chapter, we present our research for minimizing cache overhead via loaded cache block calculation and preemption point placement for linear control flow graphs [25].

Uniprocessor hard real-time systems have been widely using in an increasing number of real-time and embedded systems. These systems need to operate under strict timing and application design constraints. In this chapter, we discuss a theoretic framework to ensure hard-real-time deadlines on a uniprocessor platform by minimizing the cache overhead when selecting task preemption points in a limited preemption model. We use a new approach for computing CRPD that emphasizes the interdependence of preemption points along with accounting for the cache memory blocks that are reloaded thereby realizing significant accuracy gains. Also, we show how the real-time system designer can use our approach to minimize cache overhead via the optimal selection of preemption points to ensure taskset schedulability constraints are met.

This chapter presents a methodology for designing and analyzing linear control flow graphs in uniprocessor hard-real-time systems. The first section presents brief introduction and overview of this research. The second section provides an overview of the hardware, real-time, and cache models used throughout the chapter. The third section integrates our enhanced CRPD computation into taskset schedulability analysis for real-time systems scheduled with the Earliest Deadline First (EDF) algorithm. The fourth section describes our innovative preemption point placement algorithm leveraging the integrated WCET/CRPD computation to select a set of preemption points that minimizes cache overhead while ensuring taskset schedulability. The fifth section describes the results of our simulations and related experiments. Finally, the sixth section gives a chapter summary.

Introduction

One of the noticeable contributors to preemption overhead is due to cache related preemption delay (CRPD). CRPD occurs when a task denoted τ_i is preempted by one or more higher priority tasks denoted τ_k whose execution results in the eviction of cache memory blocks that must be subsequently reloaded when task τ_i resumes execution. Limited preemption approaches have the advantage of reduced blocking with a limited number of allowed preemptions while having the advantage of sections of non-preemptive regions (NPRs) reducing the preemption overhead. One promising approach to implementing a limited preemption approach is selecting preemption points for each task subject to the constraint on maximum

non-preemptive region execution time Q_i. A paper by Bertogna et al. [14] proposed and realized a linear time algorithm for selecting optimal preemption points for a sequential basic block structure. Basic blocks are the vertices V of a control flow graph (CFG) connected in a sequence by edges E representing the execution sequence of one or more job instructions. A sequential basic block structure implies that conditional logic and branches are fully contained within basic block boundaries. Existing research utilizes pessimistic CRPD costs that effectively limit the effectiveness of preemption point placement algorithms. The primary contributions outlined in this paper include improved accuracy for computing CRPD cost taking into account where preemptions actually occur, and providing an optimal preemption point placement algorithm implemented via dynamic programming using the more accurate CRPD cost. Furthermore, we demonstrate using a case study improved task set schedulability as compared to state-of-the-art methods.

This chapter discusses the following important contributions:

- We propose a new CRPD metric, called *loaded cache blocks (LCB)* which accurately characterizes the CRPD a real-time task may be subjected to due to the preemptive execution of higher priority tasks. Our objective is to account for the cache blocks that are reloaded between each set of potential preemption points.
- We show how to integrate our new LCB metric into our newly developed algorithms that automatically place preemption points supporting linear control flow graphs (CFGs) for limited preemption scheduling applications.
- We empirically evaluate the breakdown utilization of tasks utilizing our preemption point placement algorithm in a limited preemption scheduling environment. We demonstrate the superiority of our approach versus several state of the art methods.

Our enhanced CRPD computation method was presented in Chapter 4.

Integrated WCET/CRPD Calculation

The modified preemption cost as a function of the current and next preemption points is given by:

$$\xi_i(\delta_i^{curr}, \delta_i^{next}) \;=\; \gamma_i(\delta_i^{curr}, \delta_i^{next}) + \pi + \sigma + \eta(\gamma_i(\delta_i^{curr}, \delta_i^{next})). \tag{10}$$

where π is the pipeline cost, σ is the scheduler processing cost, and $\eta()$ is the front side bus contention resulting from the cache reload interference as described in [55–57]. The output of our algorithm is an optimal set of preemption points, with respect to our computed preemption cost $\xi_i()$, subject to the maximum allowable non-preemption region Q_i. The optimal set of preemption points obtained using the enhanced accuracy of our preemption cost computation is used to calculate each task's WCET with preemption overhead given by:

$$C_i = B_i(\rho_i) = C_i^{NP} + \sum_{m=1}^{|\rho_i|-1} [\xi_i(\rho_i^m, \rho_i^{m+1})] \tag{11}$$

where ρ_i^m is the m^{th} selected preemption point for task τ_i. To further clarify what we mean by preemption, we say that δ_i^m is a "preemption point" if the preemption between basic blocks δ_i^m and δ_i^{m+1} is enabled, meaning the scheduler may preempt between these two basic blocks. Commensurate with our preemption point placement algorithm discussed later, preemptions are always taken at basic blocks δ_i^0 and $\delta_i^{N_i}$ hence $\delta_i^0,\ \delta_i^{N_i} \in \rho_i$ for any feasible set of preemption points ρ_i. Supplemental clarification of this requirement is shown in the example of LCB interdependence in the appendix. The complete problem formulation with constraints for finding the minimum WCET with preemption overhead cost $B_i(\rho_i)$ for task τ_i is given by:

$$B_i(\rho_i) = \min_{\rho_i \in \tau_i} \left\{ \left[\sum_{m=1}^{|\rho_i|-1} \xi_i(\rho_i^m, \rho_i^{m+1}) + \sum_{s=1}^{N_i} b_i^s \right] \mid \Psi_i(\rho_i) \ = \ True \right\} \tag{12}$$

The selection of optimal preemption points is subject to the constraint $\Psi_i(\rho_i)$ that no non-preemptive region in task τ_i exceeds the maximum allowable non-preemption region parameter Q_i:

$$\Psi_i(\rho_i) = \left\{ \begin{array}{ll} \text{True}, & \text{if } q_i^m(\rho_i) \leq Q_i \text{ for } m \in [1, |\rho_i| - 1] \\ \text{False}, & \text{otherwise} \end{array} \right\} \tag{13}$$

where $q_i^m(\rho_i)$ represents the m^{th} non-preemptive-region (NPR) time for task τ_i, capturing the cost of the preemption ρ_i^m given that ρ_i^{m+1} is the next selected preemption point, plus the basic block cost of all blocks between ρ_i^m and ρ_i^{m+1}:

$$q_i^m(\rho_i) = \left[\xi_i(\rho_i^m, \rho_i^{m+1}) + \sum_{s:\delta_i^s \in \{\rho_i^m, \ldots, \rho_i^{m+1}\}} b_i^s \right] \tag{14}$$

Our approach employs the results of schedulability analysis and the aforementioned WCET + CRPD calculation with the maximum allowable non-preemption region parameter Q_i computed for each task τ_i. The objective is to select a subset of preemption points that minimizes each tasks WCET + CRPD parameter C_i. We introduce slight variations of terms Ψ_i, q_i, and B_i, used in solving intermediate sub-problems of our proposed algorithm. The selection of optimal preemption points is subject to the constraint that no non-preemptive region in task τ_i exceeds the maximum allowable non-preemption region parameter Q_i:

$$\Psi_i(\delta_i^j, \delta_i^k) = \left\{ \begin{array}{ll} \text{True}, & \text{if } q_i(\delta_i^j, \delta_i^k) \leq Q_i \text{ for } \delta_i^j, \delta_i^k \in \rho_i \\ \text{False}, & \text{otherwise} \end{array} \right\} \tag{15}$$

where $q_i(\delta_i^j, \delta_i^k)$ represents a possible candidate optimal non-preemptive-region (NPR) time with successive preemption points at basic block locations δ_i^j and δ_i^k for task τ_i:

$$q_i(\delta_i^j, \delta_i^k) = \left[\xi_i(\delta_i^j, \delta_i^k) + \sum_{n=j+1}^{k} b_i^n\right] \tag{16}$$

The next expression gives a recursive solution to the CRPD+WCET minimization problem for the subproblem of the first i basic blocks. We compute the WCET including the preemption point δ_i^k using the term $B_i(\delta_i^k)$ for task τ_i as given by:

$$B_i(\delta_i^k) = \min_{m \in \{0,1,\ldots,k-1\}} \left\{ \left[B_i(\delta_i^m) + q_i(\delta_i^m, \delta_i^k)\right] \mid \Psi_i(\delta_i^m, \delta_i^k) \; = \; True \right\} \tag{17}$$

when $k \; \in \; \{1, 2, \ldots, N_i\}$. For the base case where k = 0, we have $B_i(\delta_i^0) = 0$. This recursive formulation is necessary for developing the dynamic programming solution presented in the next section. The following theorem shows our problem has optimal substructure, and thus the formulation of Equation 17 represents an optimal solution.

Theorem 1. *The WCET + CRPD cost variable $B_i(\delta_i^k)$ utilized in Equation 17 exhibits optimal substructure.*

Proof. Let Δ_i^k be the subproblem of the sequential control flowgraph containing basic blocks $\{\delta_i^0, \delta_i^1, \ldots, \delta_i^k\}$. To prove optimal substructure, we show that we can obtain the optimal set of preemption points for minimizing the WCET+CRPD for any Δ_i^k by using the optimal solutions to subproblems $\Delta_i^0, \Delta_i^1, \ldots, \Delta_i^{k-1}$. Let $B_i(\delta_i^0), B_i(\delta_i^1), \ldots, B_i(\delta_i^{k-1})$ represent the cost of the optimal solution to these subproblems. We need to show that Equation 17 represents the optimal cost to Δ_i^k.

By way of contradiction, assume there is a better feasible solution ρ_i' for the sub-problem of determining the optimal limited preemption execution costs from basic block δ_i^0 to basic block δ_i^k; that is, $B_i(\rho_i')$ is strictly smaller than the solution to Δ_i^k obtained in Equation 17 (i.e., $B_i(\delta_i^k)$). Let δ_i^ℓ (where $\ell \in \{0, 1, \ldots, k-1\}$) be the last preemption point prior to δ_i^k in the set ρ_i', and let ρ_i'' be the set of preemption points obtained from ρ_i' by removing the preemption point after δ_i^k (i.e., ρ_i'' is a solution to Δ_i^ℓ). Thus, we can represent the cost of the solution ρ_i' (i.e., $B_i(\rho_i')$ by the left-hand-side of the following inequality:

$$B_i(\rho_i'') + q_i(\delta_i^\ell, \delta_i^k) < B_i(\delta_i^k). \tag{18}$$

Since ρ_i' is a feasible solution to Δ_i^k, it must be that $\Psi_i(\delta_i^\ell, \delta_i^k)$ is true. Hence, from Equation 17 and the $\min$ operation, we can obtain an upper bound on $B_i(\delta_i^k)$ by considering the solution to subproblem

Δ_i^ℓ:

$$B_i(\delta_i^k) \leq B_i(\delta_i^\ell) + q_i(\delta_i^\ell, \delta_i^k). \tag{19}$$

Combining the inequalities of Equations 18 and 19, we finally obtain $B_i(\rho_i'') < B_i(\delta_i^\ell)$. However, this contradicts our assumption at the beginning of the proof that $B_i(\delta_i^\ell)$ represented an optimal solution to subproblem Δ_i^ℓ. Thus, a solution ρ_i' with smaller cost than $B_i(\delta_i^k)$ cannot exist, and Equation 17 computes the minimum obtainable cost for the problem Δ_i^k. □

Preemption Point Placement Algorithm

Implementing a recursive algorithm directly from Equation 17 would lead to a computationally intractable implementation. Instead, we now propose an $O(N_i^2)$ dynamic programming algorithm for computing the optimal preemption points. Our dynamic programming preemption point placement algorithm is summarized in Algorithm 1.

Algorithm 1 D.P. Optimal Preemption Point Placement

```
1:  function Select_Optimal_PPP(N_i,b_i,Q_i,ξ_i)
2:      B_i ← ∞ ρ_prev ← {δ_i^0};
3:      if b_i^k > Q_i for some k ∈ {1,...,N_i} then
4:          return INFEASIBLE;
5:      end if
6:      B_i(δ_i^0) ← 0;
7:      for k : 0 ≤ k ≤ N_i do
8:          C_i^NP(δ_i^k,δ_i^k) ← 0;
9:          q_i(δ_i^k, δ_i^k) ← 0;
10:         for j : k − 1 ≥ j ≥ 0 do
11:             C_i^NP(δ_i^j, δ_i^k) ← b_i^{j+1} + C_i^NP(δ_i^{j+1}, δ_i^k);
12:             q_i(δ_i^j, δ_i^k) ← ξ_i(δ_i^j, δ_i^k) + C_i^NP(δ_i^j, δ_i^k);
13:             if q_i(δ_i^j, δ_i^k) ≤ Q_i then
14:                 P_cost ← B_i(δ_i^j) + q_i(δ_i^j, δ_i^k);
15:                 if P_cost < B_i(δ_i^k) then
16:                     B_i(δ_i^k) ← P_cost;
17:                     ρ_prev(δ_i^k) ← δ_i^j;
18:                 end if
19:             end if
20:         end for
21:     end for
22:     ρ_i ← Compute_PPSet(N_i, ρ_prev);
23:     return FEASIBLE;
24: end function
25:
26:
27: function Compute_PPSet(N_i,ρ_prev)
28:     Computes ρ_i from ρ_prev (Details omitted)
29: end function
```

For each task τ_i, we are given the following parameters: 1) the number of basic blocks N_i , 2) the non-preemptive execution time of each basic block b_i, 3) the maximum allowable non-preemptive region Q_i,

and 4) the preemption cost matrix ξ_i. The preemption cost matrix ξ_i is organized for each basic block δ_i^j and contains the preemption cost for all successor basic blocks of the task's control flow graph. The minimum preemption cost up to all basic blocks is computed and stored in an array denoted B_i. Each entry of the B_i array is initialized to infinity. As we consider whether each basic block δ_i^k is in the set of optimal preemption points, the location of the previous basic block δ_i^j with minimal preemption cost is stored in an array denoted ρ_{prev}. The algorithm examines each basic block from δ_i^1 to $\delta_i^{N_i}$ to minimize the preemption cost by traversing backwards from the current basic block δ_i^k under consideration in order to find the basic block δ_i^j with minimal preemption cost subject to the constraint $q_i(\delta_i^j, \delta_i^k) \leq Q_i$. While each basic block will have a predecessor with minimum preemption cost, the list of selected preemption points is obtained by starting with basic block $\delta_i^{N_i}$ and hopping to the predecessor basic block stored at $\rho_{prev}(\delta_i^{N_i})$, denoted δ_i^m. Basic blocks $\delta_i^{N_i}$ and δ_i^m are added to the optimal preemption point set ρ_i. This basic block hopping process continues until basic block δ_i^0 is reached. The set ρ_i contains the complete list of selected optimal preemption points. To exemplify how our algorithm works, consider the following example.

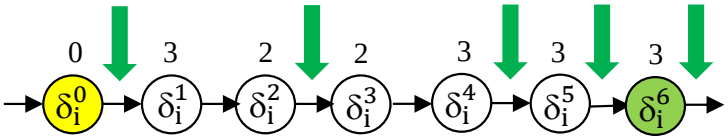

Figure 28: Linear CFG Algorithm Example.

0	-		3	-	2	-	2	-	3	-	3	-	3	-
δ_i^0	ξ_i^0		δ_i^1	ξ_i^1	δ_i^2	ξ_i^2	δ_i^3	ξ_i^3	δ_i^4	ξ_i^4	δ_i^5	ξ_i^5	δ_i^6	ξ_i^6

ξ_i	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
δ_i^0	**0**	1	2	4	4	3	2
δ_i^1	-	**0**	3	5	6	4	3
δ_i^2	-	-	**0**	8	7	5	4
δ_i^3	-	-	-	**0**	8	7	6
δ_i^4	-	-	-	-	**0**	6	7
δ_i^5	-	-	-	-	-	**0**	8
δ_i^6	-	-	-	-	-	-	**0**

Figure 29: Algorithm Example.

Let $N_i = 6$ and $Q_i = 12$ for the following basic block structure with WCET and preemption costs shown in Figure 29. The algorithm computes and stores the cumulative non-preemptive execution costs for starting and ending basic block pairs in a matrix denoted $C_i^{NP}(\delta_i^j,\delta_i^k)$. For example, $C_i^{NP}(\delta_i^1,\delta_i^3) = b_i^2 + b_i^3 = 2 + 2 = 4$. We do not include b_i^1 since the preemption occurs after execution of basic block

δ_i^1. Using this information, the combined WCET and CRPD costs for each basic block pair is computed and stored in a matrix denoted q_i. For example, $q_i(\delta_i^1,\delta_i^3) = C_i^{NP}(\delta_i^1,\delta_i^3) + \xi_i(\delta_i^1,\delta_i^3) = 4 + 5 = 9$. The remaining q_i matrix entries are computed in a similar fashion. These matrices are shown in Figure 30. The shaded cells in the q_i matrix represent cases where the combined WCET and CRPD costs for these basic block pairs exceed the maximum allowable non-preemptive region parameter Q_i. During execution of the algorithm, the minimum combined WCET + CRPD costs are computed for each basic block and stored in an array denoted B_i. Basic block pairs with preemptive costs that are less than or equal to Q_i are candidates for selection. When a lower cost is determined for a given basic block, the predecessor preemption point is updated in the ρ_{prev} array which keeps track of the selected predecessor preemption points thereby forming a daisy chain containing the entire set of selected preemption points. The final results are illustrated in Figures 30 and 31 below.

q_i	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
δ_i^0	**0**	4	7	11	14	17	19
δ_i^1	-	**0**	5	9	13	14	16
δ_i^2	-	-	**0**	10	12	13	15
δ_i^3	-	-	-	**0**	11	13	15
δ_i^4	-	-	-	-	**0**	9	13
δ_i^5	-	-	-	-	-	**0**	11
δ_i^6	-	-	-	-	-	-	**0**

Figure 30: Combined WCET and CPRD Costs.

B_i	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
δ_i^0	-	4	7	11	19	28	**39**

ρ_{PREV}	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
	$\boldsymbol{\delta_i^0}$	$\boldsymbol{\delta_i^0}$	$\boldsymbol{\delta_i^0}$	$\boldsymbol{\delta_i^0}$	$\boldsymbol{\delta_i^2}$	$\boldsymbol{\delta_i^4}$	$\boldsymbol{\delta_i^5}$

ρ_i	δ_i^0	δ_i^1	δ_i^2	δ_i^3	δ_i^4	δ_i^5	δ_i^6
	$\boldsymbol{\delta_i^0}$		$\boldsymbol{\delta_i^2}$		$\boldsymbol{\delta_i^4}$	$\boldsymbol{\delta_i^5}$	$\boldsymbol{\delta_i^6}$

0	2	3	0	2	7	2	0	3	6	3	8	3	0
δ_i^0	ξ_i^0	δ_i^1	ξ_i^1	δ_i^2	ξ_i^2	δ_i^3	ξ_i^3	δ_i^4	ξ_i^4	δ_i^5	ξ_i^5	δ_i^6	ξ_i^6

Figure 31: Algorithm Results.

The maximum blocking time Q_i that each task may tolerate utilizes the computed task WCET parameter C_i. The method for obtaining the maximum blocking time is eloquently summarized by Bertogna et al. for the Earliest Deadline First (EDF) and Fixed Priority (FP) scheduling algorithms [14] [13]. The

circular dependency between the maximum blocking time Q_i and task WCET C_i parameters suggests an iterative approach to allow the two parameters to convergence to a steady state. One such iterative approach contains the following steps as given in Algorithm 2. Convergence of Algorithm 2 results primarily from the fact that the optimal preemption points do not change unless the parameter Q_i changes. Since Q_i represents the execution time slack available to a task τ_i, as expected Q_i has been observed to exhibit non-increasing values during each subsequent algorithm iteration. The need to subsume higher level programming constructs being the prominent assumption of the linear basic block structure can potentially diminish the utility of our approach if the non-preemptive execution time of any basic block violates the constraint $C_i^{NP} > Q_i$. For this case, we need to "break-up" the basic block by permitting preemption every Q_i time units.

Algorithm 2 Iterative Schedulability and Preemption Point Placement Algorithm

1: Assume the CRPD of the task system is initially zero.
2: **repeat**
3: Run the Baruah algorithm to obtain the maximum non-preemptive region Q_i for each task.
4: Select optimal preemption points using our dynamic programming algorithm and CRPD calculation.
5: Compute the WCET + CRPD C_i from the selected preemption points.
6: **until** the selected preemption points do not change or the system is not feasible for the computed WCET + CRPD.
7: The breakdown utilization is given by U.

Evaluation

The evaluation of our preemption point placement algorithm will embody two methods: 1) characterization and measurement of preemption costs using real-time application code, and 2) a breakdown utilization schedulability comparison for various CRPD computational approaches.

Preemption Cost Characterization

To characterize the behavior and estimate the benefit of the approach proposed in this paper, a case study of representative tasks was performed. The ten tasks were randomly selected from the Malardalen University (MRTC) WCET benchmark suite [51]. Each task was built using Gaisler's Bare-C Cross Compiler [30] for the GRSIM LEON3 [31] simulated target.

Tasks were first analyzed using AbsInt's a^3 WCET [1] to determine the set of basic blocks. Next, the basic blocks $\{\delta_i^0, \delta_i^1, \delta_i^2, ..., \delta_i^{N_i}\}$ were serialized by recording their order during execution. Program points were identified as the address of the final instruction of each basic block δ_i^j for $j \in [0, N_i]$ to match the sequential basic block structure used by our preemption point placement algorithm. Each program point served as a breakpoint when running the task on the simulator. In accordance with the sequential basic

block structure used in our algorithm, only the data cache is used to compute CRPD as the linear basic block structure offers only limited opportunities for instruction cache blocks to be revisited.

During the execution of each basic block, the data cache states were saved as $\Upsilon_D(\delta_i^j)$ at each breakpoint. The cache snapshots were selected as the final visit of each program point where the data cache contents $\Upsilon_D(\delta_i^j)$ were captured and recorded. From the final $\Upsilon_D(\delta_i^j)$ snapshots, a conservative estimate of LCBs shared between program points was determined.

Shared LCBs were calculated by intersecting the cache state snapshots from δ_i^j to δ_i^k, except δ_i^k. A cache line that remains unchanged after the execution of $\{\delta_i^{j+1}, \delta_i^{j+2}, ..., \delta_i^k\}$ will be present in the cache before execution of the basic block that δ_i^k represents. It is only from these unchanged cache lines that the shared LCBs between δ_i^j and δ_i^k can be selected. The complete set of unchanged cache lines serves as a safe upper-bound on the LCBs shared between each basic block pair. The equation below formalizes this idea, using the data cache snapshots $\Upsilon_D(\delta_i^j)$.

$$LCB(\delta_i^j, \delta_i^k) \subseteq \bigcap_{m=j+1}^{k} \Upsilon_D(\delta_i^m) \tag{20}$$

Availability

This method may be verified and reproduced using the same tools and data. Gaisler's compiler and simulator are freely available. AbsInt's a^3 tool is available for educational and evaluation purposes. The programs written, and data used in our work can be found on GitHub [24] thereby permitting the research community to reproduce and leverage our work as needed.

Results

The results are presented as an illustration of the potential benefit of our proposed method, utilizing pairs of preemptions to determine costs, over methods that consider only the maximum CRPD at a particular preemption point (e.g., Bertogna et al. [14]). In terms of LCB computation this implies that the maximum LCB value over all subsequent program points must be used as the CRPD cost:

$$\max\{LCB(\delta_i^j, \delta_i^k) \mid j < k\} \tag{21}$$

In the following graphs, each point in the graph represents two points in the program. The first point of the program δ_i^j is fixed by the x-axis. The y-axis indicates the shared LCB count with a later program point. The first graph shown in Figure 32 is for the recursion program of the MRTC benchmark suite. The lines represent the minimum and maximum shared LCBs between various program points. At each point on the graph, the index of the next preemption point associated with the respective minimum or maximum

graph value at that location is shown. At program point δ_i^4, the minimum CRPD value is coupled with program point δ_i^7 having a shared LCB count of 14 whereas the single-valued CRPD computation method finds 24 shared LCBs coupled at program point δ_i^5. To compare the potential utility of a method that uses

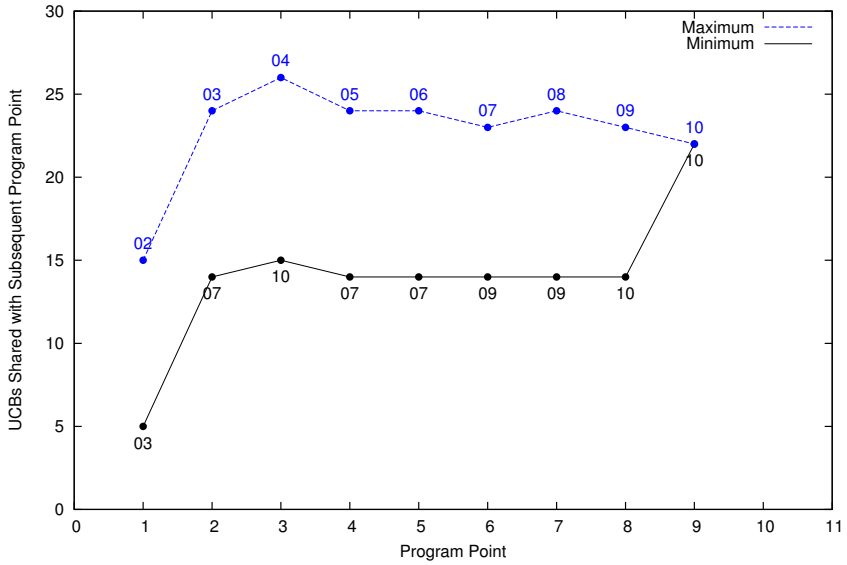

Figure 32: Recursion Data Cache.

at each preemption a single-valued CRPD and our method which determines CRPD based on a pair of adjacent preemption points, consider any program point on the horizontal access of Figures 32, 33, or 34. For any program preemption point δ_i^k, a single-valued approach would, to be safe, use the value reported in the larger-valued dashed line. However, an approach that considered pairs of preemptions, as in our approach, can reduce the value and potentially obtain a CRPD reported on the solid line. The difference between the performance of these two preemption point placement algorithms is an example of the benefit provided by considering location aware CRPD cost.

The second graph represents the lms benchmark task data cache shown in Figure 33, and the third graph represents the adpcm benchmark task data cache shown in Figure 34. Similar to the recursion benchmark data cache, the variability in the minimum and maximum shared LCBs for each program point further exemplifies the benefit of using location aware CRPD cost in preemption point placement. Maximum and minimum data cache costs for the other seven tasks show similar variability but are not shown here due to space limitations. In summary, the CRPD cost was consistently reduced or maintained compared to the single-valued approach. A maximum of 68% reduction of contributing cache lines in the bsort benchmark along with an average of 18.6% decrease over all benchmarks was notably observed. The ability of our method to leverage this reduction leads to the schedulability improvements observed in the next subsection. In examining the graphs for all tasks and data caches a few common traits were noted.

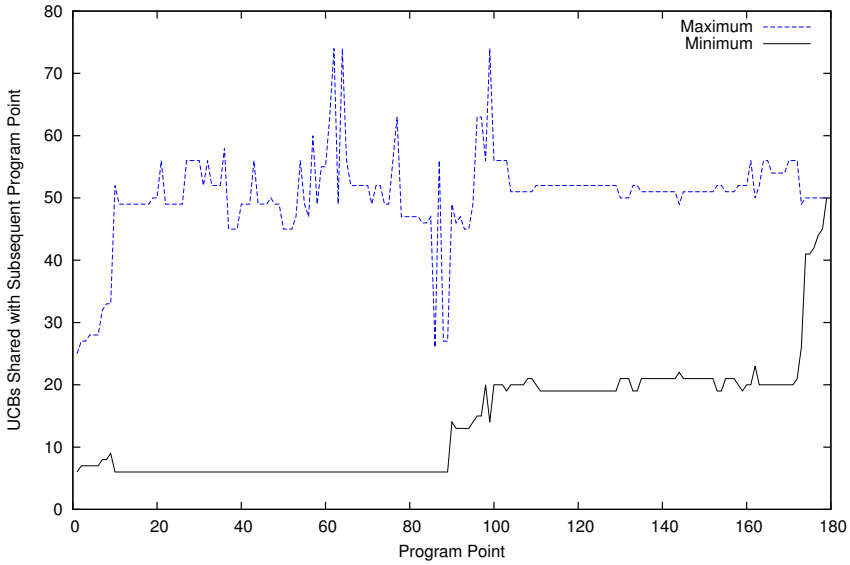

Figure 33: LMS Data Cache.

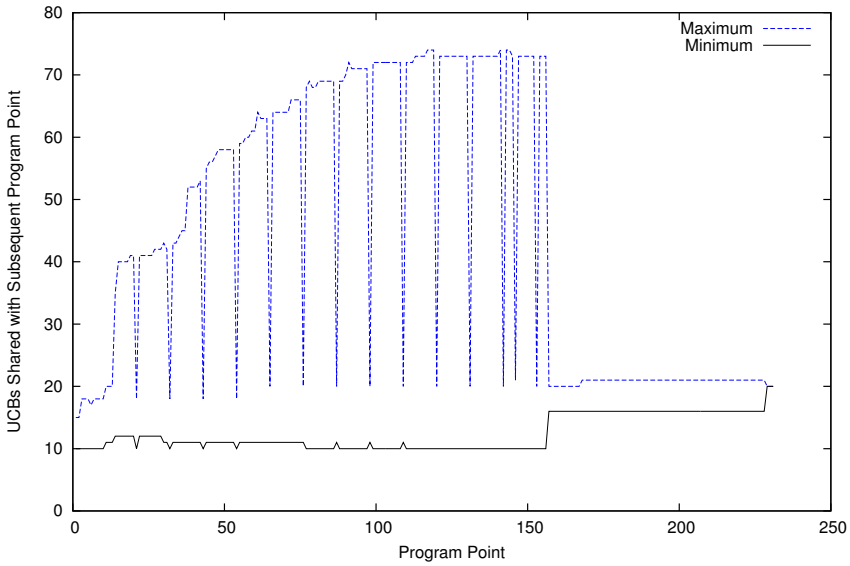

Figure 34: ADPCM Data Cache.

The minimum shared LCBs sharply increases at the end of each task's execution. This is due to the nature of the conservative estimation of shared LCBs and the tasks, and the number of instructions in the basic block between the final program points is relatively small thereby limiting the number of data cache lines that could be eliminated by the intersection.

Drastic spikes downwards in the shared LCB counts for the minimum and maximum curves coincide with function call boundaries, or large conditional blocks. At these boundaries, the maximum and minimum LCB counts are approximately the same. There is a sharp upward spike in the early program points for the maximum curves. This trend is due to the early initialization blocks built into tasks. In our analysis,

the minimum curves show a clear benefit of selecting preemption points outside of the early initialization section.

Breakdown Utilization

The previous analysis illustrates the benefit of using a more precise CRPD cost in preemption point placement thereby reducing the preemption overhead for an individual task, however, it does not address the benefits to task set schedulability. To evaluate task set schedulability benefits a second case study was performed focusing on the breakdown utilization of the MRTC WCET benchmark suite [51] for various algorithms. Our study compares the Lunniss et al. UCB only approach for EDF [47], the Bertogna et al. Explicit Preemption Point Placement algorithm [14], and our improved Explicit Preemption Point Placement algorithm. For notational convenience the UCB Only approach for EDF will be referred to as UOE, the Bertogna Explicit Preemption Point Placement algorithm as $BEPP$, and our Explicit Preemption Point Placement algorithm as EPP.

The appropriate schedulability test for UOE is comprised of three parts: $\gamma_{t,j}^{ucb}$, U_j^*, and U^* each representing the maximum CRPD for task τ_j, the utilization of task τ_j including CRPD, and the utilization of the task set respectively as documented in Lunniss *et al.* [47]. A task set is schedulable when $U^* \leq 1$.

Borrowing the breakdown utilization evaluation technique from [47], each task has its deadline and period set to $T_i = P_i = u \cdot C_i$ where u is a constant. The constant, u, begins at the number of tasks (ten) and is increased in steps of 0.25 until the task set becomes schedulable. Incremental negative adjustments are then made to determine when the set becomes un-schedulable, indicating the final breakdown utilization. For each task, the set of shared LCBs are calculated at each program point. Taking the maximum shared LCB count for any task is safe and appropriate for calculating U^*.

For $BEPP$, the maximum shared LCB counts obtained in the earlier evaluation serve as input. Lastly, for EPP, the shared LCB counts obtained in the earlier evaluation serve as input for our enhanced preemption point placement algorithm. The last input variables required for both approaches are C_i and BRT. C_i was captured as the total number of cycles required to complete the task without preemptions. The breakdown utilization study sweeps the BRT parameter from 10 μs to 390 μs, representing values across several different types of processors.

The breakdown utilization determination leverages our iterative schedulability and preemption point placement algorithm as outlined in the following steps below as given in Algorithm 3. Using ten tasks, the breakdown utilization comparison between UOE, $BEPP$, and EPP are summarized in Figure 35.

Algorithm 3 Breakdown Utilization Evaluation Algorithm

1: Start with a task system that may or may not be feasible.
2: Assume the CRPD of the task system is initially zero.
3: Run the Iterative Schedulability and Preemption Point Placement Algorithm 2
4: **if** the task system is feasible/schedulable **then**
5: Increase the system utilization by decreasing the periods via a binary search.
6: **else**
7: Decrease the system utilization by increasing the periods via a binary search.
8: **end if**
9: The breakdown utilization is given by U.

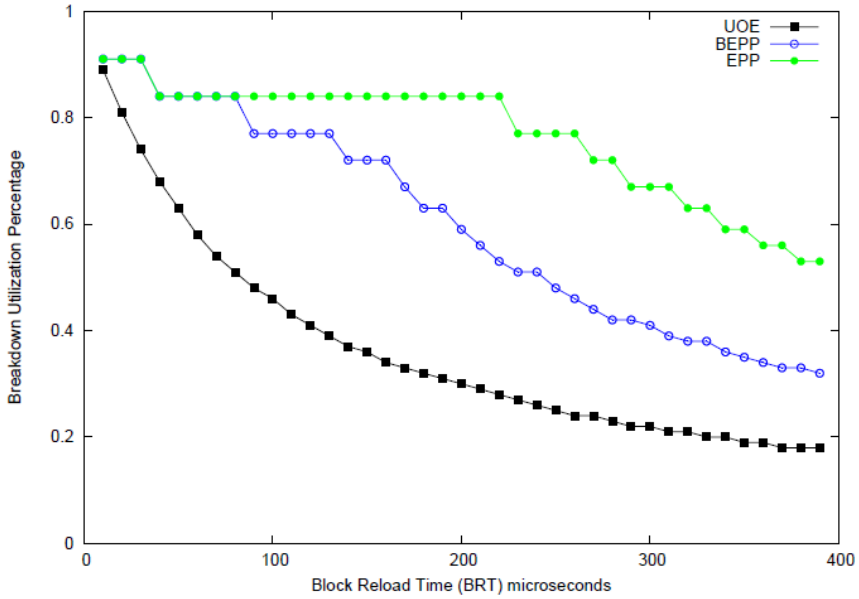

Figure 35: Breakdown Utilization Comparison.

The breakdown utilization results indicate that $BEPP$ dominates the UOE algorithm primarily due to the limited preemption model utilizing CRPD cost at the basic block level. EPP dominates $BEPP$ resulting from the more accurate location aware CRPD cost used in our preemption point placement algorithm. As expected, the breakdown utilization converges for all three methods for small BRT values as the cache-overhead becomes negligible.

Summary

In this work, we presented an enhanced approach for calculating the CRPD taking into account the selected preemption points resulting in greater accuracy. Using a more precise CRPD calculation, we also presented an improved algorithm for selecting a limited number of preemption points for each task subject to schedulability constraints. Our improved preemption placement algorithm was demonstrated to minimize the overall preemption cost, an important result in achieving schedulability in real-time systems. We highlighted the iterative nature of considering schedulability constraints in our preemption point

placement algorithm and proposed an algorithm combining schedulability analysis with limited preemption point placement. This approach effectively illustrates how the individual tasks non-preemption region parameters Q_i and the optimal selected preemption points will eventually converge. Furthermore, our enhanced algorithm was demonstrated to be optimal in that if a feasible schedule is not found, then no feasible schedule exists by any known method utilizing a static Q_i value. Our algorithm was shown to run in quadratic time complexity. Potential preemption points can be defined automatically using Gaisler's compiler and simulator along with AbsInt's a^3 tool or defined manually by the programmer during design and implementation. Our experiments demonstrated the effectiveness of the enhanced CRPD calculation by illustrating the benefits using the task set from the MRTC WCET benchmark suite [51]. We also demonstrated the benefits of our enhanced limited optimal preemption point placement algorithm and its increased system schedulability as compared to other algorithms. While our task model is defined using a linear sequence of basic blocks, it was deemed a highly suitable model to introduce our revised methods for enhanced CRPD calculation and optimal limited preemption point placement.

In future work, we plan to 1) extend the techniques described here to set-associative caches, 2) perform a schedulability comparison of synthetic task set for various preemption models, and 3) remove the linear basic block restriction thereby permitting arbitrary control flow graphs.

CHAPTER 6 REALIZING IMPROVED PREEMPTION PLACEMENT IN REAL-TIME PROGRAM CODE WITH INTERDEPENDENT CACHE RELATED PREEMPTION DELAY

In the previous chapter, we introduced innovative methods for computing accurate CRPD, integrating the enhanced CRPD with EDF schedulability analysis, and placing optimal preemption points to ensure taskset schedulability for linear control flow graphs in uniprocessor systems. In this chapter, we present our research for minimizing cache overhead via loaded cache block calculation and preemption point placement for conditional control flow graphs.

This chapter presents a methodology for designing and analyzing conditional control flow graphs in uniprocessor hard-real-time systems. The first section presents brief introduction and overview of this research. The second section provides an overview of the hardware, real-time, and cache models used throughout the chapter. The third section presents an informal problem statement outlining the various real-time disciplines that collectively comprise our work. The fourth section summarizes the fixed priority (FP) and earliest deadline first (EDF) schedulability analysis constraints for limited preemption scheduling. The real-time conditional flow graph model is detailed in the fifth section. The sixth section presents a formal problem statement describing the objective function for selecting preemptions in conditional flowgraphs. The seventh section discusses our enhanced pseudo quartic time interdependent CRPD conditional preemption point placement algorithm. A case study using MRTC benchmarks demonstrates improved task set schedulability in the eighth section. Finally, the nineth section gives a chapter summary.

Introduction

The utility of real-time system computations depends on two important properties, correctness and timeliness. The timeliness property (the subject of schedulability analysis) is concerned with ensuring real-time task computations are completed within required deadlines. Designers of real-time systems must choose the scheduling paradigm that will ultimately determine if the real-time task set will meet its timeliness objectives. The available choices are 1) non-preemptive scheduling, 2) fully preemptive scheduling, and 3) limited preemption scheduling. Non-preemptive scheduling suffers from blocking of high priority tasks and fully preemptive scheduling suffers from substantial preemption overhead (up to 44% [55–57] of a tasks WCET) each approach degrading task set schedulability. Limited preemption scheduling attempts to 1) reduce blocking by limiting the number of allowed preemptions, maximizing non-preemptive task execution and 2) reduce preemption overhead via non-preemptive regions. Regardless of the chosen scheduling paradigm, effective schedulability analysis of real-time task sets mandates accurate WCET and CRPD estimates. In this paper, the recognized benefits of limited preemption scheduling motivate our

work on PPP algorithms.

PPP algorithms select preemption points for each task to 1) minimize the contribution of CRPD to the task's overall WCET, and 2) ensure the execution time between adjacent preemptions is limited by the maximum non-preemptive region execution time. The maximum non-preemptive region execution time, denoted Q_i, is determined via task set schedulability analysis. The motivation behind our work is the utility of existing PPP algorithms are limited either by the less accurate CRPD costs or by assuming a linear code structure (i.e., no branches or loops are permitted) [13, 14, 25, 58]. Our approach removes this linear code assumption and combines the interdependent CRPD cost model with an improved PPP algorithm thereby reducing overall task WCET. The benefits of our approach will be illustrated in a case study employing real-time tasks from the MRTC benchmark suite. This chapter discusses the following important contributions:

- We revise the interdependent CRPD metric, called *loaded cache blocks (LCB)* to consider the complexity of conditional control flow graphs.
- We show how to integrate our new LCB metric into our newly developed algorithms that automatically place preemption points supporting conditional control flow graphs (CFGs) for limited preemption scheduling applications.
- Our breakdown utilization methodology evaluates the effectiveness of our conditional preemption placement algorithm using real-time code via the Malardalen MRTC benchmarks.
- We empirically evaluate the breakdown utilization of tasks utilizing our preemption point placement algorithm in a limited preemption scheduling environment. We demonstrate the superiority of our approach versus several state of the art methods.

Informal Problem Statement

In this section, we will summarize the subsequent sections and their use in our work using an informal problem statement to set the proper context.

The problem we solve in this paper is to minimize the WCET+CRPD of each real-time task τ_i in a task set τ whose code structure is represented by a real-time conditional flow graph G_i such that all tasks meet their prescribed deadlines in accordance with the employed scheduling algorithm. To achieve this objective, we propose a conditional preemption placement algorithm that computes CRPD costs using an interdependent CRPD cost model. Task preemptions are subject to the constraint that all non-preemptive regions must be less than or equal to the maximum allowable non-preemptive region parameter Q_i determined via task characteristics and the scheduling algorithm.

Our work embodies several diverse real-time disciplines, namely, graph grammars, CRPD analysis,

preemption placement, and schedulability analysis. Each discipline is discussed in a separate section for clarity. The foruth section details the fixed priority (FP) and earliest deadline first (EDF) schedulability analysis constraints for limited preemption scheduling. These scheduling constraints limit preemption placement in such a way that task set schedulability is achieved. Real-time conditional flow graphs along with an overview of graph grammars are discussed in the fifth section. Graph grammars are used to parse the real-time task code during preemption placement. Once these fundamental topics are covered, the sixth section presents a formal problem statement describing the objective function for selecting preemptions in conditional flowgraphs. Our conditional preemption placement algorithm parses the real-time task code using a set of production rules presented in the seventh section. As each production rule is applied, the WCET+CRPD cost and the associated preemption points are computed and stored for processing in subsequent production rules. Our enhanced CRPD computation method was presented in Chapter 4.

Schedulability Analysis

In this section, analysis of EDF [9, 12] and FP [72] limited preemption scheduling, summarizes the computation of the maximum non-preemptive region parameter Q_i supporting our conditional and linear preemption point placement algorithms.

The goal of schedulability analysis is to determine whether a taskset is schedulable under the worst-case task activation pattern. The worst-case activation pattern for task τ_i, known as the critical instant, results in the maximum response time. Earlier work [43] proved the critical instant for each task coincides with the synchronous activation of the task with all higher priority tasks where all jobs released immediately in accordance with the minimum inter arrival time. For FP scheduling the set of higher priority tasks is represented by:

$$k \in hp(i) = \{k \mid k < i\} \tag{22}$$

For EDF scheduling, the set of higher priority tasks is represented by:

$$k \in hp(i) = \{k \mid D_k < D_i\} \tag{23}$$

To facilitate the schedulability analysis for FP scheduling, the request bound function [42] $RBF(t)$ is used to examine the maximum cumulative execution request in an interval of length t generated by jobs of τ_i.

$$RBF_i(t) = \left\lceil \frac{t}{T_i} \right\rceil (C_i^{NP} + \gamma_i) \tag{24}$$

where γ_i denotes the preemption cost due to preemption by higher priority tasks $hp(i)$ during execution of task τ_i. To facilitate the schedulability analysis for EDF scheduling, the demand bound function $DBF(t)$ is used to examine the maximum cumulative execution request in an interval of length t generated by jobs of τ_i.

$$DBF_i(t) = \left(1 + \left\lceil \frac{t - D_i}{T_i} \right\rceil\right)(C_i^{NP} + \gamma_i) \tag{25}$$

where γ_i denotes the preemption cost due to preemption by higher priority tasks $hp(i)$ during execution of task τ_i. Starting with the critical instant, the cumulative execution request in an interval t for task τ_i and all higher priority tasks $hp(i)$ is given by:

$$W_i(t) = \sum_{j \in i, hp(i)} RBF_j(t) \tag{26}$$

Characterized by the non-preemptive regions inherent to limited preemption scheduling, the analysis must take into account the longest NP region in the lower priority tasks. The maximum NP region in task τ_i is given by:

$$q_i^{max} \leq Q_i = \min_{h \in hp(i)} \beta_h \tag{27}$$

For FP scheduling, the blocking factor B_i each task τ_i experiences is given by:

$$B_i = \max_{l \in lp(i)} \{q_l^{max}\} \tag{28}$$

where the set of lower priority tasks $lp(i)$ for FP scheduling is given by:

$$l \in lp(i) = \{l \mid l > i\} \tag{29}$$

For EDF scheduling, the set of lower priority tasks is represented by:

$$l \in lp(i) = \{l \mid D_l > D_i\} \tag{30}$$

Since the lowest priority task τ_m is subjected to no blocking, by convention we have $B_m = 0$. We can use the schedulability analysis derived for floating non-preemptive regions [15] to establish taskset schedulability:

$$W_i(t) + B_i \leq t, \; \forall \, t \; \in \; mT_i, \; m \; \in \; \mathbb{N} \tag{31}$$

Another way to look at this FP limited preemption schedulability constraint is to characterize the amount of blocking tolerance β_i that a task τ_i can withstand while meeting its deadlines.

$$\beta_i = \max_{t \in A | t < D_i} \Big\{ t - \sum_{j \in hp(i), i} RBF_j(t) \Big\} \tag{32}$$

where $A = \{mT_j, m \in \mathbb{N}, 1 \leq j < n\}$. Similarly, for EDF, we have:

$$\beta_i = \min_{t \in A | D_i \leq t < D_{i+1}} \Big\{ t - \sum_{\tau_j \in \tau} DBF_j(t) \Big\} \tag{33}$$

where $A = \{mT_j + D_j,\ m \in \mathbb{N}, 1 \leq j \leq n\}$. With suitable expressions for β_i introduced, we can express the taskset limited preemption schedulability constraint for both FP and EDF scheduling as summarized in Theorem 2 by Bertogna et al. [14].

Theorem 2. *A task set τ is schedulable with limited preemption scheduling if, for all $i \mid 1 \leq i \leq n$,*

$$B_i \leq \beta_i \tag{34}$$

We can restate the taskset schedulability constraint in terms of the maximum non-preemptive region parameter Q_i via Theorem 3 by Bertogna et al. [14].

Theorem 3. *A task set τ is schedulable with limited preemption scheduling if, for all $i \mid 1 < i \leq n$,*

$$q_i^{max} \leq Q_i = \min_{h \in hp(i)} \beta_h \tag{35}$$

In summary, each task is permitted to execute non-preemptively for a maximum amount of time denoted by Q_i thereby ensuring taskset schedulability as long as preemption points are placed such that all non-preemptive regions are less than or equal to Q_i. For convenience, Table 4 summarizes the terminology presented in this section.

Real-Time Conditional Flow Graph

The types of real-time conditional flow graphs used in our work belong to the class of graphs known as series-parallel flow graphs [37]. We use the series/parallel terminology to describe the supported graph composition steps. Series-parallel flow graphs can be created using varying sequences of four basic operations, namely graph creation, series composition, parallel composition and cyclic composition. Creation of graph G_i^A consists of two basic blocks as vertices, δ_i^s and δ_i^e, containing a connecting directed edge $e_i^{s,e} = (\delta_i^s, \delta_i^e)$ as shown in Figure 36.

Term	Description
B_i	Task τ_i blocking factor
β_i	Task τ_i blocking tolerance
$DBF_i(t)$	Demand bound function specifying the maximum cumulative execution request in an interval of length t generated by jobs of task τ_i
$hp(i)$	The set of task τ_i higher priority tasks
k	Index variable denoting the preempting task
l	Index variable denoting τ_i blocking tasks
$lp(i)$	The set of task τ_i lower priority tasks
q_j^{max}	The maximum NP execution region in task τ_j
$RBF_i(t)$	The resource bound function specifying the maximum cumulative execution request in an interval of length t generated by jobs of task τ_i
t	Cumulative task set execution time
$W_i(t)$	The cumulative execution request for task τ_i
γ_i	The preemption cost during task τ_i execution

Table 4: Schedulability Terminology

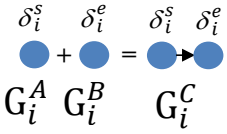

Figure 36: Graph Creation.

Series composition takes two disjoint graphs, G_i^A and G_i^B creating a new graph $G_i^C$with a connecting directed edge $e_i^{e^a,s^b} = (\delta_i^{e^a}, \delta_i^{s^b})$ where $\delta_i^{e^a}$ represents the sink node of graph G_i^A and $\delta_i^{s^b}$ represents the source node of graph G_i^B as shown in Figure 37.

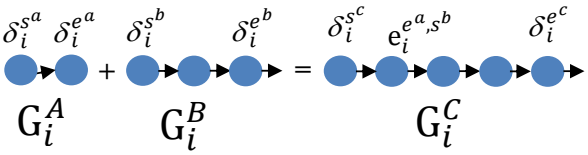

Figure 37: Series Composition.

Parallel composition takes four disjoint graphs, G_i^A, G_i^B, G_i^C, and G_i^D, creating a new graph G_i^E with edges $e_i^{s,s^b} = (\delta_i^s, \delta_i^{s^b})$, $e_i^{s,s^c} = (\delta_i^s, \delta_i^{s^c})$, $e_i^{e^b,e} = (\delta_i^{e^b}, \delta_i^e)$, and $e_i^{e^c,e} = (\delta_i^{e^c}, \delta_i^e)$ where $\delta_i^{s^b}$ and $\delta_i^{s^c}$ represent the source nodes, and $\delta_i^{e^b}$ and $\delta_i^{e^c}$ represent the sink nodes of graphs G_i^B and G_i^C respectively as shown in Figure 38.

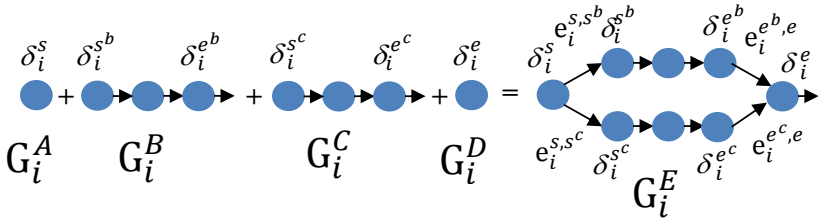

Figure 38: Parallel Composition.

Cyclic composition takes three disjoint graphs, G_i^A, G_i^B, and G_i^C, creating a new graph G_i^D with edges $e_i^{s,s^b} = (\delta_i^s, \delta_i^{s^b})$, $e_i^{e^b,e} = (\delta_i^{e^b}, \delta_i^e)$, and $e_i^{e,s} = (\delta_i^e, \delta_i^s)$ where $\delta_i^{s^b}$ represent the source node, and $\delta_i^{e^b}$ represent the sink node of graph G_i^B as shown in Figure 39.

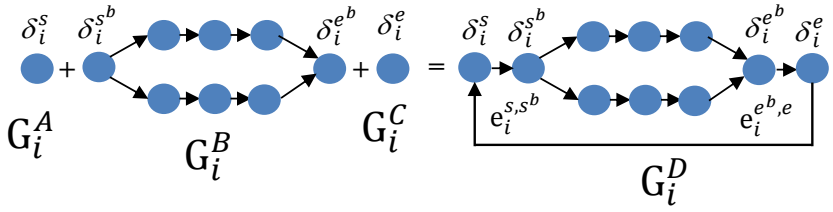

Figure 39: Cyclic Composition.

Our previous work was limited to linear control flow graphs constructed using graph creation and series composition operations only. To demonstrate the applicability of series-parallel graphs to modern real-time applications, well known and commonly used real-time structured programming language constructs such as ordered linear statement sequences, if-then statements, if-then-else statements, switch statements, bounded unrolled loops, and inline functions [3] comprise the supported software artifacts. One can readily observe that series-parallel graphs are partitioned into series and parallel connected linear code sections of basic blocks each having single entry and exit points. Due to the constrained resources of most real-time embedded processors, well written programs must employ a safe subset of programming language constructs such as conditional statements, bounded loops, and efficient functions with no recursion guaranteed to terminate within a bounded execution time in order to support cooperative tasking [71]. As a result, the real-time task code represented via series-parallel graphs can be efficiently implemented by most structured programming languages such as C. In the following sections, we present a context free graph grammar describing the series-parallel graphs supported in our work. The use of a context free grammar is compulsory for real-time conditional code preemption placement algorithms.

Grammar Background

In this section, we present a brief overview of context-free graph grammars. Historically, graph grammars have been used to facilitate the code optimization phase of program compilation [37]. In our work, we use a graph grammar to 1) recognize and construct control flow graphs conforming to the series-parallel graph structure previously described, and 2) generate intermediate structured programmatic constructs that can be efficiently solved as smaller subproblems and subsequently combined together to solve larger subproblems realizing our real-time conditional code-based PPP algorithm.

Formally, a grammar $\mathcal{G}$ defines a textual language $L(\mathcal{G})$ that is parsed and recognized via a set of production rules. Production rules are of the form $LHS \leftarrow RHS$ where the left-hand side (LHS) is a

non-terminal string and the right-hand side (RHS) contains non-terminal and/or terminal strings. A non-terminal string is a symbolic syntactic variable denoting some valid language construct. A terminal string represents some abstract or symbolic construct that is part of a textual based language $L(\mathcal{G})$. The application of a production rule means the non-terminal string on the left-hand side is substituted for the non-terminal and/or terminal strings on the right-hand side. The process of rewriting or substituting language strings in this manner is called a derivation. In the compiler domain, the set of valid language symbols are also known as tokens. Typically, these language symbols consist of numerical strings, keywords, identifiers, or symbols, comprising a program. A grammar $\mathcal{G}$ whose production rules contain only non-terminal strings on the left-hand side of each production is called a context free grammar. A context free grammar $\mathcal{G}$ where each valid string $S \in L(\mathcal{G})$ has a unique derivation sequence that recognizes S is called unambiguous.

Like their textual counterparts, context free graph grammars consist of production rules containing both non-terminal and terminal strings. However, in a graph grammar, a terminal string represents a single vertex or basic block and a non-terminal string represents a set of vertices or basic blocks and the directed edges connecting them. Therefore, we can think of a graph grammar as a set of production rules describing how basic blocks are connected thereby representing a valid control flow graph $G \in L(\mathcal{G})$. Formally, a graph G is in the language $L(\mathcal{G})$, if there exists a sequence of derivations, starting from a specified non-terminal node, that uses the productions of $\mathcal{G}$ and results in graph G [58].

In the following subsection, we present a context free graph grammar specification describing the real-time conditional CFGs that are addressed in our work. *Real time code snippets exemplifying each grammar production rule are presented along with each grammar production rule.*

Grammar Specification

Our real-time conditional graph grammar production rules are specified in this paper in Backus-Naur form (BNF) presented in the seventh section. Non-terminals are textually denoted between angle brackets $\langle$CB$\rangle$ and graphically denoted by an enclosing box. Terminals or basic blocks are textually denoted by (δ_i^j, b_i^j) where δ_i^j is the basic block identifier and b_i^j is the WCET, and graphically by a filled circle. The limitations of series-parallel graphs eliminate the use of goto statements and return statements preceding the end of a function. It is well known that real-time structured programs can be exclusively comprised of sequential statements, conditional statements, functions, and loop statements only [17].

Problem Statement

Using our series-parallel graph structure, the objective is to select a set of effective preemption points ρ_i that minimizes the WCET+CRPD of task τ_i whose real-time condition code is given by graph G_i, sub-

ject to the constraint that all non-preemptive regions must be less than or equal to the maximum allowable non-preemptive region parameter Q_i. The problem we solve in this paper is given by:

Problem Statement:

Given a real-time conditional flow graph $G_i \in L(\mathcal{G})$, an interdependent CRPD cost function $\xi_i(\delta_i^x, \delta_i^y)$ and WCET b_i^j for each basic block, find a set of Effective Preemption Points (EPPs) $\rho_i \subseteq E$ that minimizes the cost function:

$$\Phi_i(G_i, \rho_i) \stackrel{\text{def}}{=} \max_{p \in P_i(G_i, \delta_i^s, \delta_i^e)} [\sum_{\delta_i^x \in p} b_i^x + \sum_{\substack{(\delta_i^x, \delta_i^y) \in p, \rho_i \\ \delta_i^x \prec_p \delta_i^y}} \xi_i(\delta_i^x, \delta_i^y)] \tag{36}$$

subject to the constraint $\forall p \in P_i(G_i, \delta_i^s, \delta_i^e), \delta_i^w \in \rho_i, \exists e_i^{u,v} = (\delta_i^u, \delta_i^v), e_i^{x,y} = (\delta_i^x, \delta_i^y)\ where\ e_i^{u,v},\ e_i^{x,y} \in \rho_i$ ::

$$[\sum_{\substack{\delta_i^w \in p \\ \delta_i^u \preceq_p \delta_i^w \preceq_p \delta_i^x}} b_i^w + \xi_i(\delta_i^u, \delta_i^x)] \leq Q_i \tag{37}$$

The cost function $\Phi_i(G_i, \rho_i)$ evaluates to the maximum cost across all paths p through the task code.

Preemption Point Placement Algorithm

In this section, we present a dynamic-programming algorithm that achieves an improved solution ρ_i to the effective PPP problem compared to existing PPP methods. The set of selected feasible preemption points ρ_i minimizes our WCET cost objective function $\Phi_i(G_i, \rho_i)$ in that any other set of preemption points $\rho_i^{'}$ would result in a WCET cost $\Phi_i^{'}(G_i, \rho_i) \geq \Phi_i(G_i, \rho_i)$. To sufficiently describe our dynamic-programming algorithm, we first present a motivating example, then a high-level overview, followed by a recursive formulation based on our real-time conditional context-free grammar $\mathcal{G}$. The production rules described in the recursive formulation are applied to the CFG as part of the individual parsing steps in a bottom-up fashion.

Motivating Example

To present the benefits of preemption point placement using the interdependent CRPD model, consider the following example as shown in Figure 40. The WCET costs are given for each basic block along with the independent CRPD costs shown along each edge between adjacent basic blocks and summarized in Figure 41. The interdependent CRPD cost matrix summarizes the CRPD costs for each pair of connected basic blocks illustrating the opportunities for cost reduction as shown in Figure 42. Unconnected basic blocks have -1 CRPD cost entries. The upward pointing arrows denote the minimum independent CRPD

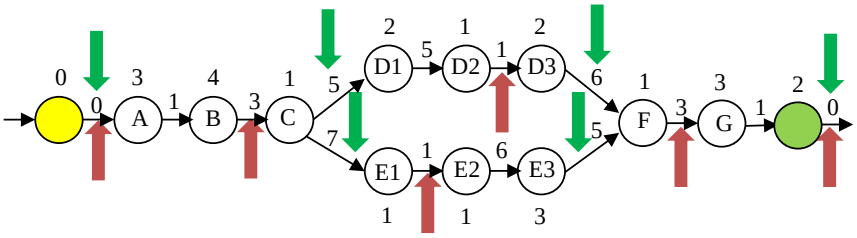

Figure 40: Motivating Example.

ξ_i	Z	A	B	C	D1	D2	D3	E1	E2	E3	F	G	H
	0	1	3	5/7	5	1	6	1	1	3	3	1	0

Figure 41: Independent CRPD Costs.

ξ_i	Z	A	B	C	D1	D2	D3	E1	E2	E3	F	G	H
Z	-1	0	0	0	0	0	0	0	0	0	0	0	0
A	-1	-1	1	1	1	1	1	1	1	1	1	1	1
B	-1	-1	-1	3	3	3	3	3	3	2	2	2	1
C	-1	-1	-1	-1	5	4	1	7	7	1	4	2	1
D1	-1	-1	-1	-1	-1	5	5	-1	-1	-1	5	3	2
D2	-1	-1	-1	-1	-1	-1	1	-1	-1	-1	1	1	1
D3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	6	4	2
E1	-1	-1	-1	-1	-1	-1	-1	-1	1	1	1	1	1
E2	-1	-1	-1	-1	-1	-1	-1	-1	-1	6	6	5	3
E3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	5	5	2
F	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	3	2
G	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	1
H	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0

Figure 42: Interdependent CRPD Costs.

cost solution whose WCET and preemption cost is 26. The downward pointing arrows denote the minimum interdependent CRPD cost solution whose WCET and preemption cost is 22 for both paths. The interdependent PPP algorithm chooses alternate preemption points (i.e. edges $e^{C,D1}$, $e^{C,E1}$, $e^{D3,F}$, and $e^{E3,F}$) in accordance with the reduced preemption cost at those locations thereby illustrating the benefits of the interdependent CRPD model as highlighted in Figure 42.

High-Level Overview

Dynamic programming algorithms are used to efficiently implement complex algorithms where solutions to smaller subproblems are computed, stored, and subsequently reused in the solutions to larger subproblems. In our approach, we compute solutions to subsets of the real-time conditional control flow graph as it is being constructed in accordance with our grammar $\mathcal{G}$ production rules. Grammar $\mathcal{G}$ is structured in such a way that solutions are computed in the following order at each level of the parse tree, namely, 1) basic blocks, 2) linear sections, 3) conditional sections, and 4) aggregate block structures. We

use the maximum non-preemptive region parameter Q_i as a suitable constraint on the number of computed solutions stored for each subgraph. We introduce two solution interface parameters, ζ_{pred}, and ζ_{succ}, denoting the non-preemptive execution times that a given solution presents to predecessor and successor subgraphs when subgraphs are combined to form solutions to larger subgraphs. To illustrate this concept, consider the following intermediate graph structure G_i^A with a proposed set of preemption points selected as denoted by the up and down arrows shown in Figure 43. Alternatively, we can visualize the solution

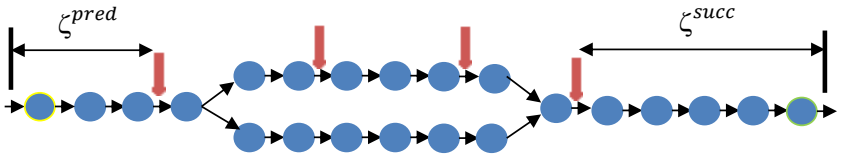

Figure 43: Subgraph Solution Interface.

interface parameters ζ_{pred} and ζ_{succ} as basic blocks whose execution times are ζ_{pred} units and ζ_{succ} units respectively as shown in Figure 44. In this simplistic model, we note that preemption is not permitted at the exterior edges as the added basic blocks denote non-preemptive execution. We use infinite weight edges to enforce non-preemption between some basic blocks. Therefore, for each subgraph, we must compute

Figure 44: Equivalent Subgraph Solution Interface.

at most $(Q_i + 1)^2$ distinct solutions for each value of ζ_{pred} and ζ_{succ} in the range of $[0 \dots Q_i - 1]$. We can think of the WCET cost and associated preemption point solutions as a set of $(Q_i + 1) \times (Q_i + 1)$ matrices, denoted as $\Phi_i(G_i^A, \zeta_{pred}, \zeta_{succ})$ and $\rho_i(G_i^A, \zeta_{pred}, \zeta_{succ})$ respectively. Later, when subgraph G_i^A is combined with other programmatic constructs in the parse tree to solve a larger subproblem, we use the ζ_{pred} and ζ_{succ} parameters to constrain which solutions from each subgraph can be combined and considered as potential solutions for the larger subgraph. We introduce two functions used to identify the visible predecessor preemption points, denoted ρ_i^{pred} and the visible successor preemption points, denoted ρ_i^{succ}. The function $\rho_i^{pred}(G_i^A, \zeta_{pred}, \zeta_{succ})$ returns the set of visible selected preemption points in the intermediate solution that may be reached along any path $p \in P_i(G_i^A, \delta_i^{s^A}, \delta_i^{e^A})$ starting at the first basic block $\delta_i^{s^A}$ and reaching some basic block $\delta_i^y \in \rho_i^{pred}$ by executing non-preemptively subject to the constraint that $\zeta_{pred} \leq Q_i$ as shown in Figure 45.

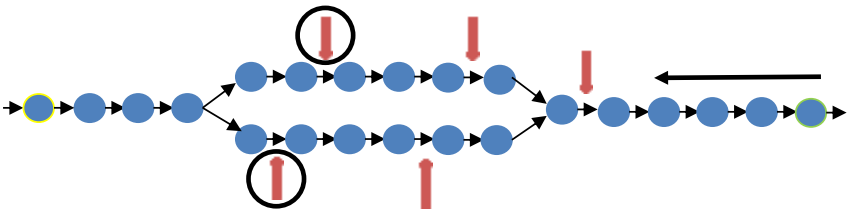

Figure 45: Visible Predecessor Preemption Points.

Similarly, the function $\rho_i^{succ}(G_i^A, \zeta_{pred}, \zeta_{succ})$ returns the set of visible selected preemption points in the intermediate solution that may be reached along any path $p \in P_i(G_i^A, \delta_i^{s^A}, \delta_i^{e^A})$ starting at basic block $\delta_i^y \in \rho_i^{succ}$ and reaching the ending basic block $\delta_i^{e^A}$ by executing non-preemptively subject to the constraint that $\zeta_{succ} \leq Q_i$ as shown in Figure 46.

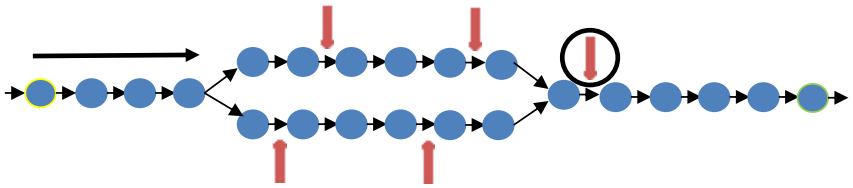

Figure 46: Visible Successor Preemption Points.

Thus, the sets ρ_i^{pred} and ρ_i^{succ} are used to determine the additive preemption cost between two subgraphs whose solutions are being combined. In the next subsection, we present a recursive formulation that achieves an effective minimized safe upper bound preemption solution for each larger subgraph as a combination of the minimized safe upper bound preemption solutions to the respective smaller subgraphs.

Recursive Formulation

In accordance with our context-free grammar $\mathcal{G}$, the various subgraphs G_i^A are created via applying the production rules on a textual based graph description that conforms to the language $L(\mathcal{G})$ presented below. As each production rule or derivation is applied, a defined subset of basic blocks and their connection relationships are assigned to subgraphs denoted by their non-terminal symbol. As each production rule is encountered, the cost function Φ_i^{SG}, and the selected preemption points ρ_i^{SG} are computed over all possible values of ζ_{pred} and ζ_{succ} in the range $[0 \ \ldots \ Q_i]$, where $< SG >$ represents the larger subgraph being constructed. For instance, the subgraph created by the linear blocks production is denoted with a suitable abbreviation to establish a proper association between the production rule, the subgraph G_i^{SG}, the WCET cost objective function Φ_i^{SG}, and the set of selected preemption points ρ_i^{SG}.

The time complexity with respect to each production rule relates to the computation of the new cost and preemption matrices using the associated formulas. The time complexity is shown for the most profound production rules in our grammar $\mathcal{G}$. Moreover, the time complexity of parsing any graph G_i using our production rules is highly dependent on the structures present in the real-time program code being

analyzed. For a graph containing N_i nodes the worst-case number of production rules executed is $2N_i - 1$ or $O(N_i)$.

We now present the grammar $\mathcal{G}$ production rules focusing on the computation of the WCET cost objective function $\Phi_i(G_i^A, \zeta_{pred}, \zeta_{succ})$ and the associated set of selected preemption points $\rho_i(G_i^A, \zeta_{pred}, \zeta_{succ})$ comprising the set of solutions generated at each level of the parse tree for all values of ζ_{pred} and ζ_{succ} in the range of $[0 \ldots Q_i]$. In this section, we present production rules $\mathcal{P}1$ through $\mathcal{P}7$ supporting the conditional CFG and block structures, while the remaining production rules $\mathcal{P}8$ through $\mathcal{P}13$ containing the other programming constructs such as loops, and functions are presented in the following subsections.

For **instruction production rule** $\mathcal{P}1$, we have:

$$<SB> \leftarrow \ (\delta_i^j, b_i^j)$$

The following graphical example shown in Figure 47 illustrates production rule $\mathcal{P}1$. A basic block is identified by a label and its corresponding WCET. The label we use is the address of the instruction contained in the basic block. Recall that by convention, basic blocks contain one or more instructions. In our approach, each basic block contains a single instruction.

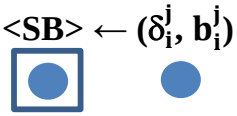

Figure 47: Production Rule $\mathcal{P}1$.

The derivation of production rule $\mathcal{P}1$ creates a subgraph G_i^{SB} containing a single basic block, δ_i^j. The associated WCET cost and preemption point functions are given by:

$$\Phi_i^{SB(j)}(\zeta_{pred}, \zeta_{succ}) = \left\{ \begin{array}{ll} \infty, & \text{if } b_i^j > Q_i \\ b_i^j, & \text{if } (b_i^j + \zeta_{pred} + \zeta_{succ}) \leq Q_i \\ b_i^j, & \text{if } (b_i^j + \zeta_{pred} + \zeta_{succ}) > Q_i \end{array} \right\} \tag{38}$$

$$\rho_i^{SB(j)}(\zeta_{pred}, \zeta_{succ}) = \left\{ \begin{array}{ll} \emptyset, & \text{if } b_i^j > Q_i \\ \emptyset, & \text{if } (b_i^j + \zeta_{pred} + \zeta_{succ}) \leq Q_i \\ \delta_i^j, & \text{if } (b_i^j + \zeta_{pred} + \zeta_{succ}) > Q_i \end{array} \right\} \tag{39}$$

There are three distinct cases to consider for production rule $\mathcal{P}1$. For case 1, the WCET b_i^j of basic block δ_i^j exceeds the maximum non-preemptive region Q_i, hence there is no solution for this graph. For case 2, the sum of basic block δ_i^j WCET b_i^j, the preceding non-preemptive region ζ_{pred}, and the succeeding

non-preemptive region ζ_{succ} is less than Q_i, so no preemption is needed ($\rho_i = \emptyset$). For case 3, the sum of basic block δ_i^j WCET b_i^j, the preceding non-preemptive region ζ_{pred}, and the succeeding non-preemptive region ζ_{succ} is greater than Q_i, so a preemption is needed ($\rho_i = \delta_i^j$).

All real-time code examples presented in this section employ assembly language code for the MIPS processor family compiled using the GCC compiler. The following real-time code example exemplifies the application of production rule $\mathcal{P}1$:

$$<SB> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i40013c, 4) \\ instruction\ code: & 0x40013c\ lw\ v1, 8(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

For **conditional production rule** $\mathcal{P}2$, we have:

$$<CB> \leftarrow <SB>\ \ [\ <Blocks>\ ]\ [\ <Blocks>\ ]*\ \ <SB>$$

The following graphical example shown in Figure 48 illustrates production rule $\mathcal{P}2$. The conditional block is comprised of two linear section blocks (for simplicity), and two connecting basic blocks, one located at the start of the conditional, and one located at the end of the conditional.

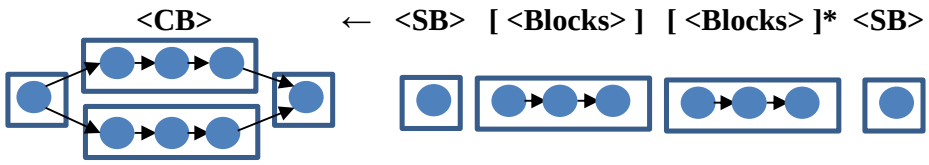

Figure 48: Production Rule $\mathcal{P}2$.

The derivation of production rule $\mathcal{P}2$ creates a subgraph G_i^{CB} concatenating a single basic block, δ_i^j in subgraph $G_i^{SB(j)}$, followed by one or more blocks each forming a conditional section in subgraph $G_i^{CS_a}$ where $a \in [1, r]$ and r denotes the number of conditional sections, ending with a single basic block, δ_i^k in subgraph $G_i^{SB(k)}$. Solutions previously computed for the r conditional sections are combined with the solutions computed for the leading and trailing basic blocks δ_i^j and δ_i^k respectively. The computation of the cost and preemptions associated with the semantics of production rule $\mathcal{P}2$ exhibits time complexity executing in $O(N_i 4 r Q_i^2)$ time. Each $<SB>$ contains 2 solutions with each of the r $<Blocks>$ structures

containing $(Q_i + 1)^2$ solutions. The associated WCET[1] cost and preemption point functions are given by:

$$\Phi_i^{CB}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = \max_{a \in \mathbb{N}: 1 \leq a \leq r} \Big\{ \min_{s,t,u} \big\{ \Phi_i^{SB(j)}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_s}) + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \Phi_i^{CS_a}(\zeta_{pred_t}, \zeta_{succ_t}) + max_{\delta_i^v, \delta_i^w}[\xi_i(\delta_i^v, \delta_i^w)] + \Phi_i^{SB(k)}(\zeta_{pred_u}, \boldsymbol{\zeta_{succ}}) \big\} \Big\} \tag{40}$$

where the variables in the min and max expressions (ζ_{succ_s}, ζ_{pred_t}, ζ_{succ_t}, ζ_{pred_u}, δ_i^m, δ_i^n, δ_i^v, and δ_i^w) represent values where the function $\Phi_i^{CB}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}})$ is minimized subject to the following constraints:

$$(\zeta_{succ_s} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_t}) \leq Q_i \tag{41}$$

$$(\zeta_{succ_t} + max_{\delta_i^v, \delta_i^w}[\xi_i(\delta_i^v, \delta_i^w)] + \zeta_{pred_u}) \leq Q_i \tag{42}$$

$$\delta_i^m \in \rho_i^{succ}(G_i^{SB(j)}, \boldsymbol{\zeta_{pred}}, \zeta_{succ_s}) \tag{43}$$

$$\delta_i^n \in \rho_i^{pred}(G_i^{CS_a}, \zeta_{pred_t}, \zeta_{succ_t}) \tag{44}$$

$$\delta_i^v \in \rho_i^{succ}(G_i^{CS_a}, \zeta_{pred_t}, \zeta_{succ_t}) \tag{45}$$

$$\delta_i^w \in \rho_i^{pred}(G_i^{SB(k)}, \zeta_{pred_u}, \boldsymbol{\zeta_{succ}}) \tag{46}$$

$$\rho_i^{CB}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = \rho_i^{SB(j)}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_s}) \bigcup_{a=1}^{r} \rho_i^{CS_a}(\zeta_{pred_t}, \zeta_{succ_t}) \bigcup \rho_i^{SB(k)}(\zeta_{pred_u}, \boldsymbol{\zeta_{succ}}) \tag{47}$$

The solutions for conditional blocks G_i^{CB} are created by considering each of the two possible solutions stored in subgraph $G_i^{SB(j)}$ and subgraph $G_i^{SB(k)}$. For each pair of solutions, we iterate through $(Q_i + 1)^2$ solutions stored in each of the r conditional section subgraphs $G_i^{CS_a}$ determining the minimized combined cost and preemption solutions. While each conditional section $G_i^{CS_a}$ cost is minimized, the overall cost of the conditional block G_i^{CB} is the maximum of all conditional sections considered selected for each value of ζ_{pred} and ζ_{succ}. While combining solutions for each conditional section, the maximum preemption cost between the solutions for combined subgraphs $G_i^{SB(j)}$ and $G_i^{CS_a}$ along with the combined subgraphs $G_i^{CS_a}$ and $G_i^{SB(k)}$ are added to the costs of the solutions for each subgraph. The first pair uses the visible successor preemptions of subgraph $G_i^{SB(j)}$ and the visible predecessor preemptions of subgraph $G_i^{CS_a}$. Similarly, the second pair uses the visible successor preemptions of subgraph $G_i^{CS_a}$ and the visible predecessor preemptions of subgraph $G_i^{SB(k)}$. The preemptions for each minimized solution contain the current preemption solutions for subgraphs $G_i^{SB(j)}$ and $G_i^{SB(k)}$ along with the union of each of the r minimized

[1] ζ_{pred} and ζ_{succ} are bolded to denote that they are constants in determining the costs for Equations 40-47; all other ζ and δ values vary over their respective ranges.

conditional section preemption solutions. The basic blocks δ_i^m, δ_i^n, δ_i^v, and δ_i^w represent elements of the predecessor and successor visible preemption sets used to determine the interdependent preemption cost of the combined solutions. It is important to note that in order to maintain a safe cost bound, we must use the cost associated with the worst-case predecessor/successor preemption points for each solution when smaller solutions are combined into larger solutions. Formally, given the Φ_i and ρ_i functions for each substructure of G_i^A where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before (resp., ζ_{succ}) after and Φ_i^A represents a safe upper bound on the total WCET and preemption cost of that solution. Thus, the solutions selected by our algorithm are minimized in accordance with our cost functions, however, our use of the maximum preemption cost when combining solutions potentially destroys global optimality. This concept applies to all presented production rules.

Theorem 4. *Given Φ_i and ρ_i functions for each substructure of CB where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production $\mathcal{P}2$ over a feasible G_i, G_i^{CB} and Q_i results in a feasible solution ρ_i^{CB} and a safe upper bound Φ_i^{CB} given by Equations 40, 41-46, and 47 respectively.*

Proof. The proof is by direct argument. We need to prove that our solution ensures that the task level Q_i constraint is not violated and the cost function $\Phi_i^{CB}(\zeta_{pred}, \zeta_{succ})$ results in a safe upper bound. To prove the Q_i constraint is not violated, we must show 1) the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface, and 2) the non-preemptive execution time of the combined solution at the new predecessor and successor interfaces does not exceed Q_i. Let $\Phi_i^{SB(j)}(\zeta_{pred}, \zeta_{succ_s})$ with ζ_{pred},$\zeta_{succ_s} \in [0 \dots Q_i]$ represent a safe upper bound cost solution for subgraph $G_i^{SB(j)}$ for basic block δ_i^j, with its corresponding set of selected preemption points denoted by $\rho_i^{SB(j)}(\zeta_{pred}, \zeta_{succ_s})$ be a limited preemption execution safe upper bound cost solution for basic block δ_i^j. We make an identical statement for subgraph $G_i^{SB(k)}$ for basic block δ_i^k, whose cost function is denoted $\Phi_i^{SB(k)}(\zeta_{pred_u}, \zeta_{succ})$, and whose set of selected preemption points are denoted $\rho_i^{SB(k)}(\zeta_{pred_u}, \zeta_{succ})$. We make a similar statement for subgraph $G_i^{CS_a}$ starting at basic block $\delta_i^{scs_a}$ and ending at basic block $\delta_i^{ecs_a}$, whose cost function is denoted $\Phi_i^{CS_a}(\zeta_{pred_t}, \zeta_{succ_t})$, and whose set of selected preemption points are denoted $\rho_i^{CS_a}(\zeta_{pred_t}, \zeta_{succ_t})$ where $a \in [1, r]$. Since we have a safe upper bound cost solution for each of the combined subgraphs, we can conclude that $\Phi_i^{CB}(\zeta_{pred}, \zeta_{succ})$ computed in Equation 40 represents a safe upper bound cost solution for the concatenated series subgraphs $G_i^{SB(j)} \bigcup_{a=1}^{r} G_i^{CS_r} \cup G_i^{SB(k)}$ starting at basic block δ_i^j, and ending at basic block δ_i^k with its corresponding selected preemption points denoted by $\rho_i^{CB}(\zeta_{pred}, \zeta_{succ})$ and computed in Equation 47. Condition 1 is met in accordance with Equations 41-46 whose purpose is to

ensure the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface. Condition 2 is met per the definition of the parameters ζ_{pred}, and ζ_{succ} respectively, whose range is given by $[0 \ldots Q_i]$. Thus, the problem finds a feasible safe upper bound cost preemption points solution when applying production $\mathcal{P}2$. □

The following real-time code example exemplifies the application of production rule $\mathcal{P}2$:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i40010c, 4) \\ instruction\ code: & 0\text{x}40010c\ beqz\ v0, 400118 \\ WCET: & 4\ cycles \end{array} \right\}$$

$$\text{<Blocks>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400110, 4) \\ & (i400114, 4) \\ instruction\ code: & 0\text{x}400110\ addiu\ v0, v0, -1 \\ & 0\text{x}400114\ j\ 400120 \end{array} \right\}$$

$$\text{<Blocks>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400118, 4) \\ & (i40011c, 4) \\ instruction\ code: & 0\text{x}400118\ lw\ v0, 12(sp) \\ & 0\text{x}40011c\ addiu\ v0, v0, 1 \end{array} \right\}$$

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400120, 4) \\ instruction\ code: & 0\text{x}400120\ lw\ v0, 18(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rules for the four sub-components have been applied, they are subsequently aggregated into a conditional block <*CB*> as follows:

$$\text{<CB>} \leftarrow \text{<SB>}\ [\ \text{<Blocks>}\ ]\ [\ \text{<Blocks>}\ ]*\ \text{<SB>}$$

For **blocks production rule** $\mathcal{P}3$, we have:

$$\text{<Blocks>} \leftarrow [\ \text{<SB>}\ ]$$

The following graphical example shown in Figure 49 illustrates production rule $\mathcal{P}3$. Here a single basic

block is subsumed into an aggregate block structure.

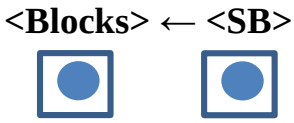

Figure 49: Production Rule $\mathcal{P}3$.

The derivation of production rule $\mathcal{P}3$ creates a subgraph G_i^{BLKS} that is equivalent to the subgraph G_i^{SB}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \;=\; \Phi_i^{SB}(\zeta_{pred}, \zeta_{succ}) \tag{48}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{SB}(\zeta_{pred}, \zeta_{succ}) \tag{49}$$

The existing single block cost and preemption solutions are copied to the aggregate blocks structure.

The following real-time code example exemplifies the application of production rule $\mathcal{P}3$:

$$\text{<}SB\text{>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i40013c, 4) \\ instruction\ code: & 0\text{x}40013c\ lw\ v1, 8(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rule for the single block sub-component has been applied, it is subsequently aggregated into a <*Blocks*> structure as follows:

$$\text{<}Blocks\text{>} \leftarrow [\ \text{<}SB\text{>}\ ]$$

For **blocks production rule** $\mathcal{P}4$, we have:

$$\text{<}Blocks\text{>} \leftarrow [\ \text{<}CB\text{>}\ ]$$

The following graphical example shown in Figure 50 illustrates production rule $\mathcal{P}4$. Here a single conditional block structure is subsumed into an aggregate block structure.

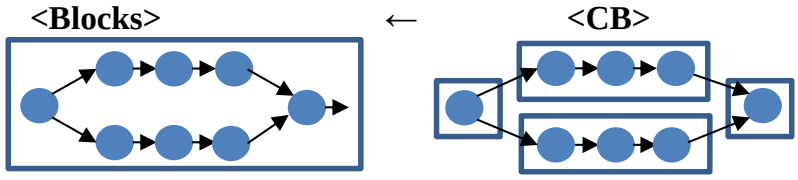

Figure 50: Production Rule $\mathcal{P}4$.

The derivation of production rule $\mathcal{P}4$ creates a subgraph G_i^{BLKS} that is equivalent to the subgraph G_i^{CB}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \; = \; \Phi_i^{CB}(\zeta_{pred}, \zeta_{succ}) \tag{50}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{CB}(\zeta_{pred}, \zeta_{succ}) \tag{51}$$

The existing conditional block cost and preemption solutions are copied to the aggregate blocks structure.

The following real-time code example exemplifies the application of production rule $\mathcal{P}4$:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i40010c, 4) \\ instruction\ code: & \text{0x40010}c\ beqz\ v0, 400118 \\ WCET: & 4\ cycles \end{array} \right\}$$

$$\text{<Blocks>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400110, 4) \\ & (i400114, 4) \\ instruction\ code: & \text{0x400110}\ addiu\ v0, v0, -1 \\ & \text{0x400114}\ j\ 400120 \end{array} \right\}$$

$$\text{<Blocks>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400118, 4) \\ & (i40011c, 4) \\ instruction\ code: & \text{0x400118}\ lw\ v0, 12(sp) \\ & \text{0x40011}c\ addiu\ v0, v0, 1 \end{array} \right\}$$

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400120, 4) \\ instruction\ code: & \text{0x400120}\ lw\ v0, 18(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rules for the four sub-components have been applied, they are subsequently aggregated into a conditional block as follows:

$$<CB> \leftarrow <SB> \ [\ <Blocks>\ ]\ [\ <Blocks>\ ]*\ <SB>$$

Once the production rule for the conditional block sub-component has been applied, it is subsequently aggregated into a *<Blocks>* structure as follows:

$$<Blocks> \leftarrow \ [\ <CB>\ ]$$

For **aggregate blocks production rule** $\mathcal{P}5$, we have:

$$<Blocks> \leftarrow <SB>\ <Blocks>$$

The following graphical example shown in Figure 51 illustrates production rule $\mathcal{P}5$. Here a single basic block is concatenated onto the front of an existing aggregate block structure.

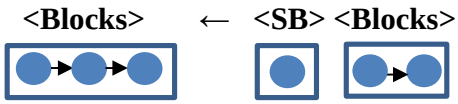

Figure 51: Production Rule $\mathcal{P}5$.

The derivation of production rule $\mathcal{P}5$ creates a subgraph $G_i^{BLKS'}$ concatenating a previously created aggregate blocks basic block subgraph G_i^{SB} in series with a previously created subgraph G_i^{BLKS}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = min_{r,s}\{(\Phi_i^{SB}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) + \max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \Phi_i^{BLKS}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}})\} \tag{52}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}})$ is minimized and valid solution combinations are subject to the following constraints:

$$(\zeta_{succ_r} + \max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i \tag{53}$$

$$\delta_i^m \in \rho_i^{succ}(G_i^{SB}, \boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \tag{54}$$

$$\delta_i^n \in \rho_i^{pred}(G_i^{BLKS}, \zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{55}$$

The associated preemption point function is given by:

$$\rho_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) \;=\; \rho_i^{SB}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \;\cup\; \rho_i^{BLKS}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{56}$$

The solutions for aggregate blocks subgraph $G_i^{BLKS'}$ are created by considering each of the two possible solutions stored in subgraph G_i^{SB}. For each of the subgraph G_i^{SB} solutions, we iterate through each of the $(Q_i + 1)^2$ aggregate block G_i^{BLKS} subgraph solutions determining the minimized cost and preemption solutions. While combining each pair of solutions, the minimum preemption cost between the combined solutions for subgraphs G_i^{SB} and G_i^{BLKS} are selected for each value of ζ_{pred} and ζ_{succ}. The algorithm uses the visible successor preemptions of subgraph G_i^{SB} and the visible predecessor preemptions of subgraph G_i^{BLKS}. The preemptions for each minimized solution contain the union of selected preemption solutions for subgraph G_i^{SB} and subgraph G_i^{BLKS}.

Theorem 5. *Given Φ_i and ρ_i functions for each substructure of $BLKS$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production $\mathcal{P}5$ over a feasible G_i, G_i^{BLKS} and Q_i results in a feasible solution ρ_i^{BLKS} and a safe upper bound Φ_i^{BLKS} given by Equations 52, 53-55, and 56 respectively.*

Proof. The proof is by direct argument. We need to prove that our solution ensures that the task level Q_i constraint is not violated and the cost function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ results in a safe upper bound. To prove the Q_i constraint is not violated, we must show 1) the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface, and 2) the non-preemptive execution time of the combined solution at the new predecessor and successor interfaces does not exceed Q_i. Let $\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ_s})$ with ζ_{pred},$\zeta_{succ_s} \in [0 \ldots Q_i]$ represent a safe upper bound cost solution for subgraph G_i^{BLKS}, with its corresponding set of selected preemption points denoted by $\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ_s})$ be a limited preemption execution safe upper bound cost solution for subgraph G_i^{BLKS}. We make an identical statement for subgraph G_i^{SB}, whose cost function is denoted $\Phi_i^{SB}(\zeta_{pred_u}, \zeta_{succ})$, and whose set of selected preemption points are denoted $\rho_i^{SB}(\zeta_{pred_u}, \zeta_{succ})$. Since we have a safe upper bound cost solution for each of the combined subgraphs, we can conclude that $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ computed in Equation 52 represents a safe upper bound cost solution for the concatenated series subgraphs $G_i^{SB} \cup G_i^{BLKS}$, with its corresponding selected preemption points denoted by $\rho_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ and computed in Equation 56. Condition 1 is met in accordance with Equations 53-55 whose purpose is to ensure the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface. Condition 2 is met per the definition of the parameters ζ_{pred}, and ζ_{succ} respectively, whose range is given by $[0 \ldots Q_i]$. Thus,

the problem finds a feasible safe upper bound cost preemption points solution when applying production $\mathcal{P}5$. □

The following real-time code example exemplifies the application of production rule $\mathcal{P}5$:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i40013c, 4) \\ instruction\ code: & \text{0x40013c } lw\ v1, 8(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rule for the single block sub-component has been applied, it is subsequently aggregated into a <*Blocks*> structure as follows:

$$\text{<Blocks>} \leftarrow [\ \text{<SB>}\ ]$$

The next instruction in the sequence will be parsed as a single block <*SB*> structure as follows:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400140, 4) \\ instruction\ code: & \text{0x400140 } sw\ v1, 4(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rule for the single block sub-component has been applied, it along with the previous blocks <*Blocks*> structure are subsequently aggregated into a <*Blocks*> structure as follows:

$$\text{<Blocks>} \leftarrow \text{<SB>}\ \text{<Blocks>}$$

For **aggregate blocks production rule** $\mathcal{P}6$, we have:

$$\text{<Blocks>} \leftarrow \text{<CB>}\ \text{<Blocks>}$$

The following graphical example shown in Figure 52 illustrates production rule $\mathcal{P}6$. Here a single conditional block is concatenated onto the front of an existing aggregate block structure.

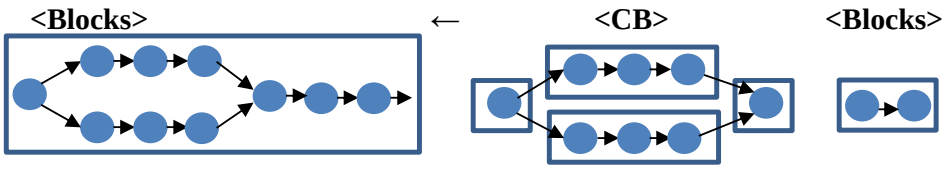

Figure 52: Production Rule $\mathcal{P}6$.

The derivation of production rule $\mathcal{P}6$ creates a subgraph $G_i^{BLKS'}$ concatenating a previously created conditional block subgraph G_i^{CB} in series with a previously created aggregate blocks subgraph G_i^{BLKS}. Pro-

duction rule $\mathcal{P}6$ exhibits the maximum time complexity for our algorithm executing in $O(N_i Q_i^4)$ time. Each <*Blocks*> and <*CB*> structure contains $(Q_i+1)^2$ solutions. The associated WCET cost function is given by:

$$\Phi_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = min_{r,s}\{(\Phi_i^{CB}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \Phi_i^{BLKS}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}})\} \tag{57}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ is minimized and valid solution combinations are subject to the following constraints:

$$(\zeta_{succ_r} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i \tag{58}$$

$$\delta_i^m \in \rho_i^{succ}(G_i^{CB}, \boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \tag{59}$$

$$\delta_i^n \in \rho_i^{pred}(G_i^{BLKS}, \zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{60}$$

The associated preemption point function is given by:

$$\rho_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = \rho_i^{CB}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \cup \rho_i^{BLKS}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{61}$$

The solutions for aggregate blocks subgraph $G_i^{BLKS'}$ are created by considering each of the $(Q_i+1)^2$ possible solutions stored in subgraph G_i^{CB}. For each of the subgraph G_i^{CB} solutions, we iterate through each of the $(Q_i+1)^2$ aggregate block G_i^{BLKS} subgraph solutions determining the minimized cost and preemption solutions. While combining each pair of solutions, the minimum preemption cost between the combined solutions for subgraphs G_i^{CB} and G_i^{BLKS} are selected for each value of ζ_{pred} and ζ_{succ}. The algorithm uses the visible successor preemptions of subgraph G_i^{CB} and the visible predecessor preemptions of subgraph G_i^{BLKS}. The preemptions for each minimized solution contain the union of selected preemption solutions for subgraph G_i^{CB} and subgraph G_i^{BLKS}.

Theorem 6. *Given Φ_i and ρ_i functions for each substructure of $BLKS$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production $\mathcal{P}6$ over a feasible G_i, G_i^{BLKS} and Q_i results in a feasible solution ρ_i^{BLKS} and a safe upper bound Φ_i^{BLKS} given by Equations 57, 58-60, and 61 respectively.*

Proof. The proof is by direct argument. We need to prove that our solution ensures that the task level Q_i constraint is not violated and the cost function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ results in a safe upper bound. To prove the Q_i constraint is not violated, we must show 1) the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface, and 2) the non-preemptive execution

time of the combined solution at the new predecessor and successor interfaces does not exceed Q_i. Let $\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ_s})$ with ζ_{pred},$\zeta_{succ_s} \in [0 \dots Q_i]$ represent a safe upper bound cost solution for subgraph G_i^{BLKS}, with its corresponding set of selected preemption points denoted by $\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ_s})$ be a limited preemption execution safe upper bound cost solution for subgraph G_i^{BLKS}. We make an identical statement for subgraph G_i^{CB}, whose cost function is denoted $\Phi_i^{CB}(\zeta_{pred_u}, \zeta_{succ})$, and whose set of selected preemption points are denoted $\rho_i^{CB}(\zeta_{pred_u}, \zeta_{succ})$. Since we have a safe upper bound cost solution for each of the combined subgraphs, we can conclude that $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ computed in Equation 57 represents a safe upper bound cost solution for the concatenated series subgraphs $G_i^{CB} \cup G_i^{BLKS}$ with its corresponding selected preemption points denoted by $\rho_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ and computed in Equation 61. Condition 1 is met in accordance with Equations 58-60 whose purpose is to ensure the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface. Condition 2 is met per the definition of the parameters ζ_{pred}, and ζ_{succ} respectively, whose range is given by $[0 \dots Q_i]$. Thus, the problem finds a feasible safe upper bound cost preemption points solution when applying production $\mathcal{P}6$. □

The following real-time code example exemplifies the application of production rule $\mathcal{P}6$:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400124, 4) \\ instruction\ code: & \text{0x400124}\ lw\ v1, 8(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rule for the single block sub-component has been applied, it is subsequently aggregated into a <*Blocks*> structure as follows:

$$\text{<Blocks>} \leftarrow [\ \text{<SB>}\ ]$$

The next instruction in the sequence will be parsed as a conditional block <*CB*> structure as follows:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i40010c, 4) \\ instruction\ code: & \text{0x40010}c\ beqz\ v0, 400118 \\ WCET: & 4\ cycles \end{array} \right\}$$

$$<Blocks> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400110, 4) \\ & (i400114, 4) \\ instruction\ code: & 0x400110\ addiu\ v0, v0, -1 \\ & 0x400114\ j\ 400120 \end{array} \right\}$$

$$<Blocks> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400118, 4) \\ & (i40011c, 4) \\ instruction\ code: & 0x400118\ lw\ v0, 12(sp) \\ & 0x40011c\ addiu\ v0, v0, 1 \end{array} \right\}$$

$$<SB> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400120, 4) \\ instruction\ code: & 0x400120\ lw\ v0, 18(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rules for the four sub-components have been applied, they are subsequently aggregated into a conditional block <*CB*> as follows:

$$<CB> \leftarrow <SB>\ [\ <Blocks>\ ]\ [\ <Blocks>\ ]*\ <SB>$$

Once the production rule for the conditional block <*CB*> sub-component has been applied, it along with the previous blocks <*Blocks*> structure are subsequently aggregated into a <*Blocks*> structure as follows:

$$<Blocks> \leftarrow <CB>\ <Blocks>$$

For **aggregate blocks production rule** $\mathcal{P}7$, we have:

$$<Blocks> \leftarrow <Blocks>\ <Blocks>$$

The following graphical example shown in Figure 53 illustrates production rule $\mathcal{P}7$. Here two aggregate block structures are concatenated together to create a new aggregate block structure.

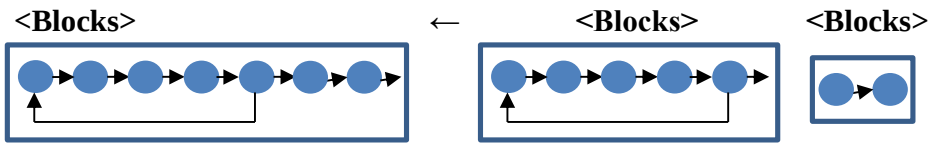

Figure 53: Production Rule $\mathcal{P}7$.

The derivation of production rule $\mathcal{P}7$ creates a subgraph $G_i^{BLKS'}$ concatenating a previously created aggregate blocks subgraph $G_i^{BLKS_a}$ in series with a previously created aggregate blocks subgraph $G_i^{BLKS_b}$. Production rule $\mathcal{P}7$ exhibits the maximum time complexity for our algorithm executing in $O(N_i Q_i^4)$ time. Each <*Blocks*> contains $(Q_i + 1)$ solutions. The associated WCET cost function is given by:

$$\Phi_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = min_{r,s}\{(\Phi_i^{BLKS_a}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \Phi_i^{BLKS_b}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}})\} \tag{62}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ is minimized and valid solution combinations are subject to the following constraints:

$$(\zeta_{succ_r} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i \tag{63}$$

$$\delta_i^m \in \rho_i^{succ}(G_i^{BLKS_a}, \boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \tag{64}$$

$$\delta_i^n \in \rho_i^{pred}(G_i^{BLKS_b}, \zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{65}$$

The associated preemption point function is given by:

$$\rho_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = \rho_i^{BLKS_a}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \cup \rho_i^{BLKS_b}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{66}$$

The solutions for aggregate blocks subgraph $G_i^{BLKS'}$ are created by considering each of the $(Q_i + 1)^2$ possible solutions stored in subgraph $G_i^{BLKS_a}$. For each of the subgraph $G_i^{BLKS_a}$ solutions, we iterate through each of the $(Q_i + 1)^2$ aggregate block $G_i^{BLKS_b}$ subgraph solutions determining the minimized cost and preemption solutions. While combining each pair of solutions, the minimum preemption cost between the combined solutions for subgraphs $G_i^{BLKS_a}$ and $G_i^{BLKS_b}$ are selected for each value of ζ_{pred} and ζ_{succ}. The algorithm uses the visible successor preemptions of subgraph $G_i^{BLKS_a}$ and the visible predecessor preemptions of subgraph $G_i^{BLKS_b}$. The preemptions for each minimized solution contain the union of selected preemption solutions for subgraph $G_i^{BLKS_a}$ and subgraph $G_i^{BLKS_b}$.

Theorem 7. *Given Φ_i and ρ_i functions for each substructure of $BLKS$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production $\mathcal{P}7$ over a feasible G_i, G_i^{BLKS} and Q_i results in a feasible solution ρ_i^{BLKS} and a safe upper bound Φ_i^{BLKS} given by Equations 62, 63-65, and 66 respectively.*

Proof. The proof is by direct argument. We need to prove that our solution ensures that the task level Q_i constraint is not violated and the cost function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ results in a safe upper bound.

To prove the Q_i constraint is not violated, we must show 1) the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface, and 2) the non-preemptive execution time of the combined solution at the new predecessor and successor interfaces does not exceed Q_i. Let $\Phi_i^{BLKS_a}(\zeta_{pred}, \zeta_{succ_s})$ with ζ_{pred},$\zeta_{succ_s} \in [0 \dots Q_i]$ represent a safe upper bound cost solution for subgraph $G_i^{BLKS_a}$, with its corresponding set of selected preemption points denoted by $\rho_i^{BLKS_a}(\zeta_{pred}, \zeta_{succ_s})$ be a limited preemption execution safe upper bound cost solution for subgraph $G_i^{BLKS_a}$. We make an identical statement for subgraph $G_i^{BLKS_b}$, whose cost function is denoted $\Phi_i^{BLKS_b}(\zeta_{pred_u}, \zeta_{succ})$, and whose set of selected preemption points are denoted $\rho_i^{BLKS_b}(\zeta_{pred_u}, \zeta_{succ})$. Since we have a safe upper bound cost solution for each of the combined subgraphs, we can conclude that $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ computed in Equation 62 represents a safe upper bound cost solution for the concatenated series subgraphs $G_i^{BLKS_a} \cup G_i^{BLKS_b}$ with its corresponding selected preemption points denoted by $\rho_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ and computed in Equation 66. Condition 1 is met in accordance with Equations 63-65 whose purpose is to ensure the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface. Condition 2 is met per the definition of the parameters ζ_{pred}, and ζ_{succ} respectively, whose range is given by $[0 \dots Q_i]$. Thus, the problem finds a feasible safe upper bound cost preemption points solution when applying production $\mathcal{P}7$. □

The following real-time code example exemplifies the application of production rule $\mathcal{P}7$:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400124, 4) \\ instruction\ code: & \text{0x400124}\ lw\ v1, 8(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rule for the single block sub-component has been applied, it is subsequently aggregated into a <*Blocks*> structure as follows:

$$\text{<Blocks>} \leftarrow [\ \text{<SB>}\ ]$$

The next instruction in the sequence will be parsed as a conditional block <*CB*> structure as follows:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i40010c, 4) \\ instruction\ code: & \text{0x40010}c\ beqz\ v0, 400118 \\ WCET: & 4\ cycles \end{array} \right\}$$

$$<\textit{Blocks}>\leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400110, 4) \\ & (i400114, 4) \\ instruction\ code: & 0\text{x}400110\ addiu\ v0, v0, -1 \\ & 0\text{x}400114\ j\ 400120 \end{array} \right\}$$

$$<\textit{Blocks}>\leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400118, 4) \\ & (i40011c, 4) \\ instruction\ code: & 0\text{x}400118\ lw\ v0, 12(sp) \\ & 0\text{x}40011c\ addiu\ v0, v0, 1 \end{array} \right\}$$

$$<\textit{SB}>\leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400120, 4) \\ instruction\ code: & 0\text{x}400120\ lw\ v0, 18(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rules for the four sub-components have been applied, they are subsequently aggregated into a conditional block <*CB*> as follows:

$$<\textit{CB}> \leftarrow <\textit{SB}>\ [\ <\textit{Blocks}>\ ]\ [\ <\textit{Blocks}>\ ]*\ <\textit{SB}>$$

Once the production rule for the conditional block $<CB>$ sub-component has been applied, it is subsequently aggregated into a <*Blocks*> structure as follows:

$$<\textit{Blocks}> \leftarrow <\textit{CB}>$$

Once the production rule for the aggregate block <*Blocks*> sub-component has been applied, it along with the previous blocks <*Blocks*> structure are subsequently aggregated into a <*Blocks*> structure as follows:

$$<\textit{Blocks}> \leftarrow <\textit{Blocks}>\ <\textit{Blocks}>$$

For convenience, Figure 54 summarizes production rules $\mathcal{P}1 - \mathcal{P}4$.

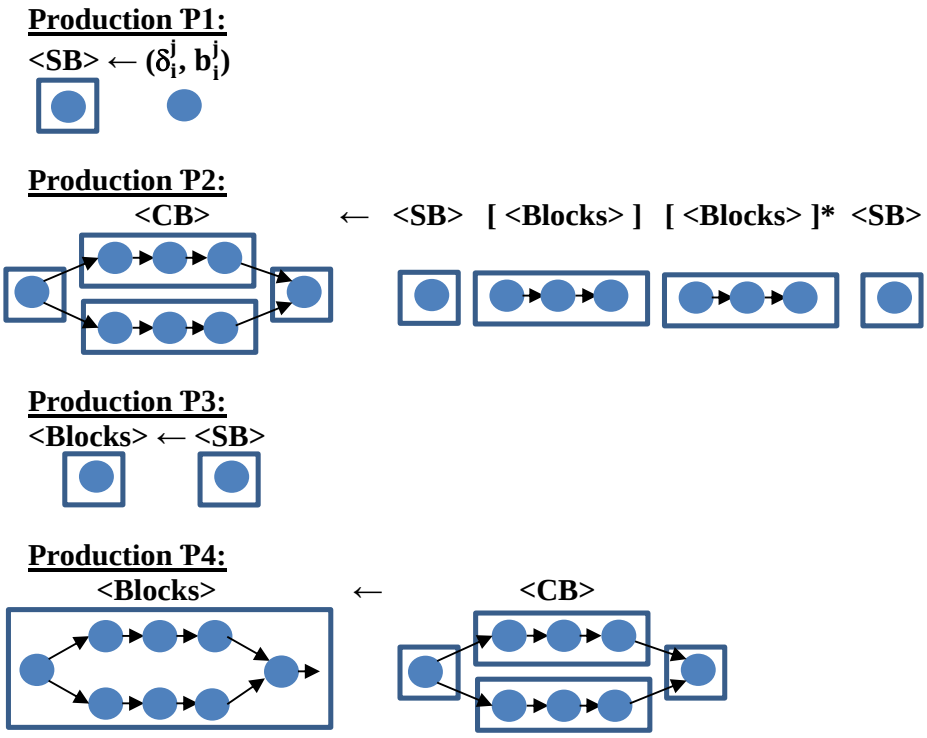

Figure 54: Production Rules $\mathcal{P}1 - \mathcal{P}4$.

For convenience, Figure 55 summarizes production rules $\mathcal{P}5 - \mathcal{P}7$.

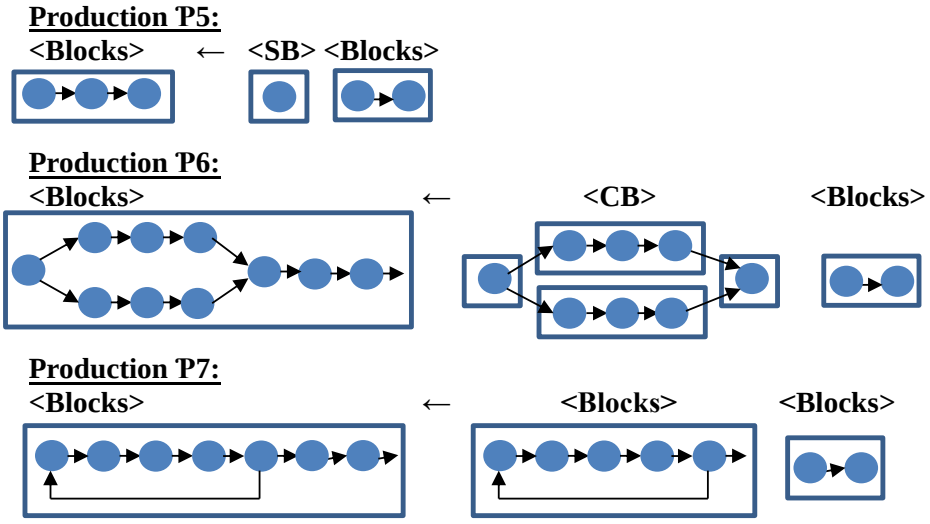

Figure 55: Production Rules $\mathcal{P}5 - \mathcal{P}7$.

The grammar we have presented thus far are focused on the production rules for conditional structures. Non-unrolled loops and functions are structured programming constructs that are also prevalent in real-time code. We present the production rules supporting these structured programming elements in the following subsections.

Non-Unrolled Loops

For **loop production rule** $\mathcal{P}8$, we have:

$$\textit{<Loop>} \leftarrow [\ \textit{<Blocks>}\ \textit{<MaxIter>}\ ]$$

The following graphical example shown in Figure 56 illustrates production rule $\mathcal{P}8$. Here a loop structure is created from an existing aggregate block structure and the maximum number of loop iterations.

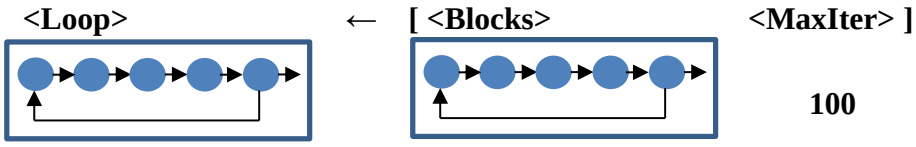

Figure 56: Production Rule $\mathcal{P}8$.

The derivation of production rule $\mathcal{P}8$ creates a subgraph G_i^{LOOP} that is equivalent to the subgraph G_i^{BLKS}. The associated WCET cost function is given by:

$$\Phi_i^{LOOP}(\zeta_{pred}, \zeta_{succ}) \ = \ \Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \ \times \ MaxIter \tag{67}$$

where valid solution combinations are subject to the following constraints:

$$(\zeta_{succ} \ + \ \zeta_{pred}) \leq Q_i \tag{68}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{LOOP}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \tag{69}$$

The existing aggregate blocks cost and preemption solutions are copied to the loop structure.

The following real-time code example exemplifies the application of production rule $\mathcal{P}8$:

$$<\textit{Blocks}> \leftarrow \left\{ \begin{array}{ll} \textit{grammar representation}: & (i400100, 4) \\ & \ldots \\ & (i400110, 4) \\ & (i400114, 4) \\ & 10 \\ \textit{instruction code}: & 0x400100\ lw\ v0, 10 \\ & \ldots \\ & 0\text{x}400110\ addiu\ v0, v0, -1 \\ & 0\text{x}400114\ bnez\ v0, 400100 \\ \textit{maximum iterations}: & 10 \end{array} \right\}$$

Once the production rule for the blocks <*Blocks*> structure has been applied, it along with the maximum loop iterations <*MaxIter*> are subsequently aggregated into a <*Loop*> structure as follows:

$$<\textit{Loop}> \leftarrow [\ <\textit{Blocks}>\ <\textit{MaxIter}>\ ]$$

For **blocks production rule** $\mathcal{P}9$, we have:

$$<\textit{Blocks}> \leftarrow [\ <\textit{LOOP}>\ ]$$

The following graphical example shown in Figure 57 illustrates production rule $\mathcal{P}9$. Here a single loop structure is subsumed into an aggregate block structure.

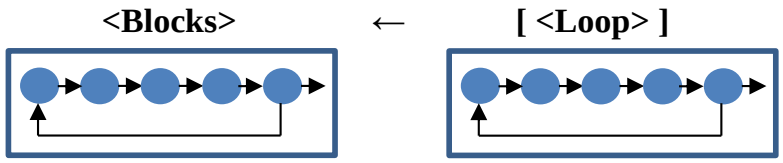

Figure 57: Production Rule $\mathcal{P}9$.

The derivation of production rule $\mathcal{P}9$ creates a subgraph G_i^{BLKS} that is equivalent to the subgraph G_i^{LOOP}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \ = \ \Phi_i^{LOOP}(\zeta_{pred}, \zeta_{succ}) \tag{70}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{LOOP}(\zeta_{pred}, \zeta_{succ}) \tag{71}$$

The existing loop block cost and preemption solutions are copied to the aggregate blocks structure.

The following real-time code example exemplifies the application of production rule $\mathcal{P}9$:

$$<\textit{Blocks}> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400100, 4) \\ & \ldots \\ & (i400110, 4) \\ & (i400114, 4) \\ & 10 \\ instruction\ code: & 0\text{x}400100\ lw\ v0, 10 \\ & \ldots \\ & 0\text{x}400110\ addiu\ v0, v0, -1 \\ & 0\text{x}400114\ bnez\ v0, 400100 \\ maximum\ iterations: & 10 \end{array} \right\}$$

Once the production rule for the blocks <*Blocks*> structure has been applied, it along with the maximum loop iterations <*MaxIter*> are subsequently aggregated into a <*Loop*> structure as follows:

$$<\textit{Loop}> \leftarrow [\ <\textit{Blocks}>\ <\textit{MaxIter}>\ ]$$

Once the production rule for the loop <*Loop*> component has been applied, it is subsequently aggregated into a <*Blocks*> structure as follows:

$$<\textit{Blocks}> \leftarrow [\ <\textit{Loop}>\ ]$$

For **aggregate blocks production rule** $\mathcal{P}10$, we have:

$$<\textit{Blocks}> \leftarrow <\textit{Loop}>\ <\textit{Blocks}>$$

The following graphical example shown in Figure 58 illustrates production rule $\mathcal{P}10$. Here an existing loop structure is concatenated with an existing aggregate block structure to create a new aggregate block structure.

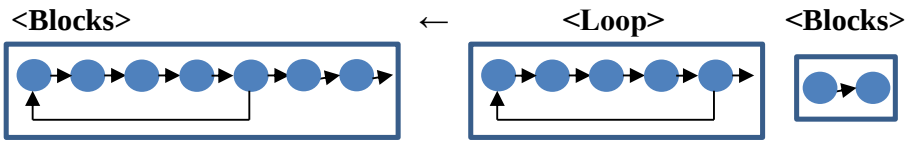

Figure 58: Production Rule $\mathcal{P}10$.

The derivation of production rule $\mathcal{P}10$ creates a subgraph $G_i^{BLKS'}$ concatenating a previously created aggregate blocks subgraph G_i^{LOOP} in series with a previously created aggregate blocks subgraph G_i^{BLKS}. Production rule $\mathcal{P}10$ exhibits the maximum time complexity for our algorithm executing in $O(N_i Q_i^4)$ time. Each $<$*Loop*$>$ and each $<$*Blocks*$>$ contains $(Q_i + 1)$ solutions. The associated WCET cost function is given by:

$$\Phi_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = min_{r,s}\{(\Phi_i^{LOOP}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \Phi_i^{BLKS}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}})\} \tag{72}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ is minimized and valid solution combinations are subject to the following constraints:

$$(\zeta_{succ_r} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i \tag{73}$$

$$\delta_i^m \in \rho_i^{succ}(G_i^{LOOP}, \boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \tag{74}$$

$$\delta_i^n \in \rho_i^{pred}(G_i^{BLKS}, \zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{75}$$

The associated preemption point function is given by:

$$\rho_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = \rho_i^{LOOP}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \cup \rho_i^{BLKS}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{76}$$

The solutions for aggregate blocks subgraph $G_i^{BLKS'}$ are created by considering each of the $(Q_i + 1)^2$ possible solutions stored in subgraph G_i^{LOOP}. For each of the subgraph G_i^{LOOP} solutions, we iterate through each of the $(Q_i+1)^2$ aggregate block G_i^{BLKS} subgraph solutions determining the minimized cost and preemption solutions. While combining each pair of solutions, the minimum preemption cost between the combined solutions for subgraphs G_i^{LOOP} and G_i^{BLKS} are selected for each value of ζ_{pred} and ζ_{succ}. The algorithm uses the visible successor preemptions of subgraph G_i^{LOOP} and the visible predecessor preemptions of subgraph G_i^{BLKS}. The preemptions for each minimized solution contain the union of selected preemption solutions for subgraph G_i^{LOOP} and subgraph G_i^{BLKS}.

Theorem 8. *Given Φ_i and ρ_i functions for each substructure of $BLKS$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production $\mathcal{P}10$ over*

a feasible G_i, G_i^{BLKS} and Q_i results in a feasible solution ρ_i^{BLKS} and a safe upper bound Φ_i^{BLKS} given by Equations 72, 73-75, and 76 respectively.

Proof. The proof is by direct argument. We need to prove that our solution ensures that the task level Q_i constraint is not violated and the cost function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ results in a safe upper bound. To prove the Q_i constraint is not violated, we must show 1) the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface, and 2) the non-preemptive execution time of the combined solution at the new predecessor and successor interfaces does not exceed Q_i. Let $\Phi_i^{LOOP}(\zeta_{pred}, \zeta_{succ_s})$ with ζ_{pred},$\zeta_{succ_s} \in [0 \dots Q_i]$ represent a safe upper bound cost solution for subgraph G_i^{LOOP}, with its corresponding set of selected preemption points denoted by $\rho_i^{LOOP}(\zeta_{pred}, \zeta_{succ_s})$ be a limited preemption execution safe upper bound cost solution for subgraph G_i^{LOOP}. We make an identical statement for subgraph G_i^{BLKS}, whose cost function is denoted $\Phi_i^{BLKS}(\zeta_{pred_u}, \zeta_{succ})$, and whose set of selected preemption points are denoted $\rho_i^{BLKS}(\zeta_{pred_u}, \zeta_{succ})$. Since we have a safe upper bound cost solution for each of the combined subgraphs, we can conclude that $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ computed in Equation 72 represents a safe upper bound cost solution for the concatenated series subgraphs $G_i^{LOOP} \cup G_i^{BLKS}$ with its corresponding selected preemption points denoted by $\rho_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ and computed in Equation 76. Condition 1 is met in accordance with Equations 73-75 whose purpose is to ensure the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface. Condition 2 is met per the definition of the parameters ζ_{pred}, and ζ_{succ} respectively, whose range is given by $[0 \dots Q_i]$. Thus, the problem finds a feasible safe upper bound cost preemption points solution when applying production $\mathcal{P}10$. □

The following real-time code example exemplifies the application of production rule $\mathcal{P}10$:

$$\text{<SB>} \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400118, 4) \\ instruction\ code: & \text{0x400118}\ lw\ v1, 8(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rule for the single block sub-component has been applied, it is subsequently aggregated into a <*Blocks*> structure as follows:

$$\text{<Blocks>} \leftarrow [\ \text{<SB>}\ ]$$

The next instruction in the sequence will be parsed as a loop block <*Loop*> structure as follows:

$$<Blocks> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400100, 4) \\ & \ldots \\ & (i400110, 4) \\ & (i400114, 4) \\ & 10 \\ instruction\ code: & 0\text{x}400100\ lw\ v0, 10 \\ & \ldots \\ & 0\text{x}400110\ addiu\ v0, v0, -1 \\ & 0\text{x}400114\ bnez\ v0, 400100 \\ maximum\ iterations: & 10 \end{array} \right\}$$

Once the production rule for the blocks *<Blocks>* structure has been applied, it along with the maximum loop iterations *<MaxIter>* are subsequently aggregated into a *<Loop>* structure as follows:

$$<Loop> \leftarrow [\ <Blocks>\ <MaxIter>\ ]$$

Once the production rule for the loop block *<Loop>* sub-component has been applied, it along with the previous blocks *<Blocks>* structure are subsequently aggregated into a *<Blocks>* structure as follows:

$$<Blocks> \leftarrow <Loop>\ <Blocks>$$

For convenience, Figure 59 summarizes production rules $\mathcal{P}8 - \mathcal{P}10$.

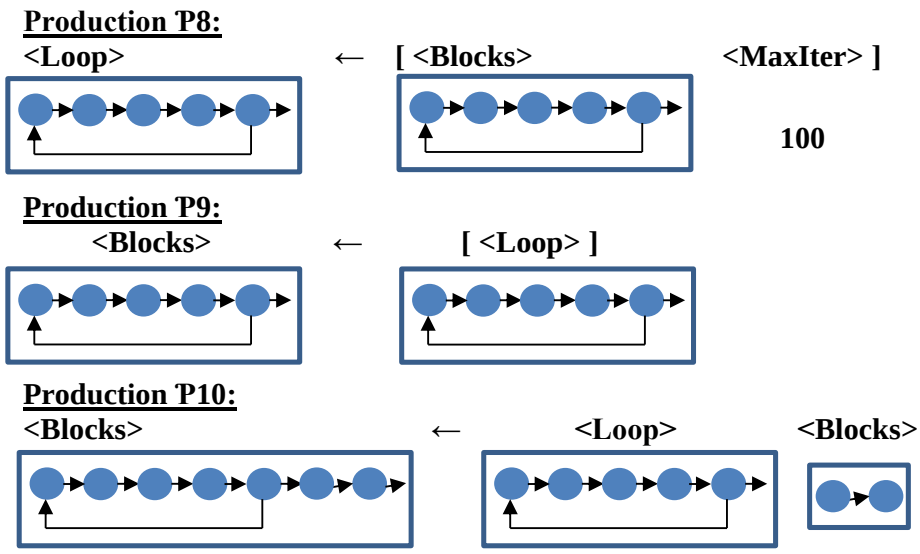

Figure 59: Production Rules $\mathcal{P}8 - \mathcal{P}10$.

Inline Functions

Functions are split into two grammar elements, namely, function definition and function invocation. The conditional PPP algorithm generates solutions for the function definition blocks consistently with the main task function. The generated function definition preemption solutions are combined with the function invocation preemption solutions at each graph location where the function is called.

For **function definition production rule** $\mathcal{P}11$, we have:

$$\textit{<Function>} \leftarrow [\ \textit{<Blocks>}\ \textit{<FunctionName>}\ ]$$

The following graphical example shown in Figure 60 illustrates production rule $\mathcal{P}11$. Here a function definition is created from an existing aggregate block structure and identified by its corresponding function name.

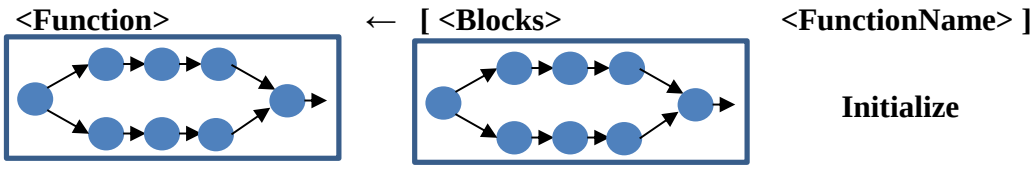

Figure 60: Production Rule $\mathcal{P}11$.

The derivation of production rule $\mathcal{P}11$ creates a subgraph G_i^{FUNC} that is equivalent to the subgraph G_i^{BLKS}. The associated WCET cost function is given by:

$$\Phi_i^{FUNC}(\zeta_{pred}, \zeta_{succ}) = \Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \tag{77}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{FUNC}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \tag{78}$$

The existing aggregate blocks cost and preemption solutions are copied to the function definition block structure.

The following real-time code example exemplifies the application of production rule $\mathcal{P}11$:

$$<\textit{Function}> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400044, 4) \\ & (i400048, 4) \\ & (i40004c, 4) \\ & (i400050, 4) \\ & \ldots \\ & (i400258, 4) \\ & (i40025c, 4) \\ & Initialize \\ instruction\ code: & 0\text{x}400044\ addiu\ sp, sp, -8 \\ & 0\text{x}400048\ li\ v0, -1 \\ & 0\text{x}40004c\ sw\ v0, -32760(gp) \\ & 0\text{x}400050\ lw\ v1, -32760(gp) \\ & \ldots \\ & 0\text{x}400258\ addiu\ sp, sp, 8 \\ & 0\text{x}40025c\ jr\ ra \\ function\ name: & Initialize \end{array} \right\}$$

Once the production rule for the blocks <*Blocks*> structure has been applied, it along with the function name <*FunctionName*> are subsequently aggregated into a <*Function*> structure as follows:

$$<\textit{Function}> \leftarrow [\ <\textit{Blocks}>\ <\textit{FunctionName}>\ ]$$

For **function call production rule** $\mathcal{P}12$, we have:

$$<\textit{FunctionCall}> \leftarrow [\ <\textit{Blocks}>\ <\textit{FunctionName}>\ ]$$

The following graphical example shown in Figure 61 illustrates production rule $\mathcal{P}12$. Here a function invocation is created from an existing aggregate block structure concatenated with the function definition identified by function name. Here the function definition is represented by a rather simple two basic block linear structure for simplicity.

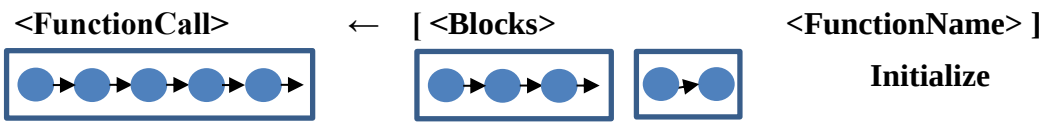

Figure 61: Production Rule $\mathcal{P}12$.

The derivation of production rule $\mathcal{P}12$ creates a subgraph G_i^{FCALL} that is equivalent to the subgraph G_i^{BLKS}. The associated WCET cost function is given by:

$$\Phi_i^{FCALL}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = min_{r,s}\{(\Phi_i^{BLKS}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \Phi_i^{FUNC}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}})\} \tag{79}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{FCALL}(\zeta_{pred}, \zeta_{succ})$ is minimized and valid solution combinations are subject to the following constraints:

$$(\zeta_{succ_r} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i \tag{80}$$

$$\delta_i^m \in \rho_i^{succ}(G_i^{BLKS}, \boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \tag{81}$$

$$\delta_i^n \in \rho_i^{pred}(G_i^{FUNC}, \zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{82}$$

The associated preemption point function is given by:

$$\rho_i^{FCALL}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = \rho_i^{BLKS}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \cup \rho_i^{FUNC}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{83}$$

The solutions for the function call subgraph G_i^{FCALL} are created by considering each of the $(Q_i + 1)^2$ possible solutions stored in subgraph G_i^{BLKS}. For each of the subgraph G_i^{BLKS} solutions, we iterate through each of the $(Q_i + 1)^2$ aggregate block G_i^{FUNC} subgraph solutions determining the minimized cost and preemption solutions. While combining each pair of solutions, the minimum preemption cost between the combined solutions for subgraphs G_i^{BLKS} and G_i^{FUNC} are selected for each value of ζ_{pred} and ζ_{succ}. The algorithm uses the visible successor preemptions of subgraph G_i^{BLKS} and the visible predecessor preemptions of subgraph G_i^{FUNC}. The preemptions for each minimized solution contain the union of selected preemption solutions for subgraph G_i^{BLKS} and subgraph G_i^{FUNC}.

Theorem 9. *Given Φ_i and ρ_i functions for each substructure of $FCALL$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production $\mathcal{P}12$ over a feasible G_i, G_i^{FCALL} and Q_i results in a feasible solution ρ_i^{FCALL} and a safe upper bound Φ_i^{FCALL} given by Equations 79, 80-82, and 83 respectively.*

Proof. The proof is by direct argument. We need to prove that our solution ensures that the task level

Q_i constraint is not violated and the cost function $\Phi_i^{FCALL}(\zeta_{pred}, \zeta_{succ})$ results in a safe upper bound. To prove the Q_i constraint is not violated, we must show 1) the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface, and 2) the non-preemptive execution time of the combined solution at the new predecessor and successor interfaces does not exceed Q_i. Let $\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ_s})$ with ζ_{pred},$\zeta_{succ_s} \in [0 \dots Q_i]$ represent a safe upper bound cost solution for subgraph G_i^{BLKS}, with its corresponding set of selected preemption points denoted by $\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ_s})$ be a limited preemption execution safe upper bound cost solution for subgraph G_i^{BLKS}. We make an identical statement for subgraph G_i^{FUNC}, whose cost function is denoted $\Phi_i^{FUNC}(\zeta_{pred_u}, \zeta_{succ})$, and whose set of selected preemption points are denoted $\rho_i^{FUNC}(\zeta_{pred_u}, \zeta_{succ})$. Since we have a safe upper bound cost solution for each of the combined subgraphs, we can conclude that $\Phi_i^{FCALL}(\zeta_{pred}, \zeta_{succ})$ computed in Equation 79 represents a safe upper bound cost solution for the concatenated series subgraphs $G_i^{BLKS} \cup G_i^{FUNC}$ with its corresponding selected preemption points denoted by $\rho_i^{FCALL}(\zeta_{pred}, \zeta_{succ})$ and computed in Equation 83. Condition 1 is met in accordance with Equations 80-82 whose purpose is to ensure the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface. Condition 2 is met per the definition of the parameters ζ_{pred}, and ζ_{succ} respectively, whose range is given by $[0 \dots Q_i]$. Thus, the problem finds a feasible safe upper bound cost preemption points solution when applying production $\mathcal{P}12$. □

The following real-time code example exemplifies the application of production rule $\mathcal{P}12$:

$$<\textit{FunctionCall}> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400018, 4) \\ & (i40001c, 4) \\ & (i400020, 4) \\ & Initialize \\ instruction\ code: & 0\text{x}400018\ addiu\ sp, sp, -24 \\ & 0\text{x}40001c\ sw\ ra, 20(sp) \\ & 0\text{x}400020\ jal\ 400044 \\ function\ name: & Initialize \end{array} \right\}$$

Once the production rule for the blocks <*Blocks*> structure has been applied, it along with the function name <*FunctionName*> are subsequently aggregated into a <*FunctionCall*> block structure as follows:

<*FunctionCall*> ← [<*Blocks*> <*FunctionName*>]

For **blocks production rule** $\mathcal{P}13$, we have:

$$<Blocks> \leftarrow [\ <FunctionCall>\ ]$$

The following graphical example shown in Figure 62 illustrates production rule $\mathcal{P}13$. Here a function invocation is subsumed into an aggregate block structure.

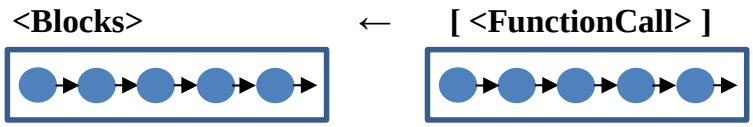

Figure 62: Production Rule $\mathcal{P}13$.

The derivation of production rule $\mathcal{P}13$ creates a subgraph G_i^{BLKS} that is equivalent to the subgraph G_i^{FCALL}. The associated WCET cost function is given by:

$$\Phi_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) \ = \ \Phi_i^{FCALL}(\zeta_{pred}, \zeta_{succ}) \tag{84}$$

The associated set of selected preemption points function is given by:

$$\rho_i^{BLKS}(\zeta_{pred}, \zeta_{succ}) = \rho_i^{FCALL}(\zeta_{pred}, \zeta_{succ}) \tag{85}$$

The existing aggregate blocks cost and preemption solutions are copied to the function call block structure.

The following real-time code example exemplifies the application of production rule $\mathcal{P}13$:

$$<FunctionCall> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400018, 4) \\ & (i40001c, 4) \\ & (i400020, 4) \\ & Initialize \\ instruction\ code: & 0\text{x}400018\ addiu\ sp, sp, -24 \\ & 0\text{x}40001c\ sw\ ra, 20(sp) \\ & 0\text{x}400020\ jal\ 400044 \\ function\ name: & Initialize \end{array} \right\}$$

Once the production rule for the blocks <*Blocks*> structure has been applied, it along with the function name <*FunctionName*> are subsequently aggregated into a <*FunctionCall*> block structure as follows:

$$<FunctionCall> \leftarrow [\ <Blocks>\ <FunctionName>\ ]$$

Once the production rule for the function call $<FunctionCall>$ component has been applied, it is subsequently aggregated into a $<Blocks>$ structure as follows:

$$<Blocks> \leftarrow [\ <FunctionCall>\ ]$$

For **aggregate blocks production rule** $\mathcal{P}14$, we have:

$$<Blocks> \leftarrow <FunctionCall>\ <Blocks>$$

The following graphical example shown in Figure 63 illustrates production rule $\mathcal{P}14$. Here an existing function call structure is concatenated with an existing aggregate block structure to create a new aggregate block structure.

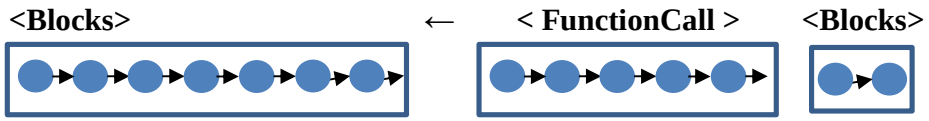

Figure 63: Production Rule $\mathcal{P}14$.

The derivation of production rule $\mathcal{P}14$ creates a subgraph $G_i^{BLKS'}$ concatenating a previously created aggregate blocks subgraph G_i^{FCALL} in series with a previously created aggregate blocks subgraph G_i^{BLKS}. Production rule $\mathcal{P}14$ exhibits the maximum time complexity for our algorithm executing in $O(N_i Q_i^4)$ time. Each $<FunctionCall>$ and each
$<Blocks>$ contains $(Q_i + 1)$ solutions. The associated WCET cost function is given by:

$$\Phi_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) = min_{r,s}\{(\Phi_i^{FCALL}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \Phi_i^{BLKS}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}})\} \tag{86}$$

where ζ_{succ_r}, and ζ_{pred_s} represent the values where the function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ is minimized and valid solution combinations are subject to the following constraints:

$$(\zeta_{succ_r} + max_{\delta_i^m, \delta_i^n}[\xi_i(\delta_i^m, \delta_i^n)] + \zeta_{pred_s}) \leq Q_i \tag{87}$$

$$\delta_i^m \in \rho_i^{succ}(G_i^{FCALL}, \boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \tag{88}$$

$$\delta_i^n \in \rho_i^{pred}(G_i^{BLKS}, \zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{89}$$

The associated preemption point function is given by:

$$\rho_i^{BLKS'}(\boldsymbol{\zeta_{pred}}, \boldsymbol{\zeta_{succ}}) \;=\; \rho_i^{FCALL}(\boldsymbol{\zeta_{pred}}, \zeta_{succ_r}) \;\cup\; \rho_i^{BLKS}(\zeta_{pred_s}, \boldsymbol{\zeta_{succ}}) \tag{90}$$

The solutions for the aggregate blocks subgraph $G_i^{BLKS'}$ are created by considering each of the $(Q_i + 1)^2$ possible solutions stored in subgraph G_i^{FCALL}. For each of the function call subgraph G_i^{FCALL} solutions, we iterate through each of the $(Q_i+1)^2$ aggregate block G_i^{BLKS} subgraph solutions determining the minimized cost and preemption solutions. While combining each pair of solutions, the minimum preemption cost between the combined solutions for subgraphs G_i^{FCALL} and G_i^{BLKS} are selected for each value of ζ_{pred} and ζ_{succ}. The algorithm uses the visible successor preemptions of subgraph G_i^{FCALL} and the visible predecessor preemptions of subgraph G_i^{BLKS}. The preemptions for each minimized solution contain the union of selected preemption solutions for subgraph G_i^{FCALL} and subgraph G_i^{BLKS}.

Theorem 10. *Given Φ_i and ρ_i functions for each substructure of $BLKS$ where each $\rho_i^A(\zeta_{pred}, \zeta_{succ})$ represents a feasible solution for substructure A given preemptions ζ_{pred} before, ζ_{succ} after, and Φ_i^A is a safe upper bound on the total WCET and preemption cost of that solution. Applying production $\mathcal{P}14$ over a feasible G_i, G_i^{BLKS} and Q_i results in a feasible solution ρ_i^{BLKS} and a safe upper bound Φ_i^{BLKS} given by Equations 86, 87-89, and 90 respectively.*

Proof. The proof is by direct argument. We need to prove that our solution ensures that the task level Q_i constraint is not violated and the cost function $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ results in a safe upper bound. To prove the Q_i constraint is not violated, we must show 1) the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface, and 2) the non-preemptive execution time of the combined solution at the new predecessor and successor interfaces does not exceed Q_i. Let $\Phi_i^{FCALL}(\zeta_{pred}, \zeta_{succ_s})$ with ζ_{pred},$\zeta_{succ_s} \in [0 \ldots Q_i]$ represent a safe upper bound cost solution for subgraph G_i^{LOOP}, with its corresponding set of selected preemption points denoted by $\rho_i^{FCALL}(\zeta_{pred}, \zeta_{succ_s})$ be a limited preemption execution safe upper bound cost solution for subgraph G_i^{FCALL}. We make an identical statement for subgraph G_i^{BLKS}, whose cost function is denoted $\Phi_i^{BLKS}(\zeta_{pred_u}, \zeta_{succ})$, and whose set of selected preemption points are denoted $\rho_i^{BLKS}(\zeta_{pred_u}, \zeta_{succ})$. Since we have a safe upper bound cost solution for each of the combined subgraphs, we can conclude that $\Phi_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ computed in Equation 86 represents a safe upper bound cost solution for the concatenated series subgraphs $G_i^{FCALL} \cup G_i^{BLKS}$ with its corresponding selected preemption points denoted by $\rho_i^{BLKS'}(\zeta_{pred}, \zeta_{succ})$ and computed in Equation 90. Condition 1 is met in accordance with Equations 87-89 whose purpose is to ensure the non-preemptive execution time of the combined solutions does not exceed Q_i at each solution interface. Condition 2 is met per the definition of the parameters ζ_{pred}, and ζ_{succ} respectively, whose range

is given by $[0 \ldots Q_i]$. Thus, the problem finds a feasible safe upper bound cost preemption points solution when applying production $\mathcal{P}14$. □

The following real-time code example exemplifies the application of production rule $\mathcal{P}14$:

$$<SB> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400024, 4) \\ instruction\ code: & 0\text{x}400024\ lw\ v1, 8(sp) \\ WCET: & 4\ cycles \end{array} \right\}$$

Once the production rule for the single block sub-component has been applied, it is subsequently aggregated into a <*Blocks*> structure as follows:

$$<Blocks> \leftarrow [\ <SB>\ ]$$

The next instruction in the sequence will be parsed as a function call block <*FunctionCall*> structure as follows:

$$<FunctionCall> \leftarrow \left\{ \begin{array}{ll} grammar\ representation: & (i400018, 4) \\ & (i40001c, 4) \\ & (i400020, 4) \\ & Initialize \\ instruction\ code: & 0\text{x}400018\ addiu\ sp, sp, -24 \\ & 0\text{x}40001c\ sw\ ra, 20(sp) \\ & 0\text{x}400020\ jal\ 400044 \\ function\ name: & Initialize \end{array} \right\}$$

Once the production rule for the blocks <*Blocks*> structure has been applied, it along with the function name <*FunctionName*> are subsequently aggregated into a <*FunctionCall*> block structure as follows:

$$<FunctionCall> \leftarrow [\ <Blocks>\ <FunctionName>\ ]$$

Once the production rule for the loop block <*Loop*> sub-component has been applied, it along with the previous blocks <*Blocks*> structure are subsequently aggregated into a <*Blocks*> structure as follows:

$$<Blocks> \leftarrow <FunctionCall>\ <Blocks>$$

For convenience, Figure 64 summarizes production rules $\mathcal{P}11 - \mathcal{P}14$.

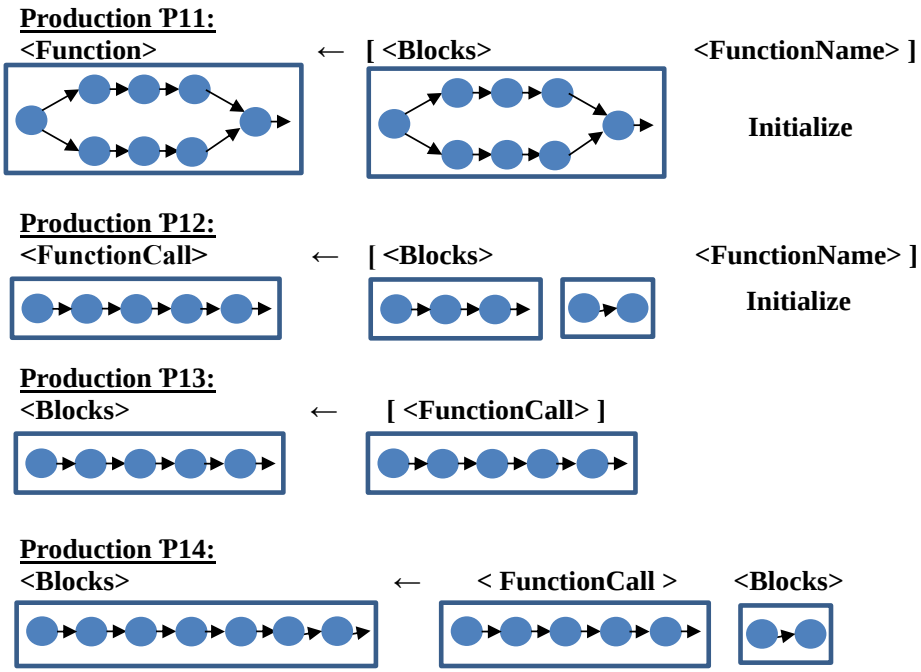

Figure 64: Production Rules $\mathcal{P}11 - \mathcal{P}14$.

The complexity of supporting non-inline functions arises from the requirement that a single function definition preemption solution be computed for all function invocations. This requirement does not mesh well with the production rule grammar approach used here and requires specialized methods for computing the preemption solution for each function definition. We use the term inline function to describe that each function invocation combined with its corresponding function definition potentially results in a different preemption solution, essentially behaving as if the function were actually inlined. The use of the notion of inline functions is a limitation in all conditional real-time preemption placement approaches. The problem of solving non-inline functions is a topic to be addressed in future work.

Interdependent CRPD Solution Handling

One of the primary motivations for our work is the interdependent CRPD cost model, which necessitates a series of modifications to the conditional PPP algorithm for proper solution handling.

Algorithm 4 Visible Section Pred. Preemptions

```
1:  function vis_pred_pps_sect(ς_curr, ρ, ρ_call)
2:      δ_i^left ← ς_curr.leftmostBB
3:      δ_i^right ← ς_curr.rightmostBB
4:      for δ_i^curr ∈ [δ_i^right, δ_i^left] do
5:          if δ_i^curr ∈ ρ then
6:              ρ_call ← ρ_call ∪ δ_i^curr
7:              Exit the For Loop.
8:          end if
9:      end for
10:     return ρ_call
11: end function
```

Algorithm 5 Visible Section Successor Preemptions

```
1:  function vis_succ_pps_sect(ς_curr, ρ, ρ_call)
2:      δ_i^left ← ς_curr.leftmostBB
3:      δ_i^right ← ς_curr.rightmostBB
4:      for δ_i^curr ∈ [δ_i^left, δ_i^right] do
5:          if δ_i^curr ∈ ρ then
6:              ρ_call ← ρ_call ∪ δ_i^curr
7:              Exit the For Loop.
8:          end if
9:      end for
10:     return ρ_call
11: end function
```

Algorithm 6 Visible Current Pred. Preemptions

```
1:  function vis_curr_pred_pps(ς_curr, ς_start, ρ, ρ_prev)
2:      α_sections ← true
3:      ρ_call ← ∅
4:      ρ_next ← ∅
5:      ρ_prev ← vis_pred_pps_sect(ς_curr, ς_start, ρ_prev)
6:      if ρ_prev ≠ ∅ then
7:          ρ_call ← ρ_call ∪ ρ_prev
8:      end if
9:      if ς_curr ≠ ς_start then
10:         for ς_next ∈ ς^pred(ς_curr) do
11:             ρ_sect ←
12:             vis_pred_pps_sect(ς_next, ς_start, ρ, ρ_call)
13:             if ρ_sect = ∅ then
14:                 α_sections ← false
15:             else
16:                 ρ_next ← ρ_next ∪ ρ_sect
17:             end if
18:         end for
19:         if α_sections = false then
20:             ρ_next ← ρ_next ∪ ρ_call
21:         end if
22:     else
23:         ρ_next ← ρ_call
24:     end if
25:     return ρ_next
26: end function
```

Algorithm 7 Visible Current Successor Preemptions

```
1:  function vis_curr_succ_pps(ς_curr, ς_start, ρ, ρ_next)
2:      α_sections ← true
3:      ρ_call ← ∅
4:      ρ_prev ← ∅
5:      ρ_next ← vis_succ_pps_sect(ς_curr, ς_start, ρ_next)
6:      if ρ_next ≠ ∅ then
7:          ρ_call ← ρ_call ∪ ρ_next
8:      end if
9:      if ς_curr ≠ ς_end then
10:         for ς_prev ∈ ς^succ(ς_curr) do
11:             ρ_sect ←
12:             vis_succ_pps_sect(ς_prev, ς_end, ρ, ρ_call)
13:             if ρ_sect = ∅ then
14:                 α_sections ← false
15:             else
16:                 ρ_prev ← ρ_prev ∪ ρ_sect
17:             end if
18:         end for
19:         if α_sections = false then
20:             ρ_prev ← ρ_prev ∪ ρ_call
21:         end if
22:     else
23:         ρ_prev ← ρ_call
24:     end if
25:     return ρ_prev
26: end function
```

One challenge interdependent CRPD presents (that independent CRPD does not) is the preemption cost cannot be determined when the preemption solutions for basic blocks are processed using production rule $\mathcal{P}1$ since the successor preemption is not known. Interdependent CRPD costs may only be determined for preemption pairs in contrast to independent CRPD, a function of a single preemption location only.

Algorithm 8 Visible Predecessor Preemptions

1: **function** $vis\_pred\_pps(\beta, \rho)$
2: Solution block CFG β, preemption solution ρ
3: $\rho_{prev} \leftarrow \emptyset$
4: $\varsigma_{start} \leftarrow \beta.startSection$
5: $\varsigma_{end} \leftarrow \beta.endSection$
6: $\rho_{vis} \leftarrow vis\_curr\_pred\_pps(\varsigma_{end}, \varsigma_{start}, \rho, \rho_{prev})$
7: **return** ρ_{vis}
8: **end function**

Algorithm 9 Visible Successor Preemptions

1: **function** $vis\_succ\_pps(\beta, \rho)$
2: Solution CFG block β, preemption solution ρ
3: $\rho_{next} \leftarrow \emptyset$
4: $\varsigma_{start} \leftarrow \beta.startSection$
5: $\varsigma_{end} \leftarrow \beta.endSection$
6: $\rho_{vis} \leftarrow vis\_curr\_succ\_pps(\varsigma_{end}, \varsigma_{start}, \rho, \rho_{next})$
7: **return** ρ_{vis}
8: **end function**

This means we have to determine the maximum preemption cost for pairs of solutions that are combined as higher-level block structures are processed. To accomplish this, we must have a way of determining the set of preemption points that are visible externally to adjacent solutions in order to compute the maximum preemption cost of the combined solutions. A preemption point has external visibility to adjacent blocks if there exists a path from the starting or ending section block to the preemption point with no intervening preemption points encountered. Determining the visible preemption points while combining preemption solutions adds an $O(N_i)$ factor to the algorithm for the current block.

Algorithm 8 illustrates the method of computing the visible predecessor preemption points for an existing solution. For the visible predecessor preemption points, we start with the ending section of the block and work our way backwards towards the starting section of the block. As we move backward, the preemption points encountered replace those in the current set if all sections have preemption points. If not, then the existing preemption points are still visible and copied to the preemption set for the next iteration. This is shown in Algorithm 6. For each section block processed, the first successor preemption point encountered moving from right to left is added to the preemption point set as shown in Algorithm 4. Computing the visible successor preemption points works in an identically symmetric way using three similar algorithms.

Algorithm 9 illustrates the method of computing the visible successor preemption points for an existing solution. For the visible successor preemption points, we start with the starting section of the block and work our way forwards towards the ending section of the block. As we move forward, the preemption points encountered replace those in the current set if all sections have preemption points. If not, then the existing preemption points are still visible and copied to the preemption set for the next iteration. This is shown in Algorithm 7. For each section block processed, the first predecessor preemption point encountered moving from left to right is added to the preemption point set as shown in Algorithm 5. For convenience, Table 5 summarizes the terminology presented in this section.

Term	Description
a	Current conditional section index
$\alpha_{sections}$	All sections containing preemptions
δ_i^{left}	Left most basic block of the current block
δ_i^{right}	Right most basic block of the current block
$\mathcal{G}$	Production rules grammar
$L(\mathcal{G})$	Textual language recognized by grammar $\mathcal{G}$
ζ_{pred}	Predecessor non-preemptive execution
ζ_{succ}	Successor non-preemptive execution
MaxIter	Maximum number of loop iterations
ρ_i	Preemptions for task τ_i
$\rho_i^A(G_i^A, \zeta_{pred}, \zeta_{succ})$	Preemptions for single subgraph G_i^A solution
ρ_{call}	Visible section preemptions
ρ_{next}	Visible successor preemptions
ρ_i^{pred}	Visible predecessor preemptions concept
$\rho_i^{pred}(G_i^A, \zeta_{pred}, \zeta_{succ})$	Visible predecessor preemptions for single subgraph G_i^A solution
ρ_{prev}	Visible predecessor preemptions
ρ_{sect}	Visible section preemptions
ρ_i^{succ}	Visible successor preemptions concept
$\rho_i^{succ}(G_i^A, \zeta_{pred}, \zeta_{succ})$	Visible successor preemptions for single subgraph G_i^A solution
ρ_{vis}	Visible predecessor/successor preemptions
r	Number of blocks in conditional
s	Current graph solution index variable
ς_{curr}	Current linear section variable
ς_{end}	Current graph ending linear section variable
ς_{next}	Next linear section variable
$\varsigma^{pred}(\varsigma_{curr})$	Predecessor sections of section ς_{curr}
ς_{start}	Current graph starting linear section variable
$\varsigma^{succ}(\varsigma_{curr})$	Successor sections of section ς_{curr}
t	Current graph solution index variable
u	Current graph solution index variable
$\Phi_i(G_i, \rho_i)$	WCET + preemption cost for task τ_i graph G_i
$\Phi_i^A(G_i^A, \zeta_{pred}, \zeta_{succ})$	WCET + preemption cost for single subgraph G_i^A preemption solution

Table 5: Preemption Placement Terminology

Evaluation

Our conditional PPP algorithm will be evaluated using two methods: 1) characterization and measurement of preemption costs using real-time application code, and 2) a breakdown utilization schedulability comparison of various PPP and fully preemptive CRPD algorithms. Each PPP algorithm evaluated either uses an independent or interdependent CRPD cost model.

Preemption Cost Characterization

Our study will utilize a subset of real-time tasks from the Malardalen University (MRTC) WCET benchmark suite [51] for comparing various PPP algorithms. The task code was compiled using the GCC

MIPS Cross Compiler for MIPS series processors with separate instruction and data $1KB$ direct-mapped caches with a line size of 32 bytes with 32 cache blocks. Tasks are scheduled using Earliest Deadline First (EDF) scheduling. Our method also supports fixed priority scheduling (FPS).

The compiled real-time task code was processed by the Heptane Static WCET analysis tool [34]. Heptane is used to determine the set of section blocks, the section block WCETs, the CFG structure, and the cache state at each instruction by analyzing the program executable. The analysis results are then imported into a Java benchmark parser program, designed to generate the task code grammar used by our conditional PPP algorithm. To accomplish this, high level programming constructs such as loops, conditionals, functions, and block statements must be recognized from the low-level compilation output.

Once the benchmark CFG structure has been constructed, the results of the Heptane cache state analysis are imported and used to compute the instruction and data UCBs ($\Upsilon_I(\delta_i^j)$ and $\Upsilon_D(\delta_i^j)$ respectively) at each program location. These sets are then used to compute the shared LCBs, along with the interdependent preemption cost matrix.

The intersection of the cache state snapshots from δ_i^j to δ_i^k are used to calculate shared LCBs. Shared LCBs represent the set of cache lines whose contents remain un-evicted after execution of basic blocks $\{\delta_i^{j+1}, \delta_i^{j+2}, ..., \delta_i^k\}$. As such, shared LCBs will continue to be present in the cache prior to the execution of basic block δ_i^{k+1}. Thus, a safe upper bound on the LCBs shared between each basic block pair can be represented by the set of unchanged cache lines. The following equation below formalizes this computation where the instruction cache snapshots are denoted $\Upsilon_I(\delta_i^j)$ and the data cache snapshots denoted $\Upsilon_D(\delta_i^j)$.

$$LCB(\delta_i^j, \delta_i^k) \subseteq \bigcap_{m=j+1}^{k} \left[\Upsilon_I(\delta_i^m) \cup \Upsilon_D(\delta_i^m)\right] \tag{91}$$

Availability

The following tools and data sets may be used to verify and reproduce our work. The MIPS GCC cross compiler and the Heptane static worst-case execution time tool are freely available. The research community may reproduce and leverage our work via the developed programs and analyzed data archived at GitHub [23].

Results

The results are presented as an illustration of the potential benefit of our proposed method, utilizing pairs of preemptions to determine costs, over methods that consider only the maximum CRPD at a particular preemption point (e.g., [14] and [58]). In terms of LCB computation this implies that the maximum

LCB value over all subsequent program points must be used as the CRPD cost:

$$\max\{LCB(\delta_i^j, \delta_i^k) \mid \delta_i^j \preceq \delta_i^k\} \tag{92}$$

The interdependent CRPD approach, representing CRPD cost for pairs of preemptions, is illustrated in the following graphs. The graph lines shown characterize the minimum and maximum shared LCBs between program points. The x-axis represents the first program preemption point, denoted δ_i^j. The y-axis measures the shared LCB count with the secondary program point, denoted δ_i^k, annotated at each graph point as shown. The first graph shown in Figure 65 characterizes the instruction cache CRPD costs for the FFT benchmark program. Each point on both curves plots the minimum or maximum CRPD value for the first preemption point given by the x-axis, and the next preemption point annotated on the graph. At program point δ_i^1, the minimum CRPD value is coupled with program point δ_i^8 having a shared LCB count of 175 whereas the single-valued CRPD computation method finds 250 shared LCBs coupled at program point δ_i^5. Comparison of the dual-valued interdependent CRPD with the single-valued independent

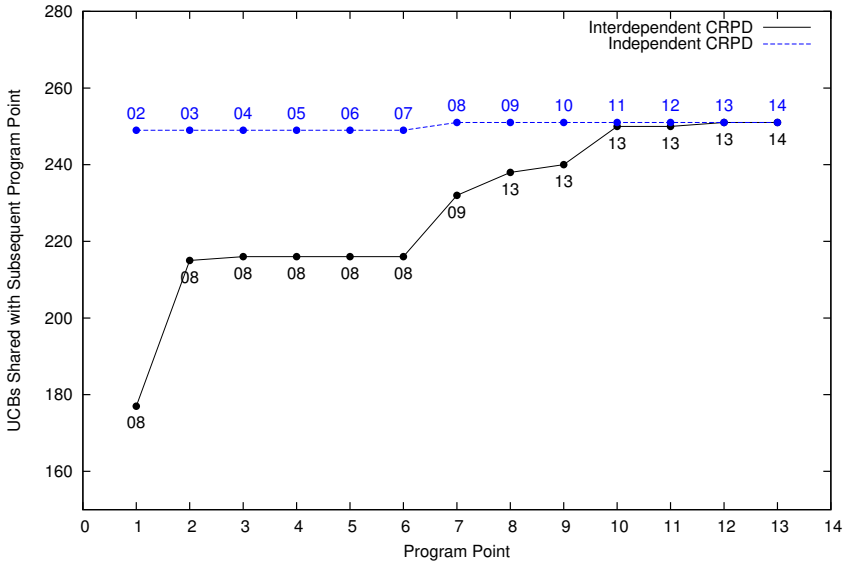

Figure 65: FFT Instruction Cache.

CRPD methods can be visualized by comparing the vertical distance between the minimum and maximum CRPD curves of Figures 65 through 70. The maximum CRPD cost at any program location represents a mandatory safe value for any single-valued independent CRPD approach. In contrast, our interdependent CRPD method offers the potential minimum value reported on the solid line. The benefit provided in considering location aware interdependent CRPD cost is captured by the difference between the minimum

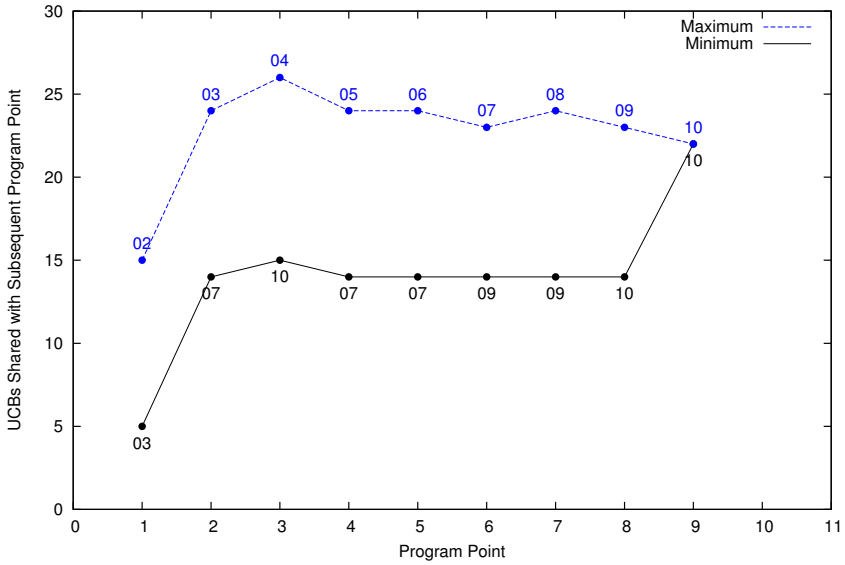

Figure 66: Recursion Data Cache.

and maximum CRPD cost curves fueling the improved performance of our conditional PPP algorithm. The variability in the minimum and maximum CRPD costs further exemplifies the benefits as illustrated in the second and third graphs, representing the lms benchmark task instruction cache in Figure 67 and the cover benchmark task instruction cache in Figure 69 respectively. In this paper, we have presented the variability witnessed in the instruction cache graphs, as the conditional CFG structure emphasizes the instruction cache effect on CRPD. Maximum and minimum instruction and data cache costs for the other MRTC tasks exhibit similar variability.

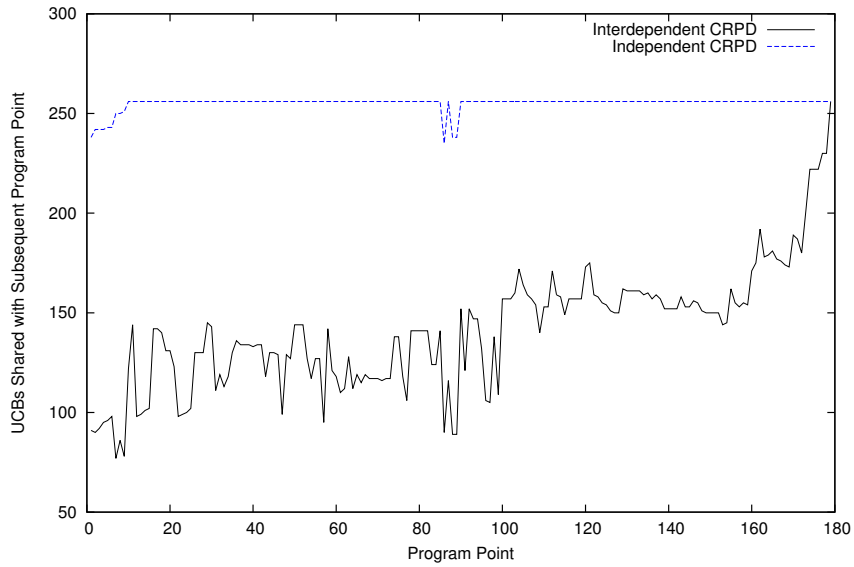

Figure 67: LMS Instruction Cache.

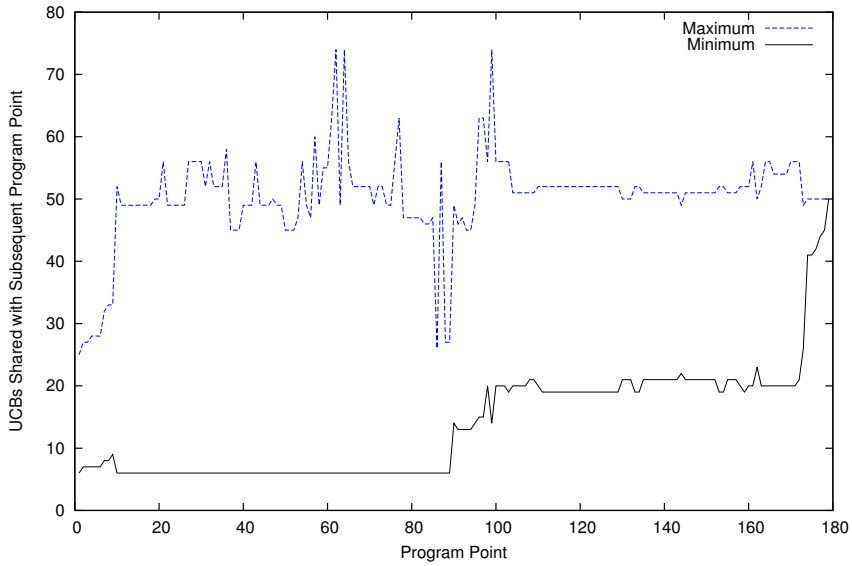

Figure 68: LMS Data Cache.

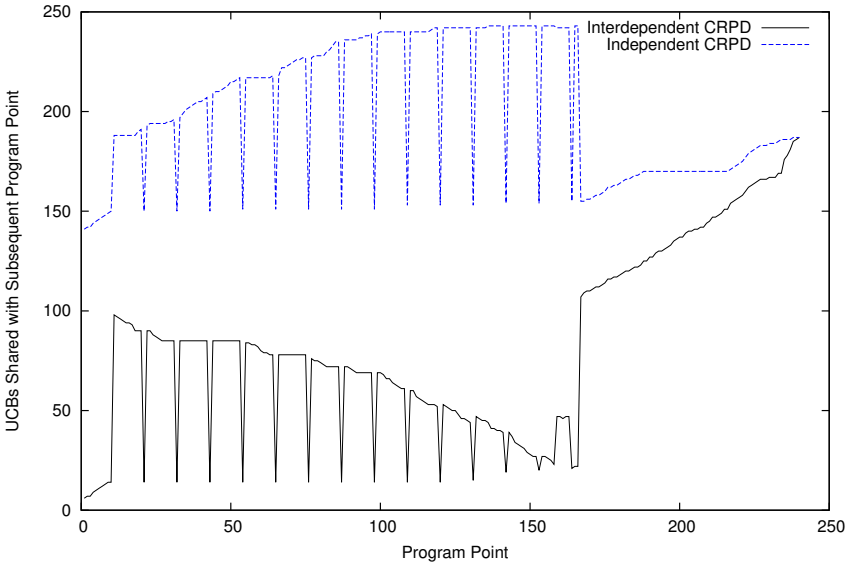

Figure 69: Cover Instruction Cache.

Review of the instruction and data cache graphs led to some notable observations. The maximum and minimum LCBs converged towards the end of each tasks CFG due to the decreasing CFG structure remaining thereby reducing the LCB count variability. Downward spikes are well aligned with task block boundaries, such as loops, conditionals, and functions. Early upward trends result from task initialization code. The separation between the two curves illustrates the accuracy improvement of our interdependent CRPD method versus independent CRPD methods. The interdependent CRPD costs for each graph are identical to the independent CRPD costs at the edges which coincide with the end of the task code. Our proposed PPP algorithm utilizes the more accurate interdependent CRPD cost leading to substantial

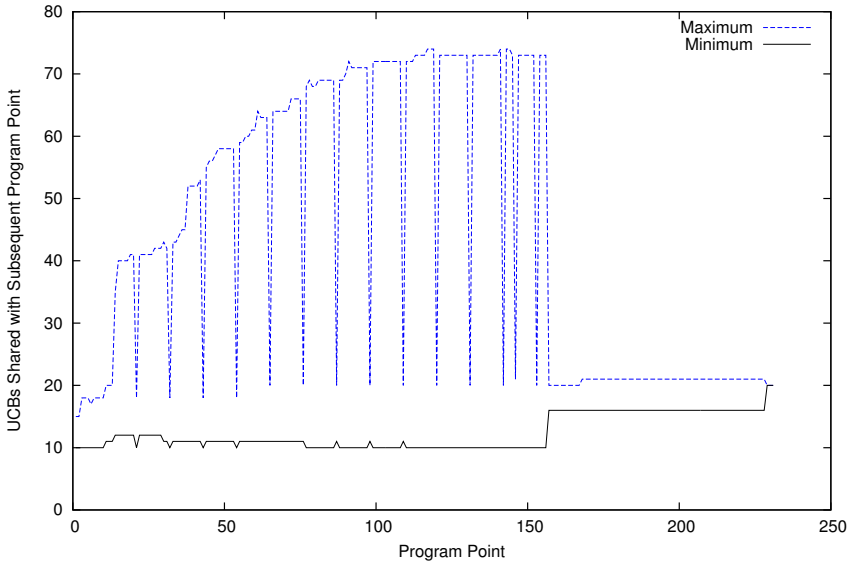

Figure 70: ADPCM Data Cache.

schedulability improvements.

Breakdown Utilization

Now that the reduced task preemption overhead benefits of the more precise interdependent CRPD cost model has been presented, we turn our attention to the benefits in task set schedulability. To evaluate task set schedulability, breakdown utilization performance was compared for several PPP algorithms on selected MRTC benchmark [51] tasks. The goal of breakdown utilization analysis [47] is to determine the utilization at which a task set becomes un-schedulable. The PPP algorithms compared in our study include the BEPP algorithm [14], our linear PPP (LEPP) algorithm [25], the PEPP conditional algorithm [58], and our proposed conditional PPP algorithm (CEPP). We compare these PPP algorithms against several fully preemptive CRPD approaches that have been integrated into task set response time analysis, namely JCR, ECB Only, UCB Only, ECB Union, UCB Union, ECB Union Multi Set, and UCB Union Multi Set [7].

The detailed steps of our iterative schedulability algorithm integrated with our conditional PPP algorithm are described in Algorithm 10. Task set utilization, given by U, is controlled by setting each tasks deadline and period to $D_i = T_i = u \cdot C_i^{NP}$. The constant, u, is binary search incremented in small steps until the task set becomes schedulable. Then u is binary search decremented in small steps until the task set becomes un-schedulable, resulting in the breakdown utilization U_B. For $BEPP$ and $PEPP$, the maximum shared LCB counts previously obtained form the independent CRPD input. Lastly, for $LEPP$ and $CEPP$, the explicit shared LCB counts previously obtained comprise the interdependent CRPD input for our linear and conditional explicit PPP algorithms. The remaining input variables required by the break-

down utilization algorithm are C_i^{NP} and BRT. C_i^{NP} was computed as the maximum number of cycles to complete task execution non-preemptively. During each run of the breakdown utilization study, the BRT parameter is swept from 1 ns to 640 ns per MIPS processor family performance.

Algorithm 10 Breakdown Utilization Evaluation Algorithm

```
1: Start with a task system that may or may not be feasible.
2: Assume the CRPD of the task system is initially zero.
3: repeat
4:     Run the Iterative Schedulability and PPP Algorithm
5:     if the task system is feasible/schedulable then
6:         Increase U by decreasing T_i values via binary search.
7:     else
8:         Decrease U by increasing T_i values via binary search.
9:     end if
10: until the utilization change is less than some tolerance.
11: The breakdown utilization is given by U.
```

Due to compiler optimizations, we had to post-process the MRTC tasks to apply our grammar. Post processing was required when the compiler generated non-structured assembly code not conforming to our grammar $\mathcal{G}$. This task consisted of transforming the assembly code into an equivalent structured form. Ideally, an automated tool would exist for this step; however, such a tool is beyond the scope of this paper due to the explicit compiler-level detail required. For this paper, we manually post-processed the following MRTC tasks: simple, bs, fibcall, lcdnum, sqrt, qurt, insertsort, ns, ud, crc, expint, jfdctint, matmult, and bsort100 [23].

Using the completed MRTC tasks, the breakdown utilization comparison between various PPP methods is summarized in Figure 71. The breakdown utilization results indicate that the $CEPP$ (resp. $LEPP$) algorithm dominates the $PEPP$ (resp. $BEPP$) algorithm primarily due to the benefits of interdependent versus independent CRPD. Both the $CEPP$ and $PEPP$ algorithms dominate $LEPP$ and $BEPP$ algorithms due to the enhanced granularity of the conditional CFGs, offering more possible preemptions than the linear CFGs. As expected, the breakdown utilization values converge for each distinct graph structure (e.g. linear, conditional) as cache-overhead becomes negligible for small BRT values. The conditional methods converge, and the linear methods converge. Specifically, the size of the individual blocks in the linear approaches are higher and hence are less schedulable than the finer grained conditional approaches. The performance of the CRPD response time analysis methods is generally hampered into two respects, namely, 1) they take a coarser view accounting for an entire task of UCBs and ECBs on every preemption, and 2) the number of higher priority task preemptions in the response time of

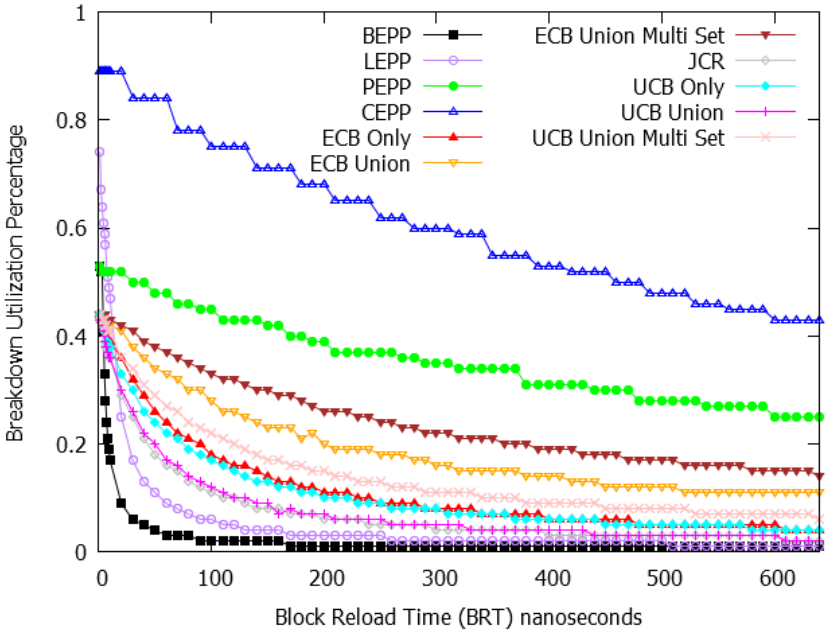

Figure 71: Breakdown Utilization Comparison.

preempted tasks is not limited in accordance with the inherent slack in the system. The ECB Union Multi Set and UCB Union Multi Set approaches improve the pessimism in terms of the number of preemptions as compared to the other CRPD methods, however the system slack is not considered. Improvements are also realized in the accounting of UCBs and ECBs, however, they are encumbered by the pessimism of the task view of UCBs and ECBs.

We also compared the various preemption placement algorithms against the corresponding fixed priority (FP) preemption threshold scheduling (FPTS) methods, namely, ECB Only, UCB Only Multi Set, ECB Union Multi Set, and UCB Union Multi Set [18, 19] as shown in Figure 72. Here we compare all CRPD methods directly without any further task layout improvements using simulated annealing. Compared to the traditional CRPD approaches, the corresponding FPTS methods realize substantial improvement due to 1) reduced preemption exposure to higher priority tasks, and 2) reduced job execution time frame known as the hold time where higher priority tasks may preempt. However, FPTS shares a similar trait in its pessimistic task view of UCBs and ECBs thereby limiting performance. For convenience, Table 6 summarizes the terminology presented in this section.

Term	Description
$\Upsilon_D(\delta_i^m)$	Data cache snapshot at basic block δ_i^m
$\Upsilon_I(\delta_i^m)$	Instruction cache snapshot at basic block δ_i^m

Table 6: Evaluation Terminology

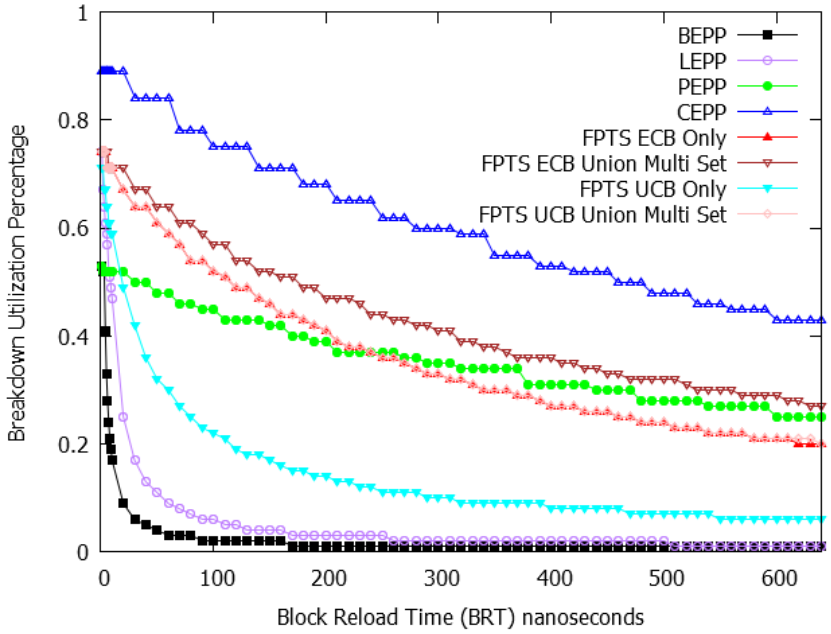

Figure 72: FPTS Breakdown Utilization Comparison.

Summary

In this work, we presented a conditional PPP algorithm using a more precise interdependent CRPD metric. By extending the interdependent CRPD cost to conditional CFG structures, further reductions in task preemption overhead were realized, leading to substantial schedulability improvements. These improvements were achieved by integrating our conditional interdependent CRPD PPP algorithm with algorithms from well-established task set schedulability theory. Our iterative schedulability algorithm demonstrates the convergence of selecting preemptions balanced by the task maximum non-preemptive execution region constraint Q_i. Our experiments demonstrated improved schedulability on real-time code using interdependent CRPD.

In future work, we plan to 1) extend the breakdown utilization analysis to the remaining MRTC benchmark tasks, 2) perform a timing analysis of various PPP algorithms, 3) add support for non-inline functions, 4) add support for goto statements, 5) extend the CRPD techniques described here to set-associative caches, and 6) conduct an in-depth case study on commercial real-time code.

CHAPTER 7 INTEGRATING PREEMPTION THRESHOLDS WITH LIMITED PREEMPTION SCHEDULING

In Chapter 6, we introduced innovative methods for computing accurate CRPD, integrating the enhanced CRPD with EDF schedulability analysis, and placing optimal preemption points to ensure taskset schedulability for conditional control flow graphs in uniprocessor systems. In this chapter, we present our research for integrating limited preemption scheduling with preemption threshold scheduling by combining preemption placement with optimal threshold assignment.

Introduction

The utility of real-time system computations depends on two important properties, correctness and timeliness. The timeliness property (the subject of schedulability analysis) is concerned with ensuring real-time task computations are completed within required deadlines. Designers of real-time systems must choose the scheduling paradigm that will ultimately determine if the real-time task set will meet its timeliness objectives. The available choices are 1) non-preemptive scheduling, 2) fully preemptive scheduling, and 3) limited preemption scheduling. Non-preemptive scheduling suffers from blocking of high priority tasks and fully preemptive scheduling suffers from substantial preemption overhead (up to 44% [55–57] of a tasks WCET) each approach degrading task set schedulability. Limited preemption scheduling attempts to 1) reduce blocking by limiting the number of allowed preemptions, maximizing non-preemptive task execution and 2) reduce preemption overhead via non-preemptive regions. Regardless of the chosen scheduling paradigm, effective schedulability analysis of real-time task sets mandates accurate WCET and CRPD estimates. The recognized benefits of limited preemption scheduling have motivated recent work on PPP algorithms. In a complimentary fashion, preemption threshold scheduling assigns preemption thresholds allowing tasks to execute non-preemptively where feasible. The two methods take advantage of inherent task set execution slack to enhance schedulability. The primary difference arises in the non-preemptive granularity. Limited preemption scheduling divides each task into non-preemptive chunks whereas preemption threshold scheduling facilitates non-preemptive execution at the task level.

The importance of CRPD in schedulability analysis stems from it comprising the majority of preemption overhead. CRPD occurs when a task denoted τ_i is preempted by one or more higher priority tasks denoted τ_k. The execution of high priority tasks results in the eviction of cache memory blocks that must be subsequently reloaded when task τ_i resumes execution. Two primary models of CRPD computation exist, 1) the independent CRPD cost model, and 2) the interdependent CRPD cost model. The vast majority of CRPD research falls under the independent CRPD model. Here, costs are solely a function of the preemption location under consideration. Since the next preemption may occur at any forward point in the

task code, independent CRPD methods must conservatively utilize the next code location corresponding to the maximum CRPD cost. The interdependent CRPD cost model, however, overcomes this limitation by considering and computing costs between each pair of task code locations thereby achieving more accuracy. A key factor in scheduling decisions, CRPD cost accuracy is of paramount importance to preemption placement, optimal threshold assignment, and schedulability algorithms.

PPP algorithms select preemption points for each task to 1) minimize the overall task WCET, and 2) ensure the execution time between adjacent preemptions is limited by the maximum non-preemptive region execution time. The maximum non-preemptive region execution time, denoted Q_i, is determined via task set schedulability analysis. The motivation behind our work is the potential benefits of preemption threshold scheduling can be integrated with existing PPP algorithms to further enhance task set schedulability. Our approach effectively integrates limited preemption scheduling via preemption placement with preemption threshold scheduling via optimal preemption threshold assignment resulting in improved schedulability. The benefits of our approach will be illustrated in a case study employing synthetically generated real-time tasks.

Schedulability Analysis

In this section, analysis of FP [72], and PTS [18, 19] limited preemption scheduling, summarizes the computation of the maximum non-preemptive region parameter Q_i supporting conditional and linear preemption point placement algorithms.

The goal of schedulability analysis is to determine whether a taskset is schedulable under the worst-case task activation pattern for a particular scheduling paradigm. The worst-case activation pattern for task τ_i, known as the critical instant, results in the maximum response time. Earlier work [43] proved the critical instant for each task coincides with the synchronous activation of the task with all higher priority tasks where all jobs released immediately in accordance with the minimum inter arrival time.

Limited Preemption FP Scheduling

For limited preemption FP scheduling the set of higher priority tasks is represented by:

$$k \in hp(i) = \{k \mid \pi_k > \pi_i\} \tag{93}$$

where π_j represents the assigned nominal priority for task τ_j. To facilitate the schedulability analysis for FP limited preemption scheduling, the request bound function [42] $RBF(t)$ is used to examine the

maximum cumulative execution request in an interval of length t generated by jobs of τ_i.

$$RBF_i(t) = \left\lceil \frac{t}{T_i} \right\rceil (C_i^{NP} + \gamma_i) \tag{94}$$

where γ_i denotes the preemption cost due to preemption by higher priority tasks $hp(i)$ during execution of task τ_i. Starting with the critical instant, the cumulative execution request in an interval t for task τ_i and all higher priority FP tasks is given by:

$$W_i(t) = \sum_{j \in i, hp(i)} RBF_j(t) \tag{95}$$

One way to look at this limited preemption FP schedulability constraint is to characterize the amount of blocking tolerance β_i that a task τ_i can withstand while meeting its deadlines.

$$\beta_i = \max_{t \in A | t < D_i} \left\{ t - \sum_{j \in hp(i), i} RBF_j(t) \right\} \tag{96}$$

where $A = \{mT_j, m \in \mathbb{N}, 1 \leq j < n\}$. We can also characterize the blocking factor B_i each task τ_i experiences which is given by:

$$B_i = \max_{l \in lp(i)} \{q_l^{max}\} \tag{97}$$

where the set of lower priority tasks $lp(i)$ is given by:

$$l \in lp(i) = \{l \mid \pi_l < \pi_i\} \tag{98}$$

Characterized by the non-preemptive regions inherent to limited preemption scheduling, the analysis must take into account the longest NP region in the lower priority tasks. The maximum NP region in task τ_i is given by:

$$q_i^{max} \leq Q_i = \min_{h \in hp(i)} \beta_h \tag{99}$$

Limited Preemption Threshold Scheduling

For PTS scheduling, the set of higher priority tasks is represented by two different conditions: 1) when a job completes, and the scheduler must select the next task job to execute, and 2) when a higher priority task preempts an existing task during its execution at the end of a non-preemptive region (NPR). The first condition is given by Equation 93. The second condition is given by:

$$k \in ht(i) = \{k \mid \theta_k > \pi_i\} \tag{100}$$

In PTS, the tasks that may preempt task τ_i have a nominal priority θ_k exceeding the computed preemption threshold π_i. Higher priority tasks that can block task τ_i are those that cannot be preempted by τ_i due to preemption threshold assignments is given by:

$$b_h(i) = hp(i) \setminus ht(i) \tag{101}$$

The set of lower priority tasks that may block task τ_i in cases where they are activated just before task τ_i is given by:

$$b_l(i) = lp(i) \setminus lt(i) \tag{102}$$

where the set of lower priority tasks task τ_i can preempt is given by:

$$l \in lt(i) = \{l \mid \theta_l < \pi_i\} \tag{103}$$

To facilitate the schedulability analysis for PTS scheduling, the request bound function [42] $RBF(t)$ is also used to examine the maximum cumulative execution request in an interval of length t generated by jobs of τ_i per Equation 94. Starting with the critical instant, the cumulative execution request in an interval t for task τ_i and all higher priority $ht(i)$, higher priority blocking $b_h(i)$ and lower priority blocking $b_l(i)$ FPTS tasks is given by:

$$W_i(t) = \sum_{j \in ht(i), b_h(i), i} RBF_j(t) + \max_{l \in b_l(i)} RBF_l(t) \tag{104}$$

Like FP scheduling, we can look at this limited preemption PTS schedulability constraint by characterizing the amount of blocking tolerance β_i that a task τ_i can withstand while meeting its deadlines.

$$\beta_i = \max_{t \in A | t < D_i} \left\{ t - \sum_{j \in ht(i), b_h(i), i} RBF_j(t) - \max_{l \in b_l(i)} RBF_l(t) \right\} \tag{105}$$

where A is identical to the FP scheduling expression. We can also characterize the blocking factor B_i each task τ_i experiences which is given by:

$$B_i = \max(\max_{l \in lp(i)} \{q_l^{max}\}, \max_{b \in b_l(i)} \{RBF_b(t)\}) \tag{106}$$

Characterized by the non-preemptive regions inherent to limited preemption scheduling, the analysis must take into account the longest NP region in the lower priority tasks. The maximum NP region in task τ_i is

given by:

$$q_i^{max} \leq Q_i = \min_{h \in ht(i)} \beta_h \tag{107}$$

Limited Preemption Scheduling Analysis

Since the lowest priority task τ_m is subjected to no blocking, by convention we have $B_m = 0$. We can use the schedulability analysis derived for floating non-preemptive regions [15] to establish task set schedulability in each case:

$$W_i(t) + B_i \leq t, \; \forall \, t \; \in \; mT_i, \; m \; \in \; \mathbb{N} \tag{108}$$

With suitable expressions for β_i introduced, we can express the task set limited preemption schedulability constraint for FP, and PTS scheduling as summarized in Theorem 11 [14].

Theorem 11. *A task set τ is schedulable with limited preemption scheduling if, $\forall i \mid 1 \leq i \leq n$,*

$$B_i \leq \beta_i \tag{109}$$

We can restate the taskset schedulability constraint in terms of the maximum non-preemptive region parameter Q_i via Theorem 12 [14].

Theorem 12. *A task set τ is schedulable with limited preemption scheduling if, $\forall i \mid 1 < i \leq n$,*

$$q_i^{max} \leq Q_i \tag{110}$$

In summary, each task is permitted to execute non-preemptively for a maximum amount of time denoted by Q_i thereby ensuring taskset schedulability as long as preemption points are placed such that all non-preemptive regions are less than or equal to Q_i.

Preemption Placement Objective

Using the series-parallel graph structure, the objective of any preemption placement algorithm is to select a single set of effective preemption points ρ_i that minimizes the WCET+CRPD of each task whose real-time conditional code is given by graph G_i, subject to the constraint that all non-preemptive regions must be less than or equal to the maximum allowable non-preemptive region parameter Q_i. This objective is formally stated as:

Preemption Placement Objective:

Given a real-time conditional flow graph $G_i \; \in \; L(\mathcal{G})$, an independent or interdependent CRPD cost function $\xi_i(\delta_i^x, \delta_i^y)$ and WCET b_i^j for each basic block, find a set of Effective Preemption Points (EPPs)

$\rho_i \subseteq E$ that minimizes the cost function:

$$\Phi_i(G_i, \rho_i) \stackrel{\text{def}}{=} \max_{p \in P_i(G_i, \delta_i^s, \delta_i^e)} [\sum_{\delta_i^x \in p} b_i^x + \sum_{\substack{(\delta_i^x, \delta_i^y) \in p, \rho_i \\ \delta_i^x \prec_p \delta_i^y}} \xi_i(\delta_i^x, \delta_i^y)] \tag{111}$$

subject to the constraint $\forall p \in P_i(G_i, \delta_i^s, \delta_i^e), \delta_i^w \in \rho_i, \exists e_i^{u,v} = (\delta_i^u, \delta_i^v), e_i^{x,y} = (\delta_i^x, \delta_i^y)\, where\, e_i^{u,v},\ e_i^{x,y} \in \rho_i$ ::

$$[\sum_{\substack{\delta_i^w \in p \\ \delta_i^u \preceq_p \delta_i^w \preceq_p \delta_i^x}} b_i^w + \xi_i(\delta_i^u, \delta_i^x)] \leq Q_i \tag{112}$$

The cost function $\Phi_i(G_i, \rho_i)$ evaluates to the maximum cost across all paths p through the task code.

Response Time Analysis

In this section, response time analysis of FP [72], and PTS [18, 19] limited preemption scheduling, summarizes the computation of the worst-case response time of tasks under each scheduling paradigm. Response time analysis is used in the PTS optimal threshold assignment algorithm and in checking taskset schedulability as a means of comparing the various scheduling methods in our evaluation. In each subsection, we begin by computing the time period associated with the worst-case task activation pattern, known as the level-i active period, for each task τ_i in the task set τ. For limited preemption FP threshold scheduling, the worst-case activation pattern occurs when all higher priority tasks are activated coincident with the activation of task τ_i with the activation of the blocking task with the largest execution cost just prior to that of task τ_i. We subsequently compute the worst-case starting and finishing times in accordance with the scheduling algorithm. Finally, the worst-case response time is the maximum difference between the finish time $F_{i,k}$ any of the k jobs in the level-i active period and the absolute deadline of the k^{th} job.

Limited Preemption FP Scheduling

For limited preemption FP scheduling, the level-i active period is given by:

$$L_i = \sum_{j \in hep(i)} E_j(L_i) \cdot \Phi_j(G_j, \rho_j) + \max_{l \in lp(i)} Q_l \tag{113}$$

where the $\max_{l \in lp(j)} Q_l$ represents the worst-case blocking experienced by task τ_i inherent to limited preemption scheduling and the term $E_j(t)$ is given by:

$$E_j(t) = \lceil t/T_j \rceil \tag{114}$$

The start time $S_{i,k}$ of the k^{th} job of task τ_i is given by:

$$S_{i,k} = k \cdot \left[\Phi_i(G_i, \rho_i) + \max_{l \in lp(i)} Q_l\right] + \sum_{j \in hp(i)} E_j(S_{i,k}) \cdot \Phi_j(G_j, \rho_j) \tag{115}$$

The finish time $F_{i,k}$ of the k^{th} job of task τ_i is given by:

$$F_{i,k} = S_{i,k} + \left[\Phi_i(G_i, \rho_i) + \max_{l \in lp(i)} Q_l\right] + \sum_{j \in hp(i)} (E_j(F_{i,k}) - E_j(S_{i,k})) \cdot \Phi_j(G_j, \rho_j) \tag{116}$$

The worst-case response time R_i of task τ_i is given by:

$$R_i = \max_{k \in E_i(L_i)} (F_{i,k} - kT_i) \tag{117}$$

Limited Preemption FP Threshold Scheduling

For limited preemption FP threshold scheduling, the level-i active period is given by:

$$L_i = \sum_{j \in hep(i)} E_j(L_i) \cdot \Phi_j(G_j, \rho_j) + \max\Big(\max_{l \in lt(i)} Q_l, \max_{b \in b_l(i)} \Phi_b(G_b, \rho_b)\Big) \tag{118}$$

where $b_l(i)$ represents the set of tasks that may block task τ_i which for limited preemption threshold scheduling is given by Equation 102 and where $lt(i)$ is given by Equation 103. For limited preemption threshold scheduling, the start time $S_{i,k}$ of the k^{th} job of task τ_i is given by:

$$S_{i,k} = k \cdot \Phi_i(G_i, \rho_i) + \max\Big(\max_{l \in lt(i)} Q_l, \max_{b \in b_l(i)} \Phi_b(G_b, \rho_b)\Big) + \sum_{j \in hp(i)} E_j(S_{i,k}) \cdot \Phi_j(G_j, \rho_j) \tag{119}$$

The finish time $F_{i,k}$ of the k^{th} job of task τ_i is given by:

$$F_{i,k} = S_{i,k} + \Phi_i(G_i, \rho_i) + \sum_{j \in ht(i)} (E_j(F_{i,k}) - E_j(S_{i,k})) \cdot \Phi_j(G_j, \rho_j) \tag{120}$$

The worst-case response time R_i of task τ_i is given by Equation 117.

Implementation

In this section, we present the integration of limited preemption FP scheduling with FP preemption threshold scheduling. The integration consists of an iterative algorithm combining preemption placement with optimal threshold assignment.

High-Level Overview

Limited preemption scheduling and preemption threshold scheduling are complimentary in that they use the execution time slack inherent in the task set τ to permit tasks to execute non-preemptively. Limited preemption scheduling accomplishes this via a task parameter Q_i specifying task τ_i's maximum non-

preemptive execution time within which the task set τ remains schedulable. In essence, limited preemption scheduling divides each task τ_i into a sequence of non-preemptive execution regions bounded by Q_i. In preemption threshold scheduling, preemption thresholds are adjusted for circumstances where the execution slack of task τ_i can completely subsume the execution time of task τ_b while still meeting its deadline, the preemption threshold of task τ_b is adjusted upwards to prevent it from being preempted by task τ_i. Each method attempts to introduce non-preemptive execution into a task set τ commensurate with its real-time execution characteristics. One of the primary differences in these two methods is the level of granularity at which the non-preemptive regions are established. Limited preemption scheduling divides each task into non-preemptive regions or chunks whereas preemption threshold scheduling permits non-preemptive execution at the task level.

Preemption Placement and OTA Integration

In this section, we present an algorithm that integrates preemption placement with optimal threshold assignment as summarized in Algorithm 11.

Algorithm 11 Integrated Preemption Placement With OTA

Input Task set $\tau = \tau_1, ..., \tau_n$ with $\{V_i, E_i, C_i^{NP}, T_i, D_i, \pi_i\}\ \forall \tau_i \in \tau$.
Output Task set schedulable, $\rho_i \subseteq E_i$, and $\Theta \in \Pi$, $\forall \tau_i \in \tau$.

for all $\tau_i \in \tau$ **do**
    $\theta_i \leftarrow \pi_1$;
    $Q_i \leftarrow C_i^{NP}$;
    $C_i \leftarrow C_i^{NP}$;
    $\rho_i \leftarrow \emptyset$;
end for
for $\tau_i \leftarrow [\tau_1, ..., \tau_n]$ **do**
    Compute L_i using Equation 121;
    Compute β_i using Equations 122, 123;
    if $\beta_i < 0$ **then**
        return $unschedulable$;
    end if
    for $\forall \tau_l$ with $l \in lp(i)$ **do**
        if $Q_l > \beta_i$ *or* $C_l > \beta_i$ **then**
            $\theta_l \leftarrow \pi_{i+1}$;
            $Q_l \leftarrow \min(Q_l, \beta_i)$;
            $\rho_l \leftarrow$ Perform task τ_l preemption placement
            using Q_l and the set $hep(i)$ as preempting tasks;
            $C_l \leftarrow \Phi_l(G_l, \rho_l)$;
        end if
    end for
end for
return $schedulable$;

We use a slightly modified version of Equation 118 to compute the level-i active period term L_i given by:

$$L_i = \sum_{j \in hep(i)} E_j(L_i) \cdot \Phi_j(G_j, \rho_j) \; + \; \max\Big(\max_{b \in b_l(i)} C_b, \max_{l \in lt(i)} Q_l\Big) \tag{121}$$

We also use a slightly modified version of Equation 106 to compute the task τ_i blocking tolerance β_i term as given by:

$$\beta_i = \min_{k=0}^{\lceil \frac{L_i}{T_i} \rceil - 1} \{\beta_i^k\} \tag{122}$$

where the term β_i^k is given by:

$$\beta_i^k = \max_{t \in [kT_i, kT_i + D_i]} \Big\{ t - \sum_{j \in hep(i)} RBF_j(t) \Big\} \tag{123}$$

The algorithm begins by initializing the variables θ_i, Q_i, C_i, and ρ_i. We start with setting each task's preemption threshold θ_i to the highest priority with the default state resulting in each task executing non-preemptively (line 6). Consistent with the non-preemptive execution defaults, we initialize the variables Q_i and C_i to be the task non-preemptive execution time C_i^{NP} (lines 7,8). Lastly we initialize the task preemption solution ρ_i to be empty set (line 9). At a high level, we process each task in the task set, computing the level-i active period L_i and the blocking tolerance β_i (lines 13,14). If the blocking tolerance is less than zero, we return the task as un-schedulable (lines 16-18). Next we analyze the potential blocking introduced by all lower priority tasks (line 20). If either the maximum non-preemptive region parameter Q_l or the WCET+CRPD variable C_l exceeds the blocking tolerance of task τ_i (line 21), then we set the maximum preemption threshold θ_l such that task τ_l cannot block task τ_i (line 22). We also constrain the maximum non-preemptive region Q_l of task τ_l to the blocking tolerance β_i of task τ_i (line 23). With the modified maximum non-preemptive region parameter Q_l, we perform preemption point placement using Q_l and Θ with the set $hep(i)$ as the preempting tasks (lines 25,26). Following preemption placement, we update the WCET+CRPD variable C_l with the computed preemption placement costs (line 28). Once each task has been considered along with analyzing the potential blocking introduced by lower priority tasks, we return schedulable (line 33). A proof of correctness and discussion of our integrated preemption placement and maximum threshold assignment is given in the following subsection.

Integrated PP/OTA Algorithm Proof of Correctness

In this section, we prove the correctness of the integrated preemption placement and maximum preemption threshold assignment algorithm as detailed in Algorithm 11.

Theorem 13. *Given a schedulable task set τ, the task set is schedulable with limited preemption scheduling if $\forall i \mid 1 \leq i \leq n$,*

$$Q_l \leq \beta_i \; \forall \; l \; \in lp(i) \tag{124}$$

Proof. The proof is by direct argument. Using the result of Theorem 109, we have:

$$B_i \leq \beta_i \; \forall i \mid 1 \leq i \leq n \tag{125}$$

Substituting Equation 106 for B_i, we have:

$$\max(\max_{l \in lp(i)} \{q_l^{max}\}, \max_{b \in b_l(i)} \{RBF_b(t)\}) \leq \beta_i \tag{126}$$

Substituting Equation 107 for q_l^{max}, and simplifying the RBF term with C_b, we have:

$$\max(\max_{l \in lp(i)} \{Q_l\}, \max_{b \in b_l(i)} \{C_b\}) \leq \beta_i \tag{127}$$

The first term represents blocking via limited preemption scheduling with the second term representing blocking via non-preemptive execution. We can reduce the constraint to the limited preemption scheduling case since $Q_l \leq C_l$. Therefore, we have:

$$Q_l \leq \beta_i \; \forall l \in lp(i) \tag{128}$$

Thus, proving the schedulability constraint. □

We use the schedulability constraint to prove the correctness of Algorithm 11.

Theorem 14. *Given a schedulable task set τ, Algorithm 11 assigns the maximum non-preemptive region set Q and maximum preemption threshold set Θ to tasks achieving schedulability if such an assignment exists.*

Proof. The proof is by induction.

Basis: We start with the highest priority task τ_1. Being the highest priority task, τ_1 executes completely non-preemptively and its schedulability can be evaluated using the expression $C_i^{NP} \leq D_i$ or $\beta_i \geq 0$. If the condition $\beta_i \leq 0$ is true, we return unschedulable (lines 16-18). Otherwise, β_i has the computed execution slack or blocking tolerance with both Q_1 and C_1 set to the non-preemptive execution time C_i^{NP}.

Induction: For each task τ_i, assuming the previous $i-1$ tasks $\{\tau_1, \tau_2, ..., \tau_{i-1}\}$ are schedulable, each iteration of the outer loop, results in a schedulable task set $\{\tau_1, \tau_2, ..., \tau_i\}$. If the task set is schedulable, then Algorithm 11 assigns Q_l and $\theta_l \; \forall l \in lep(i)$ such that $\{\tau_1, \tau_2, ..., \tau_i\}$ is schedulable. For each lower

priority task τ_l, the inner loop evaluates the extent to which τ_l can execute non-preemptively with respect to task τ_i. Two non-preemptive execution possibilities exist: 1) execute completely non-preemptively if $(C_l \leq \beta_i)$, or 2) execute in limited preemption fashion $(Q_l \leq \beta_i)$. As required to maintain schedulability, if either parameter Q_l or C_l exceeds task τ_i blocking tolerance β_i, then we must execute using limited preemption scheduling only. We adjust Q_l if needed to ensure task τ_l blocking imposed on task τ_i is bounded by β_i (line 23). The θ_i is correspondingly adjusted to a level where task τ_i must be able to preempt task τ_l. Revisions to Q_l and θ_l are followed by computation of a new preemption solution (lines 25,26). Otherwise if Q_l and C_l are already bounded by task τ_i blocking tolerance β_i, then no adjustments are made and task τ_l may execute completely non-preemptively with respect to task τ_i. Thus, with each iteration of the outer loop, the task set $\{\tau_1, \tau_2, ..., \tau_i\}$ remains schedulable with the assigned maximum non-preemptive region set Q and maximum preemption threshold set Θ, if such an assignment exists. □

Evaluation

Our integrated PPP/OTA algorithm will be evaluated using the following method, namely the average preemption cost swept across the range of system utilization, and 2) the weighted schedulability ratio as a function of cache utilization.

Task Set Generation

Our study will utilize synthetically generated task sets of linear CFGs along with task execution parameters for comparing various linear PPP algorithms against those augmented with our integrated PPP/OTA algorithm. Each experiment generates 1000 different task sets. Target task utilization is swept in accordance with the breakdown utilization algorithm. The number of tasks is randomly selected using a uniform distribution in the interval $[8, 15]$. The number of basic blocks in each task is randomly selected using a uniform distribution in the interval $[20, 200]$. The number of instructions in each basic block is randomly selected using a uniform distribution in the interval $[10, 100]$. The WCET of each basic block is randomly selecting using a gaussian distribution with $\mu = 200$ and $\sigma = 10$. The absolute value of all generated WCET values are used. Our system employs a separate instruction and data direct mapped caches containing $N^C = 1024$ cache blocks with a total cache utilization $U^C = 4$. The cache block size s is randomly generated using a uniform distribution selected from the set $\{8, 16, 32, 64, 128\}$ bytes. The total cache memory size is given by the expression $N^C \cdot s$. The total number of task set ECBs is given by $N^C \cdot U^C = 4096$. Individual task cache utilization U_i^C is generated via the UUniFast algorithm [16]. Tasks occupy contiguous memory ordered from highest priority to lowest priority. Individual instruction ECBs are determined from the generated memory layout. Individual data ECBs are generated via a two-pronged approach. Within each basic block, a median RAM memory location MA is generated using a

uniform distribution over the RAM memory space. The RAM memory accessed by each instruction is generated using a gaussian distribution with $\mu = MA$ and $\sigma = 512$ bytes. Task data UCBs have a targeted overlap of 40% with its data ECBs. Task periods T_i are generated using a uniform distribution in the interval $[10, 1000]$ mS. Arbitrary task deadlines D_i are generated using a uniform distribution in the interval $[(C_i + T_i)/2, 4T_i]$. Individual task utilization are generated via the UUniFast algorithm [16]. Nominal task priorities Π are assigned in descending order commensurate with a Deadline Monotonic policy.

Availability

The research community may reproduce and leverage our work via the developed programs and analyzed data archived at GitHub [23].

Results

The results are presented as an illustration of the potential benefit of our integrated PPP/OTA algorithm combining limited preemption scheduling with preemption threshold scheduling, in comparison to existing limited preemption methods.

System Utilization Schedulability

In this section, we study the performance of limited preemption scheduling using preemption placement only as a baseline, as compared to preemption placement integrated with preemption threshold scheduling via optimal threshold assignment. We perform a breakdown utilization comparison between the two approaches along with a comparison of the average preemption cost swept across the system utilization range for schedulable task sets. By comparing these two methods we illustrate the superiority of the integrated preemption placement/preemption threshold scheduling algorithm versus limited preemption scheduling via preemption placement only. Figure 73 illustrates the breakdown utilization comparison. The limited preemption scheduling via preemption placement is denoted as LEPP whereas the integrated preemption placement/preemption threshold scheduling method is denoted as LEPPOTA.

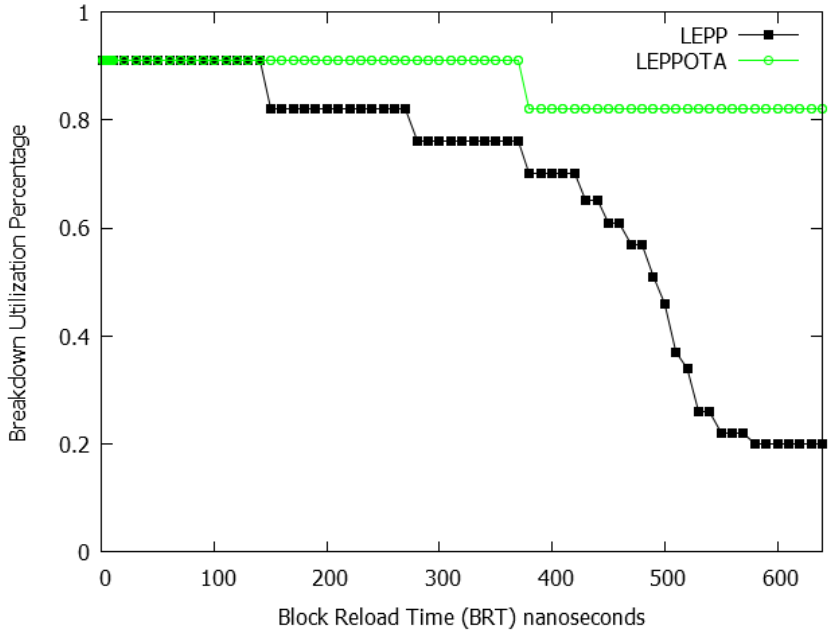

Figure 73: Breakdown Utilization Comparison.

For large block reload times, the integrated preemption placement/preemption threshold scheduling algorithm dominates the preemption placement only algorithm. This trend is also seen in Figure 74.

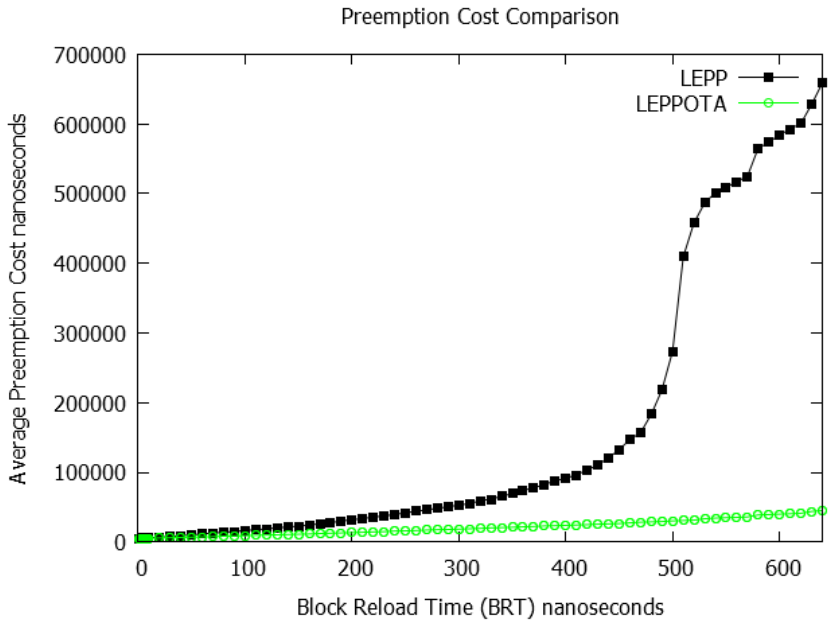

Figure 74: Average Preemption Cost Comparison.

The ability of the integrated approach to adjust the preemption thresholds to permit tasks to execute non-preemptively where possible has further benefits in reduced preemption placement costs. As expected, both methods converge as the block reload time approaches zero.

Summary

In this work, we presented an algorithm that integrates limited preemption scheduling via conditional preemption placement with preemption threshold scheduling via optimal preemption threshold assignment. Our experiments demonstrated the schedulability dominance of the integrated approach over existing preemption placement algorithms. In future work, we plan to evaluate this integrated approach on a wider range of linear and conditional real-time programs.

CHAPTER 8 FUTURE WORK

In this chapter, we discuss the portion of the research that has not been completed. As we explained in the previous chapters, our goal is to develop a generalized schedulability framework supporting the hard-real-time system design process for the limited preemption task execution model. This framework addresses taskset schedulability determination for real-time systems with physical, hardware and timing challenges. So far, we have completed a subset of our main research and proposed a new CRPD metric, called *loaded cache blocks (LCB)* which accurately characterizes the CRPD a real-time task may be subjected to due to the preemptive execution of higher priority tasks. We showed how to integrate our new LCB metric into our newly developed algorithms that automatically place preemption points supporting linear control flow graphs (CFGs) for limited preemption scheduling applications. Finally, towards the end of our defense, we complete the generalized schedulability framework that will ensure that hard-real-time tasksets will be schedulable at design time. In the next section, we discuss the remaining work to be completed.

Future Work

While significant progress has been made in preemption placement supporting limited preemption scheduling in recent years, there are several open problems that remain to be solved in order to permit limited preemption scheduling using PPP algorithms to migrate into industry applications supporting the development cyber-physical systems. A few of these open problems are summarized below.

Interdependent CRPD for Set Associative Caches with Preemption Placement

We extend our LCB metric to handle the complexity of N-way set associative caches. In this design, the chosen cache replacement policy significantly impacts the algorithms needed to compute a safe bound or the CRPD. While algorithms exist to compute CRPD for set associative caches, we will specifically address the problem of computing interdependent CRPD for the purposes of eventual preemption point placement. The challenge this work presents is to design the LCB metric in such a manner as to apply to the broadest range of cache replacement policies as possible. Cache replacement policies under consideration are the algorithms covered in Chapter 3. Cache replacement policies not fitting into the broader framework will require custom extensions to provide the requisite data needed by the preemption placement algorithm. Previous work focused on direct mapped cache designs, thereby engendering a need to extend the benefits of limited preemption scheduling models to hard-real-time systems employing N-way set associative cache designs. We will evaluate the applicability or suitability of these algorithms to the various cache replacement policies in use today. Lastly, we integrate our revised LCB metric with

preemption point placement algorithms applicable to both linear and arbitrary control flow graphs.

Real-Time Systems Computational Framework

We implement a real time systems computational framework integrating our CRPD computation algorithms with existing tools supporting WCET analysis. Subcomponents of this framework include WCET analysis, CRPD analysis, preemption point placement, and schedulability analysis. Figure 75 illustrates a high-level diagram of the various analysis steps, their input data, and output data to give an idea of what is envisioned. Blocks with a red accent denote existing tools, while blocks with a blue accent denote our contributions. WCET analysis tools analyze program structure computing the worst-case execution time

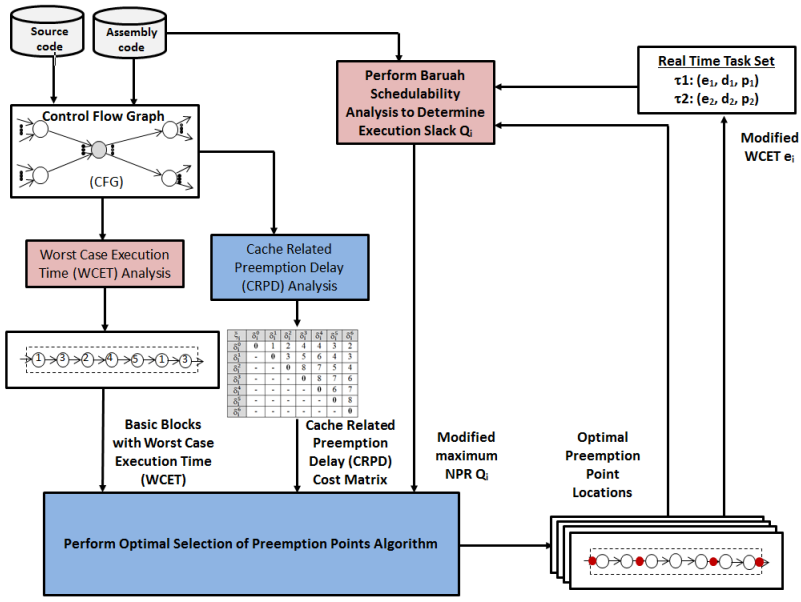

Figure 75: Real-Time Systems Computational Framework.

of each instruction or basic block. CRPD analysis tools analyze program structure computing the set of useful cache blocks thereby determining the potential impact task preemption has at each basic block. Preemption point placement algorithms utilize the computed WCET and CRPD values at each basic block to determine optimal preemption point locations with the goal of minimizing preemption overhead for hard-real-time tasksets. Schedulability analysis techniques analyze modified taskset characteristics with the CRPD overhead considered to determine the amount of available scheduling slack key to determining if a taskset is schedulable. The schedulability analysis serves as the input to preemption point placement supporting linear and arbitrary control flow graphs. The output of schedulability analysis, the allowable scheduling slack, also known as the maximum non-preemptive region Q_i, is a key parameter used in our optimal preemption point placement algorithm. Each preemption point placement solution results in new scheduling slack that affects preemption point placement during subsequent iterations. The algorithm

converges when the selected preemption points no longer change or the slack reaches some minimum threshold. In summary, our method uses an iterative approach to refine the preemption points of tasks factoring in the updated scheduling slack until the chosen preemption points or scheduling slack no longer changes. The means to making these various tools usable for the hard-real-time systems designer is the integration of the iterative schedulability analysis with interdependent CRPD based preemption point placement. In this context, our work will implement the schedulability analysis for two commonly used scheduling algorithms. These algorithms are the earliest deadline first (EDF) and fixed priority (FP). The framework will be designed in such a manner to offer the ability to customize for future scheduling algorithms supporting a flexible, open source software organization. The evaluation will demonstrate the requisite function on the Malardalen Real Time Code (MRTC) task suite [51], deemed to be of suitable size and complexity to support an initial tool feasibility assessment.

Real-Time Systems Computational Framework Case Study Evaluation

We utilize our newly developed real time systems computational framework to analyze a real time systems application of suitable size and complexity. Subcomponents of this framework include WCET analysis, CRPD analysis, preemption point placement, and schedulability analysis. The overall tool improvement focus is essentially a collection of graphical based methods for visualization and automated back annotation of downstream scheduling design decisions onto real time code, as briefly described below. Visualization of WCET analysis results would include numerical display of basic block worst-case execution time annotated in the graphical representation of the control flow graph structure. Visualization of CRPD analysis would take the form of a two-dimensional graph showing the CRPD cost assuming a given basic block is selected as a preemption point. The hard-real-time systems designer would access this information by selecting a specific edge in the control flow graph. Visual methods supporting back annotation of real time code with the selected optimal preemption points will be implemented. Within our graphical design framework, designers will have the option of selecting required preemption points to augment the preemption point placement algorithms. The inherent challenge in this work is to design the requisite information exchange between the various sub-components in such a way as to allow support for new scheduling algorithms, cache replacement policies, and new taskset characteristics. Breakdown utilization studies will illustrate for each task where the taskset schedulability breaks down so refactoring of the design can readily occur in problematic software components. The evaluation of our real time systems computational framework will focus on a case study involving analysis of a real-time application of moderate complexity deemed suitable in accordance with analysis time constraints.

Non-Inline Function Support for Preemption Placement in Real-Time Program Code

One of the most significant limitations of existing preemption placement algorithms is they only provide support for inline functions, thereby excluding non-inline functions. One of the primary motivations for working on this problem is the future commercial viability of limited preemption scheduling via interdependent CRPD cost model, by extending the supported real-time task code constructs to non-inline functions. To understand the context in which we solve for and obtain a single minimized preemption solution for each function definition each having k_j invocations in the real-time task code where j is the function definition index consider the aggregate subgraph G_i^F shown in Figure 76.

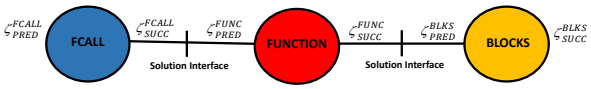

Figure 76: Function Definition Preemption Placement Problem

Each function invocation in the real-time task code consists of a function call subgraph G_i^{FCALL}, a function definition subgraph G_i^{FUNC}, and a post function execution subgraph G_i^{BLKS}. To solve this problem, we intuitively must compute the minimized function definition at the point where the preemption solutions are available for each of the k_j invocations in the subgraphs G_i^{FCALL}, G_i^{FUNC}, and G_i^{BLKS}. Using the brute force approach where all $(Q_i + 1)^2$ preemption solutions for each of the k function invocations are simultaneously iterated results in an intractably exponential time complexity $O(kN_iQ_i^{2^k})$. To visualize the complexity of computing a single preemption solution for each function definition, and to demonstrate the feasibility of solving this problem, consider the following graphical depiction as shown in Figure 77. The blue dots represent the function call G_i^{FCALL} subgraphs for each of the k invocations of

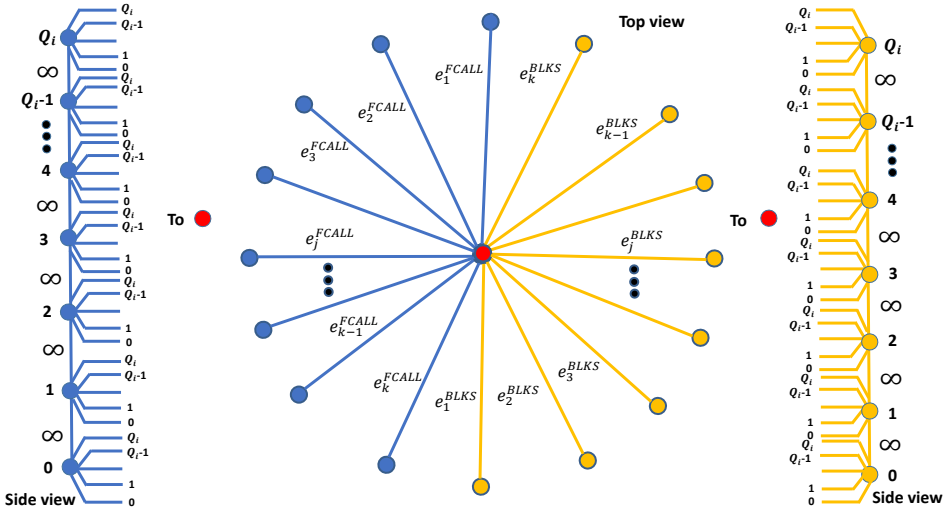

Figure 77: Compute Function Definition Preemption Problem

function f. Similarly, the yellow dots represent the post function execution G_i^{BLKS} subgraphs following each of the k invocations of function f. Lastly, the red dot denotes one of the $(Q_i + 1)^2$ preemption solutions stored for the function f definition subgraph G_i^{FUNC}. The side views depict separate vertices for each of the $(Q_i + 1)^2$ preemption solutions stored in the G_i^{FCALL} and G_i^{BLKS} subgraphs respectively. Incorporating support for non-inline functions to preemption placement algorithms will serve to extend limited preemption scheduling to a wider class of real-time program code.

Since function invocations can occur anywhere in the task code, this creates challenges in the context of the progression of in line production rules in our graph grammar $\mathcal{G}$. With the objective of a single preemption solution for each function definition subgraph G_i^{FUNC}, semantic processing of production rules must be suspended at each function invocation until the subgraph solutions for all invocations are available. To illustrate this concept, consider the following abstract parse tree shown in Figure 78.

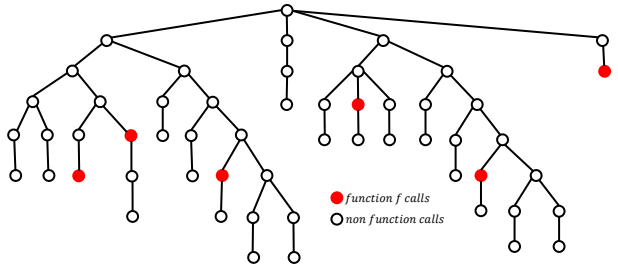

Figure 78: Parse Tree Function Invocations

Function calls are indicated by the filled red dots and the non-function calls are indicated by the hollow dots. This example demonstrates the conditions for computing a single function definition preemption solution does not coincide with the ordering of production rules. Figure 79 illustrates the parsing progress limited due to the presence of function f calls in the task code.

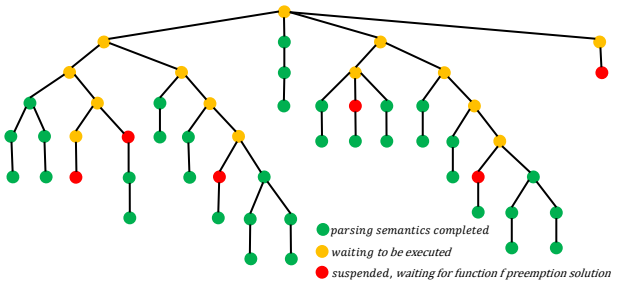

Figure 79: Function Call Parsing Progress

The green dots denote subgraphs where the preemption solutions have been computed during parsing. The red dots denote function call subgraphs where preemption solutions cannot be computed until the corresponding function definition preemption solution has been computed. The yellow dots denote higher level subgraphs whose preemption solutions are dependent on the function call preemption solutions. Once

all the calls to function f have been encountered in the parse tree, we compute the minimized function f definition preemption solution as shown in Figure 80.

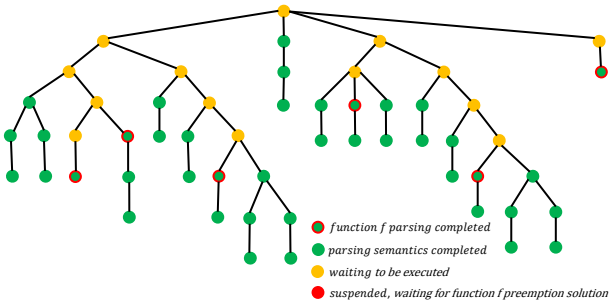

Figure 80: Function Call Preemption Solutions Computed

The green dots outlined in red denote the function call subgraphs where the preemption solutions have been computed from the single function f preemption solution. Once these preemption solutions are available, the remaining subgraph nodes whose preemption solution computations that were previously suspended may now be processed as shown in Figure 81.

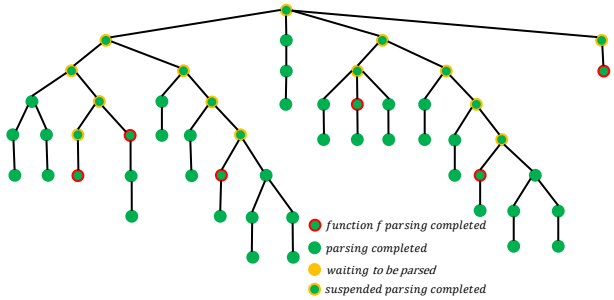

Figure 81: Graph Parsing Completed

To accomplish this modified ordering of parsing semantics, we must store suspended productions in a FIFO queue $\mathcal{Q}$ until the requisite function definition preemption solution dependencies have been resolved.

Goto Statement Support for Preemption Placement in Real-Time Program Code

Similar to the incorporation of non-inline function support in preemption placement algorithms, the addition of support for goto statements would extend preemption placement to a profoundly broader set of real-time program code. The complexity of goto statement support far exceeds that of non-inline functions since the use of goto statements adds preemption placement dependencies between program constructs in different locations. We envision the primary tenets of the proposed solution to non-inline functions to be directly applicable to goto statements. Goto statements are ubiquitous in low level assembly code due to compiler optimizations employed during the code generation phase. Because the grammar supporting existing preemption placement algorithms is largely comprised of structured programming constructs, it is necessary to semi-manually post process the generated assembly code to fit the inherent grammar limitati-

ons. This post processing step is arduous, time consuming, and error prone regardless of whether manual or automated methods are applied, thereby limiting the feasibility of preemption placement algorithms to small applications. Thus, by integrating goto statement support into preemption placement algorithms, the need for post processing of real-time program code would be eliminated.

Interdependent Probabilistic CRPD with Preemption Point Placement

We extend our LCB metric to handle the complexity of N-way set associative caches with random cache replacement policy. In this design, the random cache replacement policy requires a probabilistic approach to the analysis in order to compute a safe bound or the CRPD. In this probabilistic analysis, we will specifically address the problem of computing interdependent CRPD for the purposes of eventual preemption point placement. The challenge this work presents is the probabilistic nature the random cache replacement policy engenders is the inherent non-deterministic behavior prominently affects the methods for computing CRPD. Unlike previous methods, it is impossible to determine the exact cache contents at any program location. It will be imperative that our research offers the hard-real-time systems designer a statistical theoretical framework that will result in safe yet reasonably accurate CRPD metrics based on sound probabilistic analysis methods. Lastly, we integrate our revised LCB metric with preemption point placement algorithms applicable to both linear and arbitrary control flow graphs.

Real Time Systems Computational Framework

Our future objective is to make the capabilities of our real time systems computational framework prototype available to companies developing cyber-physical systems. To accomplish this, it will be imperative that a robust, commercial tool implementation with suitable customer support must exist moving forward. Opportunities to partner with existing WCET analysis tool vendors with the goal of incorporating our research into their tools will be realized.

Multiprocessor Systems

Multiprocessor or multi-core CPUs are being ubiquitously used in cyber-physical systems to meet the rapidly increasing computational complexity witnessed in many application domains today. Real-time systems research pertaining to multiprocessor systems is highly active with the schedulability challenges posed by task execution concurrency. Opportunities to extend our CRPD metric into a multiprocessor concurrent execution environment will be explored.

Multi-Level Cache Analysis

Multi-level cache analysis addresses the complexity introduced by cache memory designs containing more than one or multiple levels. Multi-level cache designs are primarily used in multiprocessor or multi-

core CPUs to economically improve cache hit performance. Multi-level caches may employ disparate inclusion properties that determine how cache block evictions at lower cache levels affect upper cache levels. Opportunities to extend our CRPD metric into multi-level cache designs will be explored, proposed, and validated.

Summary

Hard-real-time systems frequently subjected to various physical, hardware and timing constraints are a growing research area. There are many emerging and growing career opportunities in cyber-physical systems (CPS) design that require a broad understanding of a diverse set of real-time system concepts and disciplines.

CHAPTER 9 CONCLUSION

In this dissertation, we introduced a new CRPD metric, called *loaded cache blocks (LCB)* which accurately characterizes the CRPD a real-time task may be subjected to due to the preemptive execution of higher priority tasks. We showed how to integrate our new LCB metric into our newly developed algorithms that automatically place preemption points supporting linear and conditional CFGs for limited preemption scheduling applications. Essentially, this framework addresses taskset schedulability determination for real-time systems with physical, hardware, and timing challenges. Furthermore, our enhanced algorithm was demonstrated to be optimal for linear CFGs in that if a feasible taskset schedule cannot be constructed, then no feasible schedule exists by any known method utilizing a static Q_i value. For conditional CFGs, our proposed PPP algorithm was suboptimal due to the method of handling of preemption costs between successive block structures. Nonetheless, our conditional PPP algorithm demonstrated superior performance as compared to existing schedulability approaches. Finally, our design and analysis framework may be classified as *limited preemption point placement and schedulability analysis* for hard-real-time systems.

As a proof of concept, we demonstrate the effectiveness of our approach via experiments using tasksets from the MRTC WCET benchmark suite [51]. We show how to solve some of the issues and challenges of designing predictable real-time systems that must guarantee hard deadlines. In our framework, the system designer specifies the required deadlines that hard-real-time taskset must meet where the system automatically adjusts the taskset preemption points to ensure taskset schedulability.

The impact of our work to the hard-real-time systems development community is to facilitate the migration of limited preemption scheduling from academia to mainstream cyber-physical systems applications in industry. With respect to minimizing preemption overhead, limited preemption scheduling has been shown to exhibit superior performance as compared to fully preemptive and non-preemptive scheduling in hard-real-time tasksets. The benefit to cyber-physical systems (CPS) is improved utilization of hardware resources via reduced overhead enabled by increasing the accuracy of CRPD computations and automating optimal preemption point placement. Furthermore, the future integration of WCET analysis, CRPD analysis, schedulability analysis, and preemption point placement into a single unified real-time systems computational framework will serve to reduce the overall system schedulability design time allowing real-time system designers to focus on the more critical and important tasks of ensuring correct and robust system operation. As new product features are added, the integrated real-time systems schedulability tools will permit system designers to quickly redo the previous analysis to ensure the system remains schedulable. Therefore, these improvements in CPU utilization and reduced design time will result in substantial

cost reductions in CPS products thereby benefiting both system producers and consumers alike.

CHAPTER 10 LIST OF PUBLICATIONS

JOURNAL

Under Review

(a) John Cavicchio and Nathan Fisher. Realizing Improved Preemption Placement in Real-Time Program Code with Interdependent Cache Related Preemption Delay, *Real-Time Systems Journal* (Status: Submitted).

CONFERENCE & WORKSHOP

Published

(a) Brandon Szeliga, John Cavicchio, and Weisong Shi. DIMM: Distributed Metadata Management for Data-Intensive HPC Environments, *Proc. of the IEEE Workshop on Data-Aware Distributed Computing HPDC Symp.*, Boston, 2008.

(b) John Cavicchio, Nathan Fisher, and Corey Tessler. Minimizing Cache Overhead via Loaded Cache Blocks and Preemption Placement, *Proc. of the Euromicro Conference on Real-Time Systems Symp.*, Sweden, 2015.

IN PREPARATION

1. John Cavicchio and Nathan Fisher. Integrating Preemption Thresholds with Limited Preemption Scheduling, will submit to *Euromicro Conference on Real-Time Systems Symp.*, 2019.

REFERENCES

[1] AbsInt. AbsInt a^3 WCET Tool, 2018.

[2] T. Acharya and M. Ladlow. Cache replacement algorithms in hardware. *Technical Report Swarthmore College, May 2008*.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools Second Edition*. 2007.

[4] A.V. Aho, P.J. Denning, and J.D. Ullman. Principles of optimal page replacement. *J. ACM*, 18(1):80–93, 1971.

[5] S. Altmeyer and C. Burguiere. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture (JSA), Elsevier*, 2011.

[6] S. Altmeyer, R. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *In RTSS*, 2011.

[7] S. Altmeyer, R. Davis, and C. Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. *Real Time Systems, Springer*, 2012.

[8] S. Bansal and D.S. Modha. CAR: Clock with adaptive replacement. *In USENIX Conference on File and Storage Technologies (FAST 04), San Francisco, CA, March 31-April 2, 2004*.

[9] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. *In ECRTS*, 2005.

[10] S. Baruah and J. Goossens. Scheduling real-time tasks: Algorithms and complexity. In Joseph Y.-T Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press LLC, 2003.

[11] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Sys. J.*, 5(2):78–101, 1966.

[12] M. Bertogna and S. Baruah. Limited preemption EDF scheduling of sporadic task systems. *In RTSS*, 2011.

[13] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. *In ECRTS*, 2010.

[14] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. Optimal selection of preemption points to minimize preemption overhead. *In ECRTS*, 2011.

[15] E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *In IEEE Transactions on Computing*, 53(11):1462–1473, 2004.

[16] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, 2005.

[17] C. Bohm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, May 1966.

[18] R. Bril, S. Altmeyer, M. van den Heuvel, R. Davis, and M. Behnam. Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: integrated analysis and evaluation. *Real-Time Systems*, 53(4):403–466, Jul 2017.

[19] R. Bril, S. Altmeyer, M. van den Heuvel, R. Davis, and M. Benham. Integrating cache-related pre-emption delays into analysis of fixed priority scheduling with preemption thresholds. *In RTSS*, 2014.

[20] A. Burns. *Preemptive priority-based scheduling: an appropriate engineering approach*. 1995.

[21] G. Buttazzo. *Hard Real Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 2005.

[22] G. C. Buttazzo, M. Bertogna, , and G. Yao. Limited preemptive scheduling for real-time systems: A survey. *IEEE Transactions on Industrial Informatics*, 9(1), February 2013.

[23] J. Cavicchio and N. Fisher. Paper Programs and Data Repository, 2018.

[24] J. Cavicchio, C. Tessler, and N. Fisher. Paper Programs and Data Repository, 2014.

[25] J. Cavicchio, C. Tessler, and N. Fisher. Minimizing cache overhead via loaded cache blocks and preemption placement. *In ECRTS*, 2015.

[26] J.E.G. Coffman and P.J. Denning. *Operating Systems Theory*. 1973.

[27] F.J. Corbato. A paging experiment with the multics system. *MIT Project MAC Report MAC-M-384, May 1968*.

[28] S. Dar, M.J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. *In Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1996.

[29] C. Ding and Y. Zhong. Predicting whole-program locality through reuse-distance analysis. *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June, 2003*.

[30] Gaisler. Gaisler Bare-C Cross Compiler (BCC), 2018.

[31] Gaisler. GRSIM, 2018.

[32] G. Goud, N. Sharma, K. Ramamritham, and S. Malewar. Efficient real-time support for automotive applications: A case study. *Proc. of the Embedded and Real-Time Computing Systems and Applications*, 2006.

[33] J. Handy. *The Cache Memory Book: The Authoritative Reference on Cache Design, Second Edition*.

1998.

[34] D. Hardy, B. Rouxel, and I. Puaut. The heptane static worst-case execution time estimation tool. *In WCET*, 2017.

[35] S. Jiang, X. Zhang, , and F. Chen. Clock-pro: An effective improvement of the clock replacement. *ATEC Proceedings of the Annual Conference on USENIX Anuual Tecchnical Conference, pp.35, 2005*.

[36] S. Jiang and X. Zhang. LIRS: An efficient low interreference recency set replacement policy to improve buffer cache performance. *In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2002.

[37] K. Kennedy and L. Zucconi. Applications of a graph grammar for program control flow analysis. *In Proceedings ACM SIGACT/SIGPLAN Symposium on Principles of programming languages*, 1977.

[38] H. Kikuchi, R.K. Kalia, A. Nakano, P. Vashishta, F. Shimojo, and S. Saini. Scalability of a low-cost multi-teraflop linux cluster for high-end classical atomistic and quantum mechanical simulations. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 66.2, Washington, DC, USA, 2003. IEEE Computer Society.

[39] C.-G. Lee, J. Hahn, S.L. Min, R. Ha, S. Hong, C.Y. Park, M. Lee, and C.S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[40] D. Lee, J. Choi, J.-H. Kim, S.H. Noh, S.L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS Performance Evaluation Review*, volume 27, pages 134–143. ACM, 1999.

[41] D. Lee, J. Choi, J.-H. Kim, S.H. Noh, S.L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.

[42] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. *In Proceedings of the Real-Time Systems Symposium - 1989, Santa Monica, California, USA, IEEE Computer Society Press*, pages 166–171, December 1989.

[43] 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.

[44] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, 2000.

[45] Advantech Co. Ltd. Advancech product hardware. http://www.advantech.com/products/, September

2013.

[46] Analog Devices Co. Ltd. Linux industrial input-output subsystems. http://wiki.analog.com/software/linux/docs/iio/iio, July 2012.

[47] W. Lunniss, S. Altmeyer, C. Maiza, and R. Davis. Integrating cache related pre-emption delay analysis into EDF scheduling. *In RTAS*, 2013.

[48] J. Manuel Marinho, V. Nélis, S. M. Petters, and I. Puaut. Preemption delay analysis for floating non-preemptive region scheduling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 497–502. IEEE, 2012.

[49] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Sys. J.*, 9(2):78–117, 1970.

[50] N. Megiddo and D.S. Modha. ARC: A self-tuning, low overhead replacement cache. *In Proceedings of the USENIX File and Storage Technologies (FAST)*, 2003.

[51] MRTC. Malardalen WCET research group MRTC benchmarks, 2018.

[52] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache related preemption delay. *In CODES*, 2003.

[53] E.J. O'neil, P.E. O'neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *In ACM SIGMOD*, 22(2):297–306, 1993.

[54] E.J. O'neil, P.E. O'neil, and G. Weikum. An optimality proof of the LRU-K page replacement algorithm. *Journal of the ACM (JACM)*, 46(1):92–112, 1999.

[55] R. Pellizzoni, E. Betti, S. Bak, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. *In RTAS*, 2011.

[56] R. Pellizzoni, B.D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. *In RTSS*, 2008.

[57] R. Pellizzoni and M. Caccamo. Toward the predictable integration of real-time COTS-based systems. *In RTSS*, 2007.

[58] B. Peng, N. Fisher, and M. Bertogna. Explicit preemption placement for real-time conditional code. *In ECRTS*, 2014.

[59] P. Puschner and Ch. Koza. Calculating the maximum execution time of real time programs. *In Real-Time Systems*, 1989.

[60] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. *In RTAS*, 2006.

[61] J.T. Robinson and M.V. Devarakonda. *Data cache management using frequency-based replacement*,

volume 18. ACM, 1990.

[62] J. Sifakis. Modeling real-time systems - challenges and work directions. In *In Proceedings of the 1st International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science*, pages 373–389. Springer Verlag, 2001.

[63] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating Systems Concepts (Seventh Edition)*. 2005.

[64] J. Simonson and J.H. Patel. Use of preferred preemption points in cache-based real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 316–325. IEEE, 1995.

[65] A.J. Smith. Design of CPU cache memories. *In Proceedings of IEEE Region 10 Conference (TENCON)*, 1987.

[66] J. Staschulat and R. Ernst. Scalable precision cache analysis for real-time software. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4), September 2005.

[67] Y. Tan and V. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. *In SCOPES*, 2004.

[68] A.S. Tanenbaum and A.S. Woodhull. *Operating Systems, Design and Implementation*. 1997.

[69] H. Tomiyamay and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. *In CODES*, 2000.

[70] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. *In the International Conference on Real Time Computing Systems and Applications*, 1999.

[71] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36:1–36:53, May 2008.

[72] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. *In RTCSA*, 2009.

ABSTRACT

EFFECTIVE AND EFFICIENT PREEMPTION PLACEMENT FOR CACHE OVERHEAD MINIMIZATION IN HARD REAL-TIME SYSTEMS

by

JOHN C. CAVICCHIO

May 2019

Advisor: Dr. Nathan Fisher

Major: Computer Science

Degree: Doctor of Philosophy

Schedulability analysis for real-time systems has been the subject of prominent research over the past several decades. One of the key foundations of schedulability analysis is an accurate worst case execution time (WCET) for each task. In preemption based real-time systems, the CRPD can represent a significant component (up to 44% as documented in research literature) of variability to overall task WCET. Several methods have been employed to calculate CRPD with significant levels of pessimism that may result in a task set erroneously declared as non-schedulable. Furthermore, they do not take into account that CRPD cost is inherently a function of where preemptions actually occur. Our approach for computing CRPD via *loaded cache blocks* (LCBs) is more accurate in the sense that cache state reflects which cache blocks and the specific program locations where they are reloaded.

Limited preemption models attempt to minimize preemption overhead (CRPD) by reducing the number of allowed preemptions and/or allowing preemption at program locations where the CRPD effect is minimized. These algorithms rely heavily on accurate CRPD measurements or estimation models in order to identify an optimal set of preemption points. Our approach improves the effectiveness of limited optimal preemption point placement algorithms by calculating the LCBs for each pair of adjacent preemptions to more accurately model task WCET and maximize schedulability as compared to existing preemption point placement approaches. We utilize dynamic programming technique to develop an optimal preemption point placement algorithm. Lastly, we will demonstrate, using a case study, improved task set schedulability and optimal preemption point placement via our new LCB characterization.

We propose a new CRPD metric, called *loaded cache blocks (LCB)* which accurately characterizes the CRPD a real-time task may be subjected to due to the preemptive execution of higher priority tasks. We show how to integrate our new LCB metric into our newly developed algorithms that automatically place preemption points supporting linear control flow graphs (CFGs) for limited preemption scheduling

applications.

We extend the derivation of *loaded cache blocks (LCB)*, that was proposed for linear control flow graphs (CFGs) to arbitrary CFGs. We show how to integrate our revised LCB metric into our newly developed algorithms that automatically place preemption points supporting arbitrary control flow graphs (CFGs) for limited preemption scheduling applications.

For future work, we will verify the correctness of our framework through other measurable physical and hardware constraints. Also, we plan to complete our work on developing a generalized framework that can be seamlessly integrated into real-time schedulability analysis.

AUTOBIOGRAPHICAL STATEMENT

JOHN C. CAVICCHIO

John Cavicchio received his Master's degree in Computer Science in 2013 from Wayne State University, Detroit, Michigan USA. He previously received a Master's degree in Electrical Engineering from the University of Michigan and a Bachelor's degree in Computer Science from Michigan State University. He currently works for Harman International as a Principal Lead Software Engineer in audio embedded systems and plans on continuing his research in real-time systems. His research interests include real-time systems, scheduling theory, distributed systems, algorithm design and analysis, embedded systems, worst-case execution time (WCET) analysis, cache analysis, cache related preemption delay (CRPD) analysis, limited preemption scheduling, and real-time system architecture.