BUNDLE: TAMING THE CACHE AND IMPROVING SCHEDULABILITY OF MULTI-THREADED HARD REAL-TIME SYSTEMS

by

COREY TESSLER

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. Advisor Date

_____

Frank Mueller, Ph.D.

_____

Daniel Grosu, Ph.D.

_____

Abusayeed Saifullah, Ph.D.

ACKNOWLEDGMENTS

This research presented was supported, in part, by the National Science Foundation under Grant Numbers CNS-1618185, CNS-0953585, CNS-1205338, and IIS-1724227.

This research would not have been possible without the clear, dedicated, and gentle guidance provided by Professor Nathan Fisher of Wayne State University. He is a treasure to the University and to all who know him. Thank you.

To the committee members: Professor Daniel Grosu, Professor Frank Mueller, and Professor Abusayeed Saifullah. Please accept my gratitude for your willingness to sit on the committee along with your thoughtful feedback and suggestions. This work is made better by your contributions, thank you.

I am fortunate to have a wonderful role-model for a mother, Anastasia Tessler. Her lifelong dedication to her children extends beyond her biological children to those she adopted in the classroom. As a teacher in Detroit, her thoughtful creativity excited students to excel academically and her compassion forged life-long relationships with them and their families. She stands as an example of service to others through education to all of us. Thank you for your unwavering support and all of the lessons.

And lastly, thank you to two very important friends: Michael Noonan and Kathryn Horner. They are wonderful, unique people with generous spirits. May you rely upon me as I have relied upon you for companionship, distraction, and wholly inappropriate humor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

Computational systems and their applications have become ubiquitous. From racks of super-computers, to smart phones, to pin sized personal sensors, there are a scant few environments where a microprocessor is absent. As the use of computational systems increases, so too does our reliance and trust in them to operate in safety critical environments such as nuclear power plants, pace-makers, automotive and flight controls. When accuracy and safety depend on a computational system, we find hard real-time systems.

As a discipline, computer science provides the theoretical and practical tools necessary to guarantee the safety of hard real-time systems. However, the features of the underlying computational platforms (the architecture) and the programming models applied to them are constantly evolving – so too must the analysis of safety critical systems. Herein, the impact of cache memory and threaded execution is examined in the context of hard real-time systems. The *classical* perspective of threads and cache is advanced to an *integrated* one, resulting in safe and reliable systems that execute upon smaller and less powerful platforms. Additionally, the integrated perspective could be applied outside of the hard real-time setting they were developed for to improve the performance of threaded computational systems.

Currently, the evolution of computational platforms is focused upon increasing the number of concurrent threads of execution by providing processors with an increased number of cores and cache dedicated to those cores. As an example, the AMD Threadripper 2990WX [3] processor can execute sixty four threads simultaneously. It contains

over eighty megabytes of cache memory, with three megabytes of private cache dedicated to individual cores.

Simultaneously, the scope of safety critical applications is increasing to encompass applications that cannot be addressed by single-threaded processors. NVIDIA's Jetson TX2 plaftform [4] is designed for autonomous vehicles. It carries 256 cores to handle the demanding multi-threaded tasks associated with image processing, route planning, and motor control. Like autonomous vehicles, it is reasonable to expect the number of hard real-time systems which demand multi-threaded platforms to increase. Improving the performance of these platforms through the proposed integrated perspective increases the efficiency of these systems, thus reducing the number of processors needed, power consumed, weight, and overall cost.

In a hard real-time system, as with other systems, all computations must produce the correct answer; they must be logically correct. However, hard real-time systems have a second requirement: all computations must complete on time. A correct answer delivered too late, such as "apply emergency braking" is not only useless, it is a failure of the system with catastrophic results. Similarly, a correct answer delivered too early is also considered a failure. For a hard real-time system there are two types of correctness: logical, and temporal. Both must be met for any hard real-time system to be reliable, safe, and deployed.

Logical correctness is determined by context, depending on the inputs and parameters of the system deployed. Temporal correctness is the subject of study for *schedulability analysis*. It is uncommon for any computational system to have a single logical operation to complete, this is also true for hard real-time systems. For hard real-time systems logical

operations are divided into tasks (individual programs), which execute on the shared processor(s). Tasks compete for execution time on the processor(s) and if the competition is too great the temporal correctness of the system is at risk.

Schedulability analysis determines if a set of tasks will always be temporally correct for a given computational architecture. Each task has a frequency with which it will be requested, a window of time which it must be completed within called its deadline, and a worst-case execution time (WCET). The WCET of a task bounds the amount of time a task takes to complete on the shared processors. The limitations of the architecture are included in a *schedulability test* including the processor(s). When a schedulability test determines every request to execute a task will be temporally correct using a specific *scheduling algorithm*, the task set is said to be *schedulable*. A schedulable task set guarantees a system is safe to deploy for its stated operation (given that it is logically correct).

An important component to schedulability analysis is the calculation of worst-case execution times for tasks. For classical models of real-time systems, shared resources are often considered detractors to schedulability analysis and exclusively increase WCETs. Cache memory is one such shared resource viewed from this exclusively negative perspective. It is a natural perspective, derived from a preempting task invalidating cache lines, thus extending a preempted task's execution time.

Using the classical periodic task model [5] as an example, it is implied that a task has a single thread of execution. The model lacks a representation for tasks with multiple threads. To apply WCET and schedulablility techniques developed for the periodic and other classical models, a task that executes multiple threads is treated as several duplicate tasks with a single thread of execution. These single threaded tasks duplicated from multi-

threaded tasks are referred to as *synthetic tasks*. Any multi-threaded task that releases a job with m threads will be converted to m synthetic tasks each releasing one job.

Figure 1.1 serves as an example of converting a multi-threaded task set to a single-threaded task set. The complete task set is $\tau = \{\tau_1, \tau_2, \tau_3\}$. Threads of tasks are represented by small black squares. For τ_1, there is one thread of execution, for τ_2 three threads, and τ_3 two threads. The system designer's perspective is given on the left side of the figure, where the three tasks encapsulate their threads. On the right side of the figure is the analytical perspective, where tasks must have exactly one thread of execution. Worst-case execution time and schedulability analysis is performed on the six (rather than three) tasks.

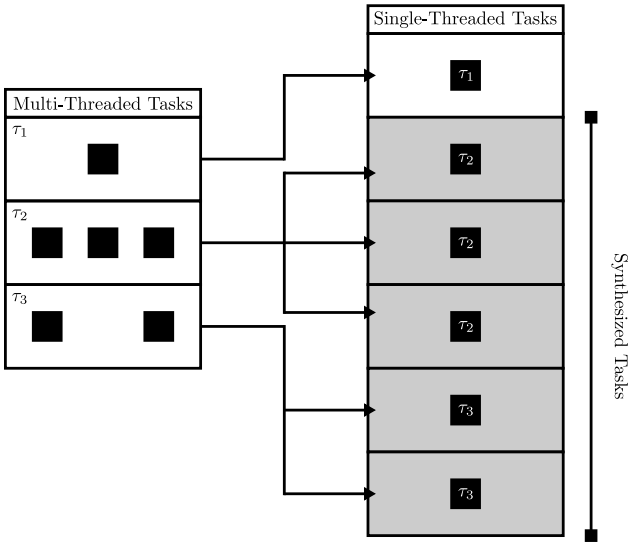


Figure 1.1: Synthesized Tasks

From the analytical perspective, the synthetic tasks are *independent* of one another competing for the shared resources of target architecture. One classical analytical model where threads are treated independently is the fork-join model [6]. Where each thread has a WCET calculated from its longest execution path. Each thread then contributes its execution demand independently of others to the schedulability test.

For classical models, tasks are assumed to be in competition for cache space. The inclusion of threads, which are converted to tasks, only amplifies the negative affect. Cache-related preemption delay analysis (CRPD), as the name implies, is the delay of a tasks completion time due to preemptions by other tasks. These delays impact schedulability

negatively by increasing worst-case execution times on a per task basis.

Figure 1.2 highlights the increase in execution demand the classical perspective necessitates. Since *all* tasks compete for cache space, all tasks must be considered in calculated CRPD values. This includes competition between synthetic tasks which were duplicated from the same multi-threaded task; eg. two threads of τ_3 compete for cache space when converted to synthesized tasks.

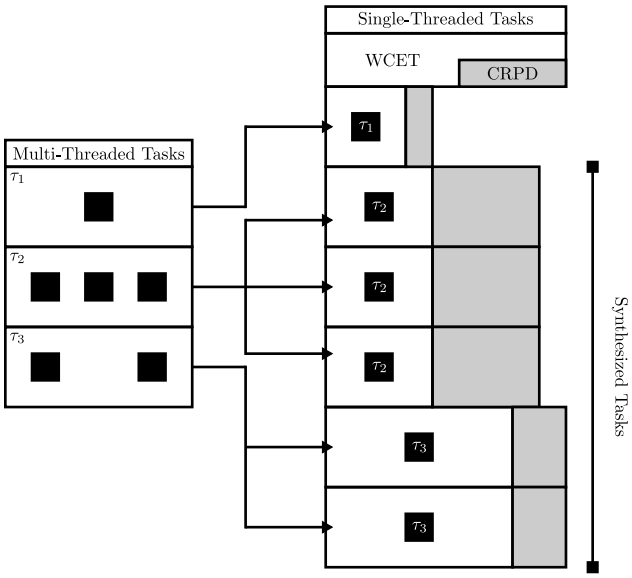


Figure 1.2: CRPD and Synthesized Tasks

Threads are not always in competition with other threads for cache space. In fact threads may mutually benefit from reusing the same cached values by virtue of sharing the same memory space. A cache miss during the execution of one thread can place values into the cache that produce a cache hit for a second thread. These unexpected cache hits reduce the execution time of the second thread and the system overall. This speed up is called the *inter-thread cache benefit*. The efforts of this work focus on quantifying the inter-thread cache benefit to decrease individual task WCET bounds and increase system schedulability.

Herein an argument is made for a new task model, scheduling algorithm, and schedulability analysis techniques. Classical approaches to Worst Case Execution Time (WCET), Cache-Related Preemption Delay (CRPD), and schedulability analysis typically produce separate values. Accounting for the inter-thread cache benefit requires an approach that

integrates the disciplines.

1.1 Thesis

> Scheduling individual threads of a multi-threaded task in a cache cognizant manner improves system schedulability through predictable and quantifiable inter-thread cache benefits. When compared to classical scheduling algorithms and analysis, this positive perspective reduces WCET and CRPD values. Realizing the benefit is achievable with the addition of a familiar (yet novel) low cost hardware mechanism.

1.2 Contributions

In support of the thesis, the following contributions are made. As an initial theoretical work, BUNDLE [7] describes the negativity of the classical perspective. In BUNDLE a positive perspective of caches is introduced along with central mechanisms for scheduling and worst-case execution time calculation.

1. A positive perspective on caches in the form of the inter-thread cache benefit. (Section 3.1)
2. A novel model of multi-threaded tasks that allows the inter-thread cache benefit of instruction caches to be quantified. (Chapter 2)
3. The introduction of the concepts of worst case execution time and cache overhead (WCETO), conflict free regions, and conflict free region graphs. (Section 5.2)
4. The BUNDLE cache cognizant scheduling algorithm for a single task with multiple threads. (Section 5.1)
5. A WCETO method for a task scheduled by BUNDLE. (Section 5.2.2)

Figure 1.3: Contributions of BUNDLE

Improving upon BUNDLE is the purpose of BUNDLEP [8]. Prioritizing bundles based upon their longest path maximizes the inter-thread cache benefit between threads. Priorities also improve the worst-case execution time calculation method by reducing the complexity of their calculation.

1. The BUNDLEP cache cognizant scheduling algorithm for a single task with multiple threads. (Section 6.4.2)
2. A WCETO method for a task scheduled by BUNDLEP. (Section 6.5)
3. Proof of optimal cache sharing under BUNDLEP scheduling.
4. A novel hardware interrupt mechanism to anticipate execution named XFLICT which uses an XFLICT_TABLE of addresses. (Section 6.4.1)
5. A toolset for BUNDLEP analysis and simulation for programs compiled for MIPS processors [9].

Figure 1.4: Contributions of BUNDLEP

Both BUNDLE and BUNDLEP are limited to the analysis and execution of a single multi-threaded task. Non-preemptive multi-task BUNDLE [10] (NPM-BUNDLE), expands the applicability of the scheduling and analysis techniques to multiple tasks.

1. A hierarchical scheduling mechanism using non-preemptive EDF for jobs scheduled by BUNDLEP with intra-task thread-level preemptions named Non-Preemptive Multi-Task BUNDLE (NPM-BUNDLE). (Chapter 7)
2. The introduction of task division for multi-threaded task sets. (Section 7.1.1)
3. A scheduling algorithm and task dividing process named Threads per Job (TPJ) for NPM-BUNDLE. (Section 7.2.3)
4. Proof of TPJ's non-preemptive multi-threaded feasibility. (Theorem 7.2.3)
5. A toolset for NPM-BUNDLE analysis of synthetic tasks [11].

Figure 1.5: Contributions of NPM-BUNDLE

NPM-BUNDLE brings the BUNDLE techniques and analysis to multiple tasks. However, the application is limited to a single processor. As a first step toward expanding to a multiprocessor setting, the following contributions are made as part of ITCB-DAG:

1. Augmenting the parallel directed acyclic graph (DAG) model to include executable objects and threads per node. (Section 8.1.2)
2. The concepts of collapsing nodes and candidacy for collapse. (Section 8.2)
3. The Dedicated Core Reduction Algorithm which increases schedulability by (potentially) allocating fewer cores per high-utilization task. (Section 8.3)
4. Two heuristics for ordering nodes to be collapsed. (Section 8.4)
5. A synthetic evaluation demonstrating the positive impact of collapse and BUNDLE scheduling for DAG tasks. (Section 8.6)

Figure 1.6: Contributions of ITCB-DAG

In series, these contributions can be viewed as increasing the scope of the inter-thread cache benefit. Starting with BUNDLE, which introduces the inter-thread cache benefit, scoped to a single task running on a uniprocessor systems. BUNLDEP improves upon BUNDLE in the uniprocessor setting, but does not increase the scope. The subsequent work NPM-BUNDLE increases the scope, bringing the inter-thread cache benefit to multi-task systems. Finally, ITCB-DAG further increases the scope by the inclusion of multiprocessor systems.

Figure 1.7 summarizes the scope of contributions. However, these contributions should not be considered final in their respective scopes. While these contributions are substantial, they do not address every theoretical or practical opportunity to incorporate inter-thread cache benefits in all settings; they are necessary fundamental steps towards greater adoption. There are numerous opportunities to improve upon the BUNDLE analysis and scheduling techniques.

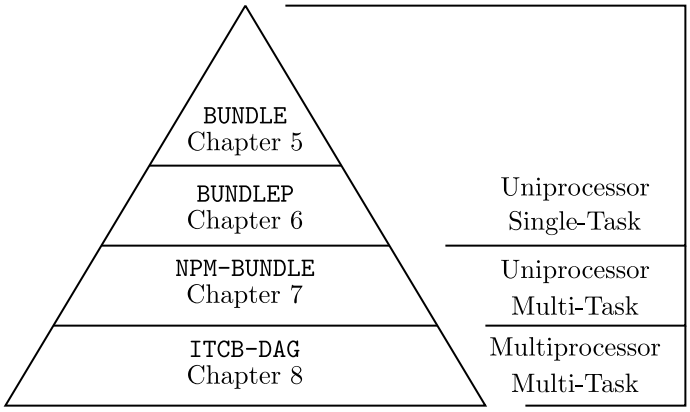


Figure 1.7: Scope of Contributions

These contributions are novel in the context of hard real-time systems, creating a positive perspective of caches for multi-threaded tasks. Previous works applicable to the worst-case execution time of single-threaded tasks [2, 12, 1] take a positive perspective on cache memory, reducing execution time bounds. In contrast, cache-related preemption delay [13, 14, 15, 16, 17] takes a negative perspective of cache memory, accounting for the execution time penalty preemptions induce in both single and multi-threaded tasks (through synthesis). These contributions are the *first* to bring a positive perspective of cache memory to multi-threaded tasks for hard real-time systems, with the goal of reducing the total system execution time. Table 1.1 places these contributions under the BUNDLE umbrella in comparison to the disciplines of WCET and CRPD analysis.

	Single Threaded	**Multi-Threaded**
Positive	WCET	BUNDLE
Negative	CRPD	CRPD

Table 1.1: Cache Perspectives in Hard-Real Time Analysis

CHAPTER 2 MODEL AND NOTATION

In this chapter, an introduction is given for the models and concepts used or augmented to obtain a positive perspective on caches. It defines the task model, schedulability conditions, program model, architecture model, and notation shared between the different BUNDLE scheduling algorithms and analysis. Additionally, the shared concepts of ribbons, threads, inter-thread cache benefit, and control flow graphs are defined.

2.1 Models and Perspectives

Throughout this work, there are references to *classical* models and the *negative* perspective. To clarify, an existing task model, scheduling algorithm, schedulability test, or WCET calculation method that cannot account for the inter-thread cache benefit is said to take the negative perspective, or is a classical model. The language is derived from the treatment of cache as a shared resource which can only extend execution times through conflicts.

In contrast, the *positive* perspective allows for caches to benefit execution. A task model, scheduling algorithm, schedulability test, or WCET method that includes the inter-thread cache benefit is termed *integrated*. The integrated methods proposed in this work which take the positive perspective are placed in the BUNDLE family (referring to common thread-level scheduling technique).

Typically, classical models assume a single thread of execution per task. Therefor, analysis of multi-threaded tasks for a classical model depends on each thread being converted to an independent task with one thread of execution. Such tasks are referred to as *synthetic*

tasks. For one multi-threaded task with m threads, m synthetic tasks (with one thread of execution) will be included in the model.

2.2 Sporadic Task Model

The sporadic task model [18] is used as a representative of the classical perspective and as the basis for modification to suit BUNDLE's task model. The set of tasks n is represented by the symbol $\tau = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$. A *task* is a computation that performs a specific function by executing on a processor. A *job* released by a task is a request to perform the computation within a specific time frame. Each task i is an ordered triple of minimum inter-arrival time p, relative deadline d, and worst case execution time c, $\tau_i = (p_i, d_i, c_i)$. The minimum inter-arrival time of a task is the fewest number of processor cycles between job releases, which will take no more than the cycles given by the worst case execution time to complete. Jobs are indexed by their release k for a task i, $J_{i,k}$ with an absolute release time of $R_{i,k}$. Each job also has an absolute deadline calculated from its release time and the task's relative deadline $D_{i,k} = R_{i,k} + d_i$. If the job does not complete its execution before the absolute deadline it is called a deadline miss.

A *scheduling algorithm* selects which job will execute on a processor at any moment. Scheduling algorithms make their decisions online, or offline. Online algorithms make scheduling decisions while the system is executing jobs, offline algorithms predetermine job and processor assignments. Jobs are given a static or dynamic priority depending on the algorithm which influences which job will be scheduled.

A *schedulability test* determines if all jobs that potentially released by τ will always meet their deadlines when scheduled by an algorithm A. If all potential job releases will meet

their deadlines the task set is deemed *schedulable*. A test is said to be *sufficient* if a task set is schedulable by A satisfies the test. However, if the test is not satisfied the task set may still be schedulable. A test is said to be *necessary* if all schedulable task sets satisfy the test.

2.3 Architecture Model

This work is focused on a single processor with a single level direct-mapped cache, in Chapter 8 the scope is expanded to multiple processors. Instructions and data are loaded from main memory into the cache before they are used by the processor. The smallest unit of storage for main and cache memory is a *block*. A block holds one or more *words* with a size expressed in bytes.

When a block is moved from main memory to cache the number of cycles required to perform the operation is called the block reload time, abbreviated BRT and represented by the symbol $\mathbb{B}$. Regardless of the instruction type, all instructions take the same number of cycles per instruction (CPI) to complete, represented by the symbol $\mathbb{I}$. Execution of an instruction with values exclusively found in the cache is referred to as a *cache hit* consuming $\mathbb{I}$ cycles. If a value is not found in the cache, it is called a *cache miss* incurring the cost of a BRT before execution, taking $\mathbb{B} + \mathbb{I}$ cycles.

Generally, caches may have multiple levels and replacement policies. This work is limited to single level direct-mapped caches where each block of main memory maps to exactly one block in the cache. The size of the cache is given by the number of blocks s. Typically, cache memory is segregated by purpose: one cache for instructions and another cache for data. This work applies only to the more predictable instruction cache.

Cache memory is typically smaller (and faster) than main memory. To be able to cache

any value from main memory a mapping between the two is needed. For a direct-mapped cache [19], each block of main memory is mapped to exactly one block in the cache. For a given program address a in main memory, the block of main memory a belongs to is denoted $\hat{M}(a)$. The cache block that a belongs to and $\hat{M}(a)$ maps to is given by $M(a)$.

2.4 Objects, Tasks, Threads, Ribbons, Entry Points

Programs take many forms, from scripting languages, compiled programs, assembly, and machine instructions. An *executable object* or object, are the machine encoded instructions that execute upon the processor. Every instruction has an absolute address in main memory denoted a. An object of a task is loaded into main memory as part of releasing a job. Instructions of an object may access, by in memory address, any other instruction or data value of the job. The combined set of reachable addresses for a job is referred to as the *memory space* of the job or task. For simplicity, we assume all jobs of the same task are loaded into the same absolute location in main memory with the same memory space for each release.

Tasks in the sporadic model implicitly represent the object, and memory space of each execution request; a job. Jobs also have an implicit entry point, a single instruction from which execution may begin. In contrast, a multi-threaded program makes multiple requests to execute the same object within the same memory space. These requests are referred to as threads. Additionally, a multi-threaded program may have multiple entry points. As such, a multi-threaded program does not align well with the sporadic model.

To address these shortcomings, the sporadic model is modified for multi-threaded programs. The set of task $\tau = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$ is preserved. However, each task is represented

by a triple of period, relative deadline, and initial ribbon, $\tau_i = (p_i, d_i, \rho_i)$. A *ribbon* is the set of instructions reachable from a single entry point and is identified by the address of its entry point a_i. A *thread* t_i is a single request to execute the instructions of a ribbon ρ_i. When referring to an arbitrary thread of ρ_i the notation t_i will be used. However, when referring to a specific thread the notation includes an index, i.e. the third thread of ρ_i is $t_{i,3}$. When a job is released, execution begins with threads of the initial ribbon called *initial threads*.

A ribbon ρ_i that is within the object of a task τ_j is said to *belong* to a task, denoted $\rho_i \in \tau_j$. Similarly, a thread belongs to a ribbon $t_i \in \rho_i$, the thread also belongs to the task the ribbon belongs to $t_i \in \tau_j$, and a thread belongs to the job it was released from $t_i \in J_{i,k}$. All threads that belong to the same job may access the entire memory space of the job. No thread belonging to one job may access the memory space of a different job. Although the model supports the release of additional threads by the initial and subsequent threads, such release patterns are not explored in this work.

2.5 Control Flow Graphs

A key concept in BUNDLE's approach to program analysis and scheduling is the control flow graph (CFG) [20]. A *control flow graph*, is a weakly connected directed graph G given by the triple of nodes, edges, and entry instruction $G = (N, E, h)$. Typically, the nodes $n \in N$ of a CFG are basic blocks. A *basic block* is a set of instructions that execute serially; if the first instruction is executed the remaining instructions of the basic block will always execute one after another (unless an error or interrupt occurs). Basic blocks are identified by their starting instruction. Directed edges between nodes $(u, v) \in E \wedge u, v \in N$

represent the possible changes in the execution path through the CFG. Execution begins with the entry instruction h, which can reach any other node in the graph. All paths through the CFG begin with h and end in a single terminal node denoted z.

Treatment of control flow graphs within this work differs from their typical use in a simple but important way. Nodes of a CFG are single instructions rather than basic blocks. A compatible definition would be that all basic blocks are of length one. Several operations will be described that divide and reassemble the control flow graph of a ribbon. The control flow graph will be separated into conflict free regions (these are also control flow graphs). Conflict free regions will then be assembled into a conflict free region graph (also a control flow graph) where conflict free regions act as nodes not graphs. The relationship between these graphs is detailed Chapter 5.

2.6 Notation Summary

The following table summarizes the notation given in this section, it also lists symbols that will be used consistently throughout later sections.

Symbol	Meaning
τ	Set of n tasks $\{\tau_0, \tau_1, ..., \tau_{n-1}\}$
$\tau_i = (p_i, d_i, \rho_i)$	A task with minimum inter-arrival time, relative deadline, and initial ribbon
$J_{i,k} = (R_{i,k}, D_{i,k})$	Job release k of task i with absolute release time and absolute deadline
$\mathbb{B}$	Block Reload Time (BRT)
$\mathbb{I}$	Cycles Per Instruction (CPI)
ρ_i	A ribbon
a_i	An address in main memory
$G = (N, E, h)$	Control Flow Graph (CFG) of nodes, edges, and entry instruction
$R = (N, E, h)$	Conflict Free Region Graph (CFRG)
$M(a)$	Block of cache memory utilized by absolute memory address a
$\hat{M}(a)$	Block of main memory a resides in
s	Size of the cache in blocks
m	Number of initial threads released with each job
H	A set of CFR entry points
T	Set of threads
t_i	A thread of ribbon ρ_i
$t_{i,k}$	The k^{th} thread of ribbon ρ_i
π	A path
C	Simulated cache with methods: present, insert, clear, conflicts
$p(n)$	Set of next intra-thread cache conflicts
$\times$	Intra-thread cache conflict
$P(n)$	Set of next inter-thread cache conflicts
$\bigotimes$	Inter-thread cache conflicts
L	Length of path
ϖ_i	Priority of CFR (bundle) n_i
$c_{n_i}(m)$	WCETO of CFR n_i for m threads
$c_i(m)$	WCETO of task τ_i for m threads

Table 2.1: List of Symbols

CHAPTER 3 INTER-THREAD CACHE BENEFIT

Central to BUNDLE's positive perspective is the inter-thread cache benefit. This chapter provides a definition of the benefit. Additionally, the impact of the benefit is directly compared to the classical perspectives of WCET and CRPD analysis by means of an example. To ease the presentation for the reader, the following table summarizes the symbols required to follow the example.

Symbol	**Meaning**
τ	Set of tasks
τ_i	Task i
$J_{i,k}$	k^{th} job of task i
ρ_j	A ribbon j
$t_{i,k}$	k^{th} thread of ribbon ρ_i
m	Number of threads released per job of τ_1
a	Main Memory Address
$M(a)$	Cache Block Containing a
$\mathbb{B}$	Block Reload Time
$\mathbb{I}$	Cycles Per Instruction

3.1 Defining the Inter-Thread Cache Benefit

As part of scheduling a job on the processor, the object of the job's task is copied into main memory. Additional memory may be reserved or requested by each job, increasing its address space. In Figure 3.1, the address spaces of the fourth job of task one and the second job of task three are shown in main memory. The shaded area is the copy of the executable object, and the sinuous area is the additional reservation made by the job.

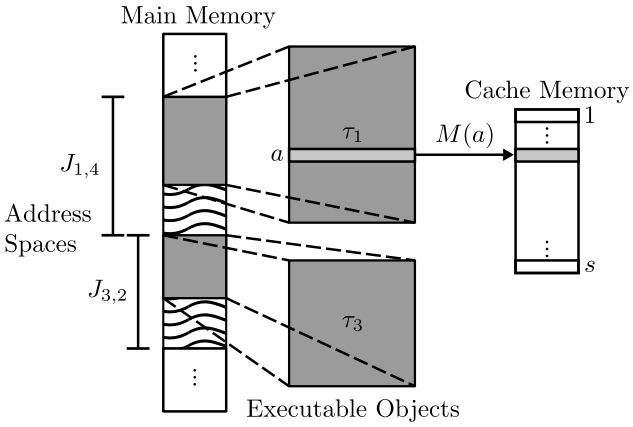


Figure 3.1: Address Space for Two Jobs

Threads share the address space of their job. A thread t_j that belongs to a job of task τ_i ($t_j \in \tau_i$), resides in the memory space of a job J_i of task τ_i. Within an address space, instructions have an address a which maps to a cache block $M(a)$ (illustrated in Figure 3.1).

When a thread t_k executes without interruption by preemption, an instruction access that results in a cache miss is called an *opportunity instruction*, or simply an *opportunity*. Similarly, during uninterrupted execution, any instruction access that hits the cache is called an *expected instruction* or an *expectation*.

When multiple threads are executed, the execution time of one or more threads may be influenced by cache interactions. When a thread t_j preempts a thread t_k, t_j may evict cache blocks of t_k placed there. If those evicted cache blocks correspond to expected

instructions, t_j will increased t_k's execution time since t_k must now pay $\mathbb{B}$ for each evicted block. Conversely, a thread t_j may unexpectedly place opportunity instructions of t_k in the cache during a preemption of t_k, reducing t_k's execution time.

Inter-Thread Cache Benefit: Thus, the inter-thread cache benefit for a thread of t_j is the speed-up of $t_j \in \tau_i$ due to the conversion of opportunities into expectations by the placement of values in the cache from a thread of $t_k \in \tau_i$ when t_k is scheduled before t_j.

3.2 Comparison of Perspectives

Tasks	Task	Ribbons	Thread Releases
$\tau = \{\tau_1\}$	$\tau_1 = (p_1, d_1, \rho_1)$	$\rho = \{\rho_1\}$	$m = 2$
Cache Size (Number of Blocks)		CPI	BRT
$s = 200$		$\mathbb{I} = 1$	$\mathbb{B} = 10$

Table 3.1: Example Model Parameters

Using the model parameters in Table 3.1, an example ribbon ρ_1 releasing two threads is presented as a CFG in Figure 3.2. The purpose of this example is to clarify the inter-thread cache benefit and expose the pessimism in the classical WCET, CRPD, and scheduling analysis techniques.

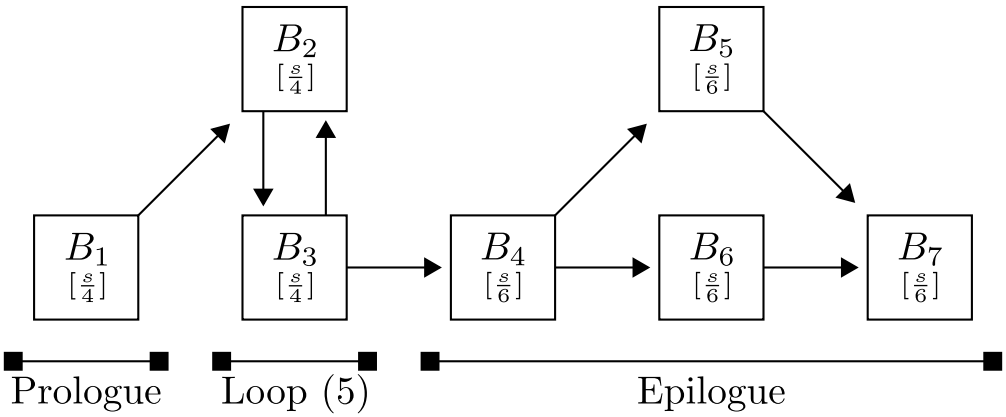


Figure 3.2: Control Flow Graph for: ρ_1

Block	Category
B_1	must-miss
B_2 B_3	first-miss
B_4 B_5 B_6 B_7	must-miss

Cache Part n of 12	Blocks		
1	B_1	B_5	B_6
2		B_7	
3			
4 ⋮ 11	B_2 B_3 B_4		
12		B_5	B_6

Table 3.2: Categories from [1, 2] and Cache Assignment

The CFG in Figure 3.2 utilizes basic blocks (serialized sets of instructions) with lengths expressed in terms of the cache size. Below each block labeled with a B, is the block's length expressed as fraction of the cache size s. The parenthesized value at the bottom of the figure indicates the maximum number of iterations the loop will execute. Figure 3.2 differs from the common use of CFG's in this work, other figures will typically use basic blocks of length one.

For the purpose of the example, the ribbon ρ_1 is analyzed by the WCET calculation methods of Arnold [1] and Mueller [2]. CRPD costs are determined using Lee et al.'s [21] useful cache block (UCB) technique. Although simpler and less accurate than modern techniques, these methods were chosen for illustrative purposes and their continued use in subsequent works.

A necessary step in WCET calculation is the categorization of instructions, such as *must-miss* and *first-miss*. A must-miss never hits the cache. A first-miss always hits the cache after its initial miss. To find first-miss instructions the CFG is searched iteratively looking for return paths. Only instructions with return paths are candidates for first-misses. Table 3.2 presents the cache mapping and categorizations.

Lee et al.'s [21] useful cache block (UCB) approach to CRPD calculation borrows the iterative return path approach. From Figure 3.2, the only candidates for first-miss and UCB instructions are contained in basic blocks B_2 and B_3. No other blocks have a return path and would be categorized as must-miss, and not useful.

3.2.1 WCET

Using these categorizations and the loop bound, the worst case execution time of ρ_1 is the sum of the execution times of the prologue, the entry executions of B_2 and B_3, the repetitions of B_2 and B_3, and the epilogue. Table 3.3 gives the intermediate values; the total execution time taking into consideration reloads is: $\frac{s(\mathbb{B}+\mathbb{I})}{4} + \frac{2s(\mathbb{B}+\mathbb{I})}{4} + \frac{8s(\mathbb{I})}{4} + \frac{3s(\mathbb{B}+\mathbb{I})}{6} = \frac{s(5\mathbb{B}+13\mathbb{I})}{4} = 3150$

Section	**Basic Blocks**	**WCET**
Prologue	B_1	$\left(\frac{s}{4} \cdot (\mathbb{B} + \mathbb{I})\right)$
Loop Entry	$B_2 + B_3$	$\left(\frac{s}{4} \cdot 2 \cdot (\mathbb{B} + \mathbb{I})\right)$
Loop Repetition	$(B_2 + B_3) \cdot 4$ (repeats)	$\left(\frac{s}{4} \cdot 2 \cdot 4 \cdot (\mathbb{I})\right)$
Epilogue	$B_4 + (B_5$ or $B_6) + B_7$	$\left(\frac{s}{6} \cdot 3 \cdot (\mathbb{B} + \mathbb{I})\right)$

Table 3.3: Segment WCET

Under the classical model two synthetic tasks are created for the two threads of ρ_1. Assigning the WCET of 3150 to both synthetic tasks, the total execution requirement for one job is 6300 every p_1 time units.

However, this is overly pessimistic. The worst possible execution scenario and schedule for the two threads is the sequential execution of $t_{1,1}$ followed by $t_{1,2}$, where $t_{1,1}$ takes the "high" road executing B_5 and $t_{1,2}$ takes the low "road" through B_6. This maximizes the number blocks $t_{1,2}$ will miss from the cache. Even so, blocks B_2, B_3, B_4 are present in the cache when $t_{1,2}$ reaches them.

Using the worst case schedule, the WCET of $t_{1,2}$ is: $\frac{s(\mathbb{B}+\mathbb{I})}{4} + \frac{5s(\mathbb{I})}{4} + \frac{s(\mathbb{B}+\mathbb{I})}{4} + \frac{s(\mathbb{I})}{4} = \frac{2s(\mathbb{B}+4\mathbb{I})}{4}$ $= 1400$. The total job execution requirement is $3150 + 1400 = 4550$ cycles, less than the 6300 cycles calculated from the synthetic task analysis and application of the Arnold and Mueller approaches. Figure 3.3 illustrates the worst possible schedule of $t_{1,1}$ and $t_{1,2}$ including a summary of cache contents at time 3150, which is compared to the WCET bound calculated from classical perspective.

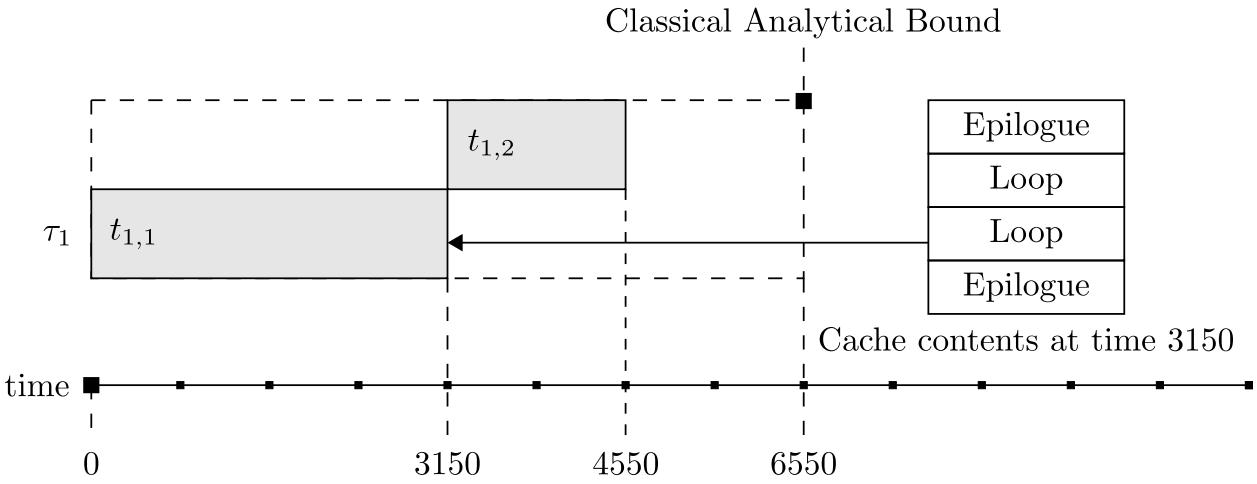


Figure 3.3: Worst Schedule of τ_1, 4550 Cycles

3.2.2 CRPD

Cache Related Preemption Delay (CRPD) is an analytical technique that accounts for the execution time extension of one task due to the cache interference of another. A task executing in isolation may store and reuse values from the cache. When preempted, those stored cache values may be invalidated before they are reused. Upon resumption the preempted task must pay the BRT for each invalidated cache block.

A method for CRPD calculation is the Lee et. al [21] useful cache block (UCB) approach. A UCB is "a cache block that contains a memory block that may be referenced before being replaced by another memory block." CRPD for a task is limited by the num-

ber of UCBs within it.

From Figure 3.2 there are two basic blocks that contribute UCBs to the ribbon ρ_1: B_2 and B_3. Applying Lee's method, the CRPD of a preemption of thread of ρ_1 is $\frac{2s(\mathbb{B}+\mathbb{I})}{4} = 1100$. However, this bound is overly pessimistic.

By construction (and shown in Table 3.2) once the "Loop" instructions are cached they cannot be invalidated. If $t_{1,1}$ were preempted by $t_{1,2}$ after the first iteration of the loop, the instructions of the loop body (B_2 and B_3) would be cached in parts 4-10. No other instructions of $t_{1,2}$ map to those cache lines and cannot invalidate them. Furthermore, there is no schedule of $t_{1,1}$ and $t_{1,2}$ which incurs any CRPD.

Lee's approach to CRPD calculation is known to be an overestimate, there are refinements such as the UCB-ECB [22], UCB-Union, and UCB-Union Multiset [13] approaches. However, the UCB calculation is a component of each of them and the advanced techniques suffer from the same inability to address cache memory as a benefit rather than a detriment. Similarly, the Arnold [1] and Mueller [2] approaches play a role in subsequent WCET methods and none incorporate the inter-thread cache benefit.

CHAPTER 4 RELATED WORK

While no existing work focuses on the inter-thread cache benefit to improve schedulability, this chapter provides a survey of related publications from the classical and positive perspectives. Chapter 1 gave a brief introduction to hard real-time systems and cache memory, the reader may find Liu's [23] and Hennessey's [19] work helpful on the topics.

4.1 Worst-Case Execution Time and Cache Memory

Cache memory brings to additional complexity to worst-case execution time analysis through non-uniform execution times due to cache misses or hits and has received considerable attention [24, 25]. A central concept of accounting for non-uniform execution times is the categorization of memory references (including instructions). A reference will be categorized as *first-miss*, *must-miss*, or *must-hit*. A must-miss reference will never be found in the cache during execution, where a must-hit will always be present. A first-miss will absent for its initial execution reference and present for others.

The works of Arnold [1] and Mueller [2, 12] use static cache simulation for direct-mapped caches to classify references. Their techniques involve repeatedly searching the CFG of the task for return paths to references. These techniques have been refined and expanded: White et al. [26] incorporated data caches, Li et al. [27] included set-associative caches and pipelines, among others.

Arnold and Mueller's work are of particular value to this work due to their role in Heptane. An open-source WCET analysis tool, Heptane extends Mueller's work in part to demonstrate the incorporation of branch prediction into WCET analysis [28]. In this work,

the Heptane toolset is extended to support our proposed WCETO analysis for `BUNDLEP` in Chapter 6.

4.2 Cache Related Preemption Delay

As an area of study cache related preemption delay (CRPD) is the examination of the extension of execution time of one job due to the preemption of another. Taking the perspective of a task being preempted, before preemption there are blocks in the cache that will be reused later. When preempted, the preempting task will execute instructions evicting the reusable blocks. Upon resuming the preempted task, the time required to reload those blocks that would have been reused is the CRPD.

Typically denoted γ in the literature, the CRPD of a task is an upper bound on the amount of time required to reload cache blocks evicted during a preemption. Schedulability tests incorporate γ in one of two ways 1.) increasing a task's WCET value 2.) adding (a factor of) γ to the task's response time [29, 30, 13, 14, 22, 31, 15].

Calculating the CRPD for a task or task set is made from one of three perspectives. The *preempting perspective* where the bound on the number of cache blocks affected is determined by the preempting task. The *preempted perspective* where the bound on the number of cache blocks affected is determined by the preempted task. The *combined perspective* bounds the number of cache blocks affected by considering both the preempted and preempting tasks.

Tomiyama and Dutt [17] developed an approach based on the preempting perspective. They are credited with creating the concept of the *evicting cache block* (ECB). Defined as "a memory block of the preempting task is called an evicting cache block, if it may be

accessed during the execution of the preempting task." [13].

Lee [21] take the perspective of the preempted task, noting that a preemption may not be harmful. For a preempted task, only those cache blocks which are reused can extend execution times if evicted. Cache blocks which are not reused may be evicted without penalty. They call these cache blocks *useful cache blocks* or UCBs, and are defined as "a cache block that contains a memory block that may be referenced before being replaced by another memory block".

The combined perspective considers the possible harm a preempting task could inflict on a preempted task. The number of useful cache blocks limit the total number of affected blocks in the preempted task. Negi et al. [22] developed the UCB-ECB approach using the intersection of the preempted tasks UCBs and the preempting tasks ECBs to bound the CRPD. Tan and Mooney [15] improved upon the UCB approach, observing the CRPD cost included multiple evictions for the same UCB per preemption. Their approach, named the UCB-Union also considers the UCBs of the preempted task with the ECBs of preempting tasks.

Among the combined approaches, Altmeyer's [14] ECB-Union Multiset, and UCB-Union Multiset out-perform the others. Principally, the use of a multi-set prevents over counting of evictions due to multiple levels preemptions during a single preemption of a lower priority task. Altmeyer employs an alternative response time function developed by Staschulat [31] with a cumulative CRPD value, rather than a per preemption value. Lastly, Altmeyer proposes what he coins the "Combined" approach which takes the minimum of the ECB-Union Multiset and UCB-Union Multiset which out-performs all methods.

Tighter analysis (reducing) of CRPD values increases schedulability by limiting the im-

pact of each preemption. A complimentary method to reducing the impact of CRPD on schedulability is to limit or defer preemptions. In the limited or deferred preemption setting, a higher priority task may preempt a lower priority task only when some condition is satisfied [32, 33, 34, 35]. Heuristics for placing preemption points to reduce CRPD values were proposed in [36, 21]. Bril et al. [37] augment preemption threshold scheduling by incorporating CRPD values into schedulability analysis. Bertogna et al. [38] provide a more formal approach for optimally determining preemptions in programs that can be represented by linear control flowgraphs given the CRPD overhead of each preemption and a bound on the maximum non-preemption region [34]. Later work, extended this to more general control flowgraphs [39] or more precise CRPD characterizations of the preemption costs [40].

Each of the CRPD methods described are limited to a single threaded task. The proposed techniques of this work focus on multi-threaded tasks. While not directly applicable, the concepts developed for CRPD calculation of ECBs and UCBs are leveraged in the proposed work. These existing methods also serve as a basis for comparison of the classical perspective to the proposed methods.

4.3 Cache Analysis in Multi-Threaded Programs

Multi-threaded WCET analysis typically takes the classical perspective on cache memory. An example is feasibility analysis for the fork-join [6] model, where the WCET of a thread is the longest single threaded execution through the object of the thread. Each thread's WCET value contributes to the overall demand independently of other threads.

Concurrent program analysis [41] of has been extended to consider variable configura-

tions such as shared multi-level caches [42]. These methods take the negative perspective, where cache interactions exclusively increase execution times. For shared caches, the analysis includes the maximum extension due to cache sharing, by constructing the worst-case interleavings of threads.

4.4 Predictable Cache Behavior

Refinements of the classical perspective include techniques that attempt to mitigate or manage the cache impact between tasks. Their goal is to reduce or eliminate conflicts between jobs by creating predictable cache behavior. On multi-core systems, Memory-Centric Scheduling [43] limits execution of tasks by considering its access to main and cache memory. Memory-Centric Scheduling depends on tasks that fit the PRedictable Execution Model (PREM) [44].

PREM-compliant tasks are divided (by the programmer) into intervals in one of two categories. Compatible fall into the first category, accessing main memory at any point during execution. Predictable intervals are the second category, and are further divided into loading and execution phases. During the loading phase all main memory accesses are placed in the lowest level cache. When the loading phase is complete, the execution phase may begin where no memory accesses will result in a cache miss.

Under PREM, great care is taken to avoid concurrent memory access between tasks. No two loading phases may take place simultaneously. Furthermore, compatible intervals are treated as loading phases. Isolation of tasks is by design due to the negative perspective of caches, as such PREM is unable to account for the potential inter-cache benefit between threads.

PREM tasks require the programmer to define compatible and predictable intervals. When active participation in memory management is infeasible or undesirable, passive predictable cache behavior may involve several techniques. An example of combined management efforts is made in Ward's allocation framework for mixed-criticality on multi-core MC^2 [45].

Ward applies three techniques simultaneously. Page coloring (also referred to as partitioning) [46] is used, where pages of memory are assigned colors in such a manner that no two pages can conflict in the cache. Tasks are assigned colored pages as their working set of memory during execution. Cache locking is introduced [45], which requires a task to hold a color lock for each of the colored pages it needs before execution. Cache scheduling considers the colors of each task when scheduling them, (possibly preemptively scheduling them) to avoid conflicts with other tasks. Similar to PREM, the focus is on isolation to reduce the negative impact of cache interference between tasks without considering the positive impact of caches.

4.5 Positive Perspectives on Caches

We are aware of two techniques that take a positive perspective on caches. Calandrino [47] limits the cache spread of threads (called subtasks) for multi-threaded tasks. The empirical results show higher cache hit rates. However, no analytical method to bound the cache spread is given.

Persistent Cache Blocks [48, 49] (PCBs) take a positive perspective on caches for subsequent job releases. A PCB is a cache block that remains in the cache after a job has completed which is then reused by a subsequent job. As such, PCBs are limited to tasks.

Additionally, the PCB approach requires modification to existing worst case response time (WCRT), WCET and CRPD analytical methods. Over-simplistically, PCBs are removed from WCET calculations and included once in response time analysis. The result is a benefit to system schedulability.

CHAPTER 5 SINGLE-TASK BUNDLE

The BUNDLE scheduling algorithm and WCETO analysis serve as the basis of our positive perspective on instruction caches. It is limited to a single processor, and single task releasing m initial threads per job release. Scheduling and analysis operate upon derivatives of the control flow graph of the initial ribbon called conflict free regions and the conflict free region graph. The description of BUNDLE's techniques begins with a definition of scheduling algorithm, followed by formal definitions of conflict free regions and conflict free region graphs before detailing the WCETO method.

5.1 BUNDLE Scheduling

BUNDLE takes its name from the manner in which it schedules threads of a job. Threads are placed in a container called a *bundle*, only one bundle is active, and only threads of the active bundle are scheduled on the processor at any time. A bundle is associated with a conflict free region (CFR): a subset of instructions of the ribbon where no two instructions conflict in the cache. When a bundle is *active* or *inactive*, it is also said that the associated conflict free region is active or inactive. A thread leaves the active bundle by attempting to execute an instruction of a different region. When leaving the active bundle and entering a new bundle a thread is blocked until the bundle it enters becomes active. The active bundle is *depleted* when all threads leave it. Scheduling threads by their bundle allows the sharing of cache values (hits) to be quantified and the penalty of cache misses reduced.

To provide context, BUNDLE's scheduling algorithm is presented as pseudocode in Figure 1. Since the algorithm is limited to one task, the task and job indices are omitted.

Introduced in the pseudocode are several new symbols. T is the set of m threads per job release. H is the set of entry instructions of CFRs. Though previously unstated, every CFR (which is a CFG) has an entry instruction that distinguishes it from the others. Line 8 utilizes a unique mechanism for halting threads, before a thread can execute the entry instruction of an inactive CFR the thread is blocked and the scheduler is invoked; we call this *anticipating execution*. We are unaware of any hardware platform that supports anticipating execution, nor is an implementation suggested in this section. For `BUNDLE` we assume the mechanism exists, a hardware mechanism is proposed in Section 6.4.1 suitable for both `BUNDLE` and `BUNDLEP`.

Algorithm 1 `BUNDLE` Scheduling Algorithm

```
1:  T                                                  ▷ Set of m threads
2:  H                                                  ▷ Set of CFR entry instructions
3:  procedure BUNDLE
4:      A ← T                                          ▷ Active bundle
5:      B ← ∅                                          ▷ Array of inactive bundles (blocked threads)
6:      while true do
7:          Select t ∈ A
8:          RUN(t) until t's next instruction is h ∈ H
9:          B[h] ← B[h] ∪ a                            ▷ Place t in a new bundle
10:         A ← A ∖ t                                  ▷ Remove t from the active bundle
11:         if |A| = 0 then                            ▷ The active bundle has been depleted
12:             Select z ∈ H, |B[z]| ≠ 0               ▷ Choose a non-empty bundle as active
13:             A ← B[z]
14:             B[z] ← ∅
15:         end if
16:     end while
17: end procedure
```

At the beginning of the `BUNDLE` procedure, the active bundle A contains all threads of the job waiting to execute the initial instruction of the ribbon. For each iteration of the loop a thread is selected from the active bundle on Line 7. The thread t executes until it encounters an entry instruction of a different bundle on line 8, it is then removed from the

active bundle placed in its new inactive bundle and blocked. If removing t depleted the active bundle A, an *arbitrary* non-empty bundle is selected as active in the body of the **if** block on Line 11. The loop repeats until all threads complete.

5.2 Conflict Free Regions and Conflict Free Region Graphs

The scheduling algorithm relies upon bundles and their associated conflict free regions (CFRs) to make scheduling decisions. Calculations for WCETO values also depend on conflict free regions over the structure of the conflict free region graph (CFRG). CFRs and the CFRG are derived from the CFG of the ribbon. This section provides formal definitions and methods of creation for CFRs and CFRGs suitable for `BUNDLE`.

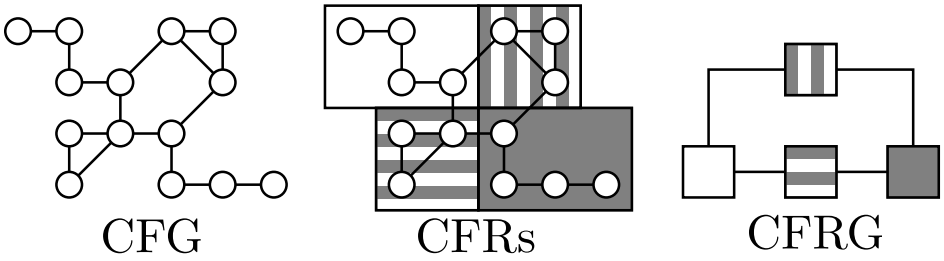


Figure 5.1: CFG, CFRs, and CFRG of a ribbon

The following definitions refer to the CFG of a ribbon as $G = (N, E, h)$ and the CFRG $R = (N, E, h)$. When necessary to remove ambiguity the sets are given a superscript of their origin, e.g. N^G identifies the nodes of the CFG and N^R the nodes of the CFRG. Recall the nodes of a CFG are individual instructions, edges represent potential paths of execution between them. Nodes of the CFRG are CFRs, when referring to a specific CFR n_i is used, nonspecific CFRs are given by F. CFRs are extracted from the CFG and placed in the CFRG with their connectivity preserved. For an edge (n_1, n_2) in the CFG, if n_1 and n_2 reside in distinct CFRs, then the CFRG must contain an edge between those CFRs. The relationship

between the CFG, CFRs, and CFRG of a ribbon is illustrated in Figure 5.1.

Regions of a Control Flow Graph: a region is a selection of the vertexes and edges of a CFG G. When extracting a region from G, the graph's connectivity is preserved. I.e, two vertexes connected in G must also be connected in any region containing both.

Formally, for a region $r = (N, E, h)$ of a control flow graph $G = (N^G, E^G, h^G)$, where $N \subset N^G$ and $E \subset E^G$. For all pairs of vertexes $(u, v) \in N$, $(u, v) \in E \iff (u, v) \in E^G$. Regions contain an entry instruction $h \in N$ that is weakly connected to all other vertexes in N.

Conflict Free Region: a region $F = (N, E, h)$ of G is conflict free if no two instructions of F in distinct memory blocks utilize the same cache block.

$$\forall n_i, n_j \in N, \hat{M}(n_i) \neq \hat{M}(n_j) \iff M(n_i) \neq M(n_j)$$

To restate, the requirements of a CFR F are:

1. No two instructions (outside of the same main memory block) map to the same cache block
2. All instructions of F are weakly connected to the entry instruction h
3. For any two instructions $(n_i, n_j) \in F$, if there was an edge between them in G then $(n_i, n_j) \in E$ (of F)

Figure 5.2: Requirements of Conflict Free Region

Conflict Free Region Graph: a conflict free region graph $R = (N^R, E^R, h^R)$ is a CFG of

CFRs of $G = (N^G, E^G, h^G)$ where connectivity between CFRs is preserved and all instructions of G are included in some CFR $n \in N^R$. In the definitions below, a CFR i is denoted $n_i \in N^R$. For a CFR n_i the triple is given by $n_i = (N_i, E_i, h_i)$.

$$\forall n \in N^G, n \in \bigcup_{n_j \in N_i \wedge n_i \in N^R} n_j \tag{5.2.1}$$

$$\forall (u, v) \in E^G, u \in N_i \wedge n_i \in N^R \wedge v \notin N_i \implies \exists (n_i, n_j) \in E^R, v \in N_j \tag{5.2.2}$$

Equation 5.2.1 ensures that each node of the CFG is included in some CFR, the set of instructions $n_j \in N_i \wedge n_i \in N^R$ are those n_j found in the set of instructions N_i from the CFR i which is contained in the CFRG as $n_i \in N^R$. Equation 5.2.2 preserves connectivity from the CFG in the CFRG, when an edge from the CFG is not contained within a CFR there must exist an edge in the CFRG.

5.2.1 Extracting Conflict Free Regions

The process of analyzing the control flow graph and assigning instructions to conflict free regions is called *extraction*. Support for the process is given by the definition of intra and inter-thread cache conflicts. These definitions ensure extraction meets the first requirement for CFRs, that no conflicts exist. The conventional use of symbols will continue for the following definitions, in context of the ribbon being analyzed: CFG $G = (N^G, E^G, h^G)$, an arbitrary CFR $F = (N, E, h)$, CFRG $R = (N^R, E^R, h^R)$, CFRs $n_i = (N_i, E_i, h_i)$ where $n_i \in N^R$.

When referring to any type of control flow graph of a ribbon, CFR, CFRG, or CFG the notation remains the same, since all of the structures are CFGs. For a CFG $G = (N, E, h)$ a node $n \in N$ is an instruction. For a CFR $F = (N, E, h)$ a node $n \in N$ is also an instruction. For a CFRG $R = (N, E, h)$ a node $n \in N$ is a CFR. The context determines which type of node or edge is being referred to.

Valid Path: a path $\pi = \langle n_0, n_1, ... \rangle$ of ordered nodes is valid if and only if, for every adjacent pair of nodes n_i, n_{i+1} in the path there exist a directed edge in the CFG.

$$\forall (n_i, n_{i+1}) \in \pi, (n_i, n_{i+1}) \in E^G$$

Intra-Thread Cache Conflict: an intra-thread cache conflict is an eviction that may occur during the non-preempted execution of a thread. For instructions $n_i, n_j \in N^G$, $\hat{M}(n_i) \neq \hat{M}(n_j)$ and a valid path $\pi = \langle n_i, ..., n_j \rangle$, n_j is an intra-thread cache conflict if $M(n_i) = M(n_j)$.

Along a valid path starting with n_i there may be multiple intra-thread cache conflicts. The *next* conflict is defined as the conflict with the shortest distance on the path from n_i.

Next Intra-Thread Cache Conflict: for $n_i \in N^G$, a next intra-thread cache conflict is an intra-thread cache conflict n_x on a valid path $\pi = \langle n_i, ..., n_j, n_x \rangle$ containing no intra-thread cache conflicts between any two nodes $(n_a, n_b) \in \pi'$ for $\pi' = \langle n_i, ..., n_j \rangle$.

Next Intra-Thread Cache Conflicts: is the set of all possible n_x values that are next intra-thread cache conflicts from n_i. The set is given by $p(n_i)$.

Figure 5.3 provides an illustration of the next intra-thread cache conflicts from n_i in the CFG of the task's ribbon. All nodes are equal in the graph, though some are unnamed.

Below each node is the cache block it maps to, in the case of n_i its value would be cached in block 3. Execution beginning with n_i could lead to an eviction of n_i by n_j. Another eviction could occur if n_b were executed after n_a. Although n_a could evict n_y, n_a is not a next conflict starting from n_i. Only two next intra-thread conflicts exist from $p(n_i) = \{n_b, n_j\}$.

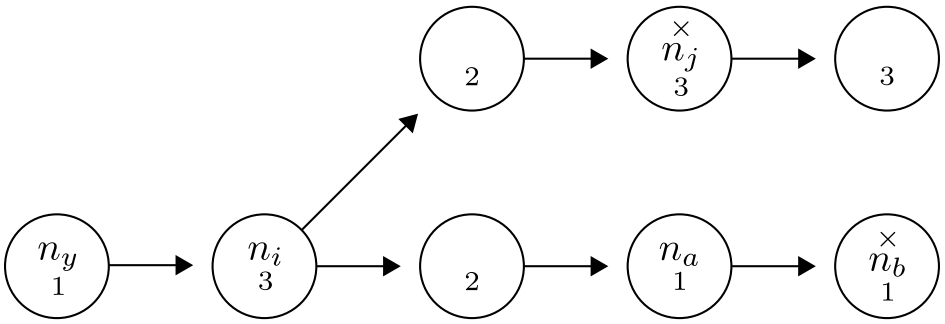


Figure 5.3: Next Intra-Thread Cache Conflicts from n_i marked with a $\times$

A pair of algorithms is presented as pseudocode for $p(n)$. The first is a recursive algorithm named *paths of conflict*, given in Algorithm 2. This procedure returns a set of valid paths starting with n and terminated with the first conflict on the path. It uses a simulated cache object C and its methods: C.insert(n) caches n's memory block, C.clear() removes all blocks, C.present(n) returns true if n's block is already cached, and C.conflicts(n) returns true if C.insert(n) would evict a cache block. Algorithm 3 returns $p(n)$, the set of next intra-thread conflicts from n: the union of the last elements of each path returned by POC(n).

A recursive depth first search (DFS) is the basis for POC(n). There are two exit conditions found on Line 8 when there are no more subsequent nodes on the path, or a conflict has been found. A conflict is determined by checking the global simulated cache state used for all recursive calls. If the exit condition is not satisfied, the current node is inserted into the cache before working on the subsequent nodes. The work of the recursive call takes

Algorithm 2 Paths of Conflict from n

```
1:  G = (N, E, h)                                   ▷ CFG of the ribbon
2:  C                                               ▷ Simulated Cache
3:  procedure POC(n)
4:      P ← ⟨n⟩                                     ▷ One path
5:      ℙ ← ∅                                       ▷ All paths, the return value
6:      mark(n)                                     ▷ Marks n as visited
7:      K ← {v | (u, v) ∈ E}
8:      if |K| = 0 or C.conflicts(n) then           ▷ Path exploration terminated or hit a conflict
9:          ℙ ← {P}
10:         return ℙ
11:     end if
12:     C.insert(n)
13:     for all v ∈ K and not visited(v) do
14:         C′ ← C                                  ▷ Copy the cache
15:         for all P′ ∈ POC(v) do
16:             ℙ ← ℙ ∪ ⟨P, P′⟩
17:         end for
18:         C ← C′                                  ▷ Restore the cache
19:     end for
20:                                                 ▷ Remove paths with cross-path conflicts
21:     for all P ∈ ℙ do
22:         for all T ∈ {ℙ \ P} do
23:             if T.last ∈ P and T.last ≠ P.last then
24:                 ℙ ← {ℙ − P}
25:             end if
26:         end for
27:     end for
28:     return ℙ
29: end procedure
```

place on lines 13-19, for each subsequent node v the cache state is copied and POC(v) invoked. The recursive call returns a set of sub-paths, starting with v that terminate in conflicts. To these paths the current path is pre-pended to make a complete path, each complete path is added to the set $\mathbb{P}$. Since the cache state is copied and restored for each recursive call, it is possible that some paths in $\mathbb{P}$ contain conflicts with others before they terminate. The double loop starting on Line 21 removes those paths that contain conflicts before their terminal node.

Algorithm 3 Next Intra-Thread Cache Conflicts $p(n)$

```
1: procedure P(n)
2:     R ← ∅
3:     ℙ ←POC(n)
4:     for all P ∈ ℙ do
5:         R ← R ∪ P.last
6:     end for
7:     return R
8: end procedure
```

Algorithm 3 completes the pair for $\text{P}(n)$ and is straightforward. Taking the last element of the paths of conflict and adding each to the set R, only the next intra-thread cache conflicts are returned.

Inter-Thread Cache Conflict: an inter-thread cache conflict is a possible eviction due to the execution of multiple threads of the same ribbon. For instructions $n_i, n_j \in N^G$, $\hat{M}(n_i) \neq \hat{M}(n_j)$, n_i and n_j are inter-thread cache conflicts if $M(n_i) = M(n_j)$.

Next Inter-Thread Cache Conflict: for $n_i \in N^G$, a next inter-thread cache conflict from n_i is an instruction n_j, where $M(n_j) = M(n_k)$ for some n_k with valid paths $\pi_j = \langle n_i, ..., n_j \rangle$, $\pi_k = \langle n_i, ..., n_k \rangle$ and no other conflicts between π_j and π_k.

$$\forall (n_a, n_b \in (\pi_j \cup \pi_k) \setminus \{n_j, n_k\}),\ \hat{M}(n_a) \neq \hat{M}(n_b) \implies M(n_a) \neq M(n_b)$$

Next Inter-Thread Cache Conflicts: is the set of all possible n_x values a that are next inter-thread cache conflicts from n_i. The set is given by $P(n_i)$.

Figure 5.4 illustrates the relationship between intra and inter-thread cache conflicts from a node n_i for a particular CFG. An intra-thread cache conflict by definition is an inter-thread cache conflict. Consider an intra-thread cache conflict $n_x \in p(n_i)$, n_x conflicts with some instruction n_j on the path $\pi = \langle n_i, ..., n_j, ..., n_x \rangle$. For two threads, one may execute

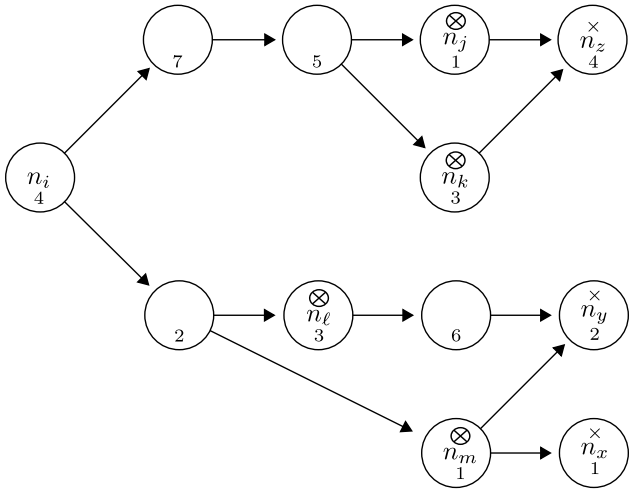


Figure 5.4: Next Inter-Thread Cache Conflicts from n_i marked with a $\otimes$

and cache n_j, and the other may execute n_x evicting n_j.

Leveraging the definition of intra-thread cache conflicts (marked with a $\times$ in Figure 5.4), the next inter-thread cache conflicts are marked with a $\otimes$. Note the inter-thread cache conflicts occur across paths. In particular, n_k and n_ℓ share cache block 3 but are not reachable from one another. The complete set of inter-thread cache conflicts are $\{n_j, n_k, n_\ell, n_m\}$, Figure 5.5 depicts the largest region without conflicts.

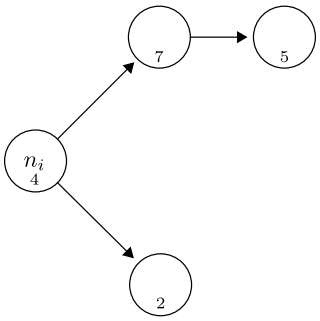


Figure 5.5: Largest region of Figure 5.4 with no conflicts from n_i

The set of next inter-thread cache conflicts is denoted $P(n)$ and described by pseudocode in Algorithm 4. It relies on the paths of conflict algorithm POC(n) to bound its search. Since POC returns the set of paths that terminate in the intra-thread cache conflicts

from n. Those paths are examined as pairs P and Q for cross-path conflicts, and if one is found the paths are cut (shortened) to the conflicting instructions.

Algorithm 4 Next Inter-Thread Cache Conflicts $p(n)$

```
1:  procedure P(n)
2:      ℙ ← POC(n)
3:      for all (P, Q) ∈ ℙ do
4:          for all n_i ∈ P, n_j ∈ Q do
5:              if M(n_i) = M(n_j) then
6:                  P ← subpath(P, n, n_i)
7:                  Q ← subpath(Q, n, n_j)
8:              end if
9:          end for
10:     end for
11:     return {P.last | P ∈ ℙ}
12: end procedure
```

When examining the CFG, the set of next inter-thread cache conflicts $P(n_i)$ from the instruction n_i identify the first reachable conflicts on any valid path from n_i. Thus, any node on any valid path from n_i up to (but not including) those of $P(n_i)$ cannot conflict. Furthermore, all nodes in this set are weakly connected to n_i and no edge is excluded from G. The set of nodes and edges satisfies the requirements of a CFR with initial instruction n_i. These observation allows the set of inter-thread cache conflicts to serve as boundaries of CFRs.

Utilizing the boundary property of next inter-thread cache conflicts, extraction of the complete set of conflict free regions from the CFG is an iterative process. Starting with the entry instruction from the CFG, the set of next inter-thread conflicts bounds the initial CFR. To extract the CFR F, nodes and edges are added to F by a depth first search from the entry instruction h^G halting at instruction $n_x \in P(h^G)$. Subsequent CFRs are created by using the set of next inter-thread conflicts as entry instructions. The process is repeated

until the terminal instruction of the CFG is reached.

In Figure 5.6 the extraction of the initial CFR is illustrated. The next inter-thread cache conflicts $P(h^G)$ are $\{n_i, n_j, n_k\}$ which are not included in the initial CFR $F = (N, E, h \leftarrow h^G)$ of the CFRG R. After extraction, F is added to R. Edges are added to R from F to each of its successors identified by their entry instruction n_i, n_j, n_k. The process is repeated using the next inter-thread cache conflicts as entry instructions of successor CFRs until each path through the CFG reaches the terminal instruction.

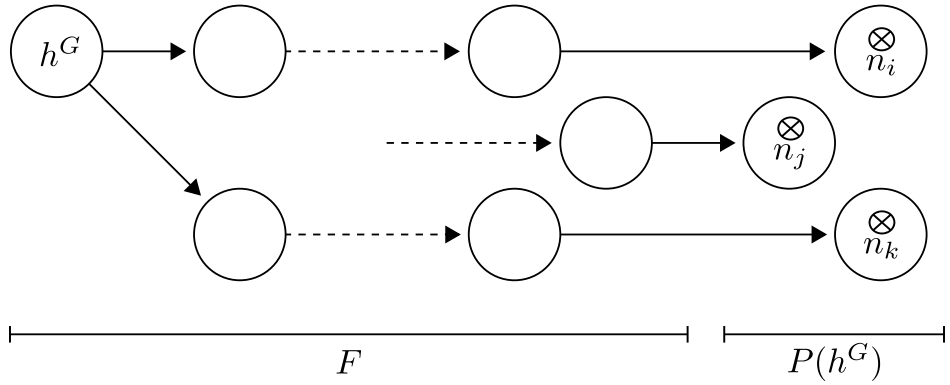


Figure 5.6: Extraction of the initial CFR from the CFG

5.2.2 Worst Case Execution Time with Cache Overhead (WCETO)

WCETO analysis for a one ribbon task releasing m threads per job depends on the structure of the program, conflict free regions, `BUNDLE`'s scheduling decisions and the conflict free region graph. The result is a bound $c(m)$ for all m threads to complete their execution.

Graphical Notation

In Figure 5.7, the CFG of a ribbon is given with entry instruction h and terminal instruction z. CFRs have been extracted from the CFG and placed in the CFRG below. The figure uses a graphical notation that is consistent within this work, instructions in the CFG are

circular nodes, and CFRs in the CFRG are square nodes[1]. Additionally, the shorthand of aligning instructions of the CFG with their CFRs in the CFRG is used throughout this work. This mapping is acceptable when treating each node n_i as the value of its main memory address a_i, therefore n_i in the CFG refers to the instruction at address a_i and the node n_i in the CFRG refers to the CFR beginning with the instruction at address a_i.

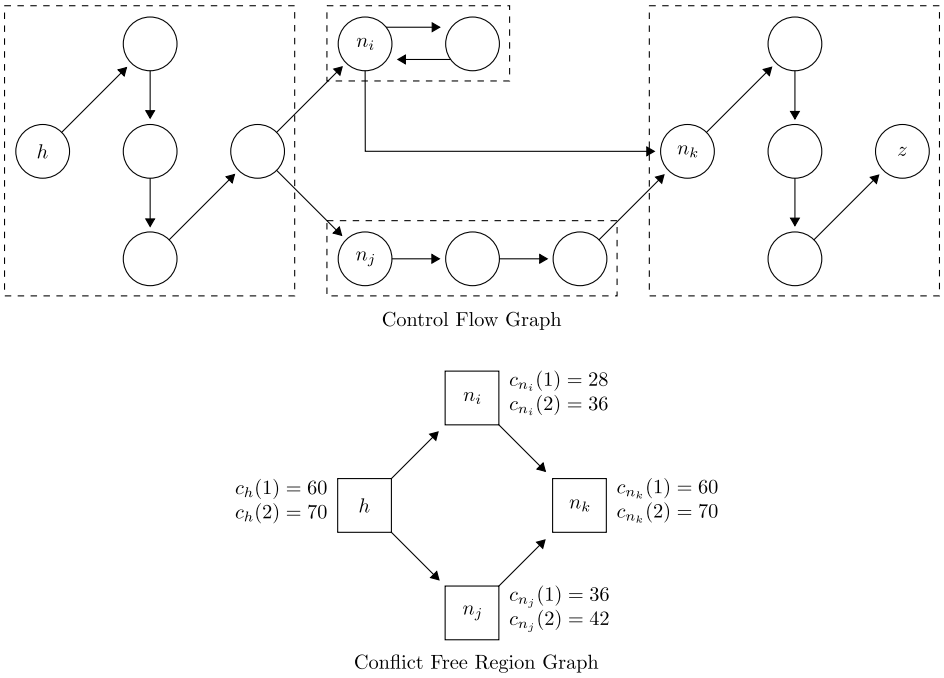


Figure 5.7: CFG to CFRG with WCETO Values

Next to each CFR of the CFRG are two WCETO values, the first for a single thread, the second for two threads. The parameter m to the per node function $c_n(m)$ is the number of threads assigned to execute over the node as scheduled by `BUNDLE`. The value of these functions depend on the structure of the CFR (definitions of structures are provided later in this section). Each $c(m)$ function is used as a portion of the per path WCETO calculation. A path π through the CFRG always begins with h and terminates with the CFR containing z, in Figure 5.7 there are two possible paths $\pi = \langle h, n_j, n_k \rangle$ or $\pi = \langle h, n_i, n_k \rangle$.

[1]Hexagons will be used for summary nodes CFRs in later sections

Given any path π and a number of threads assigned to that path m, the WCETO for a path π is $c_\pi = \sum_{n \in \pi} c_n(m)$. The set of all distinct paths through the CFRG is denoted Π, which serves as support for the worst-case selection of paths multiset Π_S. Elements of Π_S are the distinct paths and the number of threads assigned to them $\langle \pi, m \rangle$. The cardinality of Π_S is m, $|\Pi_S| = m$. A path π_i with n threads assigned to it is denoted $\pi \in^n \Pi_S$. For Π_S, the completion of all m threads is bounded by the following equation.

$$c(m) = \sum_{\pi_i \in \Pi_S} \sum_{n \in \pi_i} c_n(k \mid \pi_i \in^k \Pi_S)$$

The worst-case selection of paths Π_S is the set that produces the greatest $c(m)$ value. This entails searching the space of all paths for all possible assignments of m threads. No heuristic for finding the worst-case selection is known at this time and remains an open topic of research.

$$\Pi_S = \underset{\Pi_S \subseteq \Pi}{\operatorname{argmax}} \left\{ \sum_{\pi_i \in \Pi_S} \sum_{n \in \pi_i} c_n(k \mid \pi_i \in^k \Pi_S) \right\}$$

To illustrate, the example in Figure 5.7 is extended in Figure 5.8. It includes four WCETO of increasing m values (the number of threads per job) where m is in the range $[1, 4]$. There are two paths through the CFRG R, labelled π_1 and π_2. As the number of threads are increased the selection of worst-case paths changes. For π_1 the WCETO of one thread is $c_{\pi_1}(1) = 156$, compared to $c_{\pi_2}(1) = 148$ thus $\Pi_S = \{\pi_1\}$. For $m = 2$, the candidates for Π_S are $\{\pi_1, \pi_1\}$, $\{\pi_1, \pi_2\}$, and $\{\pi_2, \pi_2\}$ with $c(2)$ values of 182, 304, and 164 respectively; thus $\Pi_S = \{\pi_1, \pi_2\}$. For three and four threads, the incremental cost of π_2 is greater for each thread assigned to the paths.

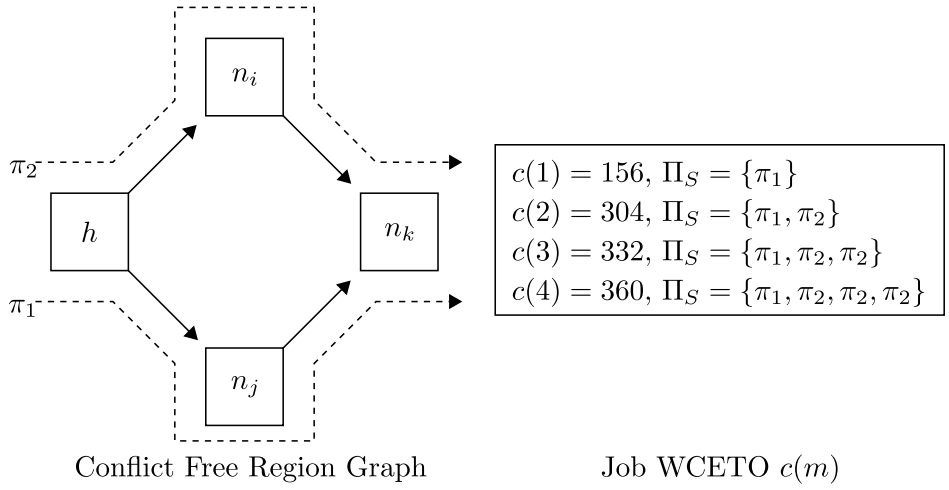


Figure 5.8: WCETO from CFRG

Through this example BUNDLE's sub-optimal behavior is exposed. The selection order of bundles is arbitrary, therefor multiple threads may be scheduled over the same CFR without benefiting from cache reuse. This behavior results in a greater WCETO bound, which can be seen in Figure 5.8 by considering the effect of coordinating bundle execution such that n_k is activated only once.

Central to WCETO calculation are the CFR WCETO bound functions (the $c_n(m)$ functions). To create these functions, some assumptions are made about each CFR, several of which are guaranteed by extraction. Other requirements come from the model and BUNDLE's scheduling algorithm. The complete set of assumptions are listed below.

1. Any CFR $F = (N, E, h)$ has a single entry instruction h.
2. When activated, All m threads of are ready and waiting to execute h.
3. Any thread attempting to execute an instruction $n_x \in P(h)$ is blocked.
4. Preemptions between threads take no time.
5. Loops have pre-determined iteration bounds.

Figure 5.9: CFR Requirements for WCETO Calculation

Individual CFR WCETO functions also depend on the CFR containing a single logical *structure*. There are three types of structures: linear, branching, and looping. Descriptions of the types are given in the following subsection. Each CFR is partitioned by structure into smaller CFRs.

An iterative process similar to the extraction of CFRs from the CFG is taken. The first structure in the CFR $F = (N, E, h)$ is detected and extracted as F_h with a set of boundary instructions K. Each boundary instruction $n_k \in K$ serves as the entry instruction of a successor structure F_{n_k}. Edges are added to the CFRG R between F_h and each of the successors. When the process completes by reaching the end of the CFR, the single node F in R is replaced by a graph of CFRs containing one structure each.

Within this work we see no need to detail the extraction of structures, given the well established techniques of pathfinding [50] and loop detection [51]. Instead, we describe the requirements placed upon those structures necessary for calculating a safe WCETO value.

5.2.3 Structures

Linear Structure: a linear structure begins with an instruction h followed by a set of serial instructions and no branches. The out-degree of any node in the structure is at most one. It terminates at a node z, an instruction which precedes a branching or looping instruction x. The terminal instruction z is within the structure, while the boundary instruction x is without.

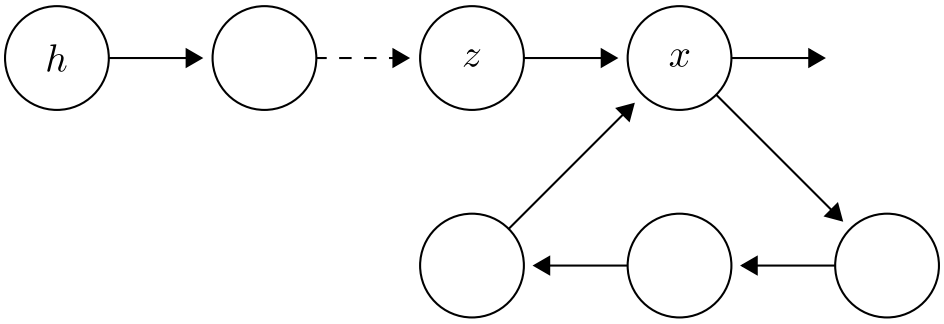


Figure 5.10: Linear Structure from h to z Preceding a Loop

Branching Structure: a branching structure contains at least on node with out-degree greater than one and no loops.

A branching structure terminates at a set of nodes Z. A node $z \in Z$ is defined as a node that precedes a node within a loop, or having out-degree zero. When a node z is determined to be in Z, all outgoing edges are pruned at z. Immediate successors of z are added to the set of boundary instructions $X = \{x \mid (z, x) \in E^G\}$. Terminal instructions are included in the structure while boundary instructions are not.

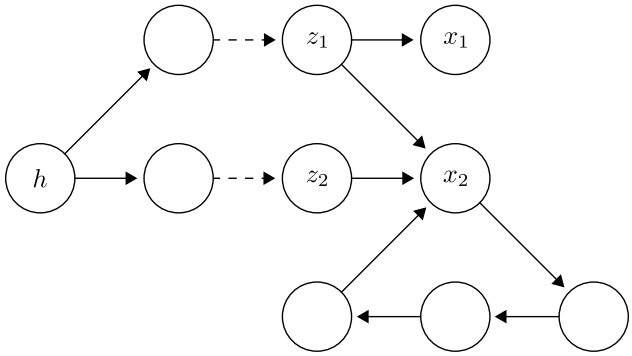


Figure 5.11: Branching Structure from h to $Z = \{z_1, z_2\}$ with Boundary Nodes $X = \{x_1, x_2\}$

Looping Structure: a looping structure contains the nodes of a cycle starting with node h. It contains no nodes outside of the cycle. The structure of the loop is further restricted, no path from a node within the cycle may leave it without passing through h. This restriction is met by precluding `GOTO` and `LONGJMP` instructions. Within the loop, linear, branching and looping structures are permitted.

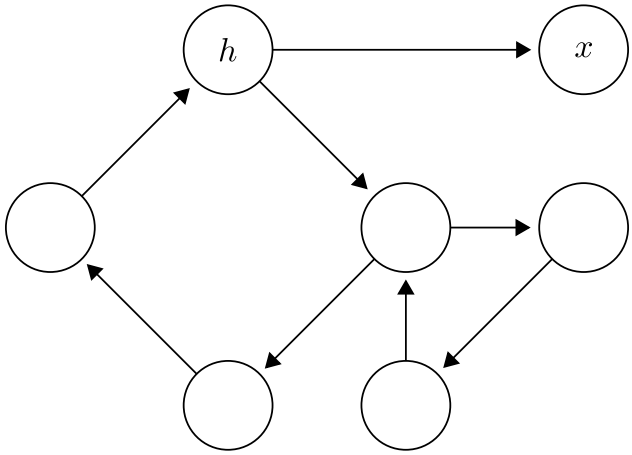


Figure 5.12: Looping Structure with Loop Head h and Boundary Nodes $X = \{x\}$

5.2.4 Structure WCETO Calculation

The remainder of this section is dedicated to the theorems and proofs of WCETO bounds for CFRs of linear, branching, and looping structures. The setting is a single CFR $F = (N, E, h)$, of one structure type, and set of next inter-thread cache conflicts $X = P(h)$.

All m threads are scheduled by BUNDLE, blocked waiting to execute h, and will block attempting to execute any instruction $x \in X$.

Theorem 1 (Eviction-less Execution)**.** *For m threads blocked waiting to execute h of F, an instruction $n \in N$ cannot be evicted during the execution of any thread over the CFR if any thread is blocked before executing an instruction $x \in X$.*

Proof. By definition of a conflict free region, $\forall n_i, n_j \in N \wedge \hat{M}(n_i) \neq \hat{M}(n_j), M(n_i) \neq M(n_j)$. Consider a cached instruction n_i, if the execution of $n_j \in N$ evicts n_i then $\hat{M}(n_i) \neq \hat{M}(n_j)$ Further, $M(n_i) = M(n_j)$ contradicting the definition of a conflict free region. Therefore, n_i cannot be evicted by execution of any $n_j \in N$. □

Corollary 5.2.0.1 (Single Load)**.** *During the execution of F by BUNDLE, any instruction $n \in N$ can be loaded into the cache no more than once for any number of threads.*

Time Bound for Linear Structures: When F contains a single linear structure with entry instruction h and terminal instruction z, there is a single path $\pi = \langle h, ..., z \rangle$. The length of this path is referred to as $L = |\pi|$.

Theorem 2 (WCETO for Linear Structures)**.** *When F contains a single linear structure with terminal instruction z and m threads waiting to execute h, an upper bound on the execution time from h to z for all threads is: $c_h(m) = L(\mathbb{I} \cdot m + \mathbb{B})$.*

Proof. Each of the m thread executes L instructions since there are no alternative paths from h to z. By Corollary 5.2.0.1, at most one of the m threads will cache each of the L instructions taking $L \cdot \mathbb{B}$ time. Execution of L instructions by m threads takes $L \cdot \mathbb{I} \cdot m$ time. Combining the time required to cache and execute, yields the bound of $c_h(m) = L(\mathbb{I} \cdot m + \mathbb{B})$. □

Time Bound for Branching Structures: When F contains a single branching structure it has an entry instruction h and set of terminal instructions Z. With multiple paths $\pi = \langle h, ..., z \rangle$, where $z \in Z$. The length of the longest path to any $z \in Z$ from h is referred to as L.

Theorem 3 (WCETO for Branching Structures)**.** *For a conflict free region F with a branching structure and m threads waiting to execute h, an upper bound on the execution time from h to $z \in Z$ for all threads is*

$$c_h(m) = L \cdot \mathbb{I} \cdot m + |N| \cdot \mathbb{B}$$

Proof. From Corollary 5.2.0.1 at most one of the m threads will cache any $n \in N$, the worst possible case is that all $|N|$ instructions are cached taking $|N| \cdot \mathbb{B}$ time. For execution, the worst case is for all m threads to execute the longest path of length L taking $L \cdot \mathbb{I} \cdot m$ time. Combining the bounds produces: $c_h(m) = L \cdot \mathbb{I} \cdot m + |N| \cdot \mathbb{B}$. □

Timing Bound for Looping Structures: When F contains a single looping structure it has an entry instruction h and a predetermined bound on the number of iterations I_h. There may be multiple distinct cycles from h to h, among these the one with the longest path is referred to as L.

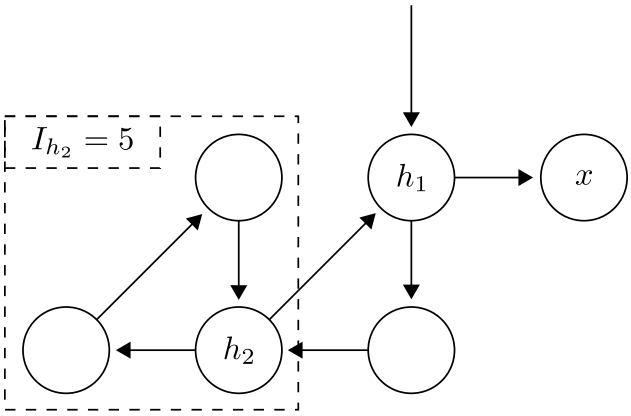


Figure 5.13: Embedded loop of h_2 within h_1

When an embedded loop is present the contribution to any path is the product of the longest path through the embedded loop and the number of iterations. In Figure 5.13, h_2 is embedded in h_1. The longest (only) cycle in

h_2 is 3, with at most 5 iterations and contributes 15 cycles to the longest path of h_1.

Theorem 4 (WCETO for Looping Structures)**.** *For a conflict free region F with a looping structure and m threads waiting to execute h, an upper bound on the execution time for all threads to complete I_h iterations is given by*

$$c_h(m) = I_h \cdot m \cdot L \cdot \mathbb{I} + |N| \cdot B$$

Proof. Consider the execution and caching of instructions separately. Since L is the longest path through the cycle and, one cycle executed by one thread can take no more than $L \cdot \mathbb{I}$ time. For m threads and I_h iterations the upper bound on execution is $I_h \cdot m \cdot L \cdot \mathbb{I}$.

Cache misses are limited by Theorem 1, since F is conflict free, no instruction can be evicted during the execution of F. Only the initial load of any instruction into the cache demands consideration. The number of initial loads is bounded by the total number of instructions in the region which takes $|N| \cdot \mathbb{B}$ time. Combining the bounds on execution and caching of instructions result in $c_h(m) = I_h \cdot m \cdot L \cdot \mathbb{I} + |N| \cdot B$ □

Special Cases for Looping Structures: Theorem 4 assumes looping structures are contained within a single conflict free region. Linear and branching structures may be divided at boundaries defined by next inter-thread conflicts. However, for looping structures, this is not always the case. To derive a bound for loops containing inter-thread cache conflicts, the concept of bounded inter-thread cache conflicts is introduced.

A **bounded inter-thread cache conflict** from a given instruction $n \in N^G$ up to and including $z \in N^G$ is an inter-thread cache conflict on a valid path from $\pi = \langle n_i, ..., z \rangle$.

Bounded Inter-Thread Cache Conflicts: Are the set of all possible x values that are a

bounded inter-thread cache conflict from n to z. The set is given by $P(n, z)$. No algorithm is given for $P(n, z)$, since it requires two small modifications to $P(n)$, 1.) accept and pass a bound to POC(n) 2.) return all conflicts of all paths found by POC(n) rather than the final element. Only one modification is required of POC(n), to cease searching at the bound rather than at a conflict. These simple modifications do not seem to warrant an additional algorithm.

When a cycle in the CFG G contains an inter-thread cache conflict it cannot be contained within a single CFR, a separate time bound for the cycle must be calculated. Cycles have the restricted form of an entry instruction h with two outgoing edges, one that enters the cycle and another exiting through the boundary instruction x.

The set of bounded inter-thread cache conflicts, calculated by $P(n, z)$, differ from the set of next inter-thread cache conflicts by including all conflicts on all paths $\pi = \langle n, ..., z \rangle$. Utilizing the initial instruction h as the start and bound for inter-thread cache conflicts $P(h, h)$produces the set of all conflicts within the loop starting with h. These are used in calculation of bound for m threads over the looping structure F. As a note during extraction, the set of entry instructions H is increased only by h and x (the boundary instruction of the loop). This differs from the typical extraction which would increase H by the inter-thread cache conflicts $P(h)$.

An additional concept is required to complete the bound calculation, that of maximum per iteration invocation. For an instruction $n \in N$ of a region $F = (N, E, h)$ that is a looping structure, the **maximum per iteration invocation** of n given by $n_{\max}$ is the greatest number of times n may be executed during a single cycle starting with h. The value is the product of the maximum iterations of the embedded loops n belongs to. For example, if n

belongs to the region F with initial instruction h, and also belongs to the embedded loops h_1 and h_2 then $n_{\max} = I_{h_1} \cdot I_{h_2}$.

To ease the bound calculation, for a set of nodes P, the operator $\lceil P \rceil$ is defined as follows. Note, this definition will not be reused outside of Theorem 5.

$$\lceil P \rceil = \sum_{n \in P} n_{\max}$$

Theorem 5 (WCETO For Special Case Looping Structures)**.** *For a CFR F with a looping structure that contains inter-thread cache conflicts with m threads waiting to execute h, and longest path L an upper bound on the execution time for all threads to complete I_h iterations of the cycle is given by*

$$c_h(m) = \mathbb{B}(|N \setminus P(h,h)|) + I_h \cdot m \left(L \cdot \mathbb{I} + \mathbb{B} \cdot \lceil P(h,h) \rceil\right)$$

Proof. Consider the time to cache all instructions of the loop separately from the time to execute a single iteration. The product of the block reload time and number of instructions $|N \cdot \mathbb{B}|$ bounds the time to populate the cache.

For a single iteration of a loop by a single thread, the execution time is bounded by $L \cdot \mathbb{I}$, for all m threads $m \cdot L \cdot \mathbb{I}$. In any iteration, an inter-thread conflict instruction x may be evicted at most $x_{\max}$ times because x may be evicted by another instruction in the closest embedded loop to which x belongs. Therefore, in one iteration, a single thread will incur at most $\lceil P(h,h) \rceil$ evictions.

Combining the execution time, block reloads, and iterations produces the time bound of $I_h \cdot m \left(L \cdot \mathbb{I} + \mathbb{B} \cdot \lceil P(h,h) \rceil\right)$ of executing I_h iterations of the loop after all instructions

are cached. Before incorporating the time to populate the cache, the double counting of $|P(h,h)|$ reloads are subtracted from $|N|$. Summing the time to populate the cache and iterate over the loop for m threads yields the bound

$$c_h(m) = \mathbb{B}(|N \setminus P(h,h)|) + I_h \cdot m \left(L \cdot \mathbb{I} + \mathbb{B} \cdot \lceil P(h,h) \rceil\right)$$

□

5.3 Evaluation of BUNDLE

The positive perspective taken by BUNDLE has been compared to the classical approach using two methods. The first method performs a static WCET and WCETO analysis of a single multi-threaded program. The second examines BUNDLE's run-time performance and overhead costs. Both analysis are performed in the same setting, CPI of one, BRT of ten, for a direct mapped instruction cache with block size of one instruction. Additionally, context switch costs for threads or tasks are not considered. To state explicitly, any context switch takes zero cycles. Figure 5.14 summarizes the shared evaluation parameters.

Block Size	1 (word or instruction)
Context Switch Cost	0 cycles
$\mathbb{I}$	1 cycles
$\mathbb{B}$	10 cycles

Figure 5.14: BUNDLE Evaluation Parameters

5.3.1 WCET vs WCETO Analysis

To compare the classical methods of Arnold [1] and Mueller [2] to BUNDLE, a parallel program written using the POSIX Thread library (pthread) is analyzed. The program

`ppi.c`, is a multi-threaded estimator of the ratio of a circle's circumference to its diameter (π). Full source is provided as Listing 5.1.

Listing 5.1: `ppi.c` a Multi-Threaded π Estimator Using `PTHREAD`

```
#define M 150
#define L 10000

void *part(void *count) {
        double x, y, d;
        int i;
        long *c = (long *) count;
        *c = 0;

        for (i = 0; i < L; i++) {
                x = rand() / (double) RAND_MAX;
                y = rand() / (double) RAND_MAX;
                d = sqrt(x * x + y * y);
                if (d <= 1) {
                        (*c)++;
                }
        }
        pthread_exit((void *) c);
}

int main (int argc, char *argv[]) {
        for (t = 0; t < M; t++) {
                pthread_create(&threads[t], NULL, part,
                    (void *)&count[t]);
        }
        total = 0;
        for (t = 0; t < M; t++) {
                pthread_join(threads[t],
                    (void *) &found);
                total += *found;
        }
        pi = (double) 4 * total / (M * L);

        printf("M:%i L:%i pi =~ %0.05f\n",
               M, L, pi);
        return 0;
}
```

Initialization and accumulation are handled by the `main` function in the initial thread. The initial thread's contribution to execution time and cache contents is constant and is ignored by the analysis. Only the object code of the `part` function is analyzed, representing a ribbon ρ for which m threads execute per job release.

The flow of ρ is divided into three sections, a prologue, loop body, and epilogue. After compilation, the prologue and epilogue are serialized sets of instructions corresponding to linear structures. However, the loop body contains a conditional resulting in a branching structure within a looping structure.

Analysis of the sections by the methods of Arnold [1] and Mueller [2] produce a WCET

value in terms of m, $\mathbb{I}$, and $\mathbb{B}$. Using these methods, all instructions of the prologue and epilogue are categorized as "never cached". Instructions within the loop body are categorized as "first miss" where some "may" be cached and others "must" be. To favor the classical approach, the number of cache misses are reduced by re-classifying all "may be cached" instructions as "must be cached". Performing the analysis yields a bound of c_1 found in Equation 5.3.1 for one thread to execute the object of ρ with a maximum of i iterations of the loop.

$$c_1 = 86 \cdot \mathbb{I} \cdot \mathbb{B} + 35 \cdot \mathbb{I} + (i-1)(100 \cdot \mathbb{I} + 8 \cdot \mathbb{B}) \tag{5.3.1}$$

Figure 5.15: c_1: Classical WCET for One Thread to Execute ρ

Similarly, a representative CRPD value for the classical perspecive is determined using the UCB [21] method over ρ. For the prologue and epilogue there can be no useful blocks. However, all instructions within the loop body are useful, they are also evicting. The bound γ_1 is the product of the number of UCBs and the block reload time, found in Equation 5.3.2. This preemption cost will be incorporated using Lunniss' [29] approach, increasing the WCET of each job by the CRPD value per preemption.

$$\gamma_1 = 65 \cdot B \tag{5.3.2}$$

Figure 5.16: γ_1 Classical CRPD for One Preemption of ρ

To avoid favoring the positive perspective, the number of cache blocks s is selected to ensure the loop body of ρ could not be contained within a single CFR. Doing so requires the loop body to be bounded by Theorem 5 instead of the smaller bound given by Theorem 4. Using a maximum iteration bound of 1,000 for the loop body, the results of increasing the number of threads m in the classical analysis and `BUNDLE` are compared in Figure 5.17.

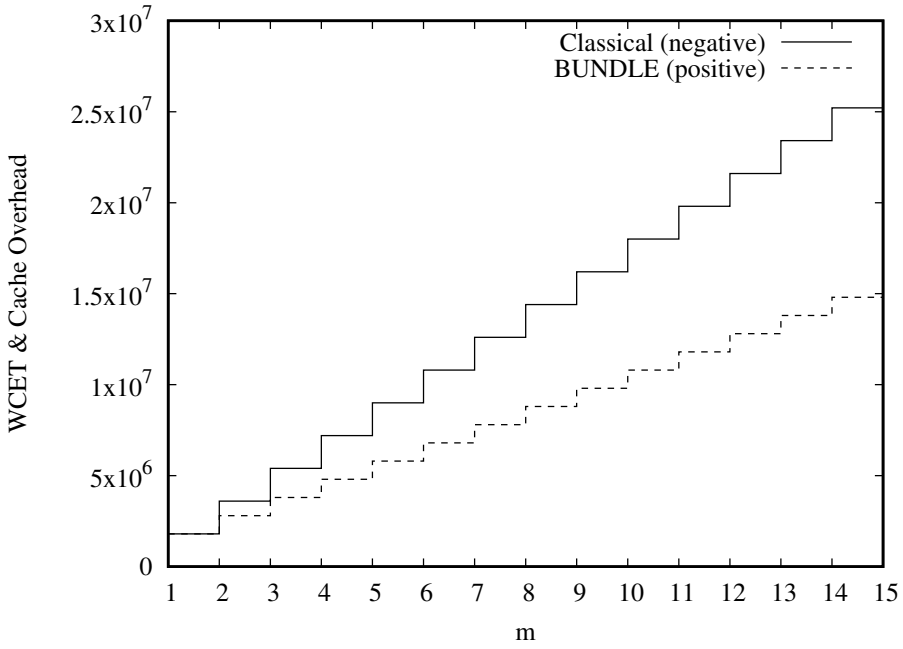


Figure 5.17: Comparison of WCET and WCETO for m threads and i = 1,000

In Figure 5.17, the solid line is product of the classical WCET bound and the number of threads, i.e. $c_1 \cdot m$. The dashed line is the WCETO value for BUNDLE scheduling, i.e. $c(m)$ calculated from Theorems 2, 3, and 5. For this ribbon as the number of threads increases, the difference between the WCET and WCETO value increases. This illustrates the analytical benefit of BUNDLE's positive perspective and cache reuse across threads.

Figure 5.18 compares the impact of preemptions to the classical approaches WCET to BUNDLE's WCETO value. The number of threads m is fixed to two, and the number of preemptions increased. Using the Luniss [29] approach each preemption increases the WCET for a task. For BUNDLE preemptions are restricted and accounted for in the WCETO, which is why the total execution time remains constant and below the classical analysis.

Preemptions		1	5	10	15
Method	Classical	181365	183965	187215	190465
	BUNDLE	101143			

Figure 5.18: WCET + Preemption Cost When $i = 10000$

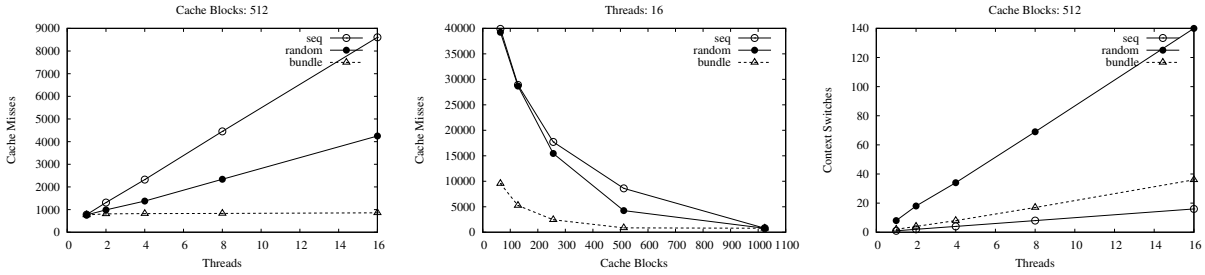
5.3.2 BUNDLE Run-Time Performance

To complement the analytical comparison, BUNDLE's scheduling algorithm is compared to two different thread scheduling algorithms. The first mirrors the classical approach, running threads to completion one after another, sequentially. This first scheduling algorithm is named `seq`. The second scheduling algorithm is named `random`, which executes a random number of instructions from one thread before preempting it with another.

Each scheduler is implemented as part of a path tracing simulator for synthetic programs. The path trace simulator is available on github at http://github.com/ctessler/pathsim.git. It takes generated programs as input, tracing the execution of multiple threads using one of the three scheduling algorithms and produces a count of instructions executed without a pipeline. When executing a branching instruction, the branch to take is randomly selected. When encountering a looping instruction, the maximum number of iterations is randomly selected from zero to the analytical loop bound.

Programs generated for the simulator were based on characteristics of the Mälardalen Real-Time Research Center's WCET benchmark suite [52]. Averages of number of branching statements, looping directives, program length, and basic block sizes were taken from the compiled objects of the set of benchmarks. The program generator uses these parameters in a Guassian distribution to generate 100 programs. Each program is executed for a given cache size s in the range of $s = (64, 128, 256, 512)$, number of threads

Figure 5.19: Run Time Overhead Results



(a) Misses vs. Threads (b) Misses vs. Size (c) Threads vs. Switches

$m = (1, 2, 4, 8, 16)$, and scheduler: (`BUNDLE`, `seq`, `random`). For the total combination of 7,500 program runs, the results are averaged and presented in Figures 5.19 and 5.20.

The run-time performance of `BUNDLE` in terms of cache misses dominates both `seq` and `random` in all circumstances. It strictly dominates (is always less than) both when the number of threads is greater than one, and the cache size is smaller than the program length. There are two circumstances for which the cache miss results are ignored, for one thread, and when the cache size is greater than the program length. Scheduling a single thread requires no scheduling decisions, each of the algorithms will exhibit the same number of misses. When the cache size is greater than the program length (e.g. $s = 1024$) the number of misses is limited to the number of instructions in the program and does not change between scheduling algorithms.

Cache miss rates favor `BUNDLE` most when the cache size is roughly two thirds the average program length ($s = 512$) shown in Figure 5.19a. For `BUNDLE` as the number of threads is increased, the number of misses remains constant. However, the other scheduling algorithms increase their misses with each thread. As the number of threads increase, so too does `BUNDLE`'s benefit.

Conversely as the cache size is increased in Figure 5.19b the benefit of `BUNDLE` de-

creases. For a fixed number of threads, increasing the number of cache lines benefits all algorithms. While BUNDLE maintains lower total cache misses, the relative benefit naturally decreases with the cache size.

BUNDLE's lower miss rate comes at a cost: an increased number of thread-level context switches, shown in Figure 5.19c. Although thread-level context switches are (by design) far less costly than process-level (job-level) context switches the cost of BUNDLE scheduling warrants consideration.

Cache Lines	64	128	256	512	1024
Threads					
1	0	0	0	0	0
2	248	413	697	1372	0
4	372	618	1046	2057	0
8	434	719	1210	2284	0
16	464	770	1296	2447	0

Figure 5.20: Minimum Context Switch Cost (in Cycles) for seq to Dominate BUNDLE

Figure 5.20 presents a bound on the thread-level context switch cost derived from the simulation results. The individual cell values represent the minimum number of cycles a single thread-level context switch must cost which for seq to execute and complete in the same number of cycles as BUNDLE. The values are computed by taking the difference in cache misses between the two schedulers and dividing the difference by the block reload time $\mathbb{B} = 10$.

Considering the thread-level context switch costs in terms of the synthetic programs, each of the 100 programs averaged 7,900 instructions of execution. The worst-case configuration of 64 cache blocks and two threads results in thread-level context switch costs

above 248 instructions for `seq` to outperform `BUNDLE`; roughly 3% of a thread's execution time. For the best-case, thread-level context switches would have to exceed 30% of the average execution time! These thread-level context switch values appear unreasonably high; therefore, the cache miss reduction of `BUNDLE` outweighs the increase in thread-level preemptions.

5.4 Summary

These initial experiments demonstrate the benefit of `BUNDLE`'s positive perspective of instruction caches for a single multi-threaded program. The classical perspective's pessimistic WCET bounds are due to its inability to account for the inter-thread cache benefit. Instead, caches are seen only as detractors in WCET bounds.

The run-time performance of the positive perspective also favors `BUNDLE` over the classical scheduling algorithms for the examined task set. Permitting thread-level context switches to consume up to 3% of a threads execution time before a naive sequential scheduler will outperform `BUNDLE`.

1. A single multi-threaded task
2. Loops are restricted to a single entry and exit point
3. Sub-optimal scheduling with respect to cache sharing and WCETO
4. Intractable WCETO calculation
5. An undefined hardware mechanism for anticipating execution
6. An evaluation limited to synthetic programs

Figure 5.21: Constraints of Single Task `BUNDLE`

However, `BUNDLE` operates in a constrained setting. These constraints limit the applicability of its positive perspective and adoption in deployed systems. Those constraints are summarized in Figure 5.21. As a proof of concept, `BUNDLE` reflects the benefits of the positive perspective and encourages further investigation into its approach.

CHAPTER 6 SINGLE-TASK BUNDLEP

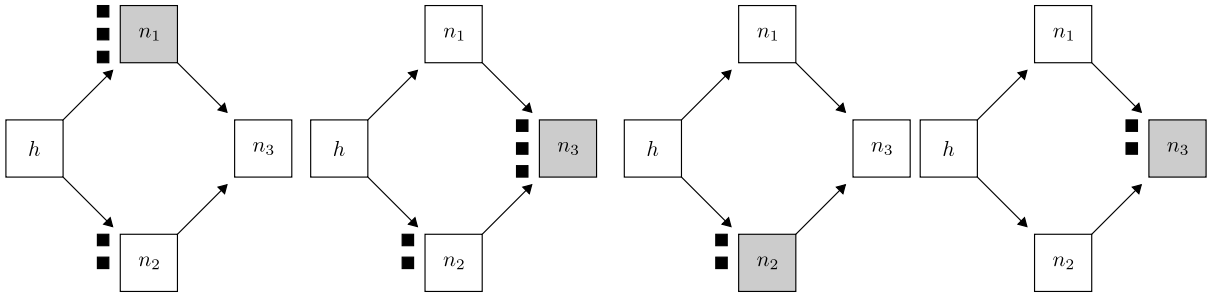
Exploration of the positive perspective continues with BUNDLEP, a modification of BUNDLE's scheduling algorithm with a novel approach to CFRG creation and WCETO calculation. Additionally, BUNDLEP's complete approach has been implemented for evaluating programs compiled for MIPS [53] architectures. Summarily, BUNDLEP extends BUNDLE by the following:

1. An evaluation and simulation platform for MIPS programs
2. A WCETO method of suitable complexity
3. Formal proof of optimal scheduling with respect to cache sharing
4. A novel and concrete hardware mechanism proposal
5. A novel CFG representation
6. Algorithms for unambiguous CFR extraction
7. Algorithms for novel CFRG formation
8. Removal of structures from CFR and WCETO analysis
9. Incorporation of context switch costs in WCETO bounds

Figure 6.1: Summary of BUNDLEP improvements

6.1 BUNDLE Sub-Optimal Cache Sharing

One of the problems addressed by BUNDLEP is BUNDLE's sub-optimal scheduling with respect to cache sharing, caused by arbitrary activation selection of CFRs. Figure 6.2 illustrates the sub-optimal behavior over the CFRG of a ribbon. The active CFR (node) is shaded light gray and the small black squares next to each node are the threads within the bundle of the associated CFR.



(a) n_1 active (b) n_3 active (c) n_2 active (d) n_3 active

Figure 6.2: Sub-Optimal BUNDLE Execution

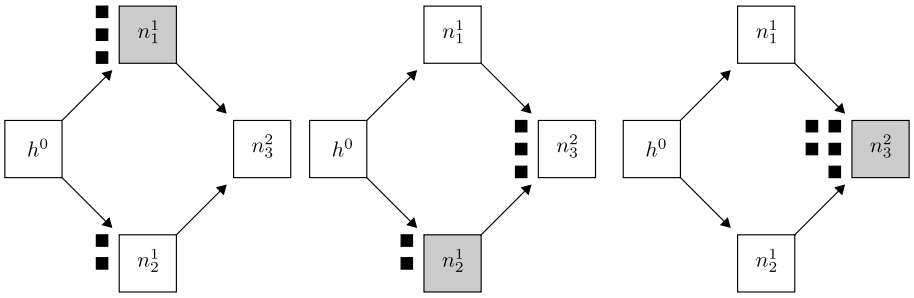
Moving from left to right follows the sequence of bundle activations in Figure 6.2. In 6.2a n_1 is active and when depleted its three threads are blocked waiting on n_3. At this point, the next potentially active bundle is n_2 or n_3. Since the BUNDLE scheduling algorithm selects the next bundle arbitrarily, n_3 is a valid choice in 6.2b. There is only one choice for the active bundle in 6.2c and 6.2d, n_2 then n_3.

The result of arbitrary selecting n_3 in 6.2b instead of n_2 is that n_3 is activated a second time in 6.2d. Two activations of n_3, one for three threads and a second for two threads has a higher WCETO bound than one activation for five threads. The bound for two activations is greater because the time required to cache all instructions of n_3 must be included twice. In other words for the WCETO bound of n_3, $c_3(m)$ for m threads: $c_3(3) + c_3(2) > c_3(5)$.

6.2 BUNDLEP Overview

Arbitrary selection of the active bundle has the additional deleterious effect of increasing the complexity of WCETO analysis, due to the number of paths being increased by potential multiple activations of individual CFRs. BUNDLEP addresses both the sub-optimal cache sharing and WCETO complexity with a simple solution: assign priorities to CFRs in the CFRG. At run-time, a bundle inherits the priority of its associated CFR. The bundle with the best (lowest) priority is always activated. A detailed description of BUNDLEP's scheduling algorithm is given in Section 6.4.

Assigning priorities to CFRs in a manner that guarantees the minimum number of activations maximizes the inter-thread cache benefit of each activation. Details of constructing such a priority assignment are provided in Section 6.4.3. Intuitively, the priority assignments are based on their distance to the terminal CFR. The terminal CFR has the highest priority, nodes immediately preceding the terminal CFR have the second highest, etc.



(a) n_1 active (b) n_2 active (c) n_3 active

Figure 6.3: Optimal BUNDLEP Execution

Figure 6.3 illustrates one such assignment of priorities based on Figure 6.2. The priority of each node is given as a superscript. Node h^0, as the initial CFR has the best priority 0, n_3^2 the worst priority 2. In 6.3a, n_1 and n_2 are the only bundles with threads waiting to

execute. Both CFRs have equal priority, selecting either one is valid. In the example, n_1 is selected. There are no decisions available in 6.3b or 6.3c, the activations are set by the priorities of n_2 and n_3. The result is n_3 is activated once, halving the cache loads in the analysis and (potentially) at run-time.

A priority assignment that minimizes the number of activations of a node n_i is one where the priority ϖ_i of n_i is equal to its longest path in the CFRG from h; where the longest path is determined by the number of edges traversed in the CFRG from h to n_i. Theorem 6 provides a proof of minimal CFR activations for CFRG which is a directed acyclic graph (DAG). Being applicable only to CFRGs that are DAGs requires `BUNDLEP`'s extraction of CFRs and CFRGs to differ from `BUNDLE`'s. Using novel methods of CFR extraction and CFRG formation to guarantee a DAG, the assignment of priorities to CFRs is performed in polynomial time, with the added benefit of reducing the WCETO complexity.

Theorem 6 (Maximum Bundle Activations)**.** *For a CFRG $R = (N, E, h)$, which is a DAG, where each node $n_i \in N$ has priority ϖ_i equal to the length of the longest path from h to n_i the bundle n_i will be activated at most once per job using `BUNDLEP`.*

Proof. To illustrate a contradiction assume a CFR n_i is activated more than once. Then there must exist a node n_j with a higher priority $\varpi_j > \varpi_i$ on a valid path $\langle h, ..., n_j, ..., n_i, ... \rangle$. Given that R is a DAG, there can be no path from n_i to n_j. Since priorities are assigned equal to the longest path from h to a node, then $\varpi_j < \varpi_i$ contradicting $\varpi_j > \varpi_i$. Therefore, n_i can be activated at most once. □

6.3 Conflict Free Region Extraction and Conflict Free Region Graph Creation

An optimal priority assignment based on the longest path requires the CFRG be a DAG[1]. Additionally, the process of converting a ribbon's executable object into a CFG, then CFRs, and finally a CFRG must not introduce loops or ambiguity. To meet these two requirements, `BUNDLEP` divides the analysis of ribbons into two stages: 1.) create an expanded CFG 2.) create CFRs from the expanded CFG and link them in the CFRG. The following two subsections are dedicated to the separate processes.

6.3.1 Expanded Control Flow Graphs

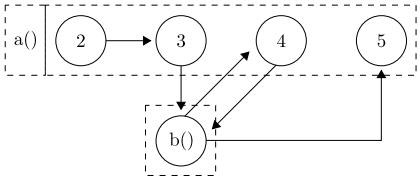
In `BUNDLE` for a CFG $G = (N, E, h)$, a node $n \in N$ is a single instruction identified by its address. Similarly, for `BUNDLEP` nodes of G are a single instruction. However, nodes are not identified by their address but their address and callstack. This prevents loops from being introduced into the CFG.

Common to programs of other hard real-time systems, ribbons are prohibited from including infinite loops, function pointers, `long jmps`, or unbounded recursions. Even with these restrictions in place, it is still possible to introduce loops into the CFG during analysis of a ribbon. Figure 6.4 provides an example where a loop is introduced into the CFG of a linearly structured program.

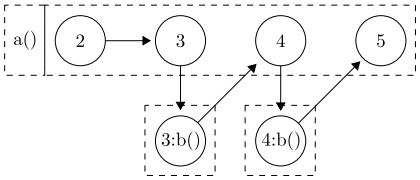
[1]User defined loops also prevent the CFRG from being a DAG, to force the DAG structure loops are collapsed – described in Section 6.4.3

```
1 || procedure a(x, y)
2 ||   c = x
3 ||   x = b(y)
4 ||   y = b(2 * c)
5 ||   return x + y
6 || end procedure
```

(a) Procedure a()



(b) Introduced Loop



(c) Avoiding the Loop

Figure 6.4: Summary of BUNDLEP improvements

For illustration, the number next to each line of the procedure in 6.4a corresponds to the in memory address of the statement. For clarity, each statement is presented as a single instruction in the CFGs found in 6.4b and 6.4c. Figure 6.4b is the CFG that results from each node being identified solely by its address. There are no cycles in the procedure, however the CFG in 6.4b contains one between 4 and `b()`. This cycle is added because there is a call to `b()` by instruction with address 3, and another call by instruction 4. By identifying each node by its address a cycle is introduced. Instead, if the callstack is included in the identification of a node as is the case in 6.4c, no cycle is introduced.

When a nodes of a CFG are identified by their callstack and address it is called an *expanded CFG*. For BUNDLEP all CFG operations take place over the expanded CFG of ribbons. Formally, for a CFG $G = (N, E, h)$, a node $n \in N$ is identified by its address a and callstack s of depth k, where $s = \langle n_1, n_2, ..., n_k \rangle$. Each entry of the callstack is a node in N, where the first node in the stack is the top of the stack – the node calling n's function. For the initial instruction h (or nodes reachable without a function call), the callstack has a single element $\varnothing$ indicating no parenting call.

In comparison to common CFG creation [54] and program analysis [55], creation of an expanded CFG is a straightforward modification of existing approaches. As such, we do not provide a detailed description of expanded CFG creation.

6.3.2 Conflict Free Region Graph Creation

An expanded CFG is the source of extraction for conflict free regions, and the construction of the conflict free region graph. In BUNDLE nodes (instructions) of the CFG could belong to multiple CFRs. Permitting membership in multiple CFRs can introduce cycles and ambiguity into the CFRG. In BUNDLEP nodes are prohibited from participating in multiple CFRs to avoid the introduction of loops and support for a novel tractable WCETO calculation.

Augmenting the requirements of a CFRs and CFRGs from BUNDLE found in Section 5.2, the complete set of BUNDLEP requirements for a CFG $G = (N^G, E^G, h^G)$, CFRG $R = (N^R, E^R, h^R)$, and set of CFRs N^R become:

Requirements of individual CFRs $n_i \in N^R$, $n_i = (N_i, E_i, h_i)$

1. No two instructions (outside of the same main memory block) map to the same cache block
2. All nodes $n \in N_i$ are weakly connected to the entry instruction h_i
3. For any two nodes $n_j, n_k \in N_i$, if there was an edge between them in G then $(n_j, n_k) \in E_i$.

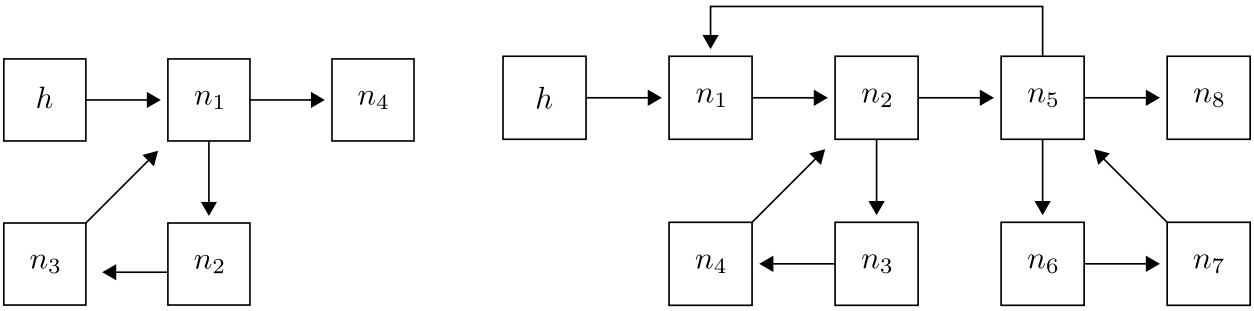
Requirements of CFRGs

1. A node in the CFG $n \in N^G$ is present in exactly one CFR: $\forall n \in N^G,\ \exists n_i \in N^R,\ (n \in N_i \wedge \forall_{k \neq i} n \notin N_k)$
2. Connectivity of the CFG is preserved: $\forall (u, v) \in E^G, u \in N_i \wedge n_i \in N^R \wedge v \notin N_i \implies \exists (n_i, n_j) \in E^R, v \in N_j$

Figure 6.5: Requirements of Conflict Free Regions and Conflict Free Region Graphs for BUNDLEP

In addition to the requirement that each instruction of the CFG is annotated with its address and callstack, nodes of the expanded CFG must also include a reference to their loop heads. All loops must have a head, a starting instruction that includes a condition which determines if the loop will repeat. For CFR assignment as well as WCETO calculation, each node must have its inner-most loop head identified.

Figure 6.6 illustrates the expectations of loop head annotations. Beginning with Figure 6.6a where nodes $\{h, n_1, n_4\}$ do not belong a loop and nodes $\{n_2, n_3\}$ belong to a loop with head n_1. For n_2 and n_3 their inner-most loop head is n_1, the others have no loop head indicated by a $\varnothing$. In Figure 6.6b there are three loops with heads n_1, n_2, and n_5. Loops n_2 and n_5 are embedded within loop n_1. When loops are embedded within one another, a node's *inner-most loop head* is the one closest to the node in the hierarchy. For example, n_7 is a *member* of both loops n_1 and n_5. The inner-most loop head of n_7 is n_5, and n_5's inner-most loop head is n_1. Any suitable algorithm may be used to identify the inner-most loop head of nodes in the expanded CFG such as [56].



Head	∅	∅	n_1	n_1	∅
Node	h	n_1	n_2	n_3	n_4

(a) Single Loop

Heads			n_1	n_1
	∅	n_1	n_2	n_5
Nodes	n_1	n_2	n_4	n_6

(b) Embedded Loops

Figure 6.6: Loop heads and Inner-Most Loop Heads

Given a properly constructed and annotated expanded CFG, conflict free regions can be extracted and assembled into a CFRG. The process of converting the CFG to a CFRG takes two phases 1.) assignment 2.) linking. In the *assignment* phase, nodes of the CFG are assigned to exactly one CFR. In the *linking* phase, CFRs are joined by edges in the CFRG. The following subsections 6.3.3 and 6.3.4 describe those phases.

6.3.3 Assignment

Assignment is responsible for placing each node of the CFG into one CFR. The process is completed by the cooperative efforts of two depth first searches (DFS): TAGCFRS() and LABELNODES(). As the top-level (or outer) search, TAGCFRS() marks all the nodes of the CFG G that are entry nodes of CFRs. The bottom-level (inner) search, LABELNODES(n) identifies those instructions that belong the CFR with entry instruction n, it also returns a set of conflicts which TAGCFRS() will use to continue its search.

Algorithm 5 TAGCFRS()

```
 1: G = (N, E, h)                          ▷ Expanded CFG G
 2: C                                      ▷ Simulated Cache
 3: procedure TAGCFRS
 4:     s.clear()                          ▷ Local stack
 5:     v.clear()                          ▷ Visited node array
 6:     s.push(h)                          ▷ Starting node
 7:     while not s.empty() do
 8:         n ← s.pop()                    ▷ Take a node
 9:         v[n] ← true                    ▷ Mark the node as visited
10:         C.clear()                      ▷ Reset the cache
11:         X ← LABELNODES(n)              ▷ Label CFR nodes
12:         for x ∈ X do
13:             s.push(x) if not v[x]      ▷ Conflict begins a CFR
14:         end for
15:     end while                          ▷ v[n] = true indicates n is a CFR entry.
16: end procedure
```

Pseudocode for TAGCFRS() is provided by Algorithm 5. It uses a simulated cache object

C (identical to the one used in Algorithm 2) with methods insert(), clear(), present(), and conflicts(). The procedure is similar to DFS, starting with the initial node of the CFG being pushed to the local search stack s. During each iteration of the while loop, a node n is popped from the search stack and marked as visited in the array v. Being marked by TAGCFRS() indicates that n is an entry instruction of a CFR. After being marked, n is processed by LABELNODES(n). A typical DFS would push the successors of n to the search list before the next iteration. This is where TAGCFRS() differs.

On Line 11, the call to LABELNODES(n) identifies the nodes that belong to the CFR F with initial instruction n and returns a set of nodes X that begin subsequent CFRs. Only those $x \in X$ which have not been visited are added to the search list instead of the immediate successors of n.

By definition, CFRs cannot contain conflicts. For a node n_i with an immediate successor n_x that conflicts in the cache, n_x cannot belong to the same CFR as n_i. Since n_x belongs to a different CFR than n_i and it is the first reachable instruction from the CFR n_i belongs to, n_x must be the entry instruction of a subsequent CFR. It is this observation that guides the behavior of LABELNODES(n).

An example of LABELNODES(n_3) is given in Figure 6.7. In the CFG of the figure, the subscript identifies the node, and the value below the node is the cache block it maps to, e.g. $\underset{2}{n_6}$ maps to block 2. For node n_3, the entry instructions of subsequent CFRs is $X = \{n_7, n_8, n_9\}$. This determination is made because n_7 and n_9 conflict with n_5 through cache block 3, while n_8 conflicts with n_3 through cache block 4.

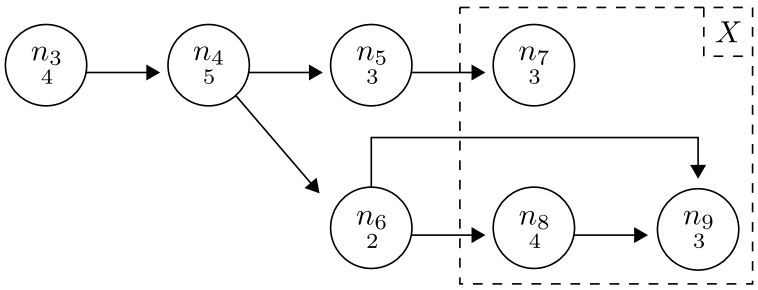


Figure 6.7: Call to LABELNODES(n_3)

In addition to return the set of entry nodes of subsequent CFRs, LABELNODES() labels nodes of the CFG with the CFR they belong to. A CFR $F_i = (N_i, E_i, n_i)$ can be identified by its entry node n_i. In Figure 6.7 the nodes $\{n_3, n_4, n_5\}$ are labeled with their CFR n_3. Pseudocode for LABELNODES() is provided by Algorithm 6.

Algorithm 6 LABELNODES()

```
1:  G = (N, E, h)                                   ▷ CFG G, shared with TAGCFRS()
2:  C                                               ▷ Simulated cache, shared with TAGCFRS()
3:  procedure LABELNODES(n)
4:      s, x                                        ▷ Local stacks (not shared with TAGCFRS())
5:      v                                           ▷ Local visited array (not shared with TAGCFRS())
6:      if n.label ≠ ∅ then
7:          ℓ ← n.label                             ▷ Breaking an existing CFR
8:      end if
9:      s.push(n)
10:     while not s.empty() do
11:         u ← s.pop()
12:         if (n.isHead() ∧ not n.inLoop(u)) ∨
13:                                                 ▷ Case 1, Loop Exit
14:             (u.isHead() ∧ u ≠ n) ∨
15:                                                 ▷ Case 2, Loop Head
16:             (u.label ≠ ∅ ∧ u ≠ ℓ) ∨
17:                                                 ▷ Case 3, Already Assigned
18:             (C.conflicts(u.a))                  ▷ Case 4, Cache Conflict
19:              then
20:             x.push(u)                           ▷ Push the Conflict
21:             v[u] ← true                         ▷ Skip u's successors.
22:         end if
23:         next while if v[u]                      ▷ Already visited
24:         v[u] ← true
25:                                                 ▷ Case 5, Add to CFR n
26:         u.label ← n                             ▷ Label u with CFR n
27:         C.insert(u.a)                           ▷ Insert u into the Cache
28:         for y ∈ G.succ(u) do
29:             if not v[y] then
30:                 s.push(y)
31:             end if
32:         end for
33:     end while
34:     return x
35: end procedure
```

The procedure is a DFS, with a search stack s initially populated with the provided node n that begins a new CFR. During each iteration of the while loop on Line 10 a candidate node u is popped from the stack. Within the body of the loop u will be classified as a

member of the CFR starting with n or not a member. If u is not a member, it is called a conflict and begins a subsequent CFR

Four conditions lead to u being deemed a conflict. The most straight-forward case is on Line 18, if inserting u into the cache would evict another value in the cache then clearly u is a conflict. The three remaining cases are more subtle, but each is necessary to maintain the DAG structure of the CFRG and avoiding the introduction of spurious loops.

Case 1 and 2 address user defined loops; those programmed by the user into the ribbon. When n is a loop head u must belong to the loop to be within the same CFR. Case 1 will be true if n is a loop head and u is not a member of n's loop. It may be that u is loop head (and is not n), Case 2 ensures u begins a separate CFR. To maintain the CFRG DAG structure loops are collapsed into summary nodes (described in Section 6.4.3). To be able to collapse loops into summary nodes, loop heads must be the entry node of CFRs and CFRs must only contain nodes of the same loop.

Case 3 is more complex than the others because it ensures that CFRs are weakly connected while preventing spurious loops from being added to the CFRG. WCETO calculation requires that CFRs be weakly connected. Due to the structure of programs and the nature of conflicts, it is possible for the same node to be reached on multiple paths and potentially assigned to a new CFR. Case 3 only applies to nodes that have already been labeled. If the candidate node u has already been labeled and the label differs from n's label then u must belong to a different CFR and is a conflict.

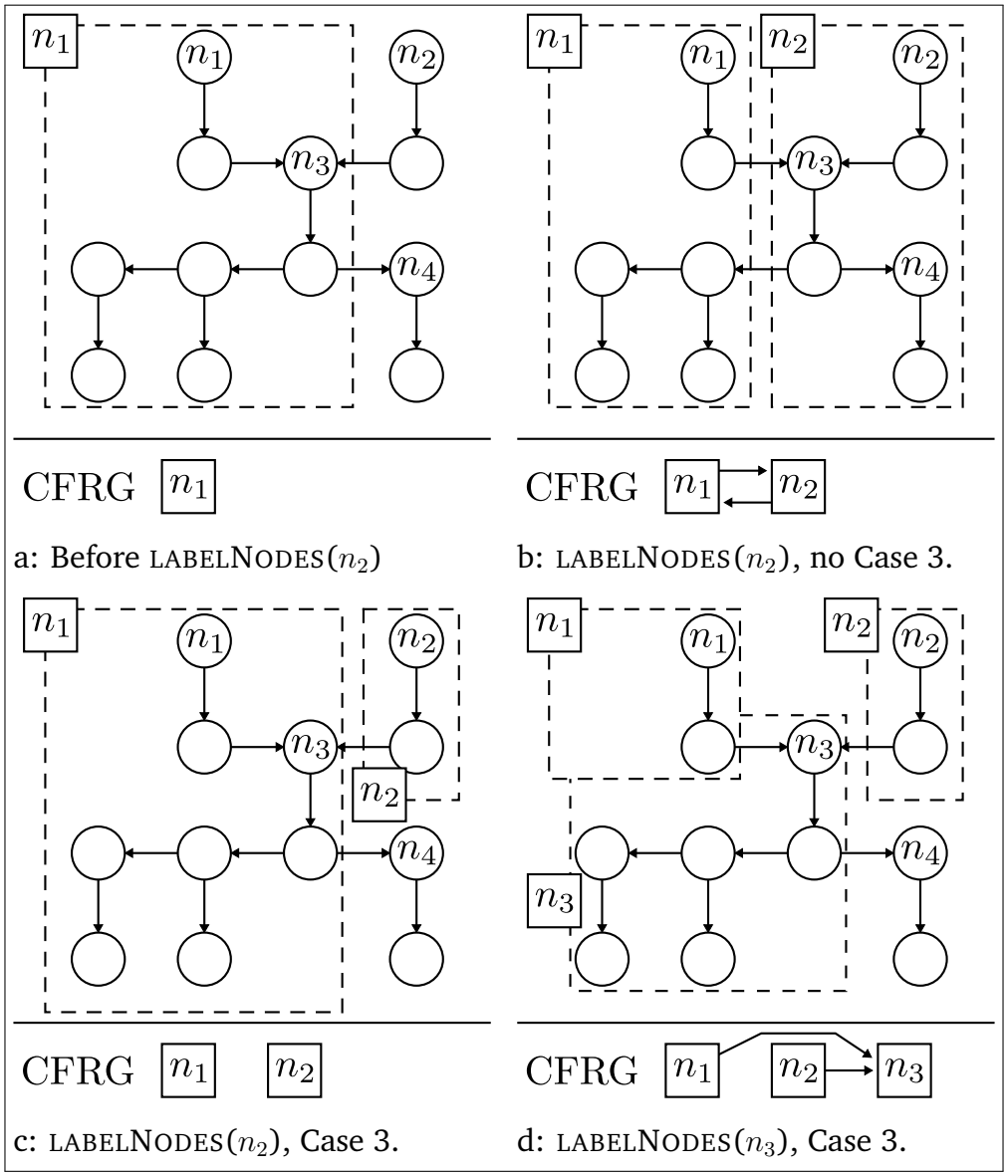


Figure 6.8: Case 3 Protection

To elaborate, Figure 6.8 presents the effects of assigning instructions with and without Case 3 protection. Figure 6.8a is the initial state, after the call to LABELNODES(n_1) has completed and n_2 is in the search stack of TAGCFRS() waiting to be labeled by a call to LABELNODES(n_2). If Case 3 protection is removed, the result of calling LABELNODES(n_2) is Figure 6.8b. There are two issues: the first is a loop has been introduced into the CFRG between n_1 and n_2, the second is that n_1 is no longer weakly connected.

Alternatively, Figure 6.8c illustrates the result of having Case 3 protection in place during the call of LABELNODES(n_2). When n_3 is encountered it has a label of n_1 which differs from n_2. This indicates that n_3 is a conflict, n_3 added to the set of conflicts x and returned to TAGCFRS() when the search completes. Figure 6.8d demonstrates how nodes previously labeled as n_1 have their label replaced by n_3 as the result of TAGCFRS() calling LABELNODES(n_3), shrinking n_1 while ensuring nodes belong to exactly one CFR and the CFRG is a DAG.

If u is determined to be a conflict by one of the cases, it is added to the set of conflicts x and marked visited to avoid further processing. If u is not a conflict, it is labeled as a member of the CFR starting with n and marked as visited. After being labeled a member, the immediate successors (which have not been visited) are added to the search stack. The search repeats until the stack is empty. When the search completes the set of conflicts x are returned to TAGCFRS().

6.3.4 Linking

Assignment is completed when TAGCFRS(h) returns. At this point, each node n has a label given by n.label. Each unique label identifies a CFR. To construct a CFRG from the labeled nodes the final step is to create CFRs add edges between them, *linking* them together. Linking pseudocode is omitted due to the simple nature of the operation: a DFS of the CFG that creates a CFR when encountering a new label and adds edges between CFRs when the endpoints of edges in the CFG have distinct labels.

After linking the CFRG $R = (N, E, h)$ is complete. The set of CFRs is N, edges between CFRs E, and entry CFR h. A CFR $F = (N, E, h)$ has nodes N edges E and entry node h.

To ease discussion over the three types of CFGs, a CFR $F_i = (N_i, E_i, h_i)$ is also identified by it's entry instruction from the CFG. For example, in Figure 6.8c n_3 is a node in the CFG and the entry node of a CFR. Since n_3 is the entry node, the CFR is labeled n_3 in the graph.

6.4 BUNDLEP

Input to the BUNDLEP scheduling algorithm is a CFRG where each CFR is assigned a priority. At run-time, BUNDLEP selects (in priority order) the active bundle. Threads of the active bundle execute until they leave the active CFR. A thread leaves the CFR by attempting to execute the entry instruction of a subsequent CFR. When the active bundle is depleted, another is selected as active. The process repeats until all threads terminate. To support this behavior, BUNDLEP relies on a hardware mechanism to anticipate execution (as did BUNDLE).

6.4.1 Hardware Support

This section proposes an anticipatory mechanism for BUNDLE scheduling. It is a new hardware interrupt named XFLICT. When raised, the interrupt represents the attempted execution of an instruction that may result in a cache conflict. A potential conflict cannot be determined solely by the instruction being executed, additional information is required. That additional information is supplied in the XFLICT TABLE.

Each entry of the XFLICT TABLE is the address of an instruction which may result in a cache conflict. While the processor executes instructions, if the program counter is set to a value present in the XFLICT TABLE, the proposed hardware mechanism halts the CPU before executing the instruction and raises an XFLICT interrupt. The interrupt and ancillary data including the inciting instruction is received by the BUNDLEP scheduling algorithm,

which then moves threads to their appropriate bundle. Hardware breakpoints [53] and the proposed XFLICT interrupt behave similarly, halting the CPU when reaching a specific program address and raising an exception.

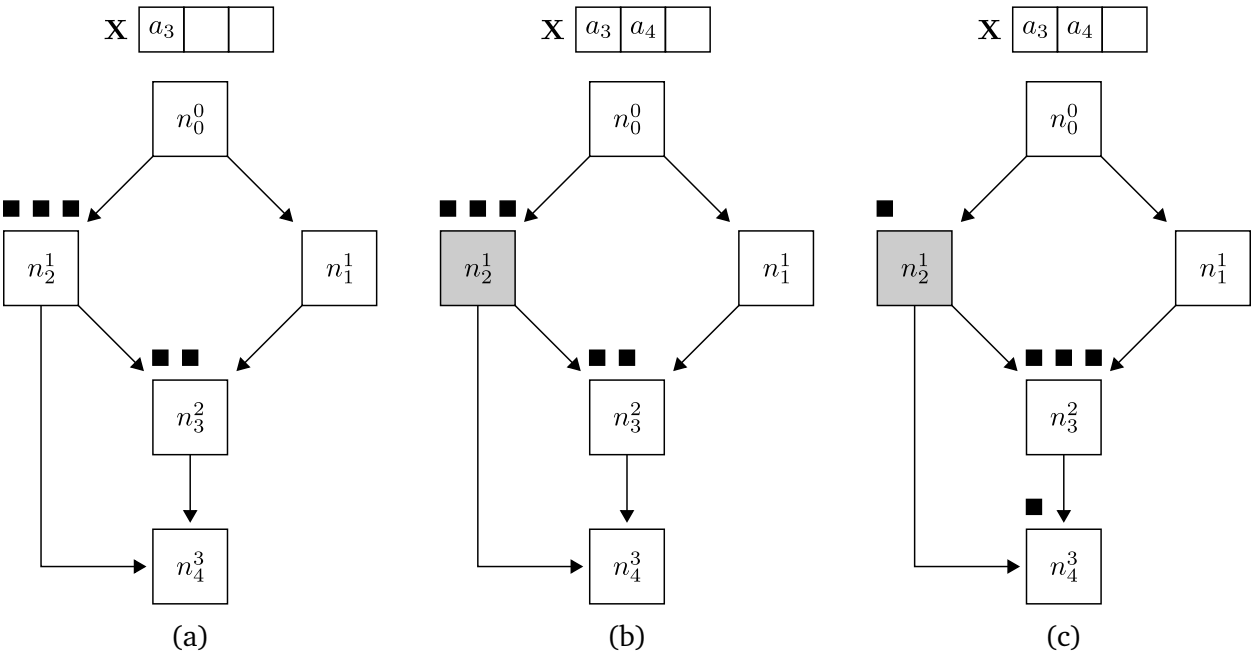


Figure 6.9: XFLICT Interrupts and BUNDLEP

An example illustrates how the interrupt, table, and scheduling algorithm cooperate in Figure 6.9. In the example, priorities are assigned in accordance with Theorem 6 to minimize the number of activations of bundles. Figure 6.9a represents the moment n_1 has been depleted and the BUNDLEP scheduling algorithm prepares to activate n_2 (since it has the lowest priority of any bundle with threads waiting to execute). In the process of selecting n_2 as active, the XFLICT TABLE denoted **X** is populated with the address of the entry nodes of the subsequent CFRs of n_2: $\{a_3, a_4\}$. Only after the XFLICT TABLE has been populated are threads of the active bundle permitted to execute. Between Figure 6.9b and 6.9c two threads have executed and raised XFLICT interrupts, one by attempting to execute a_3 and being placed in n_3, the other being placed in n_4. When each interrupt is raised, the

BUNDLEP scheduling algorithm places the thread in the appropriate inactive bundle, where it waits to be selected as active.

6.4.2 BUNDLEP's Scheduling Algorithm

Algorithm 7 BUNDLEP Scheduling Algorithm

```
 1: T                                                  ▷ Set of Threads
 2: R = (N, E, h)                                      ▷ Conflict Free Region Graph
 3: P                                                  ▷ Priority Queue of Ready Bundles
 4: B                                                  ▷ Array of bundles indexed by their node n ∈ N
 5: procedure BUNDLEP
 6:     b ← B[h]
 7:     b.t.add(T)
 8:     P.insert(b, b.ϖ)
 9:     while b ← P.removeMax() do                     ▷ Best Bundle
10:         S ← ∅                                     ▷ Clear the successor array
11:         XFLICT_CLEAR()                            ▷ Clear the XFLICT table
12:                                                   ▷ Create the mapping of address to node
13:         for s ∈ R.succs(b.n) do
14:             b_s ← B[s]
15:             S[b_s.a] ← b_s
16:             XFLICT_ADD(b_s.a)
17:         end for
18:         for t ∈ b.t do
19:             try {
20:                 TCB_RESTORE(t)
21:                 RUN(t)
22:             } catch (XFLICT x) {
23:                 TCB_SAVE(t)
24:                 b_next ← S[x.a]                   ▷ Get the next bundle
25:                 b_next.t.add(t)
26:                 P.insert(b_next, b_next.ϖ)
27:                 next for                          ▷ t has not terminated
28:             }
29:         end for
30:     end while
31: end procedure
```

Algorithm 7 presents the pseudocode of the BUNDLEP scheduling algorithm utilizing the XFLICT interrupt. It uses four global variables: the set of threads T, CFRG R, priority

queue P, and set of bundles B. Every bundle $b \in B$ has four members: address of the initial node $b.a$, node of the CFRG $b.n$, priority $b.\varpi$ inherited from $b.n$, and a set of threads $b.t$. Only when a bundle is ready (has threads waiting to execute over it) is it added to the priority queue P.

Initialization of the scheduling loop is handled by Lines 6-8, adding all threads to the bundle of the entry node of the CFRG, and placing the bundle into the priority queue. Every iteration of the while loop from Lines 9 to 30 corresponds to the activation of the best priority bundle and execution of threads until the bundle is depleted. When the priority queue is empty all threads have terminated and the job has completed.

`BUNDLEP` is responsible for managing the `XFLICT TABLE`. Lines 10-17 perform the operation of clearing the table and adding the addresses of entry instruction of subsequent CFRs to the table. There is one caveat, the same address may map to multiple CFRs. For this reason `BUNDLE` keeps the successor S array to map from address to bundle, which is used when an `XFLICT` interrupt is raised.

With the successor array populated, the for loop on Lines 18-28 handles the execution of threads until the depletion of b. It begins by selecting an arbitrary thread of the active bundle t. The thread control block (TCB) of t is then restored. TCBs for each thread are stored in the memory space of the scheluder (ie. kernel space). Within every TCB is a copy of the general purpose registers and the active program stack. Restoring the TCB correctly sets the general purpose registers and moves the stack pointer to correct location.

With the TCB restored, the processor is properly prepared to RUN(t) the thread. The thread executes until it terminates, or raises an `XFLICT` interrupt by attempting to execute the entry instruction of a subsequent CFR. In response to the interrupt, `BUNDLE` saves the

TCB of the thread, then (using the successor array S) places the thread in the appropriate bundle b_{next}. If necessary, b_{next} is added to the priority queue P to be selected as active later.

Section 6.5 will consider the context switch costs of `BUNDLEP` in terms of the complexity of each operation with respect to the maximum number of threads per job. The data structures used to store bundles and threads determine the complexity of the context switches. On Line 18 a thread t is removed from the bundle b, order is irrelevant so an array may be used to store all threads which results in $\mathbb{O}(1)$ for removal of a single thread. For selecting the active bundle on Line 9 priority order is relevant and a priority queue is used to maintain the ordering. An efficient implementation of priority queue will utilize a Fibonacci heap with complexity $\mathbb{O}(1)$ of insertion, and amortized complexity of $\mathbb{O}(\log n)$ for removeMax.

6.4.3 Priority Assignment

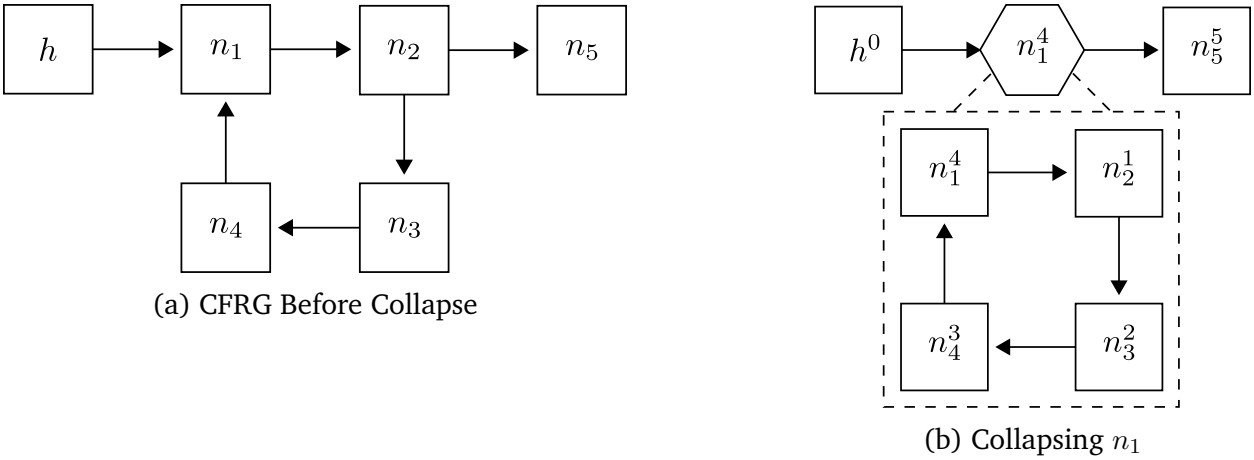
Assigning priorities to CFRs occurs during the offline analysis of the ribbon. For each node n of the CFRG $R = (N, E, h)$, the priority of n is determined by the longest path from h to n. To make this assignment, the CFRG must be a DAG. CFG and CFRG creation carefully avoid creating spurious loops from a ribbon. However, programs contain looping logical structures. These *user defined loops* may lie within a single CFR which would not negatively affect the CFRG. If user defined loops span multiple CFRs, by necessity a loop will be created in the CFRG.

To clarify, a loop in the CFRG $R = (N^R, E^R, h^R)$ has a head which is a node $n_i \in N^R$ (a CFR). The CFR $n_i = F_i = (N_i, E_i, h_i)$ has an entry node h_i which is the loop head instruc-

tion, for such a CFR we will say that F_i (or n_i) is the loop head. A node $n_m \in N^G$ (of the CFG) that has h_i as one of its loop heads is said to be a *member* of h_i's loop. All nodes within a CFR share the same loop heads. A CFR is a member of the loops of the nodes contained within it.

When a user defined loop creates a loop in the CFRG the loop head and its members are *collapsed* into a loop summary node. Unless otherwise stated, loop summary nodes are treated identically to other nodes in the CFRG. If a difference in treatment is required they will be referred to as "loop summary nodes" or "summary nodes" to distinguish them.

A summary node replaces the head and member nodes of a loop in the CFRG with a single node. Within a summary node only one loop is permitted. To enforce the loop limitation a summary node may include other summary nodes. If a member of a summary node is a loop head h_i, the loop of h_i is collapsed with its members and replaced by a summary node. Doing so ensures a summary node with loop head h_i exclusively contains members with inner-most loop head h_i. Priorities assignment leverages the structure of summary nodes.



(a) CFRG Before Collapse

(b) Collapsing n_1

Figure 6.10: Collapsing One Loop of a CFRG

Figure 6.10 illustrates the collapse of a single loop. Figure 6.10a is the CFRG with loop n_1 and members $\{n_2, n_3, n_4\}$. Before collapse the set of nodes in the CFRG is:

$$N^R = \{h, n_1, n_2, n_3, n_4, n_5\}$$

Figure 6.10b is the result of collapsing n_1's loop into a summary node also labeled n_1. After collapsing the loop, the set of nodes in the CFRG are $N^R = \{h, n_1, n_5\}$, where n_1 is a summary node. Members of the summary node are $\{n_1, n_2, n_3, n_4\}$ where n_1 is a regular node.

For a CFRG that contains no summary nodes, the priority value of each node is the longest path to each node from the initial node in terms of edge weights, where all edges are weighted one. For a CFRG containing summary nodes, each summary node has its longest path inflated by the length of the longest cycle collapsed within it.

In Figure 6.10b, the length of the longest cycle from n_1 to n_1 is four. Therefore, the loop head is given priority four, as is the summary node. Giving the loop head the highest priority of all members ensures that during each iteration all threads block before starting their next iteration.

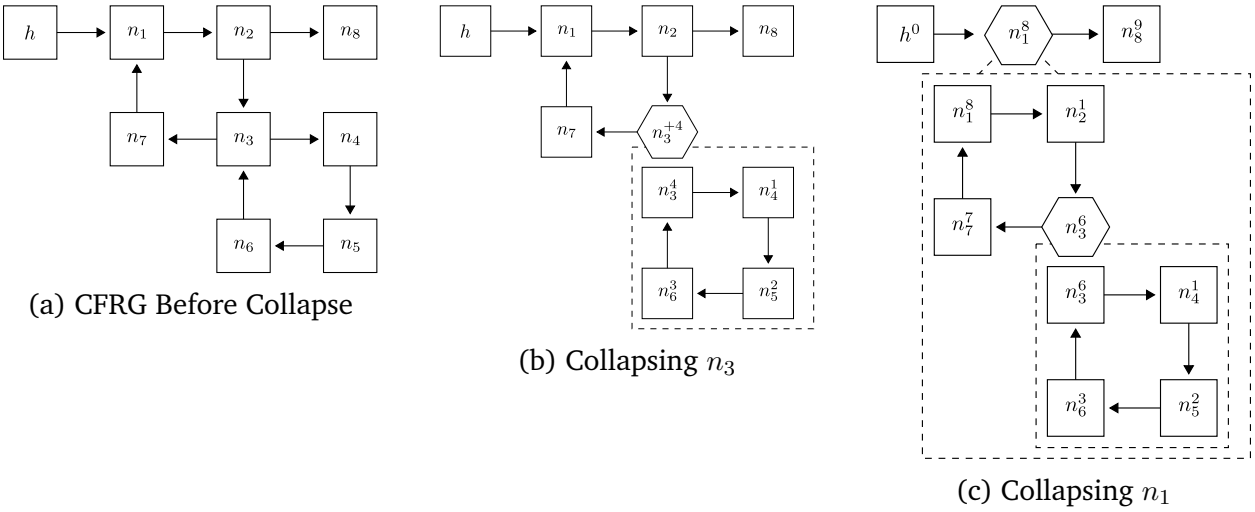
In addition to having an inflated priority value, each summary node must have a unique priority within its scope. For summary node n_1 in Figure 6.10b, the other nodes in its scope are $\{h^0, n_5^5\}$. Having a unique priority guarantees all threads complete their loop iterations before progressing. A property that will be leveraged during WCETO calculation.

To summarize, for a summary node n_i

1. The longest path (and priority) of n_i is increased by the length of the longest cycle within the summary node.

2. In its scope, n_i must have a unique priority.

Figure 6.11: Summary Node Priority Requirements



(a) CFRG Before Collapse

(b) Collapsing n_3

(c) Collapsing n_1

Figure 6.12: Collapsing Embedded Loops

For embedded loops, when a summary node contains another summary node the embedded loops are collapsed first. Figure 6.12 provides an example of embedded collapse and the resulting priority assignments. Figure 6.12a shows the CFRG before collapsing any loops. The first loop collapsed is that of n_3 in 6.12b. Nodes $\{n_4, n_5, n_6\}$ are given priorities equal to the longest path from the loop head n_3. The loop head (and summary node) are given priority four, equal to the longest path denoted as $+4$ to accentuate the temporary nature of the value. Lastly, in Figure 6.12c n_1 is collapsed, in the process the summary node n_3 is given its final priority value of 6 which is passed to the loop head n_3.

This affects the priorities of $\{n_7, n_1\}$.

Repeatedly collapsing loops into summary nodes from the inner-most to outer-most, all loops are removed from the CFRG and it is converted to a DAG. Additionally, since the contents of each summary node are restricted to a single loop, within a summary node the graph is also a DAG (if the edges returning to the loop head are excluded). Setting the priority of nodes within a summary node equal to their longest path from the loop head h_i, where the loop head has the highest priority of all members, guarantees each CFR is activated at most once per iteration of the loop h_i, as shown by Theorem 7.

Theorem 7 (Maximum Bundle Activations per Iteration)**.** *For a graph of the summary node $G = (N, E, n_0)$ with loop head n_0, set of member nodes N and edges E, where each node $n_i \in N$ has priority ϖ_i equal to the longest path from n_0 to n_i, and n_0 has priority greater than all others $\{\varpi_0 \mid \forall_{n_j \in \{N \setminus n_0\}}\ \varpi_0 > \varpi_j\}$ the bundle of n_i will be activated at most once per iteration of n_0.*

Proof. Observation I: For a member node n_i of N that is a loop head, n_i will be collapsed with all other nodes that have n_i as their inner-most loop head. Only the summary node n_i will be in N. Therefore, for any summary node, there is exactly one loop with head n_0.

Observation II: A single iteration of the loop contained within a summary node is defined as the series of activations that begins with the activation of n_0 and ends just before n_0 would be selected as active once again. Since n_0 has the greatest priority among all nodes in N, bundle's of all other nodes must have been depleted before n_0 could be activated again.

Observation III: Since n_0's priority is unique in its scope, for the n_0 summary node to

be activated all other threads must be blocked waiting on bundles of greater priority to activate. Since the regular node of n_0 has greatest priority among all members of summary node n_0, when activated the summary node n_0 will complete all of its iterations and iterations of embedded loop summary nodes before executing the bundle of any node that is not a member of summary node n_0.

Consider the graph G where the incoming edges to n_0 have been removed, removing the cycle, i.e., $E = \{(u, v)|(u, v \neq n_0) \in E\}$ as a graph $G' = (N, E, n_0)$. By Observation I, G' is now a DAG of CFRs. Treating a single iteration as a job release and applying Theorem 6 to G', each $n \in N$ is activated at most once per iteration for all threads executing the loop.

□

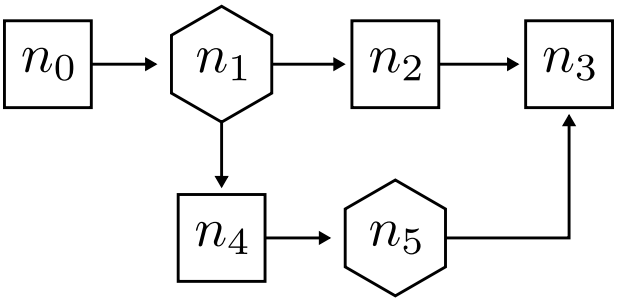
6.5 BUNDLEP WCETO Calculation

A primary goal of BUNDLEP is the practical effort of creating an effective, safe WCETO bound. To that goal, the bound calculation is formulated as an integer linear program (ILP). The number of variables grow at $\mathbb{O}(N + E)$ for N CFRs and E edges between them. This section is devoted to describing the transformation of a CFRG into a set of constraints and an objective function. To present the transformation in a concise manner the individual ILP constraints are presented in the ancillary Section 6.9.

Assigning priorities to nodes of the CFRG and collapsing loops (as described in Section 6.4.3) guarantees each node is activated at most once. As such, the contributions of individual nodes may be considered in isolation. What determines a node's individual contribution is the number of threads assigned to it.

For the ILP, the maximization problem becomes finding the greatest sum of contribu-

tions of individual nodes for a valid assignment of threads. The WCETO of an individual node is given by the function $\omega_n(t_n)$ where t_n is the number of threads assigned to node $n \in N$. Figure 6.13 illustrates the relationship between the CFRG, WCETO of nodes $\omega_n(t_n)$, and objective function $\Omega = \sum_{n \in N} \omega_n(t_n)$; it is the maximized sum of WCETO contributions of the CFRs of the CFRG given an assignment of threads per node. An additional example using representative values from the evaluation is available in the ancillary Section 6.10.



i	n_0	n_1	n_2	n_3	n_4	n_5	
t_i	4	4	2	4	2	2	Ω
ω_i	40	20	70	30	50	10	220

Figure 6.13: CFRG Individual Nodes and ILP Objective

The WCETO of a node $\omega_n(t_n)$ depends on the number of threads assigned to it t_n, the function takes the form of Equation 6.5.1. We assume a timing-compositional architecture [57]; the number of cycles required to complete a single node is divided into two parts: the *memory demand* and the *execution demand*. The memory demand of a node n is the product of two factors 1.) the set of unique cache blocks in the CFR, commonly referred to as evicting cache blocks (ECBs [13]) and 2.) the block reload time $\mathbb{B}$. The memory demand is denoted $\gamma_n = |ECB_n| \cdot \mathbb{B}$. The execution demand is the product of the worst-case execution time of a single thread over the node c_n and the number of threads assigned t_n. Two context switch costs are included to reflect the penalty of `BUNDLEP` scheduling, $\mathbb{X}_b$ is the number of cycles required to switch to a new active bundle, and $\mathbb{X}_t$ is the cost of selecting a thread from the active bundle. The costs $\mathbb{X}_b$ and $\mathbb{X}_t$ are directly related to

lines 9 and 18 of Algorithm 7.

$$\omega_n(t_n) = \begin{cases} \omega'_n(t_n), & summ(n) \\ 0, & t_n = 0 \\ c_n \cdot t_n \cdot \mathbb{X}_t + \mathbb{X}_b + \gamma_n, & t_n \geq 1 \end{cases} \tag{6.5.1}$$

For a summary node n the function $summ(n)$ returns true and false for non-summary nodes. When a summary node is supplied to Equation 6.5.1 the value is calculated by $\omega'_n(t_n)$. Described by Equation 6.5.2, $\omega'_n(t_n)$ depends on $\mathbb{I}_n$ the maximum number of iterations of the loop summary node n and $inscope(n)$. The set of nodes returned by $inscope(n)$ is the set of member nodes with inner-most loop head n, which includes nodes and loop summary nodes. For example in Figure 6.12c $inscope(n_1) = \{n_1, n_2, n_3, n_7\}$ where n_3 is a summary node.

$$\omega'_n(t_n) = \begin{cases} 0, & t_n = 0 \\ \mathring{\gamma}_n + \mathbb{I}_n \cdot \sum_{i \in inscope(n)} \mathring{\omega}_i(t_i), & t_n \geq 1 \end{cases} \tag{6.5.2}$$

The memory and execution demand of a summary node are not entirely separable. Individual nodes within scope of n have their per-iteration contribution bounded by $\mathring{\omega}_i(t_i)$, described later. An initial memory demand for summary node n is calculated as $\mathring{\gamma}_n$, it represents the number of cycles required to cache all blocks of nodes within the summary node regardless of scope. The set of nodes $allscope(n)$ includes any node that is not a summary node and has loop head n. Using Figure 6.12c, $allscope(n_1) = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$ which includes no summary nodes.

For $\overset{\circ}{\gamma}_n$ to account for the time required to cache all blocks of all members of the summary node n it must account for multiple nodes utilizing the same cache block. A multi-set containing the union of ECBs from all nodes addresses the issue. The multi-set union of ECBs is formed and labeled $\overset{\circ}{ECBs}_n = \bigcup_{i \in allscope(n)} ECBs_i$. The product of cardinality of the ECB multiset and the block reload time produces $\overset{\circ}{\gamma}_n = \mathbb{B} \cdot |\overset{\circ}{ECBs}_n|$. By virtue of the multi-set's cardinality the number of cycles required to load every cache block of all nodes collapsed under n is properly accounted for.

The per-iteration contribution of a node $\overset{\circ}{\omega}_i(t_i)$ is defined by Equation 6.5.2. When i is an embedded summary node, its contribution is calculated by Equation 6.5.2. For a non-summary node i, its per-iteration contribution includes its WCETO of i, context switch costs, execution demand, and the worst-case memory demand. A method similar to the ECB-Union cache related preemption delay approach [16] is employed to calculate the memory demand from the perspective of the affected node i. The worst-case occurs when another member node evicts the ECBs of i, forcing the blocks $ECBs_i$ to be loaded when i is activated. The number of evictions can be bounded by the ECBs of all loop members, specifically those that occur more than once in the loop. The set of ECBs found more than once within the summary node n are given by $\overset{2}{ECBs}_n = \{\bigcup_{u \cdot k} \mid u \cdot k \in \overset{\circ}{ECBs}_n \wedge k \geq 2\}$. Thus, the memory demand bound for i is $\overset{2}{\gamma}_i = |\overset{2}{ECBs}_n \cap ECBs_i| \cdot \mathbb{B}$. Incorporating per-iteration context switches, execution and memory demand into the bound for i yields Equation 6.5.3.

$$\mathring{\omega}_i(t_i) = \begin{cases} \omega'_i(t_i), & summ(i) \\ 0, & t_n = 0 \\ \mathbb{X}_b + \mathbb{X}_t \cdot t_i \cdot c_i + \overset{2}{\gamma}_i, & t_n \geq 1 \end{cases} \tag{6.5.3}$$

A valid assignment of threads takes into account the structure of the CFRG. To reflect the structure, threads are treated as flow traversing the edges of nodes. The entry node is treated as the source of flow, providing a total m threads on its outgoing edges. All threads must reach the terminal node. At each node the sum of threads along incoming edges and outgoing edges must be equal (except the entry and terminal nodes).

The ILP finds the assignment of threads according to the flow of the CFRG which maximizes the number of cycles required to complete m threads according to BUNDLEP scheduling, thus bounding the WCETO of a job.

6.6 BUNDLEP Evaluation

The evaluation takes the approach of comparing BUNDLEP's thread-level scheduling algorithm to a naive algorithm which executes threads one after another (serially). Individual benchmarks from the Mälardalen [52] MRTC suite are treated as ribbons releasing m threads per job. The WCET of each job is analyzed twice, once for a single multi-threaded task scheduled by BUNDLEP, and again for m serial threads by Heptane. Similarly, the runtime behavior is collected for each benchmark under BUNDLEP and serial execution. A fully functional virtual machine with the tools and source is available for download to recreate these results or expand upon them [9].

Ideally, BUNDLEP would also be compared with BUNDLE. However, the BUNDLE evalua-

tion used synthetic programs rather than compiled source (for any architecture). WCETO analysis for BUNDLE is also intractable with complexity $\mathbb{O}((|N|!)^m)$. This is due to the nature of the algorithm, it does not restrict the flow of threads through the CFRG, which demands all-paths be repeatedly searched. A novel BUNDLE WCETO implementation of an intractable solution, which is known to be dominated by BUNDLEP is not compelling, as such it is omitted from the evaluation.

The target platform for WCETO analysis and execution is a MIPS 74K processor with a direct mapped single level instruction cache. Cache blocks are restricted to 32 bytes. The CPI $\mathbb{I}$, block reload time $\mathbb{B}$, and number of cache blocks ℓ vary based on Table 6.1. Additionally, the number of threads per job m vary from 1 to 16 by powers of two. Jobs are executed on a MIPS simulator provided by Heptane and modified to execute BUNDLEP scheduling or a serial batch of threads.

CPI ($\mathbb{I}$)	BRT ($\mathbb{B}$)	ℓ	m
1	100	{8, 16, 32}	{2, 4, 8 ,16}
10	100	{8, 16, 32}	{2, 4, 8 ,16}

Table 6.1: MIPS 74K Architecture Parameters

Of the 27 MRTC benchmarks, 18 were evaluated. The selection is limited by Heptane's ability to perform WCET analysis using the `lp_solve` ILP solver and the 12 gigabytes of RAM available (the complete results are available in the technical report [58]).

6.6.1 Context Switch Costs

For BUNDLEP scheduling there are two types of context switches: 1.) switching between threads of the active bundle $\mathbb{X}_t$ and 2.) switching to the next active bundle $\mathbb{X}_b$. For the

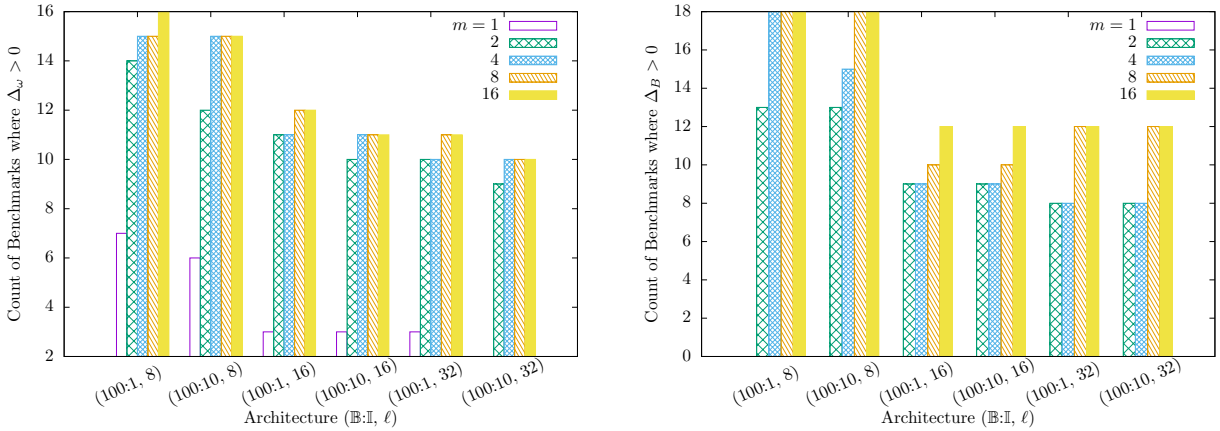
classical approach, there is a single job-level context switch cost. Thread-level switches are (by design) less costly than job-level: where virtual pages are exchanged, and task (instead of thread) level control blocks are updated, etc. To favor the classical approach, the bundle-level context switch cost $\mathbb{X}_b$ is also used as the job-level context switch cost.

Finding representative values for both $\mathbb{X}_b$ and $\mathbb{X}_t$ considers the scheduler behavior and sample programs written for the target architecture. Incorporated into both costs is the time required to TCB_SAVE and TCB_RESTORE, which on a MIPS 74K requires two instructions. For a TCB_SAVE they are 1.) a `save` instruction to copy general purpose registers and increment the stack pointer `$sp` and 2.) a `mov` instruction to copy `$sp` into the TCB. For a TCB_RESTORE they are 1.) a `mov` of `$sp` from the TCB and 2.) a `restore` of the stack.

For $\mathbb{X}_t$ the dominant operation is selecting an element from an array, setting the context, and `jmp`'ing to the previous context. Analysis of a program that performs these operations by Heptane produced a WCET of less than 10 cycles. Therefore, setting $X_t = 10$ serves as an overestimate of cyles. For $\mathbb{X}_b$, a precise value would require the implementation of a priority queue, supported by an optimized heap and analysis by Heptane. The implementation of such a queue is beyond the scope of this work. However, a limited program including queue operations over two elements was analyzed by Heptane with a WCET of less than 55 cycles. Bundle-level context switches are dominated by P.removeMax() on line 9 of Algorithm 7. Assuming an optimized queue, the operation grows at $\log_2(m)$ yielding an $\mathbb{X}_b$ in terms of threads $m : \mathbb{X}_b = 55 \cdot \log_2(m)$.

For each benchmark, Heptane produces a single WCET value for the execution of one thread through the ribbon denoted c_H. To compare Heptane's WCET c_H to `BUNDLEP`'s

WCETO Ω, the number of threads and context switch costs are incorporated and quantified as a difference $\Delta_\omega = m \cdot (c_H + \mathbb{X}_b) - \Omega$. Similarly, the number of cycles required to execute on the simulator serially is denoted E_H, cycles required for BUNDLEP execution denoted E_B, and the comparison $\Delta_B = E_H - E_B$. A positive Δ value indicates the BUNDLEP approach provides a benefit.



(a) Analytical Benefit of BUNDLEP

(b) Run-Time Benefit of BUNDLEP

Figure 6.14: Benefits of BUNDLEP

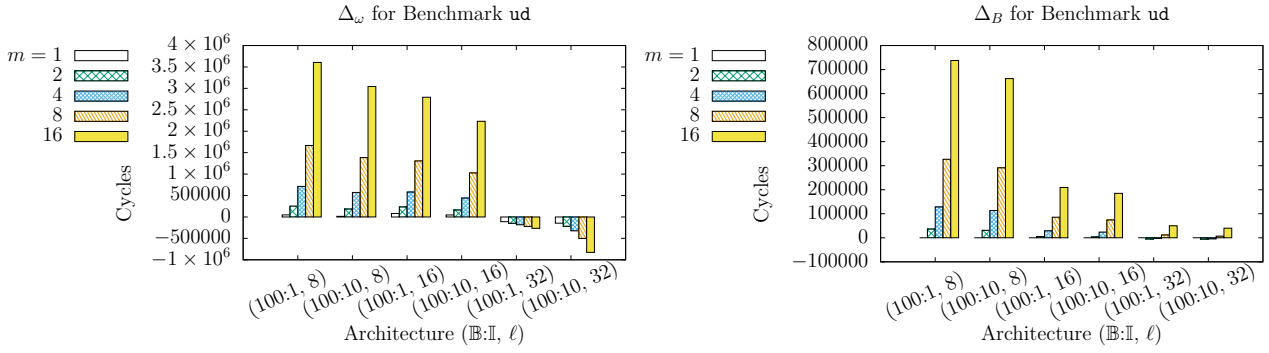
Figures 6.14a and 6.14b summarize the results of the evaluation. The y-axis represents the number of benchmarks where BUNDLEP benefits the task. Along the x-axis, the groups separate the architecture parameters which are enumerated by their "($\mathbb{B}$:$\mathbb{I}$, ℓ)". For each group the result is tallied by the number of threads per job, from 1 to 16.

There are several interesting observations to be made in Figure 6.14a. Though BUNDLEP analysis provides a benefit in the majority of cases, it does not always. As the cache size is reduced the number of benchmarks that benefit increases. Similarly, as the number of threads per job increases so do the number of benchmarks that benefit. These trends are due to the number of misses (typically) increasing as the cache size is reduced, or the number of threads is increased. BUNDLEP avoids these conflicts or converts them to cache

hits. Surprisingly, for a single thread per job BUNDLEP may provide a lower bound – this is likely due to the use of the expanded CFG instead of the conventional CFG used by Heptane's analysis.

The run-time benefit summary in Figure 6.14b more heavily favors BUNDLEP, with unsurprising trends. For a single thread per job, BUNDLEP provides no benefit since there is no reason to block and incur context switch costs. As the number of threads increases so does the run-time performance. As the cache size decreases, the number of benchmarks that see a run-time performance increases. When compared to the WCETO benefit, more benchmarks benefit from the run-time behavior than the analysis would suggest. This implies further refinements of the analysis are possible.

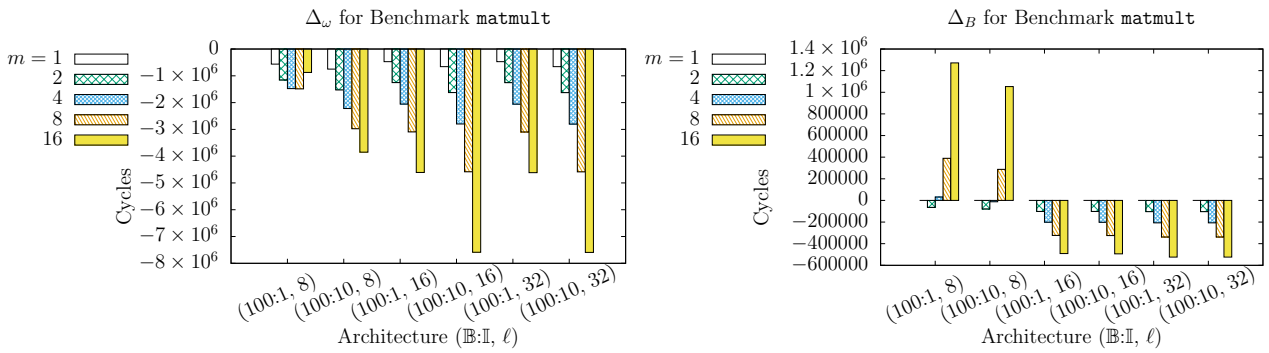
Across the four dimensions of the evaluation (cache size, BRT, CPI, and number of threads per job), the expectation of BUNDLEP's benefit will increase as the cache size decreases, increase as the BRT increases, decrease as the CPI increases, and increase as the number of threads per job increase. Many of the benchmarks match these expectations, such as the results for ud found in Figure 6.15.



(a) Analytical Benefit (b) Run-Time Benefit

Figure 6.15: Results for the ud Benchmark

The anomalies provide insights into the circumstances where BUNDLEP may be im-



(a) Analytical Benefit (b) Run-Time Benefit

Figure 6.16: Results for the `matmult` Benchmark

proved. Using the `matmult` benchmark as an example, `BUNDLEP` never produces an analytical benefit, and rarely a run-time benefit (Figure 6.16). Counter-intuitively as the number of threads increase the analytical result worsens compared to the serial bound. This is due to the structure of the CFRG, which has several small CFRs contained within multiple embedded loops. The number of bundle-level context switches with cost $\mathbb{X}_b$ outweighs the benefit of sharing cache values.

6.7 Summary

`BUNDLEP` expands upon the foundation set by `BUNDLE`. The central principles of treating the cache as a benefit to execution times by scheduling threads in a cache cognizant manner are refined and improved. `BUNDLE` allowed instructions to reside in multiple CFRs creating ambiguity in scheduling decisions. The CFR and CFRG creation methods used by `BUNDLE` created unnecessary loops increasing analytical complexity.

Refining the creation of CFRs removes ambiguity from scheduling decisions and prevents loops from being added to the CFRG. The preclusion of additional loops and the assignment of priorities to CFRs reduces the complexity of the WCETO calculation. Addi-

tionally, restricting the flow of threads through the CFRG tightens the WCETO bound.

With refined creation methods for CFRGs and tractable WCETO calculation a practical implementation of BUNDLEP has been implemented. The toolkit is available for download and reuse for future use and expansion. Use of the toolkit shows a benefit to BUNDLEP scheduling in terms of analysis and run-time behavior for specific programs and architecture parameters.

The anomalous results provide further motivation to improve BUNDLEP's approach. Of particular interest is the balance between context switch costs and WCETO values per CFR. When the inter-thread cache benefit is smaller than the context-switch cost $\mathbb{X}_b$, allowing threads to execute over the CFR could decrease the task's WCETO. However, the full implications of such behavior require further investigation.

6.8 Ancillary Preamble

The remainder of this chapter are two ancillary sections. The first provides the formal details of ILP formulation. It is followed by a second ancillary section that includes an example illustrating the constraints and functions of ILP formulation to calculate a WCETO bound. These two sections are provided as an aid to a discrete implementation. Theoretical and general contributions resume with the following Chapter 7.

6.9 Ancillary: ILP Transformation and Example

The following describes the transformation of equations 6.5.1, 6.5.2, 6.5.3 and the supply of threads into the constraints of the ILP for WCETO calculation. For a CFRG

$R = (N, E, h)$, the objective of the ILP is to maximize:

$$\Omega = \sum_{n \in N} \omega_n(t_n)$$

Several variables are added to the ILP which are not present in the formulae. A binary selector variable $b_n \in \{0, 1\}$ is added for each node, when the value is 1 the node has at least one thread assigned to it. For every edge $(u, v) \in E$, the variable $t_{(u,v)}$ represents the number of threads passed from node u to v. The terminal node of the CFRG is identified as $z \in N$, having out-degree zero.

Two functions are defined for each node. The successor and predecessor functions return the sets given by their names. Both of these functions properly obey the scope of the provided node n. Only nodes with the same inner-most loop head are included in the returned set. If a predecessor or successor is a loop head (which is not the inner-most loop head of n) a loop summary node is replaces the loop head in the set.

Functions

$preds(n) \triangleq \{u|(u, n) \in E\}$

Set of immediate predecessors of $n \in N$.

$succs(n) \triangleq \{v|(n, v) \in E\}$

Set of immediate successors of $n \in N$.

What follows are the individual constraints generated for each node. To clarify, a top-most summary node contributes its WCETO directly to the objective, being a member of

$n \in N$. Member nodes of the summary node contribute to the objective indirectly. Members of summary nodes have their WCETO reflected by their summary node's ω' value.

Node Constraints

$t_n \in \{0, m\}$

Number of threads assigned to node n.

$b_n \in \{0, 1\} \leq t_n$

Binary selector for n, indicating that n has at least one thread assigned.

$$t_n \triangleq \sum_{u \in preds(n)} t_{(u,n)}$$

Number of threads assigned to n must be equal to the sum of all entering n.

$$t_n \triangleq \sum_{v \in succs(n)} t_{(n,v)}$$

Number of threads assigned to n must be equal to the sum of all leaving n.

$$\omega_n \triangleq c_n \cdot t_n \cdot \mathbb{X}_t + \mathbb{X}_b \cdot b_n + \gamma_n \cdot b_n$$

WCETO of a non-summary node n, see Equation 6.5.1.

$$\omega'_n \triangleq (\mathring{\gamma}_n \cdot b_n) + \mathbb{I}_n \left(\sum_{i \in inscope(n)} \mathring{\omega}_i \right)$$

WCETO for a summary node n, see Equation 6.5.2.

$$\mathring{\omega}_n \triangleq c_n \cdot t_n \cdot \mathbb{X}_t + (\mathbb{X}_b + \overset{2}{\gamma}_n) \cdot b_n$$

WCETO per-iteration contribution, see Equation 6.5.3.

Special Case Constraints

$t_h \triangleq m$

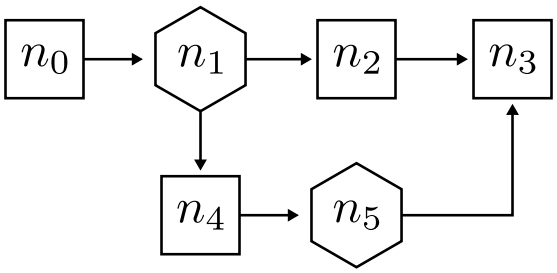
The initial node h must have all m threads assigned.

$t_z \triangleq m$

The terminal node z must have all m threads assigned.

6.10 Ancillary: WCETO Example

The ILP objective function Ω, is the sum of the contributions of the CFRs of the CFRG given an assignment of threads per node. Figure 6.17 illustrates the source of each node's contribution for four threads ($m = 4$). It reuses the structure of Figure 6.13 with detailed memory and execution demand values that are closer to those found in the evaluation.



Ω
2,680

i	n_0	n_2	n_3	n_4
γ_i	400	500	100	300
c_i	40	110	5	100
t_i	4	2	4	2
ω_i	560	720	120	500

i	n_1	n_5
$\mathring{\gamma}_i$	200	300
$\mathbb{I}_{n_i}$	8	2
$\sum \mathring{\omega}$	30	20
t_i	4	2
ω_i	440	340

Figure 6.17: CFRG Individual Nodes and ILP Objective

When completed, the ILP determines the WCETO bound is 2,680 cycles. Understanding how the bound is calculated is made easier by considering the memory demand indepedently of the execution demand. For n_0, there is no decision four threads are assigned. The memory demand for n_0 is 400 cycles, and 40 cycles per thread for 560 cycles total.

There are no decisions to be made for n_1 or n_3, the number of threads assigned to them are determined by the structure of the graph. For threads to be assigned to n_4 and n_5, their combined execution and memory demand must be compared to n_2. For one thread, n_2 has a total demand of 610 cycles. For one thread, the combined demand for n_4 and n_5 is 710 cycles (the demand for interior nodes of n_5 is 10 cycles per thread. Though not explicitly listed in the figure, this is the reason $\sum \mathring{\omega} = 20 = 2 \cdot 10$).

The execution demand for a second thread (or third) of n_2 is 110 cycles, and the combined execution demand for a second thread of n_4 and n_5 is also 110 cycles. Any assignment where t_2, t_4, and t_5 are greater than or equal to one will result in the same WCETO value. The assignment in Figure 6.17 has balanced the threads across paths.

CHAPTER 7 NON-PREEMPTIVE MULTITASK BUNDLE

BUNDLE and BUNDLEP's scheduling algorithm are limited to a single multi-threaded task on a single processor. To bring the positive perspective to multi-threaded multi-task uniprocessor scheduling the non-preemptive multi-task BUNDLE (NPM-BUNDLE) scheduling algorithm and analysis is introduced. Implied by the name, scheduling is non-preemptive with respect to jobs. However, threads of jobs are scheduled according to BUNDLE or BUNDLEP which preempt one another at CFR boundaries.

To support multiple tasks, the following contributions are made as part of NPM-BUNDLE:

1. A model of hard real-time multi-threaded tasks which is compatible with existing single-threaded models, where tasks sets may be transformed by dividing tasks while preserving the total number of threads.
2. A schedulability test named Threads Per Job (TPJ) which divides task sets (when possible) to create a job-level non-preemptive schedulable task set.
3. Proof of TPJ's optimality with respect to non-preemptive multi-threaded feasibility.
4. An improvement to Baruah's [34] non-preemptive chunks algorithm.
5. An evaluation of over 500,000 task sets, comparing the schedulability ratio of TPJ to those of non-preemptive and preemptive EDF, with an accompanying implementation available for download [11].

Figure 7.1: Summary of NPM-BUNDLE contributions

These contributions are presented in the following sections. Section 7.1 augments the BUNDLE model to suit multiple tasks, describes the application of non-preemptive EDF scheduling for thread-level schedulers, and the requirements of task transformation. Section 7.2 introduces then improves upon the non-preemptive chunk algorithm [34], fol-

lowed by the TPJ schedulability algorithm and proof of feasibility. Section 7.3 compares the schedulability ratio of TPJ to other non-preemptive and preemptive scheduling algorithms, before summarizing the contributions of NPM-BUNDLE in Section 7.4.

7.1 NPM-BUNDLE Model and Notation

In previous Chapters 5 and 6 describing BUNDLE and BUNDLEP, WCETO analysis is limited to a single task. The model and (notation) in the previous chapters efficiently utilize this limitation, foregoing task identifiers where possible. In this chapter the single task limitation is removed, requiring careful delineation of tasks in the notation. Table 7.1 summarizes the notation used within this chapter.

τ	Set of n tasks {τ_0, τ_1, ..., τ_{n-1}}
$\tau_i = (p_i, d_i, m_i, c_i(m) : \mathbb{N} \mapsto \mathbb{R}^+)$	Task i
p_i	Minimum inter-arrival time of τ_i
d_i	Relative deadline of τ_i
m_i	Threads per job of τ_i
$c_i(m)$	WCET function of τ_i in terms of m threads
o_i	Executable object of τ_i
q_i	Non-preemptive chunk size of τ_i
t	Time or interval

Table 7.1: NPM-BUNDLE Notation

The set of n multi-threaded tasks is given by $\tau = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$. Each job of a task $\tau_i = (p_i, d_i, m_i, c_i(m) : \mathbb{N} \mapsto \mathbb{R}^+)$ has a minimum inter-arrival time of p_i and relative deadline d_i. For every job release of τ_i, a positive integer m_i identical threads are released. Each thread of τ_i executes over the same object o_i on the shared processor. All threads share the same deadline as their job. The WCET of τ_i is a function of the number of threads per job, $c_i(m_i)$.

Scheduling and schedulability analysis of NPM-BUNDLE relies upon a relationship between the number of threads scheduled per multi-threaded job and the WCET of the job executed non-preemptively. To clarify, the NPM-BUNDLE scheduling mechanism precludes preemptions between jobs of different tasks. For threads within a job of a task, a thread-level scheduler may execute threads preemptively. Figure 7.2 illustrates the scheduling behavior.

In Figure 7.2, at time 1 a job of τ_2 is released. The job of τ_2 cannot be preempted by the job of τ_1 released at time 5. During the execution of τ_2, the two threads (given distinct colors) may preempt one another according to the thread-level scheduler, at time 8 for instance. Thread-level scheduling and preemption decisions are not restricted by NPM-BUNDLE analysis. The thread-level scheduling policies of τ_1 and τ_2 are independent of the non-preemptive task-level scheduling of non-preemptive EDF used herein.

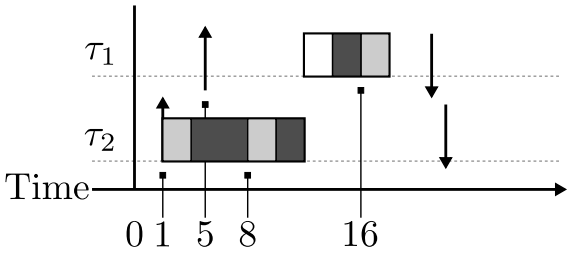


Figure 7.2: Scheduling Behavior

Thread-level scheduling algorithms must be characterized by a WCET (or WCETO) function $c_i(m_i)$ for m_i threads per job and $c_i(m_i)$ must be strictly increasing discrete and concave (detailed in Subsection 7.1.2). Thread-level schedulers that produce concave $c_i(m_i)$ functions establish a relationship between the execution requirements of a task and the number of threads, where the requirement for one job of m_i threads is less than m_i jobs of one thread. For BUNDLE-based schedulers, concavity is the result of the inter-thread cache benefit, where $c_i(m) - c_i(m-1) \geq c_i(m+1) - c_i(m)$; it is this relationship the scheduling behavior and analysis seek to exploit.

Not all tasks and thread-level schedulers will produce concave WCET functions. For a task τ_i with a convex WCET function (where there is no benefit in grouping threads together), the m_i threads of τ_i may be replaced with m_i single-threaded tasks. These single-threaded have vacuously concave WCET functions by virtue of executing no more than one thread.

The task set τ provided by the system designer to schedulability analysis is referred to as the task set *specification*. Commonly [18, 5, 34, 38, 32, 59], task set specifications are immutable in hard-real time models. The number of tasks, their WCET time, period, and deadline are provided by the system designer, not to be changed. Schedulability analysis determines if the task set specification is feasible. For `NPM-BUNDLE`, task sets are transformable (obeying some restrictions).

Transformation of a task set exploits the concavity of execution requirements, redistributing the threads of individual tasks to multiple tasks. A greater number of threads per job reduces the WCET of a task but increases the non-preemptive execution requirement. Conversely, a fewer number of threads per task increases the total WCET for all tasks while decreasing the non-preemptive execution requirement. Schedulability analysis in this non-preemptive setting encompasses the search for a distribution of the fixed number threads from the task set specification to a variable number of tasks, resolving the tension between a greater number of tasks and a greater number of threads per task to find a feasible task set.

Schedulability analysis is a process that begins by considering the current task set named the *anterior* task set $\hat{\tau}$. If the set is schedulable, the set is unmodified and processing ceases with a positive result. If the task set $\hat{\tau}$ cannot be scheduled as described, the

task set is transformed into a *posterior* task set τ, and processed again as an anterior set. Processing ceases with a negative result when there are no available transformations of $\hat{\tau}$.

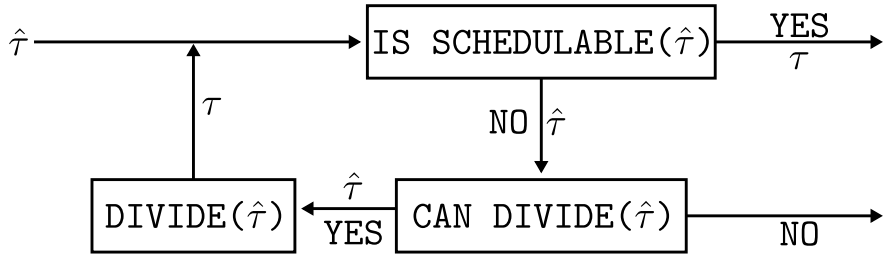


Figure 7.3: Schedulability and Transformable Task Sets

Figure 7.3 illustrates the schedulability analysis process. Division is the transformative operation of the process and is described in Subsection 7.1.1. The figure highlights the ability of a single task set to be both anterior and posterior to different sets during processing. To aid in explanation, properties of a task may be referred to in terms of the set the task was transformed from and to. By example, if the number of threads assigned to τ_i in the anterior set $\hat{\tau}$ is reduced by one in the posterior task set τ, the posterior threads of τ_i may be written as $m_i = \hat{m}_i - 1$.

As a process, schedulability analysis of the specified task set serves two purposes under this model. The first, is to determine if there exists a posterior task set which is feasible. Second, to produce the feasible posterior task set if one exists. It is the feasible posterior task set τ found by schedulability analysis that is then deployed on the target architecture. From the system designer's perspective, each task $\tau_i \in \tau$ of the specified task set is a request to execute m_i threads of the object o_i with shared periods p_i and deadlines d_i for **any** posterior task set τ. A task set specification is flexible, for one object there may be multiple tasks with variable numbers of threads per job. However, the specified m_i of a task is a ceiling on any m_i of a posterior task.

7.1.1 Dividing and Task Parts

A task set may be transformed by *dividing* tasks of the set. Dividing a task reduces the number of threads executed by each job, splitting the anterior task into two or more tasks in the posterior set.

Definition 7.1.1 (Task Division)**.** In the anterior task set $\hat{\tau}$, a task $\tau_i = (p_i, d_i, c_i(m_i))$ may be *divided* into two (or more) posterior tasks τ_j and τ_k with three restrictions: 1.) the periods of τ_j and τ_k are equal to the period of τ_i 2.) the relative deadlines of τ_j and τ_k are equal to the deadline of τ_i 3.) the sum of threads of τ_j and τ_k are equal to τ_i 4.) the objects of τ_i, τ_j, and τ_k are equal. Enumerated, the restrictions are:

1. $p_i = p_j = p_k$
2. $d_i = d_j = p_k$
3. $m_i = m_j + m_k$
4. $o_i = o_j = o_k$

Definition 7.1.2 (Partial Tasks)**.** When an anterior task τ_i is divided into τ_j and τ_k posterior tasks, τ_j and τ_k are referred to as *partial tasks* or *parts* of τ_i.

Definition 7.1.3 (Partial Task Set)**.** For convenience, the set of posterior tasks of τ_i is denoted Φ_i and called the *partial task set* of τ_i, where $m_i = \sum_{\tau_k \in \Phi_i} m_k$.

7.1.2 Worst-Case Execution Time Function Growth

Schedulability analysis for BUNDLE-based scheduling algorithms produce, for each task τ_i, a worst-case execution time combined with cache overhead (WCETO) function $c_i(m)$ in terms of m the number of threads per job scheduled in a cache-cognizant manner. NPM-BUNDLE extends the BUNDLE-based methods from a single multi-threaded to multiple (non-preemptively scheduled) tasks. For tasks that benefit from BUNDLE-based scheduling

and analysis, $c_i(m)$ is a strictly increasing discrete concave function. Tasks that do not are made vacuously concave by restricting jobs to release one thread.

In the WCETO analysis of `BUNDLE` and `BUNDLEP`, threads are assigned to paths through the conflict-free region graph of the executable object which maximize their contribution to $c_i(m_i)$. When considering the addition of a thread $m_i + 1$, only the greatest increase in $c_i(m_i)$ is permitted. Subsequently, the addition of thread $m_i + 2$ must increase $c_i(m_i)$ by less than or equal to the increase from $m_i + 1$ or the increase of $m_i + 1$ would not have been maximal. Therefore, for any $m_a < m_b < m_c$ the point $(m_b, c_i(m_b))$ lies above the straight line described by $(m_a, c_i(m_a))$ and $(m_c, c_i(m_c))$ – subsequently, $c_i(m_i)$ is concave.

A consequence of $c_i(m)$'s strictly increasing discrete concavity is a limit on the increase of the WCET as the number of threads increases. This property is referred to as the *concave restricted growth* (*concave growth* for brevity) of $c_i(m)$ and is leveraged in Sections 7.2 and 7.3.

Property 7.1.1 (Concavity Restriction on WCET Growth)**.** *For a strictly increasing discrete concave WCET function* $c_i(m)$*:*

$$\forall m \in \mathbb{N}^+ \mid c_i(m) - c_i(m-1) \geq c_i(m+1) - c_i(m) \tag{7.1.1}$$

It then follows for $m_x \geq m_y > 0$

$$\begin{aligned} c_i(m_x + 1) - c_i(m_x) &\leq c_i(m_x) - c_i(m_x - 1) \\ &\leq c_i(m_x - 1) - c_i(m_x - 2) \\ &\dots \\ &\leq c_i(m_y) - c_i(m_y + 1) \\ &\leq c_i(m_y) - c_i(m_y - 1) \end{aligned}$$

A WCET function $c_i(m)$ that obeys Property 7.1.1, will produce a value for $c_i(m+1)$ threads which is greater than $c_i(m)$. The difference between $c_i(m+1)$ and $c_i(m)$ must be less than or equal to the difference of $c_i(m)$ and $c_i(m-1)$. As the number of threads increase, $c_i(m)$ increases at a decreasing (or stable) rate.

For the purposes of comparison and evaluation in Section 7.3, an upper bound on the growth of $c_i(m)$ is called the *growth factor* $\mathbb{F}_i$ of τ_i. Growth factors relate the WCET of one thread $c_i(1)$ to the WCET of an arbitrary number of threads $c_i(m)$ for $m > 0$. A growth factor $\mathbb{F}_i \in (0, 1]$, for a task τ_i, is a real number that satisfies Equation 7.1.2.

Definition 7.1.4 (Growth Factor for τ_i)**.**

$$\forall m \mid c_i(m) \leq c_i(1) + (m-1) \cdot \mathbb{F} \cdot c_i(1) \tag{7.1.2}$$

For an $\mathbb{F}$ satisfying Equation 7.1.2, the pessimistic upper bound provides a linear function that can be rearranged to find an upper bound on the WCET of one thread in terms of m threads. The result is Equation 7.1.3, which will be used in the evaluation Section 7.3

when constructing task sets. Note, since $m \in \mathbb{N}$ each increase of m increases $c_i(m)$ by $\mathbb{F} \cdot c_i(1)$.

$$c_i(m) = c_i(1) + (m-1) \cdot \mathbb{F} \cdot c_i(1) \tag{7.1.3}$$

7.2 Non-Preemptive EDF Schedulability

Preemptive earliest deadline first (EDF) schedulability analysis for sporadic task sets has been well studied [5, 18, 60]. In the fully preemptive setting for which the algorithm is optimal, the overhead of a large number of preemptions may be a detriment to schedulability. Baruah [34] addresses this concern with an algorithm for calculating the non-preemptive chunk size q_i of each task $\tau_i \in \tau$. The non-preemptive chunk size q_i guarantees that task τ_i may execute up to q_i time units non-preemptively without introducing a deadline miss for any task in τ scheduled by preemptive EDF.

Section 7.2.3 introduces the non-preemptive feasibility algorithm Thread Per Job (TPJ) based upon the non-preemptive chunks algorithm from [34]. TPJ differs from the non-preemptive chunks algorithm by requiring the non-preemptive chunk size q_i of each task τ_i to be greater than or equal to its WCET: $c_i(m_i) \leq q_i$. As such, all jobs can be scheduled non-preemptively without fear of a deadline miss. To clearly convey TPJ, a description of the non-preemptive chunks algorithm and its dependencies is provided in the immediate subsection. Subsection 7.2.2 describes, by example, the available improvements to the non-preemptive chunks algorithm [34]. Subsection 7.2.4 defines and proves TPJ's optimality.

7.2.1 Non-Preemptive Chunks

The non-preemptive chunks algorithm depends on the demand bound function, EDF feasibility, ordering of absolute deadlines, and slack for the task set τ. Ordered absolute deadlines are given by $\{D_1, D_2, ...\}$ with $D_n < D_{n+1}$ for all n, where each task $\tau_i \in \tau$ contributes deadlines $D = k \cdot p_i + d_i$ for $k \in \mathbb{Z}^+$.

For a sporadic task τ_i the demand bound function for a task $\text{DBF}(\tau_i,t)$ is an upper bound on the amount of execution requirement generated from jobs released by τ_i over t units of time. The demand bound function is presented as Equation 7.2.1 as $\text{DBF}(\tau_i,t)$ modified from [18] to suit the NPM-BUNDLE task set model.

Definition 7.2.1 (Demand Bound Function for a Task τ_i and Interval t)**.**

$$\text{DBF}(\tau_i,t) = \max\left(0, \left(\left\lfloor \frac{t - d_i}{p_i} \right\rfloor + 1\right) \cdot c_i(m_i)\right) \tag{7.2.1}$$

When necessary for brevity, Equation 7.2.2 will be used to represent the sum of demand of all tasks over an interval of length t.

Definition 7.2.2 (Demand Bound Function for the Task Set τ and Interval t)**.**

$$\text{DBF}(\tau,t) = \sum_{\tau_i \in \tau} \text{DBF}(\tau_i,t) \tag{7.2.2}$$

Slack of the task set τ at deadline D_k is given by Equation 7.2.3. Intuitively, slack is the minimum time the processor will be idle over an interval. It is the difference between the demand over the interval and the length of the interval.

Definition 7.2.3 (Slack at Deadline D_k)**.**

$$\text{SLACK}(D_k) = \min_{j \leq k} \left(D_j - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k) \right) \tag{7.2.3}$$

For EDF, feasibility is determined by examining increasing time intervals and calculating the demand and supply. If demand exceeds supply, the system is infeasible. Equation 7.2.4 provides a formal definition of feasibility for the task set τ.

Definition 7.2.4 (EDF Feasibility Demand Bound Test)**.**

$$\forall t \geq 0, \left(\sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t) \right) \leq t \tag{7.2.4}$$

In [34], the number of time instants tested by Equation 7.2.4 is limited to the values of the ordered set of absolute deadlines $\{D_1, D_2, ...\}$. The ordered set of absolute deadlines is an infinite set, impractical for feasibility test. There is an upper bound on the value of all time instants (absolute deadlines) that must be tested and is denoted $T^*(\tau)$. Taken from [60], $T^*(\tau)$ is given by Equation 7.2.5 below. Among all tasks the largest deadline is $d_{\text{max}} = \max_{\tau_j \in \tau}(d_j)$. Utilization of τ_j is defined as $U_j = \frac{c_j(m_j)}{p_j}$. Among all tasks, the greatest difference of period and deadline is given by $\Delta_{max} = \max_{\tau_i \in \tau}(p_i - d_i)$. The hyper-period of all tasks (the least common multiple of all relative deadlines) is given by P.

Definition 7.2.5 (Feasibility Test Bound t for τ)**.**

$$T^*(\tau) = \min \left(P, \max \left(d_{\text{max}}, \frac{1}{1-U} \cdot \Delta_{max} \cdot \sum_{i=0}^{n-1} U_i \right) \right) \tag{7.2.5}$$

The non-preemptive chunks algorithm from [34] is presented (with additional details)

as pseudocode in Algorithm 8 and named NP-CHUNKS. In addition to determining if the task set is schedulable under EDF, the algorithm produces a non-preemptive chunk size q_j for each task $\tau_j \in \tau$. Jobs of τ_j may execute up to q_j time units non-preemptively without negatively impacting schedulability. This setting, where a task τ_j may execute non-preemptively for some period of time q_j is referred to as *limited-preemption*.

Algorithm 8 Non-Preemptive Chunks (NP-CHUNKS)

SLACK$(D_1) \leftarrow D_1 - \sum_{\tau_i \in \tau}$ DBF(τ_i, D_1)
for $\tau_j \in \{\tau_i \in \tau \mid (d_i = D_1)\}$ **do**
    $q_j \leftarrow c_j(m_j)$
end for
for $k \in \{D_2, D_3, ..., \}$ **do**
    if $D_k > T^*(\tau)$ **then**
        return *feasible*
    end if
    SLACK$(D_k) \leftarrow \min\left(\text{SLACK}(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k)\right)$
    if SLACK$(D_k) < 0$ **then**
        return *infeasible*
    end if
    for $\tau_j \in \{\tau_i \in \tau \mid (d_i = D_k)\}$ **do**
        $q_j \leftarrow$ SLACK(D_k)
    end for
end for

For a detailed description of NP-CHUNKS refer to [34]. To summarize, NP-CHUNKS begins by seeding the slack of the smallest interval D_1 and the non-preemptive chunk size of tasks with the smallest relative deadline equal to their WCET. During each iteration of $D_k \in \{D_2, D_3, ..., \}$, the slack for the interval D_k is calculated as the minimum of the current slack and the previous slack value. If there is less than zero slack, the system is infeasible. If the slack is zero or greater, each task with relative deadline equal to the current interval size is assigned the available slack as the task's non-preemptive chunk size. A task τ_j is assigned a non-preemptive chunk once, before assignment $q_j = \varnothing$ afterwards $q_j \neq \varnothing$. If

the interval being examined D_k exceeds $T^*(\tau)$, the task set must be schedulable.

7.2.2 Improving the Non-Preemptive Chunk Size

From the description of NP-CHUNKS in [34], there is an opportunity to improve the available slack for each of the k deadlines considered. Alogrithm 8 is pessimistic in the amount of available slack at any deadline D_k. To illustrate, consider the task set and intermediate values described by Figure 7.4.

i	p_i	d_i	m_i	$c_i(m_i)$
τ_0	4	2	1	1
τ_1	3	3	1	1
τ_2	3	3	1	1

P	D_k	$\tau_j : d_j = D_k$	DBF(τ,D_i)	SLACK(D_i)	q_j
12	$D_1 = 2$	τ_0	1	1	1
	$D_2 = 3$	τ_1	3	0	**0**
		τ_2	3	0	**0**

Figure 7.4: Example Task Set $\tau = \{\tau_0, \tau_1, \tau_2\}$

There are three tasks in the task set of Figure 7.4, with utilization of approximately 0.92. For τ_0, initialization assigns a non-preemptive chunk of $q_0 = 1$ time units. By observation, after release τ_0 may be delayed from execution by at most one time unit or it will miss its deadline. Consequently, the non-preemptive chunk size available to τ_1 and τ_2 is 1. As such NP-CHUNKS would be expected to find $q_0 = 1, q_1 = 1, q_2 = 1$.

Note, it is not possible for τ_0 to be blocked for 1 or more time units if both τ_1 and τ_2 execute non-preemptively for 1 time unit each. If τ_0 is blocked for less than 1 time unit by τ_1, then τ_0 will be the highest priority task when τ_1 completes (similarly for τ_2). It is impossible for τ_0 to be blocked 1 time unit or more by τ_1 or τ_2, τ_0 would have to be released at the same time instant as τ_1 or τ_2 and τ_1 or τ_2 would have to execute before τ_0, since the relative deadline of τ_0 is less than the other two, limited-preemption EDF executes τ_0: the task with earliest absolute deadline.

For τ_0, q_0 is calculated as expected $q_0 = c_0(m_0) = 1$, by Lines 2-4 of Algorithm 8. However, τ_1 has a non-preemptive chunk size of $q_1 = 0$. The reason is Line 9, where SLACK(D_2) is calculated which includes the execution demand of τ_1 and τ_2. Slack is an upper bound on the non-preemptive chunk size assigned to a task (in this case τ_1). Giving a task the available slack permits the task to execute longer, delaying higher priority jobs from executing in the interval by delaying them for as much time as there is slack.

By example in Figure 7.4, the available slack for τ_1 is determined from the interval of length $D_2 = 3$. The execution requirement of τ_1 and τ_2 is included in DBF(τ,3) because $d_1 = d_2 = 3$. Thus SLACK(D_2) is zero. Since τ_1's execution requirement is already included, it cannot further interfere over the interval D_2. Furthermore, τ_1 must have executed some portion without being preempted or the system would not be schedulable. Inclusion of τ_1's execution requirement within the interval over which slack is calculated for is pessimistic with respect to the non-preemptive chunk q_1 in this specific example, and q_j in general.

In the pseudocode implementation of NP-CHUNKS adopted from [34], Line 9 calculates the non-preemptive chunk size according Equation 7.2.6 (Equation 7 of Theorem 1 in [34]). Comparing Line 9 of Algorithm 8 to Equation 7.2.6 a mismatch between the algorithm and the infeasibility test is illuminated.

Definition 7.2.6 (Infeasibility Test, Equation 7, from [34])**.**

$$\exists \tau_j \in \tau, t \in [0, d_j) \mid t < q_j + \sum_{\substack{i=0 \\ i \neq j}}^{n-1} \text{DBF}(\tau_i, t) \tag{7.2.6}$$

If the condition of Equation 7.2.6 is satisfied for a task set τ, the task set is unschedulable given a limited-preemption task set with assigned non-preemptive chunks q. The

interval considered in the demand of Equation 7.2.6 is over $[0, d_j)$. The demand used in Algorithm 8 to calculate q_j is over the interval $[0, d_j]$. Extending the interval to include d_j introduces the pessimism identified by the example and is not required by Equation 7.2.6.

Figure 7.4 illustrates the pessimism of NP-CHUNKS found in [34]. The example uses the notation of assigning non-preemptive chunks to individual tasks from [34]. A later work [61] uses a different notation, assigning non-preemptive chunks to interval lengths for the remaining execution of a job. The conceptual pessimism of including demand for tasks with deadline equal to the current interval (described by Figure 7.4) is also found in [61].

7.2.3 Threads per Job (TPJ) Scheduling Algorithm

The NP-CHUNKS algorithm is modified for several purposes. First, the unnecessary pessimism is removed from chunk calculations. Second, the schedulability test is adapted to the NPM-BUNDLE task model. Lastly, when a given assignment of tasks and threads are infeasible, tasks are divided (when possible) to fit into their chunks. The division process is repeated until the task set is feasible, or no possible divisions remain and the task set is reported as infeasible. The algorithm is named the *Threads Per Job* (TPJ) scheduling algorithm.

A full description of TPJ is presented at the end of this subsection. To reach the complete description, an intermediate algorithm named *Bigger Non-Preemptive Chunks* (BNC) is presented as pseudocode in Algorithm 9. BNC removes the pessimism described in Section 7.2.2. The algorithm takes advantage of a property of the demand function DBF(τ,t) noted in [34].

Property 7.2.1 (Demand Change)**.** *Demand for a task does not change for values of* t *that do not equal an absolute deadline. In terms of the set of ordered absolute deadlines,* $\text{DBF}(\tau, D_{i-1}) = \text{DBF}(\tau, D_i - \epsilon)$*, for* $0 < \epsilon \leq (D_i - D_{i-1})$.

Algorithm 9 Bigger Non-Preemptive Chunks (BNC)

```
1:  SLACK(D_0) ← ∞
2:  for k ∈ {D_1, D_2, D_3, ...,} do
3:      if D_k > T*(τ) then
4:          return feasible
5:      end if
6:      SLACK(D_k) ← min (SLACK(D_{k-1}), D_k − Σ_{τ_i∈τ} DBF(τ_i, D_k))
7:      if SLACK(D_k) < 0 then
8:          return infeasible
9:      end if
10:     for τ_j ∈ {τ_i ∈ τ | (d_i = D_k)} do
11:         q_j ← min(c_j(m_j), SLACK(D_{k-1}))
12:     end for
13: end for
```

Line 11 of Algorithm 9 implements the improvement of BNC over NP-CHUNKS. The non-preemptive chunk q_j of task τ_j is taken from the slack of the previous interval D_{k-1} or the task's WCET $c_j(m_j)$, whichever is smaller. The algorithm verifies the condition set by Equation 7.2.6, selecting the correct interval length by Property 7.2.1, which precludes the inclusion of τ_j's execution requirement in the interval (and other tasks with deadline D_k).

The Threads per Job scheduling Algorithm 10, is a modification of BNC from limited-preemption EDF (EDF-LP) scheduling to non-preemptive EDF (EDF-NP). Input to the schedulability test is a task set specification τ, if TPJ returns a *feasibile* result there exists a posterior task set which can be scheduled by non-preemptive EDF and the posterior task set is returned as τ. An *infeasible* result indicates that TPJ could not guarantee τ would be schedulable by EDF-NP for any posterior task set. Since non-preemptive EDF is not optimal with respect to feasibility [59], TPJ is a sufficient test but cannot be necessary.

Algorithm 10 Threads-Per-Job (TPJ)

1: SLACK(D_0) $\leftarrow \infty$
2: **for** $k \in \{D_1, D_2, D_3, ..., \}$ **do**
3: **if** $D_k > T^*(\tau)$ **then**
4: return *feasible*
5: **end if**
6: **for** $\hat{\tau}_j \in \{\tau_i \in \tau \mid (d_i = D_k)\}$ **do**
7: **if** SLACK(D_{k-1}) $< \hat{c}_j(1)$ **then**
8: return *infeasible*
9: **end if**
10: $\Phi_j \leftarrow \{\hat{\tau}_j\}$
11: **if** SLACK(D_{k-1}) $< \hat{c}_j(\hat{m}_j)$ **then** ▷ Jobs must be divided
12: $\Phi_j \leftarrow$ DIVIDE($\hat{\tau}_j$,SLACK(D_{k-1}))
13: $\tau \leftarrow \tau \setminus \hat{\tau}_j$ ▷ Anterior task $\hat{\tau}_j$ is represented by Φ_j
14: $\tau \leftarrow \tau \cup \Phi_j$ ▷ Partial tasks include all threads of $\hat{\tau}_j$
15: **end if**
16: **for** $\tau_j \in \Phi_j$ **do**
17: $q_j \leftarrow c_j(m_j)$
18: **end for**
19: **end for**
20: SLACK(D_k) $\leftarrow \min\left(\text{SLACK}(D_{k-1}), D_k - \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, D_k)\right)$
21: **if** SLACK(D_k) < 0 **then**
22: return *infeasible*
23: **end if**
24: **end for**

Algorithm 10 (TPJ) modifies BNC, the modifications are limited to Lines 6-19. An additional benefit of BNC removing the pessimism of each q_j, is that each q_j can be calculated without consideration of the current task τ_j and the demand at D_k. Chunk values depend on the demand of D_{k-1} instead. This permits an efficient implementation of TPJ by moving the slack calculation of the current interval to the end of each iteration. Otherwise, if slack were calculated earlier in each iteration, the changes to demand resulting from Lines 6-19 would force the demand and slack of D_k to be recalculated.

The first notable change to BNC is introduced on Line 7, comparing the available slack to the WCET of a single thread of $\hat{\tau}_j$. If there is insufficient slack to execute just one

thread of $\hat{\tau}_j$ to completion, the task cannot be executed non-preemptively for any number of threads and the task set is infeasible non-preemptively.

Lines 11-15 introduce several subtle changes. For clarity, it is simpler to discuss the negative case (SLACK$(D_{k-1}) \geq \hat{c}_j(\hat{m}_j)$) before the positive. When there is sufficient slack for $\hat{m}_j$ threads to execute without preemption, $\hat{\tau}_j$ is given its full WCET ($\hat{c}_j(\hat{m}_j)$) as its non-preemptive chunk. In other words, no division of $\hat{\tau}_j$ is required and the posterior task set τ is unchanged (with respect to $\hat{\tau}_j$). Lines 11-15 are avoided, the algorithm progresses to the next task such that $d_i = D_k$.

However, in the positive case on Line 11 (when SLACK$(D_{k-1}) < \hat{c}_j(\hat{m}_j)$), $\hat{m}_j$ threads of $\hat{\tau}_j$ cannot feasibly execute without being preempted. Therefore, $\hat{\tau}_j$ must be divided. The DIVIDE procedure creates a partial task Φ_j set of $\hat{\tau}_j$, such that all tasks $\tau_p \in \Phi_j$ will complete within the available slack $c_p(m_p) \leq D_{k-1}$. The posterior task set τ has $\hat{\tau}_j$ removed, and is replaced by the partial set Φ_j maintaining the specified number of threads for $\hat{\tau}_j$.

For any task $\hat{\tau}_j$, the task is transformed into a partial task set Φ_j and assigned a non-preemptive chunk only once in the iteration where the absolute deadline D_k is equal to the relative deadline of the task: $D_k = \hat{d}_j$. Since tasks of τ are evaluated in strictly increasing absolute deadline order, the impact on demand and non-preemptive chunk sizes of processing $\hat{\tau}_j$ exclusively impacts demand for larger intervals $D_\ell > D_k$ and non-preemptive chunk sizes for tasks $\tau_\ell \in \tau$ with greater relative deadlines $d_\ell > \hat{d}_k$.

Property 7.2.2 (Divisions of $\hat{\tau}_j$ Exclusively Impacts Interval of Length $t \geq \hat{d}_j$)**.** *Division of $\hat{\tau}_j$ into the partial set Φ_j, and replacing $\hat{\tau}_j$ in τ with Φ_j will impact demand exclusively for intervals of length $D_k \geq \hat{d}_j$, slack of absolute deadlines $D_k > \hat{d}_j$ and therefore non-preemptive chunk values q_ℓ for tasks $\tau_\ell \in \tau$ with relative deadlines $d_\ell \geq D_k$*

By definition of DBF($\hat{\tau}_j$,t)*, no task of* Φ_j *or* $\hat{\tau}_j$ *can impact the task set* τ *demand* DBF(τ,t) *when* $t < d_j$*. Thus replacing* $\hat{\tau}_j$ *in* τ*, only affects the demand of intervals with length* $\hat{d}_j$ *or greater. Slack over the interval* D_k *is calculated from exclusively shorter intervals. Since the demand of the current interval* D_k *does not influence the slack at* D_k*, replacing* $\hat{\tau}_j$ *in* τ *only affects the slack of intervals with length greater than* D_k*. Non-preemptive chunk sizes are assigned based on the available slack, and only those assigned for an interval of length greater than* D_k *can be affected by replacing* $\hat{\tau}_j$ *in* τ*.*

Algorithm 11 DIVIDE

```
1: procedure DIVIDE(τ̂_j, q)
2:     Φ_j ← {}
3:     m ← argmax_{m∈ℤ⁺} (ĉ_i(m) ≤ q)
4:     r ← m̂_j
5:     while r > 0 do
6:         m_p ← min(r, m)
7:         τ_p ← (p̂_j, d̂_j, m_p, ĉ_j)      ▷ Posterior task, same period, deadline, WCET function.
8:         Φ_j ← Φ_j ∪ τ_p
9:         r ← r − m_p
10:    end while
11:    return Φ_j
12: end procedure
```

On Line 12 of the TPJ Algorithm 10, the task $\hat{\tau}_j$ is divided into Φ_j by the DIVIDE procedure. Pseudocode of DIVIDE is given by Algorithm 11. The number of tasks in Φ_j are determined by the maximum number of threads m of $\hat{\tau}_j$ that can execute non-preemptively within q time units. Each task $\tau_k \in \Phi_j$ is assigned m threads of $\hat{\tau}_j$ or however many remain, whichever is less. The result is that each task set has the following properties.

Property 7.2.3 (Partial Task Sets Returned from DIVIDE)**.** *The partial task set* Φ_j *of an anterior task* $\hat{\tau}$ *for a specific* q *value (and related maximum threads assigned per job* m *such that* $c_j(m) \leq q$*) contains posterior tasks where:*

1. *The exact number of posterior tasks is* $|\Phi_j| = \lceil \frac{\hat{m}_j}{m} \rceil$

2. *Exactly* $\lfloor \frac{\hat{m}_j}{m} \rfloor$ *tasks of* Φ_j *are assigned* m *threads per job.*

3. *There is at most one task* $\tau_g \in \Phi_j$ *with exactly* $m_g = \hat{m}_j \mod m$ *threads.*

7.2.4 Non-Preemptive Feasibility of TPJ and DIVIDE

The DIVIDE Algorithm 11 creates a partial task set Φ_j for an anterior task $\hat{\tau}_j$, assigning as many threads to each task in Φ_j as possible. Upon returning Φ_j to TPJ, $\hat{\tau}_j$ is replaced in the task set τ. Algorithm 11 is one method of dividing of $\hat{\tau}_j$ which TPJ could employ when creating the posterior task set τ. This section justifies DIVIDE's method by demonstrating the effect on schedulability and optimality of TPJ.

This section's ultimate objective is to clearly convey Theorem 7.2.3; concluding that TPJ is optimal with respect to task-level non-preemptive multi-threaded feasibility. The theorems that precede Theorem 7.2.3 establish minimal demand and WCET sums for partial sets created by DIVIDE necessary to illustrate TPJ's optimality.

Non-preemptive EDF scheduling of jobs of multiple threads ordered by a thread-level scheduler (such as BUNDLE or BUNDLEP) allows preemptions between threads of the same job but precludes preemptions between jobs. Each task benefits from the advantages of thread-level scheduling by the exclusive use of the processor and shared resources. Since task set specifications may be divided, a specification is feasible when threads of the specification $\hat{\tau}$ may be assigned to tasks such that the posterior task set τ is feasible by EDF-NP.

Definition 7.2.7 (npm-feasible)**.** A task set specification $\hat{\tau}$ is task-level non-preemptive multi-threaded feasible (*npm-feasible*) if there exists a posterior task set τ of $\hat{\tau}$ such that all multi-threaded jobs scheduled by EDF-NP will always meet their deadlines.

For the theorems that follow, unless necessary to discriminate between anterior and posterior tasks, the anterior task $\hat{\tau}_i$ will be written τ_i. The sum of the demand of the partial tasks of τ_i for an interval of length t is $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$.

Theorem 7.2.1 (Minimal Demand of Partial Task Sets Over All Intervals)**.** *For a partial task set Φ_i of an anterior task τ_i with m_i threads, minimizing $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,d_i)$ minimizes $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$ for all $t \geq 0$.*

Proof. Provided into two parts, when $t < d_i$ and $t \geq d_i$. The first portion is a simple direct argument. The second portion is by contradiction.

Part 1: When $t < d_i$, $0 = \sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$. By definition of the demand bound function (Equation 7.2.1) the execution requirement of a task is zero before the first possible deadline. All tasks $\tau_k \in \Phi_i$ share the same relative deadlines $d_k = d_i$ and absolute deadlines because $p_k = p_i$. These follow from the definition of division (Definition 7.1.1) and partial tasks (Definition 7.1.2). Since $t < d_i$, $\text{DBF}(\tau_k,t) = 0$ for all $\tau_k \in \Phi_i$. Therefore, $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$ will be minimal (exactly zero) when $t < d_i$, regardless of $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,d_i)$.

Part 2: When $t \geq d_i$, assume $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,d_i)$ is minimal and $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k,t)$ is not minimal. Since all partial tasks $\tau_k \in \Phi_i$ share absolute deadlines (as described in Part 1), demand for each task $\text{DBF}(\tau_k,t)$ increases only for values of t that equal absolute deadlines. Furthermore, the execution requirement of every τ_k increases exactly by $c_k(m_k)$ for each

absolute deadline of $\tau_i = \{D_1, D_2, ...\}$:

$$\text{DBF}(\tau_k, D_1) = 1 \cdot c_k(m_k)$$

$$\text{DBF}(\tau_k, D_2) = 2 \cdot c_k(m_k)$$

$$...$$

$$\text{DBF}(\tau_k, D_z) = z \cdot c_k(m_k)$$

Utilizing Property 7.2.1, for $t \geq d_i$ and D_z, where D_z is the greatest absolute deadline of τ_i less than or equal to t ($D_z \leq t$):

$$\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, t) = \sum_{\tau_k \in \Phi_i} z \cdot \text{DBF}(\tau_k, d_i) = z \cdot \sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d_i)$$

Because z depends on t (and is completely independent of the division of the partial task set), if $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, t)$ were not minimal then $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d_i)$ could not be minimal, contradicting the assumption.

Combining Parts 1 and 2, when the demand for the partial tasks of τ_i is minimized for the interval d_i, the demand of partial tasks of τ_i is minimized for all intervals of length $t \geq 0$. □

Corollary 7.2.1.1 (Minimal WCET Sum of Φ_i Minimizes Demand Over the Interval d_i)**.** *The demand of Φ_i over the interval d_i is minimized when the sum of WCET of Φ_i is minimized.*

Proof. Following directly from Theorem 7.2.1, where the demand over the interval d_i of each task $\tau_k \in \Phi_i$ is given by $\text{DBF}(\tau_k, d_i) = 1 \cdot c_k(m_k) = c_k(m_k)$. Then,

$$\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d_i) = \sum_{\tau_k \in \Phi_i} c_k(m_k)$$

Thus, minimizing $\sum_{\tau_k \in \Phi_i} c_k(m_k)$ minimizes $\sum_{\tau_k \in \Phi_i} \text{DBF}(\tau_k, d_i)$ □

Corollary 7.2.1.2 (Minimal WCET Sum of Φ_i Minimizes Demand Over all Intervals $t \geq 0$)**.** *The demand of Φ_i over alls interval $t \geq 0$ is minimized when the sum of WCET of Φ_i is minimized.*

Proof. Following directly from Theorem 7.2.1 and Corollary 7.2.1.1. □

Definition 7.2.8 (Assumptions of Theorem 7.2.2)**.** For the following theorem, there are several assumptions that must be upheld for the result to be valid. These assumptions are consequences of the non-preemptive setting and requirements of the task set specification.

1. All tasks τ_i must be characterized by strictly increasing discrete concave WCET function $c_i(m_i)$.

2. Any task $\tau_i \in \tau$ where $c_i(m_i) > q_i$ is not schedulable non-preemptively. Consequently, no assignment of m_i may cause $c_i(m_i) > q_i$ or the task set is infeasible.

3. The greatest number of threads assigned to a task τ_i such that $c_i(m_i) \leq q_i$ is named $m = \underset{m \in \mathbb{Z}^+}{\text{argmax}} \, (c_i(m) \leq q_i)$.

Theorem 7.2.2 (Minimal Sum of WCET of Φ_i for any q by DIVIDE)**.** *For an anterior task $\hat{\tau}_i$ and non-preemptive chunk size q,* DIVIDE *will produce a partial task set Φ_i with minimum WCET sum among all possible partial task sets of $\hat{\tau}_i$.*

Proof. To illustrate a contradiction, assume Φ_i returned from DIVIDE does not have the minimal WCET sum for a specific q and task $\hat{\tau}_i$. There must exist a partial task set Φ_k of $\hat{\tau}_i$ that differs, ie. $\Phi_i \neq \Phi_k$ and

$$\sum_{\tau_k \in \Phi_k} c_k(m_k) < \sum_{\tau_j \in \Phi_i} c_j(m_j)$$

By Property 7.2.3 of partial tasks created by DIVIDE, Φ_i will have at most one task with less than m threads assigned to it. For Φ_k to differ, it must have at least two tasks with less than m threads assigned to them. Call these two tasks with less than m threads $\tau_x, \tau_y \in \Phi_k$. Select τ_x as the task with the greater number of threads $m_x \geq m_y$.

Consider the impact on $\sum_{\tau_k \in \Phi_k} c_k(m_k)$ of moving one thread of τ_y to τ_x, as the operation of adding the difference of WCET values for $c_x(m_x + 1)$ and $c_y(m_y - 1)$ to the sum.

$$\begin{aligned} &\left(\sum_{\tau_k \in \Phi_k} c_k(m_k) \right) - c_x(m_x) + c_x(m_x + 1) - c_y(m_y) + c_y(m_y - 1) \\ = &\left(\sum_{\tau_k \in \Phi_k} c_k(m_k) \right) + (c_x(m_x + 1) - c_x(m_x)) - (c_y(m_y) - c_y(m_y - 1)) \end{aligned}$$

By the concave growth Property 7.1.1 and virtue of $m_y \leq m_x$, the quantity $(c_x(m_x + 1) - c_x(m_x))$ is less than or equal to $(c_y(m_y) - c_y(m_y - 1))$ so the difference must be less than or equal to zero. Therefore:

$$\left(\sum_{\tau_k \in \Phi_k} c_k(m_k) \right) + (c_x(m_x + 1) - c_x(m_x)) - (c_y(m_y) - c_y(m_y - 1)) \leq \sum_{\tau_k \in \Phi_k} c_k(m_k)$$

The WCET sum of Φ_k can be reduced by moving one thread of τ_y to τ_x. When $m_x = m$

no more threads may be assigned to τ_x or the system will be infeasible by Definition 7.2.8. While there are two (or more) tasks of $\tau_x, \tau_y \in \Phi_k$ with fewer than m threads assigned, moving one thread from τ_y to τ_x will reduce the WCET sum. By repeatedly moving tasks to reduce the WCET sum, Φ_k will satisfy all aspects of Property 7.2.3 of partial task sets created by DIVIDE, ie. $\Phi_i = \Phi_k$ after all moves have been completed. This contradicts the assumption that $\Phi_i \neq \Phi_k$ and the relationship of their WCET sums, therefor Φ_i is minimal.

□

Theorem 7.2.3 (TPJ is Optimal with Respect to npm-feasibility)**.** *For a task set specification $\hat{\tau}$,* TPJ *returns feasible if and only if there exists an npm-feasible posterior task set τ of $\hat{\tau}$.*

Proof. Forward Direction (TPJ returns feasible for $\hat{\tau} \implies \exists$ a posterior task set $\tau \mid \tau$ is npm-feasible): The TPJ algorithm returned a posterior task set τ where the infeasibility condition (Equation 7.2.6) is never satisfied across intervals of length $0 \leq t \leq T^*(\tau)$ and every job of $\tau_i \in \tau$ executes non-preemptively for $c_i(m_i) \leq q_i$ time units. Therefore, τ is npm-feasible.

Reverse Direction ($\exists$ a posterior task set $\tau \mid \tau$ is npm-feasible $\implies$ TPJ returns feasible for $\hat{\tau}$): For the purpose of demonstrating a contradiction, assume TPJ returns infeasible for an npm-feasible task set $\hat{\tau}$. Name the absolute deadline which TPJ returned infeasibility for D_x from the set ordered deadlines $\{D_1, D_2, ...\}$ and the task which generated D_x, $\hat{\tau}_x$. Name the set of tasks with relative deadlines smaller than $\hat{d}_x$, $\bar{\tau}$.

For any task $\tau_k \in \bar{\tau}$ and partial task set Φ_k of τ_k included in the posterior set τ, the number of tasks and threads assigned to each Φ_k cannot be affected by $\hat{\tau}_x$ due to $\hat{d}_x > d_k$ and Property 7.2.2. The combined set of posterior tasks of $\bar{\tau}$ in τ is denoted $\dot{\tau} = \cup_{\tau_k \in \bar{\tau}} \Phi_k$.

There are two cases where TPJ will return infeasible for $\hat{\tau}$, on Line 8 and Line 22. Both illustrate a contradiction with the respect to demand.

Line 8: If TPJ returns infeasible for $\hat{\tau}$ on Line 8 there is insufficient slack q_x to execute any one-thread job of $\hat{\tau}_x$ non-preemptively. Since slack is inversely related to demand, the demand of $\dot{\tau}$ is too great to allow any thread of τ_x as part of a feasible task set.

Line 22: If TPJ returns infeasible for $\hat{\tau}$ on Line 22, there is insufficient supply for Φ_x (the set of partial tasks of $\hat{\tau}_x$). By Corollary 7.2.1.1 and Theorem 7.2.2 the demand of Φ_x is minimal over all intervals for the available slack q_x. Due to Property 7.2.2 only tasks with shorter relative deadlines i.e. $\dot{\tau}$, can impact the demand of Φ_x by affecting q_x. In this case, the demand of $\dot{\tau}$ is too great for the demand of Φ_x to be included as part of a feasible task set.

By assumption $\hat{\tau}$ is npm-feasible, the infeasibility conditions on Lines 8 and 22 of TPJ indicate the demand of $\dot{\tau}$ is too great. However, TPJ adds each partial set Φ_k to $\dot{\tau}$ in increasing deadline order. By Property 7.2.2, every Φ_k added to $\dot{\tau}$ exclusively impacts the demand of larger deadlines. Every Φ_k increases the demand of $\dot{\tau}$ minimally starting with D_1, maximizing the slack available for partial task sets with greater deadlines; thus the demand of $\dot{\tau}$ is minimal and cannot be reduced. For $\hat{\tau}$ to be npm-feasible, there must be another partial task set that reduces $\dot{\tau}$'s demand, which is a direct contradiction. Therefore, TPJ must return feasible. □

7.3 Evaluation

Evaluation [11] of TPJ and the non-preemptive multi-threaded task model focuses on the schedulability ratio of synthetic task sets and a case study based upon the evaluation

of BUNDLEP [8]. The ratio of task set specifications deemed schedulable by TPJ for EDF-NP will be compared to NP-CHUNKS in both limited and fully preemptive settings for EDF. What follows is a description of the parameters to task set specification generation, the prescribed evaluation metrics, and analysis of the results.

7.3.1 Generating Task Sets

A specified task set τ is generated with four parameters, M the total number of threads of execution, U the target utilization, a maximum growth factor $\mathbb{F}$, and m the maximum number of threads per task. The number of threads M may be one of $\{3, 5, 7, 10, 25, 50, 100\}$ with dependent m values of $\{2, 2, 3, 4, 8, 16, 32\}$. Utilization varies from [0.1, 0.9] and the growth factor varies from [0.1, 0.9] independently by increments of 0.1.

Each task $\tau_i \in \tau$ is assigned m_i threads from a random uniform integer distribution over $[1, m]$, such that the sum of all threads is equal to $M = \sum_{\tau_i \in \tau} m_i$. A task's period p_i is from a uniform integer distribution over [10, 1000]. Utilization u_i of each task τ_i is calculated using the UUniFast(n, U) [62] algorithm, where $n = |\tau|$.

A task's WCET is assigned for m_i threads, $c_i(m_i) = \lceil p_i \cdot U_i \rceil$. Tasks are given a growth factor $\mathbb{F}_i$ in a uniform real distribution over $[0.1, \mathbb{F}]$. The remaining $m_i - 1$ WCET values are determined by substituting $\mathbb{F}_i$ into Equation 7.1.3. The relative deadline of τ_i, d_i is taken from a uniform integer distribution over $[\max(c_i(m_i), p_i/2), 1000]$.

For each combination of $(M, m, U, \mathbb{F})$, 1000 task sets specifications are generated. Figure 7.5 summarizes the parameters of task set generation. The smaller values of M are taken from [61] and the dependent m values were selected to avoid one task consuming more than half of the threads in the task set specification (where possible).

U	[0.1, 0.9]	M	{3, 5, 7, 10, 25, 50, 100}
$\mathbb{F}$	[0.1, 0.9]	m	{2, 2, 3, 4, 8, 16, 32}

Figure 7.5: Task Set Generation Parameters

Applicability of Parameters

To avoid favoring TPJ, the task set generation parameters m and $\mathbb{F}$ were carefully selected. For the threads per task m, a large m favors TPJ. Therefore, no single task my be assigned more than half the total threads: $m \leq \lfloor \frac{M}{2} \rfloor$ (except for $M = 3$).

The growth factor $\mathbb{F}$ is informed by previous results for BUNDLEP [8]. In [8], multi-threaded tasks are constructed from the Mälardalen WCET benchmarks [52]. Task analysis in [8] yields growth factors below 0.1 for several benchmarks. A lower bound (0.1) on $\mathbb{F}$ greater than observed values is pessimistic, resulting in less favorable results for TPJ.

7.3.2 Case Study

BUNDLEP's evaluation covers 18 benchmarks for distinct architecture configurations. An architecture configuration includes the block reload time (BRT), cycles per instruction (CPI), and number of cache lines. One of the least favorable in terms of the analytical benefit of BUNDLEP is a BRT of 100, CPI of one, and 32 cache lines. From this configuration, the WCET values and growth factors were extracted, growth factors ranging in the range $[0.08, 3.02]$.[1]

From these results of BUNDLEP 1000 task sets with 18 tasks (one per benchmark) and a total 100 threads were generated per utilization target. The utilization target ranged from 0.1 to 1.0 increments of 0.1. Threads were assigned to each task τ_i from a distribution

[1] Due to length restrictions the full listing of WECT and growth factors are omitted.

over $m_i \in [2, 8]$. Each tasks utilization, period, and deadline, $c_i(m_i)$ were assigned using the same method as synthetic tasks. The WCET values for fewer threads $1 \leq k < m_i$, were scaled such that the value of $c_i(k)/c_i(m_i)$ remained constant after the $c_i(m_i) = \lceil p_i \cdot U_i \rceil$ assignment.

7.3.3 Evaluation Metrics

TPJ is compared with the NP-CHUNKS schedulability test in non-preemptive (EDF-NP) and preemptive (EDF-P) settings. The focus of the evaluation is on the non-preemptive setting. The preemptive setting serves as a comparison to alternative scheduling strategies and the theoretical best case. For EDF-P, preemptions incur **no** penalty, CRPD or otherwise. In this highly advantageous setting for EDF-P, TPJ can still produce feasible non-preemptive task sets NP-CHUNKS deems infeasible in a preemptive setting!

To compare schedulability tests, each task set specification $\hat{\tau}$ is provided to TPJ without modification under EDF-NP scheduling. TPJ will transform the task set producing a posterior task set τ if a feasible one exists. A task set specification $\hat{\tau}$ cannot be provided directly to NP-CHUNKS, since NP-CHUNKS has no concept of threads per job.

To be suitable for analysis by NP-CHUNKS, a task set specification $\hat{\tau}$ is transformed into two posterior task sets. The first task set, τ^1 represents single-threaded tasks by including all threads of $\hat{\tau}$ as individual tasks. The second task set, τ^m represents the tasks of $\hat{\tau}$ as indivisible, executing all specified threads without preemption per job. Each task in τ^m benefits from the thread-level scheduler but does not expose the threaded nature of the task to the scheduling algorithm. This is achieved by modifying an anterior task $\hat{\tau}_j$ with $\hat{m}_j > 1$ and $\hat{c}_j(\hat{m}_j)$ to a posterior task τ_j with $m_j = 1$ and $c_j(1) = \hat{c}_j(\hat{m}_j)$.

Test	Task Set	EDF-NP	EDF-P
TPJ	$\hat{\tau}$	EDF-TPJ	-
NP-CHUNKS	τ^1	EDF-NP:1	EDF-P:1
	τ^m	EDF-NP:M	EDF-P:M

Figure 7.6: Schedulability Test Combinations

The NP-CHUNKS schedulability test will produce results for τ^1 and τ^m in both preemptive and non-preemptive settings. For non-preemptive schedulability analysis, each task $\tau_i \in \tau^1$ or τ^m must have a non-preemptive chunk size $q_i \geq c_i(m_i)$. When evaluating preemptive EDF schedulability for τ^1 and τ^m, the results are labeled EDF-P:1 and EDF-P:M respectively. When evaluating non-preemptive EDF schedulability, the results are labeled EDF-NP:1 and EDF-NP:M. Schedulability results for TPJ under EDF-NP scheduling are labeled EDF-TPJ. Table 7.6 gives a synopsis of the schedulability tests. Schedulability ratios for each of the combinations are calculated for every $(M, m, U, \mathbb{F})$ configuration.

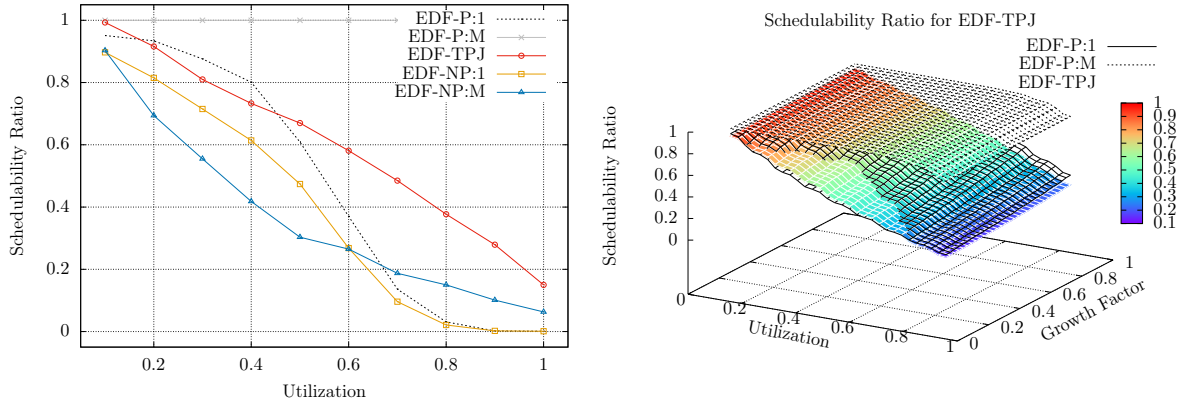
It must be noted that EDF-P:M is an unrealistic schedulability test. It serves only as a theoretical limit to the benefits of concave growth. Concave growth is a result of scheduling threads of the same job without preemption by another job with a BUNDLE-based thread-level scheduler. However, current BUNDLE implementations require that an executing task cannot be preempted by a different task. Such a preemption would destroy the cache benefits and analysis of BUNDLE scheduling. Analysis of EDF-P:M assumes preemptions between jobs are allowed and have zero cost. It is included as a reference for TPJ's performance, as a ceiling for what is theoretically possible given ideal (but likely impossible) conditions.

As a consequence of transforming multi-threaded task set specifications $\hat{\tau}$ to single-threaded task sets τ^1, some single threaded task sets may not be feasible. One reason for

a task set τ^1 to become infeasible is the utilization exceeding one, while τ^m and $\hat{\tau}$ have utilization less than one. In this setting, EDF-TPJ is capable of scheduling task sets that preemptive EDF cannot.

For a task set specification configuration $(M, m, U, \mathbb{F})$, call S the set of all task set specifications $\hat{\tau}$ generated for the configuration. Call s the set of τ^1 task sets transformed from $\hat{\tau} \in S$ such that τ^1 has utilization greater than one. The set s^{TPJ} is the subset of s deemed feasible by the TPJ schedulability test. That is, s^{TPJ} is the set of all tasks TPJ could schedule, yet EDF-P:1 could not (even) when CRPD values are zero.

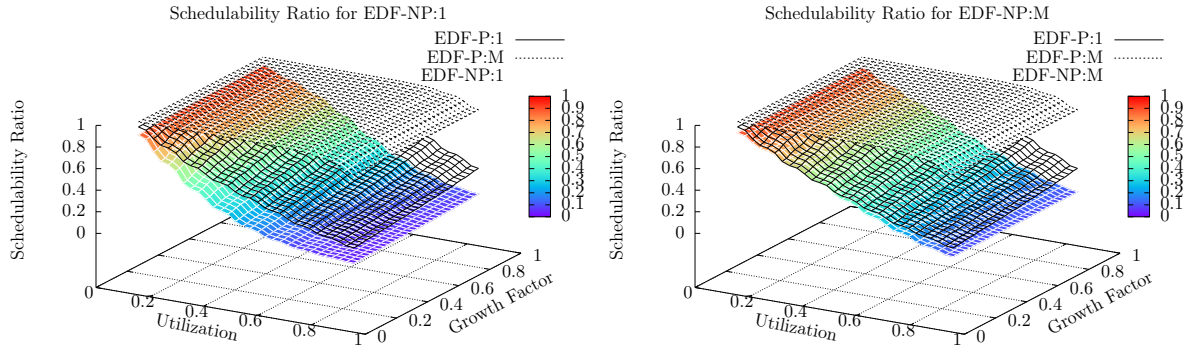
7.3.4 Results



(a) BUNDLEP Case Study (b) EDF-TPJ Summary

Figure 7.7: Case Study and EDF-TPJ Summary Results

Schedulability ratios from the BUNDLEP case study are given in Figure 7.7a. For the target architecture and 18 benchmarks, EDF-TPJ consistently outperforms the other non-preemptive algorithms. For preemptive EDF-P:1 (with zero cost preemptions), EDF-TPJ has higher schedulability ratios for the majority of target utilization values. EDF-TPJ's comparative performance increases with the target utilization. This case study demonstrates the benefit of TPJ to non-preemptive and (potentially) preemptive approaches.



(a) EDF-NP:1 Summary

(b) EDF-NP:M Summary

Figure 7.8: EDF-NP:1 and EDF-NP:M Summary

Figures 7.7b, 7.8a, and 7.8b, summarize the results for the synthetic task sets varied by the utilization and growth factor. Within each graph, the schedulability ratios provided by EDF-P:1 and EDF-P:M serve as references. The difference between EDF-P:1 and the subject of the graph illustrate the benefit of preemptive scheduling. Inclusion of EDF-PM highlights the theoretical limit of concave growth to schedulability.

Including EDF-P:1 and EDF-P:M in each of the summary graphs eases the comparison between EDF-NP:1, EDF-NP:M, and EDF-TPJ. Comparing EDF-NP:1 (7.8a) to EDF-NP:M (7.8b), illustrates the benefits of the model and scheduling mechanism. EDF-NP:M has a consistently higher schedulability ratio for all utilizations and growth factors. EDF-TPJ (7.7b) outperforms EDF-NP:M, with higher schedulability ratios for all utilizations and growth factors due to the ability to transform task sets. EDF-TPJ performs best among the non-preemptive tests across all configurations. Additionally, EDF-TPJ is able to schedule task sets deemed infeasible for EDF-P:1.

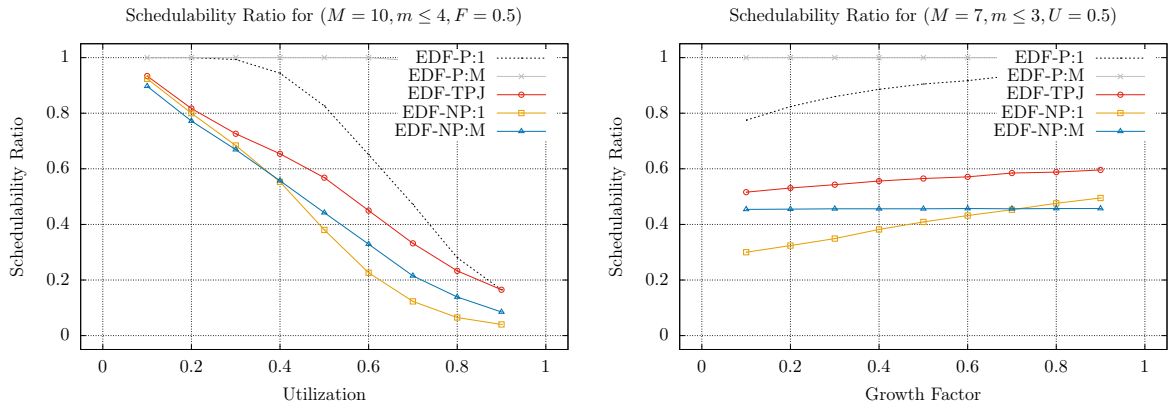
Table 7.9 summarizes the infeasible utilization findings for the synthetic tasks. For moderate and larger values of $M(\geq 25)$, the number of infeasible by utilization task sets

dominate the specifications. For 25, 50, and 100 total threads, the infeasible by utilization comprise 44, 59, and 74 percent of the task sets respectively, with EDF-TPJ finding 25, 34, and 45 percent feasible. This illustrates the large potential of the proposed model, in conjunction with concave growth WCET functions of thread-level schedulers (e.g. `BUNDLE` and `BUNDLEP`).

(M, m)	$(3, 2)$	$(5, 2)$	$(7, 3)$	$(10, 4)$	$(25, 8)$	$(50, 16)$	$(100, 32)$	Total
$\lvert S \rvert$	81000	81000	81000	81000	81000	81000	81000	567000
$\lvert s \rvert$	3131	4973	11744	18689	36565	49147	59412	183661
$\lvert s^{\text{TPJ}} \rvert$	465	291	1437	3065	9426	16912	25832	57428

Figure 7.9: $U > 1$ Feasibility

There are two noteworthy trends within the schedulability results. The simpler of the two is the relationship between utilization and schedulability ratio for a fixed growth factor. Figure 7.10a illustrates the trend common among $M \leq 10$ total threads. The trend for preemptive and non-preemptive schedulability tests when utilization increases is for the schedulability ratio to decrease. However, EDF-TPJ always outperforms the other non-preemptive tests.



(a) $(M, m, U, \mathbb{F}) = (10, 4, *, 0.5)$ (b) $(M, m, U, \mathbb{F}) = (7, 3, 0.5, *)$

Figure 7.10: $M \leq 10$ Performance

The second trend is slightly more complex. Figure 7.10b was selected for the smallest M and U values with visually distinct plots per schedulability test. The growth factor and the schedulability ratio are correlated. As the growth factor increases, so does the schedulability ratio. This is due to the utilization being held constant. When the growth factor is small, the WCET of the first thread of each task is larger. Larger WCET values are harder to schedule non-preemptively.

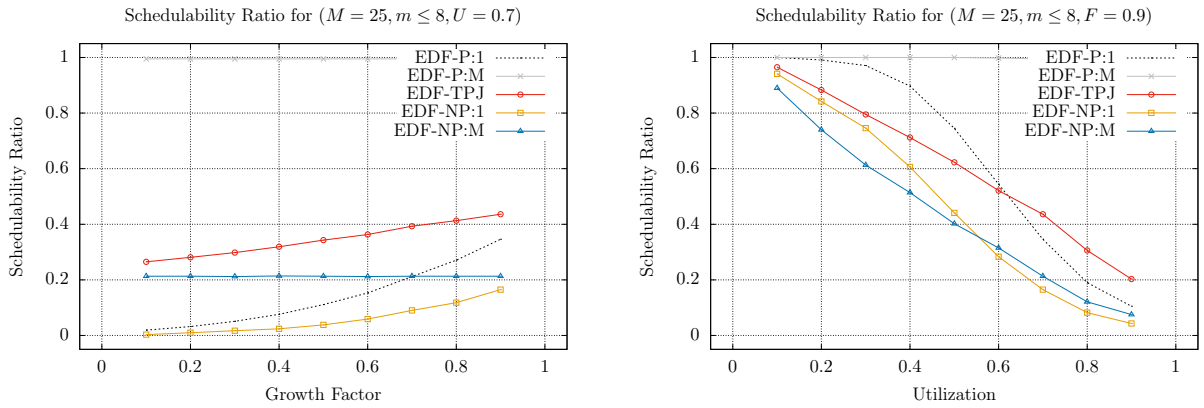


Figure 7.11: $M > 10$ EDF-TPJ Performance Above EDF-P:1

As M increases beyond 10 total threads, the number of infeasible by utilization task sets s grows. This contributes to the schedulability ratio of EDF-TPJ surpassing EDF-P:1 for threshold utilization and growth factor values. For $M = 25$, the threshold of utilization is between $[0.6, 0.7]$ shown in Figure 7.11.

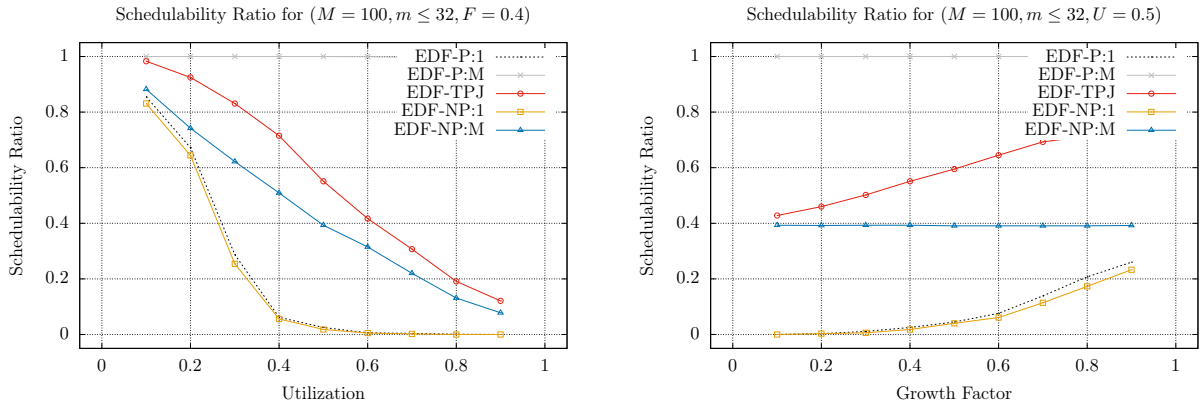


Figure 7.12: $M = 100$ EDF-TPJ Performance

For $M = 100$ and $\mathbb{F} \leq 0.4$, EDF-TPJ outperforms EDF-P:1. Figure 7.12 highlights the advantage of EDF-TPJ compared to EDF-P:1 by virtue of concave growth. It also highlights the benefit of dividing tasks, as the performance of EDF-NP:M is always below EDF-TPJ.

The comparative performance of EDF-TPJ is at its lowest for $M < 10$ threads and $U > .4$ utilization. In these ranges EDF-TPJ maintains the highest schedulability ratio among the non-preemptive methods, but the ratio is closer to EDF-NP:M or EDF-NP:1 than EDF-P:1. This suggests, the decrease in EDF-TPJ's performance is more likely due to the non-preemptive setting combined with larger WCET values for individual threads.

7.4 Summary

The primary goal of `NPM-BUNDLE` is to create a multi-task scheduling technique and schedulability test for the `BUNDLE`-based single task thread-level schedulers. In addition to achieving the primary goal, the scheduling technique and schedulability test developed for the multi-task `BUNDLE`-based scheduler can be applied to any thread level scheduler with strictly increasing discrete concave WCET functions. This allows any compatible thread-level scheduling technique to benefit from the `TPJ` approach developed in this work. As

a non-preemptive multi-threaded schedulability test TPJ is optimal with respect to npm-feasibility, always producing a feasible task set if one is schedulable by EDF-NP.

CHAPTER 8 MULTI-PROCESSOR MULTI-TASK BUNDLE

BUNDLE, BUNDLEP, and NPM-BUNDLE are limited to the scheduling and analysis of uniprocessor systems. Expanding the inter-thread cache benefit to multi-processor systems may take one of several approaches. Of the available approaches, this chapter describes the first, bringing BUNDLE's perspective to directed acyclic graph (DAG) parallel tasks and named ITCB-DAG. ITCB-DAG is seen as an initial work in the multi-processor setting. As such, the goal of ITCB-DAG is to demonstrate the potential benefit in this novel setting. To begin incorporating the inter-thread cache benefit to multi-processor systems for DAG tasks the following contributions are made as part of ITCB-DAG.

1. Incorporation of executable objects and thread counts to the DAG model named the directed acyclic graph tasks with objects and threads model, abbreviated DAG-OT.
2. The concepts of collapse and candidacy for collapse of nodes within a DAG-OT task.
3. The Dedicate Core Reduction Algorithm which reduces the number of cores reserved for a high utilization task.
4. Heuristics for ordering the collapse of nodes within a single DAG-OT task.
5. A synthetic evaluation demonstrating the impact of the inter-thread cache benefit for DAG tasks.

Figure 8.1: Summary of ITCB-DAG contributions

These contributions are presented in the following sections. Section 8.1 supplies the necessary background for DAG tasks including the existing model and proposed changes to the existing model. Section 8.2 describes the collapse operation and impact upon DAG tasks. Section 8.3 introduces the general algorithm for collapsing nodes within a task as well as the schedulability test for a DAG task set. Section 8.4 describes the proposed

heuristics for ordering candidates for collapse. Section 8.5 discusses the impact of collapse upon low utilization tasks. Section 8.6 describes the methods, metrics, and results of the synthetic evaluation.

8.1 Background and Related Work

In previous chapters, graphs represent executable objects as control flow graphs, conflict free regions, or conflict free region graphs. In this chapter, graphs will represent parallel tasks. Individual nodes within these graphs will encapsulate complete executable objects to be executed upon a single processor. This shifts the focus from the analysis and scheduling mechanisms of `BUNDLE` and `BUNDLEP`, to treating the `BUNDLE` techniques as reusable components within DAG tasks.

Existing works on parallel DAG tasks commonly share notation. These common symbols conflict with those used to describe `BUNDLE`. Within this chapter the notation and symbols in Table 8.1 supersede those previously defined in favor of the common notation found in [63] and others.

τ	Set of n tasks {τ_0, τ_1, ..., τ_{n-1}}
$\tau_i = (p_i, d_i, G_i)$	Task i
p_i	Minimum inter-arrival time of τ_i
d_i	Relative deadline of τ_i
$G_i = (V_i, E_i)$	Directed Acyclic Graph of τ_i
V_i	Nodes of graph G_i
E_i	Edges of graph E_i
o_i	Executable object of τ_i
o_v	Executable object of node $v \in V_i$
m	Number of cores in the target system

Table 8.1: `ITCB-DAG` Notation

The DAG model of hard real-time tasks [63] defines the set of n sporadic tasks τ as $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. A task $\tau_i = (p_i, d_i, G_i)$ is a tuple of minimum inter-task arrival time p_i, relative deadline d_i, and directed acyclic graph G_i. The set of n DAGs is denoted $\mathbb{G} = \{G_1, G_2, ..., G_n\}$. A task's DAG G_i represents the parallelism and dependencies of execution within the task. A DAG $G_i = (V_i, E_i)$ is a tuple of vertexes V_i and edges E_i.

A node $v \in V_i$ represents the execution of a single thread. A thread executes on exactly one of the m cores of the target architecture (or distributed system). Each node is associated with an executable object o_v: a set of machine instructions reachable from a single entry point. A worst-case execution c_v time is associated with every node v; an upper bound on the execution time required to complete the thread without interruption on a single core. An edge $(u, v) \in E_i$ indicates an execution dependency between $u, v \in V_i$. For v to begin execution on any core, all immediate predecessors $\{u|(u, v) \in E_i\}$ must run to completion.

For simplicity of analysis, every DAG G_i must have exactly one source and sink node, $s, t \in V_i$ respectively. A source s has no incoming edges, $\nexists u \mid (u, s) \in E_i$. A sink t has no outgoing edges, $\nexists v \mid (t, v) \in E_i$. It is possible for a DAG to have multiple sources and sinks. When a DAG contains multiple sources, the DAG is augmented by adding an "empty source": a single node with zero execution cost that is connected by outgoing edges to existing sources. Similarly, for a DAG with multiple sinks an "empty sink" is added with zero execution cost connected by incoming edges from the existing sinks.

Jobs of a task begin with one thread of s on one core. Jobs terminate when the single thread of t completes. During the execution of a job, up to m cores may execute any of the $v \in V$ threads in parallel. A task $\tau_i \in \tau$ generates a potentially infinite number of jobs,

each arriving no less than p_i time units after the previous job. All jobs of τ_i must complete within d_i time units.

An example DAG task is shown in Figure 8.2. Accompanying each node is a single-threaded WCET. For u and v, their WCET values are $c_u = 20$ and $c_v = 10$ respectively. Edges illustrate the dependency order of execution, such as (s, v) precluding v from executing until s has completed.

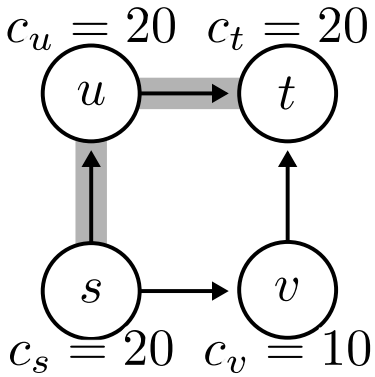


Figure 8.2: A DAG Task

For a DAG $G_i = (V_i, E_i)$, the length of a path through the graph is the sum of WCET values of all nodes along the path. The *critical path* λ_i of G_i, is a path from s to t with the greatest length L_i – named the *critical path length*. If multiple paths exist with length equal to L_i, only one is selected as the critical path. The *workload* of G_i is the sum of all WCET values $v \in V_i$. *Utilization* of the task τ_i is the ratio of its workload and minimum inter-arrival time.

Definition 8.1.1 (Critical Path Length of G_i)**.**

$$L_i = \sum_{v \in \lambda_i} c_v \tag{8.1.1}$$

Definition 8.1.2 (Workload of G_i)**.**

$$C_i = \sum_{v \in V_i} c_v \tag{8.1.2}$$

Definition 8.1.3 (Utilization of G_i)**.**

$$u_i = C_i/T_i \tag{8.1.3}$$

Definition 8.1.4 (Utilization of τ)**.**

$$U = \sum_{\tau_i \in \tau} u_i \tag{8.1.4}$$

In Figure 8.2, the critical path $\lambda = \langle s, u, t \rangle$ is highlighted. The calculated critical path length is $L = c_s + c_u + c_t = 60$ and workload $C = c_s + c_u + c_v + c_t = 70$.

8.1.1 Federated Scheduling

Federating scheduling [63] is a partitioned scheduling algorithm and analysis method developed for parallel DAG task sets. It divides the task set τ into two disjoint sets. Tasks with utilization greater than one are placed in the *high utilization task set* τ_{high}. The *low utilization task set* τ_{low} contains the remainder of τ. Every task τ_i of τ_{high} is assigned m_i dedicated cores. Only threads of τ_i may execute on the m_i cores dedicated to it. Calculating the number of dedicated cores required by a task τ_i to guarantee all jobs of the high utilization task complete before their deadlines is achieved by Equation 8.1.5.

$$m_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \tag{8.1.5}$$

The number of cores allocated to all high utilization tasks is denoted $m_{high} = \sum_{\tau_i \in \tau_{high}} m_i$. The remaining cores of low utilization tasks are denoted $m_{low} = m - m_{high}$. A task set τ is schedulable under federated scheduling if m_{low} is non-negative and all tasks of τ_{low} can be execute on the m_{low} cores without missing a deadline. Under federated scheduling, low

utilization tasks are scheduled *sequentially* where only one node of the task executes at any time on any core, and the selection of which node to execute obeys the dependency relationship of the task's graph.

Any greedy, work-conserving, parallel scheduler can be used to schedule a high utilization task $\tau_i \in \tau_{high}$ on its m_i dedicated cores. Low utilization tasks are treated as sequential tasks, executing at most one thread of a job at a time. Any multiprocessor scheduling algorithm (such as partitioned EDF) can be used to schedule all the low-utilization tasks on the m_{low} allocated cores.

Under federated scheduling DAG tasks execute on a parallel system with m identical cores. Requiring uniform cores ensures the validity of the WCET bound for each node regardless of where the thread executes. Furthermore, each core must possess identical cache configurations (hierarchy, size, etc.), memory architecture, and be timing-compositional [57]. Doing so guarantees the worst-case execution time and cache overhead of every node will be consistent across all cores. WCETO analysis is limited to the per-core dedicated instruction caches. Data caches and shared caches are not considered as part of `ITCB-DAG`

8.1.2 Proposed Model Changes

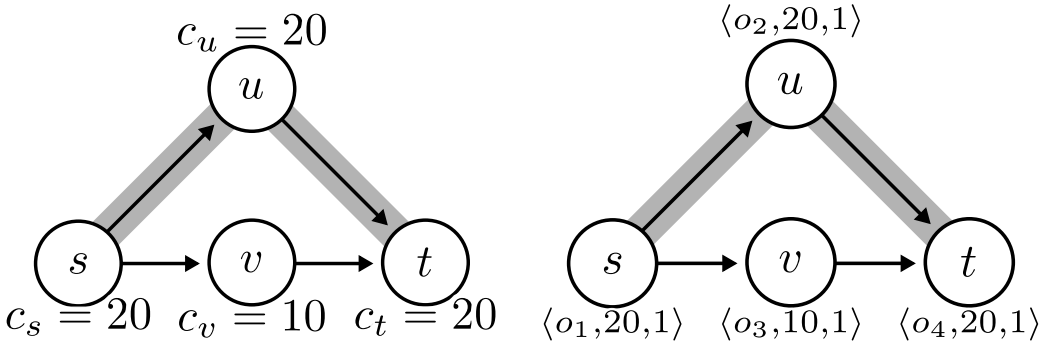
To incorporate the inter-thread cache benefit to parallel DAG tasks, a change to the model's description of nodes is proposed. For clarity, the existing model is referred to as the directed acyclic graph model of parallel tasks or simply "the DAG model", the proposed model is named the DAG with objects and threads or "the DAG-OT model".

For a DAG $G_i = (V_i, E_i)$ in the DAG model, two nodes $u, v \in V_i$ represent the release of

threads regardless of the executable object the threads execute. Inter-thread cache benefits can only be applied to identical executable objects. Thus, the first proposed change to a node $v \in V_i$ under DAG-OT is the inclusion of the executable object o_v in its description.

For a node $v \in V_i$ in the DAG model, the execution of a thread is bounded by a single WCET value c_v. WCETO analysis produces a function which bounds the execution of a specific number of threads scheduled by `BUNDLE` that includes the inter-thread cache benefit. The second proposed change is to append the number of threads assigned to a node η_v and the WCETO value as a function $c_v(\eta) : \mathbb{N}^+ \rightarrow \mathbb{R}^+$.

Combining the proposed changes, a node $v \in V_i$ in the DAG-OT model is represented by a tuple $v = \langle o_v, c_v(\eta), \eta_v \rangle$. Figure 8.3 presents the differences between the DAG and DAG-OT models visually. Herein, a consistent illustrative shorthand is used, with the order of nodes tuple's preserved and the critical path highlighted in gray.



(a) DAG model (b) DAG-OT Model

Figure 8.3: From DAG to DAG-OT

Nodes of the DAG-OT model are compatible with nodes of the DAG model [63], where nodes from the DAG model can be expressed as $v = \langle o_v, c_v(\eta_v), \eta_v = 1 \rangle$ under DAG-OT. This is illustrated by Figures 8.3a and 8.3b, which are equivalent.

For the DAG-OT model, the definitions of critical path length and workload must be

updated by Equations 8.1.6 and 8.1.7.

Definition 8.1.5 (DAG-OT Critical Path Length of G_i)**.**

$$L_i = \sum_{v \in \lambda_i} c_v(\eta_v) \tag{8.1.6}$$

Definition 8.1.6 (DAG-OT Workload of G_i)**.**

$$C_i = \sum_{v \in V_i} c_v(\eta_v) \tag{8.1.7}$$

8.1.3 Discrete Concave Functions and Growth Factors

As described in Section 7.1.2, `BUNDLE` analysis produces discrete concave WCETO functions. This property is applied directly to nodes of DAG-OT tasks, which are characterized by a growth factor $\mathbb{F}_v, v \in V_i$. Equation 7.1.2 is modified from the `NPM-BUNDLE` setting to nodes of DAG-OT tasks by Equation 8.1.8.

Definition 8.1.7 (Growth Factor $\mathbb{F}$)**.** For a node $u \in V_i$ of a DAG G_i, the *growth factor* of u is a number $\mathbb{F}_u \in (0, 1]$ that satisfies Equation 8.1.8 for all $\eta_u \geq 1$.

$$c_u(\eta_u) \leq c(1) + \mathbb{F}_u \cdot (\eta_u - 1) \cdot c_u(1) \tag{8.1.8}$$

An example for a node u, associated $c_u(\eta_u)$, and growth factor $\mathbb{F}_u = .5$ is shown in Figure 8.4. The values of $c_u(\eta_u)$ are $10, 15, 17, 18, 19$ for $\eta_u \in [1, 5]$. While any growth factor greater than .5 would satisfy the definition, the minimum was selected for illustrative purposes.

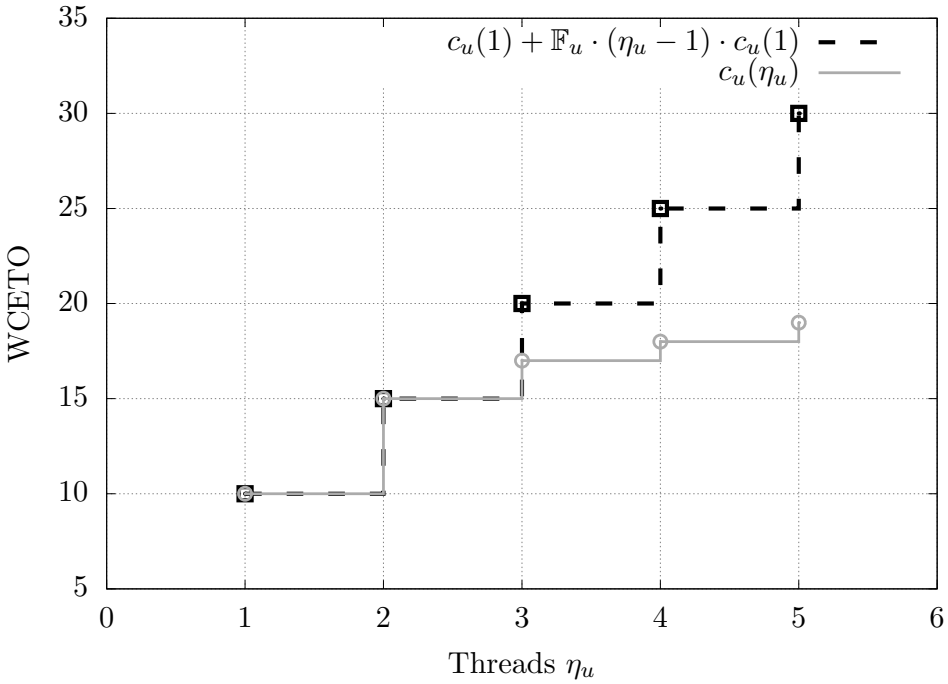


Figure 8.4: Example Growth Factor

8.1.4 Related Work

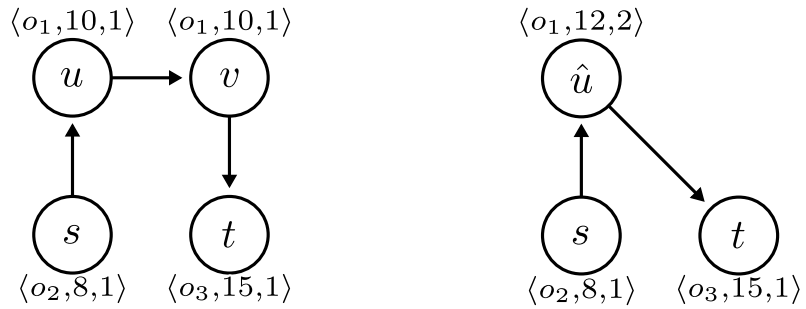
Parallel hard real-time tasks modeled by DAGs may be scheduled by federated [64, 65, 66] or global [67, 68, 69, 70] policies. Federated scheduling purports to improve the analytical bounds of global scheduling. Further improvements of federating scheduling include conditional DAGs, where edges between nodes may not be traversed unless a logical condition is met [71, 65, 72]. Resource consideration for federated scheduling has been studied in an energy-aware setting [73], and spin-lock blocking analysis performed in [74]. However, none of these works address the impact of cache memory for parallel DAG tasks. There are no known works that consider the impact of caches upon parallel DAG tasks

8.2 Collapsing Nodes

To bring the inter-thread cache benefit to the DAG-OT model, the concept of *collapsing* nodes is proposed. Under the DAG-OT model, two nodes $u, v \in V_i$ which execute the same object $o_u = o_v$ may potentially be combined into a single node. Nodes that share the same executable object are referred to as *candidates for collapse*. Collapsing two nodes into a single node turns two distinct execution requests executing on (possibly) distinct cores, into a single request to execute the combined threads on one core using BUNDLE scheduling. By virtue of BUNDLE's analysis incorporating the inter-thread cache benefit, the WCETO of the combined node may be less than the sums of the individual nodes.

Definition 8.2.1 (Candidate for Collapse)**.** For a DAG $G_i = (V, E)$ and nodes $u, v \in V$, u and v are *candidates* for collapse if and only if they share an executable object $o_u = o_v$.

To illustrate, consider Figure 8.5. Nodes u and v share the same executable object o_1. If the WCETO of one thread scheduled by BUNDLE on one core is 10 and two is 12, collapsing u and v reduces the workload (and potentially the critical path length) by 8.



(a) Pre-Collapse (b) Post-Collapse

Figure 8.5: Node Collapse

Collapse restricts the execution of threads and cores. In Figure 8.5 pre-collapse u and v may have executed on distinct cores. Post-collapse the combined threads of u and v must

execute on the same core scheduled by BUNDLE. To differentiate between pre and post-collapse values a "hat" will be used for the latter. In Figure 8.5, before collapse u and v have one thread each. Collapsing the two nodes into $\hat{u}$ joins the two threads $\eta_{\hat{u}} = 2 = \eta_u + \eta_v$. The pre-collapse workload is $C_i = 43$ and post-collapse workload $\hat{C}_i = 35$. The reduction in workload is due to the concave WCETO function $c_u(\eta) = c_v(\eta) = c_{\hat{u}}(\eta)$, where $c_u(1) = 10$ and $c_u(2) = 12$.

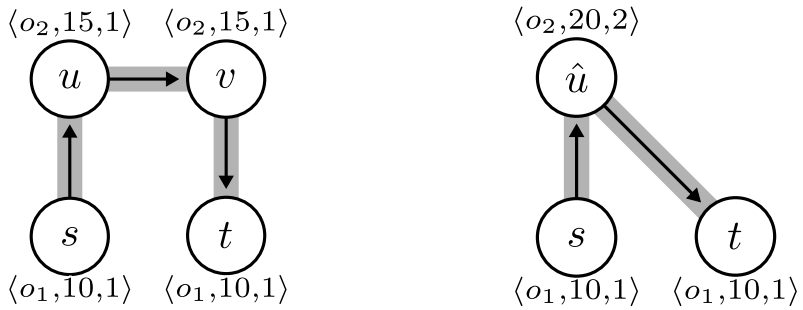
Formally, the collapse operation is defined as follows.

Definition 8.2.2 (Collapse $\hat{u} \leftarrow u \bowtie v$)**.** For pre-collapse nodes $u, v \in V_i$ of G_i, collapsing u and v (denoted $u \bowtie v$) into $\hat{u}$ modifies G_i by the following, resulting in a new DAG named $\hat{G}_i$.

1. $V_i \leftarrow V_i \cup \hat{u}$: A new blank node $\hat{u}$ is added to V_i
2. $\eta_{\hat{u}} \leftarrow \eta_u + \eta_v$: $\hat{u}$ is assigned the combined total number of threads
3. $o_{\hat{u}} \leftarrow o_u$: $\hat{u}$ is assigned the shared executable object
4. $c_{\hat{u}} \leftarrow c_u$: $\hat{u}$ is assigned the shared WCETO function
5. $\forall (x, y) \in E_i | y = u \vee y = v : E_i \leftarrow (x, \hat{u})$: incoming edges of u and v are copied to $\hat{u}$
6. $\forall (x, y) \in E_i | x = u \vee x = v : E_i \leftarrow (x, \hat{u})$: outgoing edges of u and v are copied to $\hat{u}$
7. $V_i \leftarrow V_i \setminus \{u, v\}$: u and v are removed from V_i
8. $\forall (x, y) \in E_i | y = u \vee y = v : E_i \leftarrow E_i \setminus (x, y)$: incoming edges of u and y are removed
9. $\forall (x, y) \in E_i | x = u \vee x = v : E_i \leftarrow E_i \setminus (x, y)$: outgoing edges of u and v are removed

8.2.1 Infeasibility and the Impact of Collapse

Collapsing nodes may reduce the critical path length L_i. This is illustrated by Figure 8.6, where the pre-collapse critical path length is $L_i = 50$. After collapsing $u \bowtie v \to \hat{u}$, the critical path length of $\hat{G}_i$ is $\hat{L}_i = 40$.



(a) Pre-Collapse (b) Post-Collapse

Figure 8.6: Critical Path Reduction

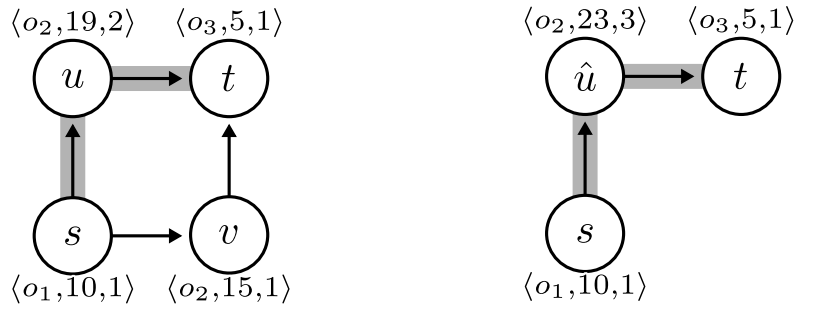
Observation 1 (Critical Path Reduction)**.** *For a DAG $G_i = (V, E)$ and candidates nodes $u, v \in V$, the collapse of $u \bowtie v$ into $\hat{u}$ may reduce the critical path length in $\hat{G}_i$: $\hat{L}_i \leq L_i$.*

Under the DAG model a task τ_i is infeasible (for any number of dedicated cores m_i) if the critical path length is greater than the deadline, i.e., $L_i > D_i$. A task τ_i deemed infeasible due to critical path length and period under the DAG model ($L_i > D_i$) may become feasible (and possibly schedulable) under the DAG-OT model by collapse and Observation 1 ($\hat{L}_i \leq D_i$). Thus the $L_i > D_i$ infeasibility test does not apply pre-collapse to the DAG-OT model. However, for any post-collapse $\hat{G}_i$ of τ_i if $\hat{L}_i > D_i$ the task set is unschedulable under DAG-OT.

Observation 2 (Critical Path Extension)**.** *For a DAG $G_i = (V, E)$ and candidates nodes $u, v \in V$, the collapse of $u \bowtie v$ into $\hat{u}$ may extend the critical path length in $\hat{G}_i$: $\hat{L}_i \geq L_i$.*

In contrast to Observation 1, collapse may extend the critical path length. This can

occur when one of the candidate nodes $u, v \in V$ lies on the pre-collapse critical path and the other does not. In Figure 8.7, u lies on the pre-collapse critical path. Collapsing $u \bowtie v \to \hat{u}$ increases the critical path length $\hat{L}_i$ compared to L_i by $c_u(\eta_u + \eta_v) - c_u(\eta_u)$.



(a) Pre-Collapse $L_i = 34$ (b) Post-Collapse $\hat{L}_i = 38$

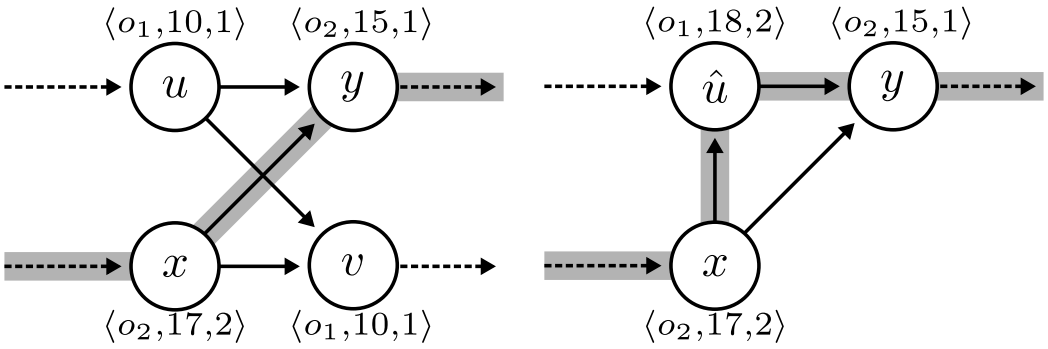
Figure 8.7: Critical Path Extension

Observation 3 (Workload Decrease)**.** *For a DAG $G_i = (V, E)$ and candidates nodes $u, v \in V$, the collapse of $u \bowtie v$ into $\hat{u}$ will reduce the workload $\hat{C}_i \leq C_i$.*

For candidates $u, v \in V$, their contribution to the workload of C_i is $c_u(\eta_u) + c_v(\eta_v)$. The contribution of $\hat{u} \leftarrow u \bowtie v$ to $\hat{C}_i$ is $c_{\hat{u}}(\eta_{\hat{u}}) = c_u(\eta_u + \eta_v)$. Since, $c_u(\eta)$ is a concave function, $c_u(\eta_u + \eta_v) \leq c_u(\eta_u) + c_v(\eta_v)$ and $\hat{C}_i \leq C_i$.

Observation 4 (Collapse Occlusion)**.** *For a DAG $G_i = (V, E)$, candidates (u, v) and (x, y), the collapse of $u \bowtie v$ may prevent the collapse of $x \bowtie y$.*

Collapsing one candidate (u, v) may preclude the collapse of another. For example, consider Figure 8.8. By collapsing (u, v) the pair (x, y) cannot be collapsed – doing so would introduce a cycle into the DAG.



(a) Pre-Collapse (b) Post-Collapse

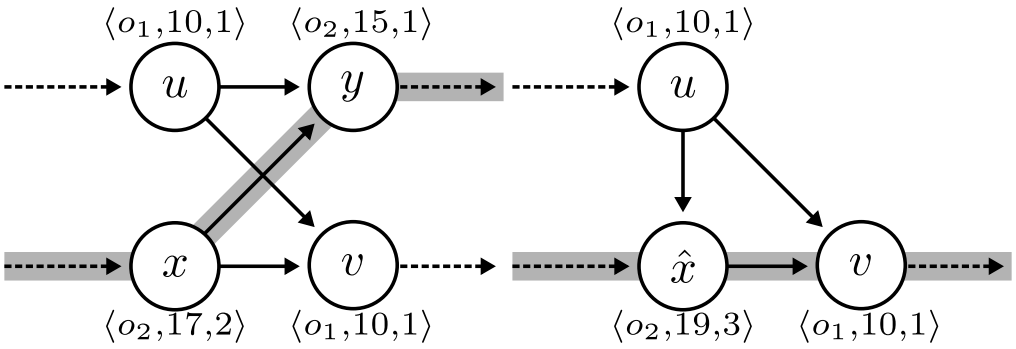
Figure 8.8: Collapse of (u, v) before (x, y)

Given a deadline $D_i = 40$ the result of collapsing (u, v) with respect to the workload, critical path length, and dedicated cores are summarized in Table 8.2.

C_i	L_i	m_i	$u \bowtie v$	$\hat{C}_i$	$\hat{L}_i$	$\hat{m}_i$
52	32	$\lceil 2.5 \rceil$	$\rightarrow$	50	33	$\lceil 2.42 \rceil$

Table 8.2: Collapse of u and v from Figure 8.8

Observation 5 (Alternate Collapse may Decrease $\hat{m}$)**.** *For a DAG $G_i = (V, E)$, candidates (u, v) and (x, y), the collapse of $u \bowtie v$ which occludes $x \bowtie y$ and resulting allocation of cores denoted $\hat{m}_{(u \bowtie v)}$ may be greater than the allocation of cores due to collapsing $x \bowtie y$, ie. $\hat{m}_{(x \bowtie y)} < \hat{m}_{(u \bowtie v)}$.*



(a) Pre-Collapse (b) Post-Collapse

Figure 8.9: Collapse of (x, y) before (u, v)

Continuing the example, collapsing (x, y) precludes the collapse of (u, v). Collapsing (x, y) instead of (u, v) results in Figure 8.9. The impact upon the workload and critical path length of $x \bowtie y$ differs from that of $u \bowtie v$ and ultimately a difference in m.

C_i	L_i	m_i	$x \bowtie y$	$\hat{C}_i$	$\hat{L}_i$	$\hat{m}_i$
52	32	$\lceil 2.5 \rceil$	$\rightarrow$	49	29	$\lceil 1.81 \rceil$

Table 8.3: Collapse of x and y from Figure 8.9

Table 8.3 illustrates the impact of ordering of collapse with respect to m_i. Where collapsing $x \bowtie y$ in place of $u \bowtie v$ yields a smaller number of dedicated cores m_i.

8.2.2 Beneficial Collapse

By Observations 1-5 collapse of any individual candidate may increase or decrease the number of cores allocated to a task. A collapse may increase or decrease the critical path length creating an infeasible task set or introduce a cycle into the graph. This subsection defines which collapses are *beneficial*.

Beneficial collapse depends on the Definition 8.2.3 of *improving* the allocation of cores. Improving the number of allocated cores balances the concepts of reducing the number of cores allocated to a feasible task, avoiding the creation of an infeasible task, and (possibly) creating feasible tasks from infeasible ones.

Definition 8.2.3 (Improved Core Allocation)**.** For a given number of cores allocated to a task m_i, $\hat{m}_i$ is an improvement upon m_i denoted $\hat{m}_i \ll m_i$ if and only if:

1.) $m_i > 0 \rightarrow 0 < \hat{m}_i \leq m_i$**:**

$$\frac{\hat{C}_i - \hat{L}_i}{D_i - \hat{L}_i} \leq \frac{C_i - L_i}{D_i - L_i}$$

2.) $m_i \leq 0 \rightarrow \hat{m}_i \geq m_i$:

$$\frac{\hat{C}_i - \hat{L}_i}{D_i - \hat{L}_i} \geq \frac{C_i - L_i}{D_i - L_i}$$

When m_i is greater than zero, an $\hat{m}_i$ less than m_i and greater than zero is an improvement, reducing the number of cores allocated to the task. When $m_i < 0$, the critical path length has exceeded the deadline $L_i > D_i$. Such a task is not feasible under the DAG model, but may be schedulable under the DAG-OT model. For m_i less than zero, a $\hat{m}_i$ greater than m_i is an improvement; an increase over m_i may result in a schedulable task under DAG-OT.

Improvement of m_i does not include the ceiling described by Equation 8.1.5. This is due to the difference in context of m_i under the DAG model compared to DAG-OT. For the DAG model, m_i is calculated once and an integer number of cores are assigned to the task τ_i for schedulability analysis. For the DAG-OT model, m_i is recalculated after every collapse operation. Only when collapse operations have ceased is the final integer ceiling of $\hat{m}_i$ assigned to τ_i for schedulability analysis. The treatment of m_i (and $\hat{m}_i$) as a real rather than integer number is consistent throughout this work.

Beneficial collapse, defined by Definition 8.2.4 includes the improvement of core allocation as one of the three conditions. The first condition maintains the integrity of the analysis, a beneficial collapse may not introduce a cycle into the graph which the critical path length calculation depends upon.

Definition 8.2.4 (Beneficial Collapse)**.** For a task τ_i, DAG $G_i = (V, E)$, and candidate nodes $u, v \in V$ the collapse of $u \bowtie v$ which results in $\hat{G}_i$ is *beneficial* if and only if:

1. $\hat{G}_i$ contains no cycles

2. $L_i \leq D_i \rightarrow \hat{L}_i \leq D_i$

3. $\hat{m}_i \ll m_i$

Condition 2 of the beneficial collapse definition provides protection against collapse increasing the critical path length L_i beyond the deadline D_i, which would create an unschedulable task. The protection of Condition 2 does not prevent unschedulable tasks becoming schedulable by collapse, due to the post-collapse critical path length being bounded by the deadline only if the pre-collapse critical path length was also less than the deadline.

8.2.3 Optimal Collapse

The primary goal of this work is to improve the schedulability of a task set by reducing the number of cores reserved for high utilization tasks. Defining optimality with respect to the number of cores assigned to a task matches the goal of minimizing the allocation for high utilization tasks. The definition of optimal follows:

Definition 8.2.5 (Optimal Collapse of a Task)**.** For a task and DAG G an *optimal* collapsing of G is a DAG $\hat{G}$ and least positive $\hat{m}$ obtainable by collapsing candidates of G.

Currently, the complexity class of selecting the optimal set of candidates to collapse for a single task is unknown and remains an open problem. Observations 1-5 along with Definitions 8.2.3 and 8.2.4 illustrate the difficulties of identifying candidates that are beneficial to collapse. The only known method to compute the optimal collapse of a task requires the exploration of all possible combinations of candidates. There may be V^2 candidates per task, exploring all possible combinations is $\mathbb{O}(2^{V^2})$. Generating the optimal formulation

and finding the optimal collapse of a task are both reserved for future work. As a practical alternative, heuristics for ordering candidates for collapse are proposed in Section 8.4.

8.3 DAG-OT Schedulability

Due to the intractability of optimal collapse for a task the following heuristic Algorithm 12 is proposed. It seeks to reduce the number of cores allocated to a high utilization task τ_i by collapsing only those candidates that are beneficial according to Definition 8.2.4.

Algorithm 12 DAG-OT Dedicated Core Reduction Algorithm

```
1:  procedure DAGOT-REDUCE(G_i)
2:      A ← CANDIDATES(G_i)
3:      A ← ORDER(A)
4:      while |A| ≠ 0 do
5:          (u, v) ← FIRST(A)
6:          A ← A \ (u, v)
7:          if BENEFIT(G_i, u, v) then
8:              COLLAPSE(G_i, u, v)
9:          end if
10:     end while
11: end procedure
```

Reduction begins by identifying the potential candidates for collapse on Line 2. Candidacy follows Definition 8.2.1, calculating the complete set of candidates is of complexity $\mathbb{O}(V^2)$. The set of candidates is prioritized for collapse consideration by ORDER, ordering heuristics are proposed in Section 8.4. Each proposed heuristic is of equal or lesser computational complexity than the while loop (and its contents) beginning on Line 4.

Only candidates that benefit the task set are collapsed. A beneficial collapse improves (Definition 8.2.3) the number of cores allocated to a task without introducing a cycle into the DAG. Checking for a cycle in $\hat{G}_i$ by a depth first search is $\mathbb{O}(V + E)$ complex. Calculating $\hat{L}_i$ of a DAG by topological sort is also $\mathbb{O}(V + E)$ complex. Deciding if the

number of allocated cores satisfy the definition of improved is an $\mathbb{O}(1)$ operation, and collapse is an $\mathbb{O}(E)$ operation. Iterating over $\mathbb{O}(V^2)$ possible candidates, Algorithm 12 is $\mathbb{O}(V^3 + V^2E)$.

During each iteration of the while loop on Line 4 of the DAGOT-REDUCE Algorithm 12 the current state of the DAG G_i serves as input and $\hat{G}_i$ is the output. A subsequent iteration of the loop consumes the previous $\hat{G}_i$ value as input when considering the next candidate for collapse.

To determine if the task set τ is schedulable after DAG-OT reduction has been applied, the low utilization tasks are given $m_{low} = m - m_{high}$ cores. The task set τ is deemed schedulable if m_{low} is non-negative, and all tasks of τ_{low} are multi-processor schedulable on the m_{low} available cores.

8.4 Candidate Ordering

Two heuristics for collapse ordering are proposed. The first "greatest benefit", orders the candidates by descending workload savings. The second "least penalty", orders candidates by increasing longest path extension. The proposed heuristics are compared against an "arbitrary" (random) ordering to highlight each heuristics impact.

8.4.1 Greatest Benefit

For the greatest benefit heuristic, intuition suggests that collapsing nodes that most reduce the total workload C_i will also reduce the number of cores m_i maximally. The difference in workload is represented by Δ in Equation 8.4.1. There is a one time cost to calculate Δ for all candidates in A and then order the set. This operation is of $\mathbb{O}(V \log V)$ complexity. Employing the greatest benefit heuristic, Algorithm 12 is then

$\mathbb{O}(V \log V + V^3 + V^2E) \iff \mathbb{O}(V^3 + V^2E)$

$$\Delta = c_u(\eta_u) + c_v(\eta_v) - c_u(\eta_u + \eta_v) \tag{8.4.1}$$

8.4.2 Least Penalty

For the least penalty heuristic, intuition suggests collapsing pairs with the least extension to the critical path length allows more nodes to be collapsed – this includes collapses which extend the critical path negatively, shortening it. Penalties γ are calculated once by Equation 8.4.2 for every candidate pair. The set of candidates A are ordered by increasing penalty for use in Algorithm 12.

$$\gamma = \hat{L}_i - L_i \tag{8.4.2}$$

Penalty calculation requires a topological sort for every candidate to find $\hat{L}_i$ with complexity $\mathbb{O}(V + E)$, for $\mathbb{O}(V^2)$ candidates. Sorting the candidates by penalty is $\mathbb{O}(V \log V)$ complex. Therefore, the initial penalty ordering complexity is $\mathbb{O}(V^3 + V^2E + V \log V)$. The complexity of Algorithm 12 utilizing the least penalty heuristic is then $\mathbb{O}(V^3 + V^2E + V \log_V + V^3 + V^2E) \iff \mathbb{O}(V^3 + V^2E)$.

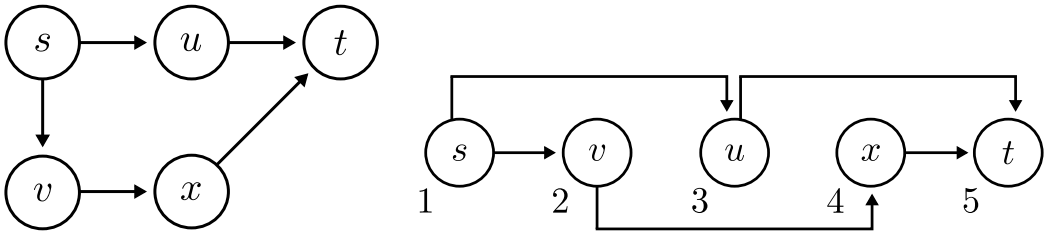
Penalty calculations apply to a single DAG $G_i = (V_i, E_i)$ instance. Collapsing two nodes $u, v \in V$ may impact the critical path length, i.e. $\hat{L}_i \neq L_i$. Since the penalty of collapse depends on the critical path length, the collapse of $u \bowtie v$ may impact the penalty γ of other candidates. This relationship, where one collapse may influence the penalty of a later collapse is ignored when ordering candidates for least penalty in favor of maintaining

the $\mathbb{O}(V^3 + V^2E)$ complexity of Algorithm 12.

8.5 Low Utilization Tasks

Previous sections have focused on reducing m_i for high ulitization tasks. Low utilization tasks may also incorporate the inter-thread cache benefit through collapse. To incorporate the benefit, a non-preemptive scheduler is required due to `BUNDLE`'s lack of preemptive schedulability analysis.

A low utilization DAG task τ_i requires no more than one core $m_i = 1$ to meet all deadlines. Therefore, τ_i may be *serialized*. To serialize τ_i a topological sort of G_i is performed and nodes are executed on the single processor in sort order. Figure 8.10 illustrates the serialization of a task τ_i.



(a) Pre-Serializing (b) Post-Serializing

Figure 8.10: Serializing a Task τ_i

Before a low utilization task is serialized all candidates $u, v \in V_i$ that are beneficial to collapse are collapsed. For a serialized task τ_i, the workload bounds the critical path length $C_i \geq L_i$. A serialized task is infeasible if $C_i > D_i$. Since the workload is only reduced by collapse, collapse preceding serialization cannot convert a feasible task into an infeasible one.

Similar to high utilization tasks, the complexity of serializing low utilization tasks depends on the number of candidates $\mathbb{O}(V^2)$, a DFS to check for cycles $\mathbb{O}(V + E)$, and a

topological sort to order execution $\mathbb{O}(V + E)$. The total complexity of the operation is

$$\mathbb{O}(V^2 \cdot (V + E) + (V + E)) \iff \mathbb{O}(V^3 + V^2 E) \tag{8.5.1}$$

Another concern shared with high utilization tasks is the order of collapse. For simplicity, collapse ordering is defined for the entire task set and shared between high and low utilization tasks. Whichever heuristic is selected for high utilization tasks is also selected for low utilization tasks uniformly for all tasks $\tau_i \in \tau$.

Every collapsed and serialized low utilization task $\tau_i \in \tau_{low}$ is scheduled non-preemptively, lest the inter-thread cache benefit of scheduling individual threads of nodes via `BUNDLE` be lost. Scheduling can be perceived as a hierarchy, where the job-level scheduler dictates which job can be run on a processor and the thread-level scheduler selects which thread of the running job will run. `BUNDLE` scheduling utilizes explicit thread-level preemptions but the analysis cannot accommodate job-level preemptions. Thus, the job-level scheduler is chosen to be non-preemptive EDF.

Each low utilization task $\tau_i \in \tau_{low}$ is assigned to exactly one of the m_{low} cores by the *Worst-Fit* [75][1] partitioning algorithm. Once assigned, jobs of τ_i will execute only upon its assigned processor. Worst-Fit assigns each task $\tau_i \in \tau_{low}$ to a per-core task set on a core m_k with the most available slack. When assigning τ_i to m_k, the assignment will not be made if it creates create an infeasible per-core task set determined by [76]. For the low utilization tasks τ_{low} to be deemed schedulable, all per-core task sets must be schedulable on their respective cores. The task set τ is schedulable if m_{low} is non-negative and τ_{low} is deemed

[1] Any non-preemptive EDF schedulability test based task assignment to cores could be chosen.

schedulable.

8.6 Evaluation

Evaluation of the proposed ITCB-DAG approach focuses on two metrics: schedulability ratios and the reduction of dedicated cores to high utilization tasks. No existing approach to federated scheduling of DAG tasks which incorporates inter-thread cache benefits or CRPD is (currently) known. To illustrate the potential of inter-thread cache benefits to DAG tasks under federated scheduling [63] high utilization tasks are scheduled by any work-conserving algorithm on the individual tasks dedicated cores. Low utilization tasks are assigned to cores by the Worst-Fit [75] partitioning algorithm and scheduled by non-preemptive EDF. In addition to non-preemptive EDF scheduling of low utilization tasks, a comparison to federated scheduling using preemptive EDF of low utilization tasks is made; to the benefit of preemptive EDF, preemptions have **no time** penalty.

An additional concession is made to all schedulability tests. Due to the nature of demand bound tests for low utilization tasks on partitioned cores, the operation may take an impractical amount of time to complete given the scale of the evaluation. To allow a greater volume of task sets to be included, if any schedulability test for a task set takes more than 10 minutes to complete the task set is deemed unschedulable for all schedulability tests and collapse heuristics.

The existing schedulability analysis approaches are compared to collapse by DAGOT-REDUCE using the proposed heuristics. Table 8.4 summarizes the existing and proposed approaches used in the evaluation along with their notation. The approaches are enumerated by their inclusion of collapse and their use of non-preemptive EDF (EDF-NP) or preemptive EDF

(EDF-P) for low utilization tasks.

Approach	EDF-NP	EDF-P
Baseline (No Collapse)	B-NP	B-P
Collapse Arbitrary	OT-A	∅
Collapse Greatest Benefit	OT-G	∅
Collapse Least Penalty	OT-L	∅

Table 8.4: Federated Schedulability Test Comparisons

Synthetic task sets are provided to each of the schedulability tests. Generation of the synthetic DAG tasks takes the form of a pipeline, where individual tasks are synthesized and then combined to make task sets. To allow a comparison to be made between the baseline and collapsed tasks, tasks are generated for the baseline first and then collapsed. DAGOT-REDUCE modifies the structure of DAG tasks, as well as the critical path length and total demand. Due collapse related changes, tasks that were trivially infeasible (ie. $L_i > D_i$) may become feasible. As such, existing approaches to task set generation which do not permit or construct trivially infeasible tasks were not suitable for evaluation of this work.

Figure 8.11 describes the pipeline in coarsest detail. Individual tasks are generated, then filtered. The filtered tasks are then duplicated, once per collapse ordering, before being assembled into task sets.

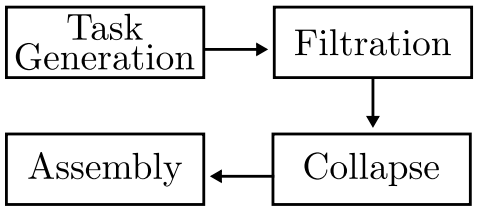


Figure 8.11: Task Set Generation Pipeline

Each stage of the pipeline is described using a tuple such as:

$$\langle A = \{a_1, a_2, ..., a_j\}, B = \{b_1, b_2, ..., b_k\} \rangle$$

A tuple abbreviates the cross product of all possible combinations i.e. $((a_1, b_1), (a_1, b_2), ..., (a_j, a_k))$. Additionally, a tuple may be preceded by iteration constant K that repeats each pair of the cross product K times. For example when $K = 2$:

$$K \cdot \langle A, B \rangle \rightarrow ((a_1, b_1), (a_1, b_1), (a_1, b_2), ..., (a_j, b_k), (a_j, b_k))$$

The size of any tuple is the product of cardinality of the elements of the tuple and the iteration constant.

Task generation is the first stage in the pipeline and is divided into smaller segments. The first segment of task generation is the creation of graph structures. There are three input parameters to graph creation: the number of nodes per graph n, the probability of an edge between any two nodes $P(u, v)$, and the number iterations S. To assign an edge to a pair of nodes u, v a random value in the range $r \in [0, 1]$ is generated, if $r \leq P(u, v)$ the edge is added. The set of graphs generated is referred to as τ_g, which is the result of $\tau_g = S \cdot \langle n, P(u, v) \rangle$. Table 8.5 enumerates these parameters with a range $[\min, \max]$ and increment, the total provided is the number of graphs generated after this segment.

Parameter	Range
n	$(16, 32, 64)$
$P(u, v)$	$(0.02, 0.06, 0.12)$
S	10
Total $\lvert\tau_g\rvert$	$\lvert S \cdot \langle n, P(u, v) \rangle \rvert = 90$

Table 8.5: Task Generation Graph Creation Parameters

The second segment of task generation is execution assignment. Each task in τ_g is repeatedly assigned objects to execute, creating a new task after each assignment. Execution

assignment begins by creating a set number of executable objects o per task. Each object is given a single thread WCET c_1 and a growth factor $\mathbb{F}$. The single thread execution value of each object is assigned a random value from the range $c_1 \in [1, 50]$. The growth factor of each object is assigned a random value from the range $[0.2, \mathbb{F}]$ Every node of the task is assigned exactly one executable object and one thread of execution. The set of tasks processed after this segment is referred to τ_e, which is the result of $\tau_e = \tau_g \times \langle o, \mathbb{F} \rangle$ Table 8.6 enumerates the execution assignment parameters, the total provided is the number of tasks generated after this segment.

Parameter	Range
o	$(4, 8, 16)$
$\mathbb{F}$	$(0.2, 0.6, 1.0)$
Total $\lvert\tau_e\rvert$	$\lvert\tau_g \times \langle o, \mathbb{F}\rangle\rvert = 90 \cdot 9 = 810$

Table 8.6: Task Generation Execution Assignment Parameters

The third and final segment of task generation is timing assignment for deadlines and periods. Each task in τ_e is repeatedly assigned a period and deadline, creating a new task after each assignment. Timing assignment is related to the critical path length of the task and one of the *task target utilization* values U_τ. For each task target utilization value, the task's period is set to $T = C/U_\tau$. For each period assignment, the task's deadline is repeatedly assigned for each of the *critical path length factors* cpf. A cpf lower bounds the deadline of the task in terms of the critical path length, the task's deadline is randomly selected in the range $[L \cdot cpf, T]$. The set of tasks after task set generation is referred to as τ, which is the result of $\tau = \tau_e \times \langle U_\tau, cpf \rangle$. After which, the set of tasks τ is sent to filtration. Table 8.7 enumerates the timing assignment parameters and provides the total

number of tasks generated.

Parameter	Range
U_τ	$(0.25, 0.50, 2.0, 4.0, 8.0$
cpf	$(0.5, 1.0, 2.5)$
Total $\lvert\tau\rvert$	$\lvert\tau_e \times \langle U_\tau, cpf\rangle\rvert = 810 \cdot 18 = 14,580$

Table 8.7: Task Generation Timing Assignment Parameters

Filtration is a single step process that removes tasks that are *always* trivially infeasible. A trivially infeasible task has a critical path length greater than its deadline. Since collapse may reduce the critical path length of a DAG task, an infeasible task may become a feasible DAG-OT task. Filtration executes each of the collapse heuristics on every task of τ. If the DAG task of τ is feasible, the task remains. If the DAG task is infeasible, and any collapse ordering produces a feasible DAG-OT task, the task remains. Only if the DAG task is infeasible, and all collapse orderings are also infeasible is the task removed from τ.

Collapse is the next stage of the pipeline, for each DAG task in τ a collapsed version of the DAG-OT task is produced. Tasks are segregated into pools one for the DAG task, and one for each of the collapse orders applied to the DAG-OT task. These collapsed task sets are referred to as τ_a for arbitrary ordering, τ_b for greatest benefit, and τ_p for least penalty. Each DAG task $\tau_i \in \tau$ shares its index i across pools, for example: $\tau_i \in \tau_p$ refers to the DAG-OT task generated from the DAG task $\tau_i \in \tau$ that was collapsed using the least penalty heuristic.

Assembly is the final stage of the task set generation pipeline. Fore every selection of cores in the system architecture c, and *target task set utilization* U, N task sets are created from the DAG tasks τ. For every task set assembled from τ, the corresponding task set

from each of the collapse orderings is also created. To clarify, for a DAG task set:

$$A = \{\tau_i, \tau_j, \tau_k\}, \tau_i, \tau_j, \tau_k \in \tau$$

The corresponding task DAG-OT task set using the greatest benefit collapse ordering is:

$$A_b = \{\tau_i, \tau_j, \tau_k\}, \tau_i, \tau_j, \tau_k \in \tau_b$$

Table 8.8 enumerates the assembly parameters and the total number of task sets created. The total reflects the total number of DAG task sets assembled, it does not reflect the equivalent DAG-OT task sets.

Parameter	Range
U	$(0.5, 1, 2, 4, 8, 12, 16, 20, 24, 28, 32, 36)$
c	$(4, 8, 12, 16, 20, 24, 28, 32)$
N	1000
Total	$N \cdot \langle c, U \rangle = 96,000$

Table 8.8: Task Set Assembly Parameters

8.6.1 Evaluation Metrics

A schedulability ratio is calculated for each of the schedulability tests. For the DAG-OT schedulability tests, the number of cores saved m_{saved} per task is calculated by Equation 8.6.1 where pre-collapse m_{high} comes from Equation 8.1.5 and $\hat{m}_{high}$ after Algorithm 12 has terminated.

$$m_{saved} = m_{high} - \hat{m}_{high} \tag{8.6.1}$$

The average change in number of cores allocated to high utilization tasks is given by Equation 8.6.2.

$$\bar{\Delta}_m = \frac{\sum_{\tau_i \in \tau} m_{saved}}{n} \tag{8.6.2}$$

Similarly, the change in workload and critical path length for each of the DAG-OT schedulability tests is compared to C_i and L_i from Equations 8.1.2 and 8.1.1 under the DAG model. Equation 8.6.3 quantifies the average change in workload and 8.6.4 the average change in critical path length.

$$\bar{\Delta}_C = \frac{\sum_{\tau_i \in \tau} C_i - \hat{C}_i}{n} \tag{8.6.3}$$

$$\bar{\Delta}_L = \frac{\sum_{\tau_i \in \tau} \hat{L}_i - L_i}{n} \tag{8.6.4}$$

8.6.2 Results

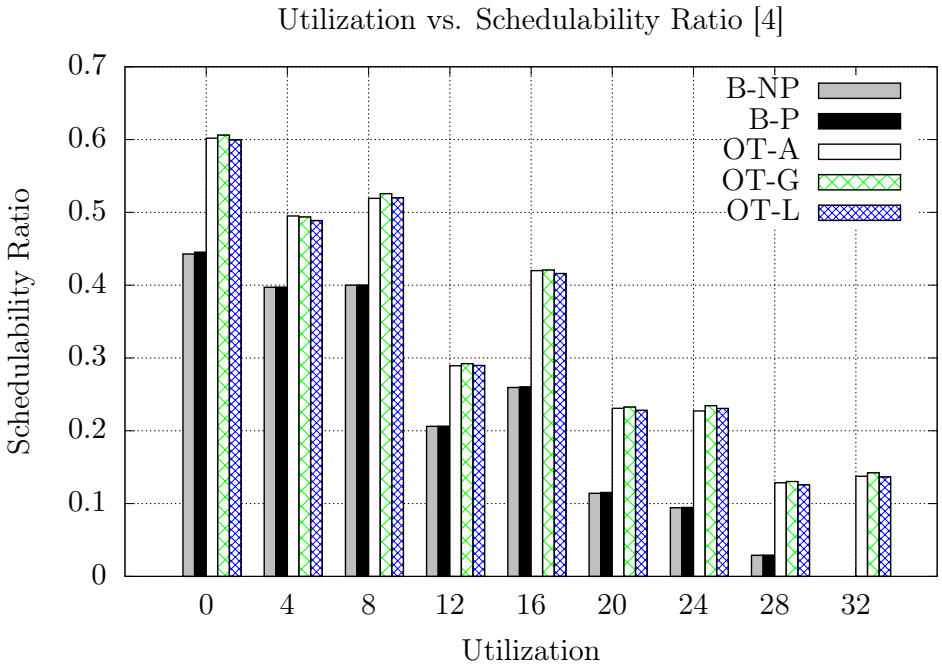


Figure 8.12: Mean Schedulability Ratio

Figure 8.12 summarizes the schedulability results. In the title the '[4]' indicates the utilization interval the column summarizes. For the histograms labeled '0', only task sets with utilization in $[0, 4)$ contribute to the ratio. The height of the bar is the average schedulability ratio over the interval. From this summary data, it is clear collapse can improve the schedulability of federated scheduled DAG tasks.

Furthermore, the deleterious DAG-OT requirement of non-preemptive scheduling for low utilization tasks does not outweigh the gains in schedulability of collapsing tasks. This can be observed by the higher schedulability ratios for collapsed task sets compared to the uncollapsed *fully preemptive* low utilization task sets of B-P. Where the fully preemptive scheduler incurs no penalty for preemptions between low utilization tasks. Of the 96,000

task sets generated, 36 were deemed unschedulable due to exceeding the 10 minute time limit.

It is unclear from Figure 8.12 which of the collapse heuristics is the most desirable. Greatest benefit (OT-G) performs best, across all intervals. However, the improvement over arbitrary (OT-A) and least penalty (OT-L) is small (less than two percent) and inconsistent across intervals.

Figure 8.13 focuses on the central purpose of collapse: to reduce the number of cores assigned to high utilization tasks. The greatest benefit heuristic (OT-G) performs better than least penalty (OT-L). Both heuristics perform better than arbitrary collapse ordering (OT-A). For these task sets, the heuristics provide a greater reduction in dedicated cores than arbitrary ordering for collapse.

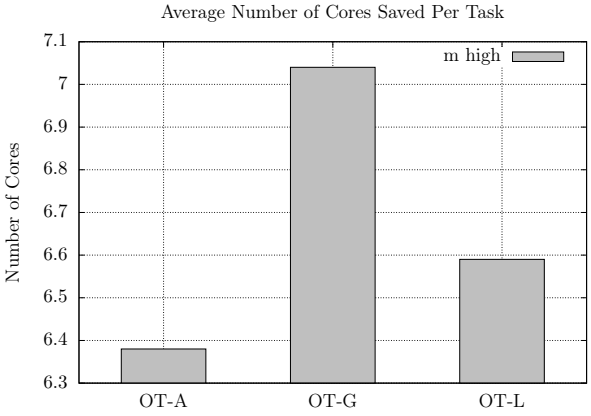
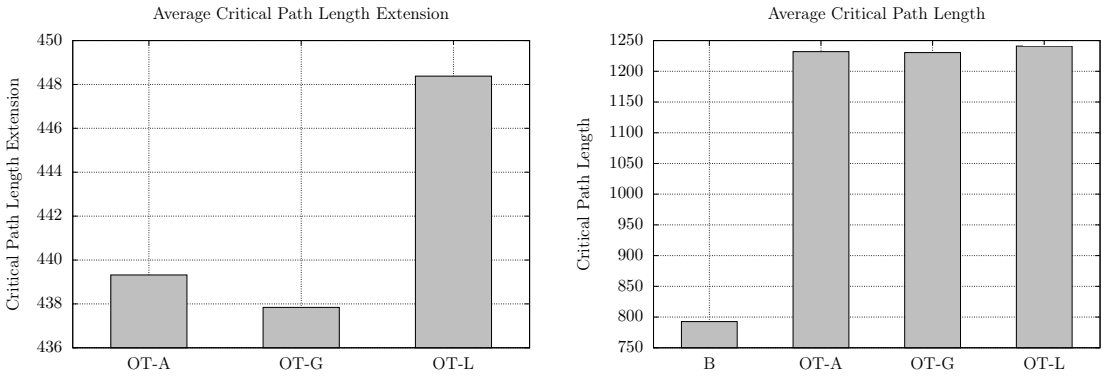


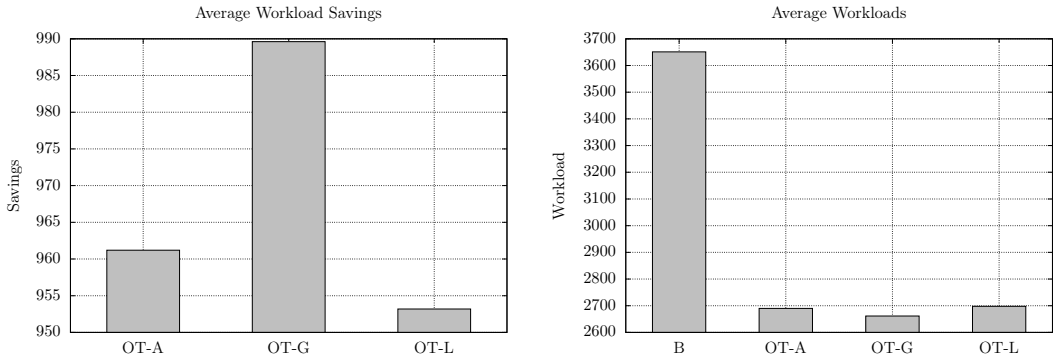
Figure 8.13: Mean Core Savings



(a) $\bar{\Delta}_L$ (b) $\bar{L}$

Figure 8.14: Mean Critical Path Lengths and Extensions

The least penalty (OT-L) heuristic seeks to collapse those nodes with the smallest increase to the critical path length before others. Surprisingly, Figure 8.14 shows the least penalty ordering of collapse may not have the intended effect. For OT-L, the average critical path length is greater than greatest benefit (OT-G) or arbitrary ordering (OT-A); although it remains within 0.5 percent of OT-G and OT-A.



(a) $\bar{\Delta}_C$ (b) $\bar{C}$

Figure 8.15: Mean Workloads and Savings

Figure 8.15 illustrates the benefits of collapse upon the workload for all orderings. Unsurprisingly OT-G providing the greatest average workload reduction of 28 percent. With the worst performance among the three, ordering candidates by least penalty provides the least improvement to workload reduction.

From the results of Figure 8.13, 8.14, and 8.15 greatest benefit performs better in terms of saving cores, critical path length extension, and workload savings for the tasks sets evaluated herein. This is due to the nature of critical path length extension in comparison to workload savings. With each collapse, there is potential for the critical path to shift from one set of nodes to another. If the critical path length shifts, the initial least penalty ordering may no longer be in descending critical path length extension order. However,

workload savings are not affected when the critical path shifts; thus greatest benefit provides more consistent behavior and overall better performance.

8.7 Summary

The ITCB-DAG approach brings BUNDLE scheduling and analysis techniques to multiprocessor parallel DAG tasks. In addition, the approach introduces the concept of collapsing nodes within a DAG task to decrease the number of cores dedicated to high utilization tasks. In context of federated scheduling, with low utilization tasks scheduled by non-preemptive partitioned EDF collapse of high utilization tasks can improve schedulability when compared to federated DAG tasks with low utilization tasks scheduled by fully preemptive partitioned EDF where preemptions incur **no** penalties. For a static ordering of collapsing node pairs, the greatest benefit heuristic provides more consistent and favorable performance improvements than the ordering generated by the least penalty heuristic.

CHAPTER 9 FUTURE WORK

The positive perspective taken by the BUNDLE approach provides a myriad of opportunities for improvement and greater applicability. The greater goal is to bring this positive perspective to deployed hard real-time systems. BUNDLE is the first step toward this goal. BUNDLEP a second, improving upon BUNDLE with a concrete implementation and reasonable complexity in WCETO calculation. NPM-BUNDLE a third, expanding the BUNDLE approach to multi-threaded multi-task systems. ITCB-DAG a fourth, demonstrating the potential of collapse for multi-core platforms.

There are many avenues available when taking the next steps: improving BUNDLE scheduling performance and WCETO calculation in the single-task setting, or increasing multi-core support with formal guarantees of performance when collapsing nodes, or adding support for architecture features such as hierarchical and shared caches. Among the available avenues, there are two which address the most significant deficiencies of the BUNDLE approach.

9.0.1 Scheduling Support

BUNDLE's scheduling techniques rely upon a mechanism that does not exist on any existing hardware platform or as part of any scheduler. A hardware mechanism has been proposed as part of BUNDLEP, implementation of the mechanism on a simulator such as riscv-angel would demonstrate the viability of the mechanism and provide deeper insights into the penalties of BUNDLE scheduling.

To complement the hardware mechanism, a software based approach has been con-

ceived and initial testing has begun. The concept behind the software based approach to support BUNDLE scheduling is to introduce trampolines at CFR boundaries. These trampolines are inserted with a user-level thread scheduler which performs the BUNDLE scheduling operations. For the software based approach, the addition of instructions complicates the CFR analysis.

The software based approach provides an immediate opportunity for BUNDLE deployment in hard real-time systems and is the next step. Comparing the software based approach to a hardware based solution follows to illustrate the benefits of each.

9.0.2 Preemptive Multi-Task BUNDLEP

NPM-BUNDLE is limited to non-preemptive multi-task scheduling and analysis. Support for preemptive scheduling and analysis is a clear next step. There are two preemptive models to investigate. Fully preemptive, where a job can be preempted during any portion of its execution by a higher priority job. Limited preemption, where preemptions are limited to bundle activations. When limited, preemptions are delayed until the active bundle of the lower priority job is depleted before the higher priority job is permitted to execute. A more granular approach of allowing the current thread to execute until blocking may be explored.

9.0.3 From Switched to Unswitched CFRs

As noted in Chapter 6, BUNDLEP may produce higher WCETO values for single tasks when compared to the serial execution of threads. One significant cause of higher WCETO values is the introduction of context switches with greater cost (more cycles) than the inter-thread cache benefit of BUNDLE scheduling for a specific CFR.

To address this issue, the proposed work will develop a method to identify unswitched CFRs. An *unswitched* CFR will allow threads to leave the active CFR by executing the entry instruction of the unswitched CFR without blocking. Since no thread will block entering an unswitched CFR the context switch of selecting the CFR is avoided. It is unclear if there is a tractable optimal (with respect to minimum WCETO value) algorithm. The work will focus on a heuristic when selecting which CFRs will be unswitched.

CHAPTER 10 CONCLUSION

Taking a positive perspective on instruction caches, two analytical techniques and scheduling algorithms have been proposed for single multi-threaded tasks: BUNDLE and BUNDLEP. The two share a common approach to scheduling a multi-threaded hard real-time task, where threads (not tasks) are executed in a cache cognizant manner that increases the inter-thread cache benefit at run-time and is quantified during WCETO analysis.

Both rely on a newly proposed model of multi-threaded tasks which includes ribbons and threads as top level objects. They also share a proposed modification of CFGs, where ribbons are divided into CFRs and assembled into a CFRG. It is the CFRG which provides information to WCETO analysis as well as run-time scheduling decisions.

BUNDLE and BUNDLEP differ in their construction of the CFRG. BUNDLE's approach is less restrictive and more descriptive, individual instructions are permitted to reside in multiple CFRs and WCETO calculations are limited to specific structures. BUNDLEP restricts individual instructions to a single CFR and ignores their structure except for loops. Utilizing priorities for CFRs, BUNDLEP improves upon BUNDLE. BUNDLEP's scheduling algorithm maximizes cache sharing and reduces the complexity of BUNDLE's WCETO calculation; avoiding multiple all paths walks of the CFRG.

Both scheduling algorithms require a novel mechanism to anticipate execution. The proposed XFLICT interrupt and XFLICT TABLE meet the needs of both algorithms. Additionally, the address table based behavior of the interrupt is similar to the accepted and tested method of hardware interrupts.

Results from BUNDLE's and BUNDLEP's evaluation produce lower WCETO bounds than the classical approach in many cases, encouraging a deeper investigation into the positive perspective. Further expansion of the positive perspective is possible in a multitude of directions. Set associative caches, alternative cache replacement policies, hierarchical caches, support for multi-core systems, alternative WCETO calculation methods and implementation on a Commercial Off-The-Shelf system are opportunities for further expansion of BUNDLE's approach.

These BUNDLE techniques and analysis have been incorporated into multi-threaded multi-task analysis with the introduction of NPM-BUNDLE. The hierarchical scheduling technique of NPM-BUNDLE non-preemptively schedules jobs of tasks, while threads within jobs are permitted to preempt one another according to BUNDLE scheduling decisions. As part of NPM-BUNDLE, a novel scheduling algorithm TPJ is introduced that divides threads among tasks and is guaranteed to produce an npm-feasible task set if one exists. Utilizing this approach, some task sets deemed unschedulable for preemptive EDF with **no** preemption penalties are schedulable under TPJ.

Incorporation of the inter-thread cache benefit to multi-core architectures is the purpose of ITCB-DAG. The DAG-OT model expands the DAG model, enumerating the executable objects and threads associated with each node. In contrast to the DAG model, nodes may execute multiple threads per release. Inclusion of objects and threads permits collapsing of nodes under the DAG-OT model. When combined with the collapse ordering heuristics, schedulability ratios of parallel DAG tasks may be increased.

Currently, the most significant areas impeding general acceptance and deployment of BUNDLE are 1.) lack of scheduling support at CFR boundaries 2.) strictly non-preemptive

scheduling of BUNDLE jobs 3.) for specific tasks, BUNDLE produces greater WCETO bounds and worse run-time performance compared to the classical perspective. Therefor, future efforts are focused on hardware and software support for BUNDLE scheduling, fully and limited preemption analysis, and conditionally switching CFRs for lower analytical bounds and better run-time performance.

LIST OF PUBLICATIONS

IN PROGRESS

1. **Corey Tessler**, Prashant Modekurthy, Nathan Fisher, Abusayeed Saifullah. Cache Cognizant Collapse of Nodes Within Directed Acyclic Graph Tasks to Reduce High Utilization Cores, submitting to *26th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Sydney, Australia, 2020.*

CONFERENCES

2. **Corey Tessler**, Nathan Fisher. `NPM-BUNDLE`: Non-Preemptive Multitask Scheduling for Jobs with `BUNDLE`-Based Thread-Level Scheduling, *Proceedings of Euromicro Conference on Real Time Systems (ECRTS)*, Stuttgart, Germany, 2019.
3. **Corey Tessler**, Nathan Fisher. `BUNDLEP`: Prioritizing Conflict Free Regions in Multi-Threaded Programs to Improve Cache Reuse, *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, Nashville, Tennessee, 2018.
4. **Corey Tessler**, Gedare Bloom, Nathan Fisher. Work-in-Progress: Reducing Cache Conflicts via Interrupts and BUNDLE Scheduling, *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Vienna, Austra, 2017.
5. **Corey Tessler**, Nathan Fisher. `BUNDLE`: Real-Time Multi-Threaded Scheduling to Reduce Cache Contention, *Proceedings of IEEE Real-Time Systems Symposium (RTSS)*, Porto, Portugal, 2016.

6. John Cavicchio, **Corey Tessler**, Nathan Fisher. Minimizing Cache Overhead via Loaded Cache Blocks and Preemption Placement, *Proceedings of Euromicro Conference on Real-Time Systems (ECRTS)*, Lund, Sweden, 2015.

BIBLIOGRAPHY

[1] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. *Real-Time Systems Symposium, 1994., Proceedings.*, pages 172–181, Dec 1994. viii, 9, 20, 23, 24, 54, 55

[2] Frank Mueller. *Static Cache Simulation and Its Applications*. Ph.d. dissertation, Florida State University, 1995. viii, 9, 20, 23, 24, 54, 55

[3] Advanced Micro Systems. Welcome to amd. 2019. http://www.amd.com. 1

[4] NVIDIA. Artificial intelligence leadership from nvidia. 2019. http://www.nvidia.com. 2

[5] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. 3, 105, 110

[6] Jinghao Sun, Nan Guan, Yang Wang, Qingxu Deng, Peng Zeng, and Wang Yi. Feasibility of fork-join real-time task graph models: Hardness and algorithms. *ACM Trans. Embed. Comput. Syst.*, 15(1):14:1–14:28, February 2016. 4, 27

[7] Corey Tessler and Nathan Fisher. BUNDLE: Real-Time Multi-Threaded Scheduling to Reduce Cache Contention. In *IEEE Real-Time Systems Symposium*, 2016. 6

[8] C. Tessler and N. Fisher. BUNDLEP: Prioritizing conflict free regions in multi-threaded programs to improve cache reuse. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 325–337, Dec 2018. 7, 128, 129

[9] Corey Tessler. BUNDLEP virtual machine toolkit. 2018. http://www.cs.wayne.edu/~fh3227/bundlep. 7, 91

[10] C. Tessler and N. Fisher. NPM-BUNDLE: Non-preemptive multitask scheduling for jobs with BUNDLE-based thread-level scheduling. In *2019 Euromicro Conference on Real-Time Systems (ECRTS)*, volume 133, 2019. 7

[11] Corey Tessler. NPM-BUNDLE artifacts, 2019. http://www.cs.wayne.edu/~fh3227/npm-bundle/. 7, 102, 127

[12] Frank Mueller. Timing analysis for instruction caches. In *The Journal of Real-Time Systems 18*, pages 217–247, 2000. 9, 24

[13] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *Journal of Systems Architecture*, 57(7):707–719, August 2011. 9, 23, 25, 26, 88

[14] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*, RTSS '11, pages 261–271, Washington, DC, USA, 2011. IEEE Computer Society. 9, 25, 26

[15] Yudong Tan and Vincent Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Software and Compilers for Embedded Systems: 8th International Workshop, SCOPES 2004, in: Lecture Notes on Computer Science*, SCOPES '04, pages 182–199. Springer, 2004. 9, 25, 26

[16] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embededed Computing Systems*, 6(1), February 2007. 9, 90

[17] H. Tomiyama and N.D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the Eighth International*

Workshop on Hardware/Software Codesign (CODES), pages 67–71, May 2000. 9, 25

[18] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE Real-Time Systems Symposium*, pages 182–190, Dec 1990. 11, 105, 110, 111

[19] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann, 2011. 13, 24

[20] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. 14

[21] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998. 20, 21, 22, 26, 27, 56

[22] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '03, pages 201–206, New York, NY, USA, 2003. ACM. 23, 25, 26

[23] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000. 24

[24] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. 24

[25] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2):157–179, May 2000. 24

[26] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings Third IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997. 24

[27] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *17th IEEE Real-Time Systems Symposium*, pages 254–263, Dec 1996. 24

[28] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, May 2000. 24

[29] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 75–84, April 2013. 25, 56, 57

[30] 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*, 48(5), 2012. 25

[31] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of the 2005 17th Euromicro Conference on Real-Time Systems (ECRTS)*, ECRTS '05, pages 41–48, July 2005. 25, 26

[32] A. Burns. *Advances in Real-Time Systems*, chapter Preemptive priority-based scheduling: an appropriate engineering approach, pages 225–248. Prentice Hall, Inc., 1995. 27, 105

[33] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the International Conference on Real Time Computing Systems and Applications*, 1999. 27

[34] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 137–144, July 2005. 27, 102, 105, 110, 112, 113, 114, 115, 116

[35] J. M. Marinho, V. Nelis, S.M. Petters, and I. Puaut. An improved preemption delay upper bound for floating non-preemptive region. In *Proceedings of IEEE International Symposium on Industrial Embedded Systems*, 2012. 27

[36] J. Simonson and J.H. Patel. Use of preferred preemption points in cache based real-time systems. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*, 1995. 27

[37] 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, July 2017. 27

[38] M. Bertogna, O. Xhani, M. Marinoni, F. Esposito, and G. Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 217–227, July 2011. 27, 105

[39] B. Peng, N. Fisher, and M. Bertogna. Explicit preemption placement for real-time conditional code. In *Proceedings of Euromicro Conference on Real-Time Systems*, 2014.

27

[40] J. Cavicchio, C. Tessler, and N. Fisher. Minimizing cache overhead via loaded cache blocks and preemption placement. In *Proceedings of the Euromicro Conference on Real-Time Systems*, 2015. 27

[41] Robert Mittermayr and Johann Blieberger. Timing Analysis of Concurrent Programs. In *12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 59–68, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 27

[42] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 57–67, Dec 2009. 28

[43] S. Bak, G. Yao, R. Pellizzoni, and M. Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309, Aug 2012. 28

[44] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, April 2011. 28

[45] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems*, pages 157–167, July 2013. 29

[46] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multitasking real time embedded systems. In *2008 14th IEEE International Conference on*

Embedded and Real-Time Computing Systems and Applications, pages 101–110, Aug 2008. 29

[47] John Michael Calandrino. *On the Design and Implementation of a Cache-aware Soft Real-time Scheduler for Multicore Platforms*. PhD thesis, Chapel Hill, NC, USA, 2009. 29

[48] S. A. Rashid, G. Nelissen, D. Hardy, B. Akesson, I. Puaut, and E. Tovar. Cache-persistence-aware response-time analysis for fixed-priority preemptive systems. In *Euromicro Conference on Real-Time Systems*, pages 262–272, July 2016. 29

[49] S. A. Rashid, G. Nelissen, S. Altmeyer, R. I. Davis, and E. Tovar. Integrated analysis of cache related preemption delays and cache persistence reload overheads. In *IEEE Real-Time Systems Symposium*, pages 188–198, Dec 2017. 29

[50] S Meena Kumari and N Geethanjali. A survey on shortest path routing algorithms for public transport travel. *Global Journal of Computer Science and Technology*, 9(5):73–75, 2010. 46

[51] Prabhaker Mateti and Narsingh Deo. On algorithms for enumerating all circuits of a graph. *SIAM Journal on Computing*, 5(1):90–99, 1976. 46

[52] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *International Workshop on Worst-Case Execution Time Analysis*, volume 15, pages 136–146, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 58, 91, 129

[53] MIPS Technologies, Inc. *MIPS® 74K™ Processor Core Family Software User's Manual*, 2011. https://www.mips.com/products/classic/. 63, 79

[54] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys*, 44(3):10:1–10:33, June 2012. 68

[55] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999. 68

[56] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A new algorithm for identifying loops in decompilation. In *Static Analysis*, pages 170–183, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 70

[57] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. Towards compositionality in execution time analysis: Definition and challenges. *ACM SIGBED Review*, 12(1):28–36, March 2015. 88, 143

[58] Corey Tessler. `BUNDLEP`: Prioritizing conflict free regions in multi-threaded programs to improve cache reuse – extended results and technical report. 2018. https://arxiv.org/abs/1805.12041. 92

[59] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011. 105, 117

[60] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Research Report RR-2966, INRIA, 1996. Projet REFLECS. 110, 112

[61] M. Bertogna and S. Baruah. Limited preemption edf scheduling of sporadic task systems. *IEEE Transactions on Industrial Informatics*, 6(4):579–591, Nov 2010. 116, 128

[62] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 196–203, July 2004. 128

[63] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 85–96. IEEE, 2014. 139, 140, 142, 144, 160

[64] Jing Li, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Federated scheduling for stochastic parallel real-time tasks. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2014. 146

[65] Sanjoy Baruah. Federated scheduling of sporadic dag task systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 179–186. IEEE, 2015. 146

[66] Niklas Ueter, Georg von der Brüggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. Reservation-based federated scheduling for parallel real-time tasks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 482–494. IEEE, 2018. 146

[67] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013. 146

[68] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014. 146

[69] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Raj Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *2010 31st IEEE Real-Time Systems Symposium*, pages 259–268. IEEE, 2010. 146

[70] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global edf scheduling of systems of conditional sporadic dag tasks. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 222–231. IEEE, 2015. 146

[71] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *2015 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 211–221. IEEE, 2015. 146

[72] Jinghao Sun, Nan Guan, Xu Jiang, Shuangshuang Chang, Zhishan Guo, Qingxu Deng, and Wang Yi. A capacity augmentation bound for real-time constrained-deadline parallel tasks under gedf. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2200–2211, 2018. 146

[73] Zhishan Guo, Ashikahmed Bhuiyan, Abusayeed Saifullah, Nan Guan, and Haoyi Xiong. Energy-efficient multi-core scheduling for real-time dag tasks. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 76. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017. 146

[74] Son Dinh, Jing Li, Kunal Agrawal, Chris Gill, and Chenyang Lu. Blocking analysis for spin locks in real-time parallel tasks. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):789–802, 2018. 146

[75] Sanjoy Baruah. Partitioned edf scheduling: a closer look. *Real-Time Systems*, 49(6):715–729, 2013. 159, 160

[76] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pages 129–139, Dec 1991. 159

ABSTRACT

BUNDLE: TAMING THE CACHE AND IMPROVING SCHEDULABILITY OF MULTI-THREADED HARD REAL-TIME SYSTEMS

by

COREY TESSLER

December, 2019

Advisor: Dr. Nathan Fisher

Degree: Doctor of Philosophy

For hard real-time systems, schedulability of a task set is paramount. If a task set is not deemed schedulable under all conditions, the system may fail during operation and cannot be deployed in a high risk environment. Schedulability testing has typically been separated from worst-case execution time (WCET) analysis. Each task's WCET value is calculated independently and provided as input to a schedulability test. However, a task's WCET value is influenced by scheduling decisions and the impact of cache memory. Thus, schedulability tests have been augmented to include cache-related preemption delay (CRPD). From this *classical* perspective, the effect of cache memory on WCET and schedulability is always negative; increasing execution times and demand. In this work we propose a new *positive* perspective, where cache memory benefits multi-threaded tasks by scheduling threads in a manner that shares values predictably.

This positive perspective is reached by integrating, rather than separating the disciplines of schedulability analysis and worst-case execution time. These *integrated* techniques are referred to as the `BUNDLE` family of worst-case execution time and cache over-

head (WCETO) analysis and scheduling algorithm. WCETO calculation divides the task's structure into conflict free regions and calculates a bound utilizing explicit understanding of the thread-level scheduling algorithm. Conflict free regions are utilized by the scheduling algorithm, which associates with each region a thread container called a *bundle*. At any time only one bundle may be active, and only threads of the active bundle may execute on the processor.

The BUNDLE family of scheduling algorithms developed in this work increase in scope from BUNDLE through ITCB-DAG. As the fundamental contribution, BUNDLE and BUNDLEP apply to a single multi-threaded task running on a uniprocessor architecture with a single level direct mapped instruction cache. NPM-BUNDLE expands the positive perspective to multiple tasks on a uniprocessor system. With ITCB-DAG bringing BUNDLE's analysis and scheduling techniques to multi-processor systems.

Each of the scheduling algorithms require a novel hardware mechanism to *anticipate* execution and make scheduling decisions. To support anticipation of execution, a novel XFLICT interrupt is proposed. It is a simple mechanism that emulates the behavior of hardware breakpoints. An implementation of the BUNDLEP analytical techniques, scheduling algorithm, and XFLICT interrupt is available as a simulated platform for further research and extension.

Future work is planned to expand BUNDLE's positive perspective and increase adoption. The most significant barrier to adoption is the ability to deploy BUNDLE's scheduling algorithm, this mandates a viable and available hardware or software mechanism to anticipate execution. NPM-BUNDLE is limited to non-preemptive multi-task scheduling and analysis, support for preemptive scheduling will increase the positive impact of BUNDLE's integrated

perspective.

AUTOBIOGRAPHICAL STATEMENT

Corey Tessler has a successful professional career behind him as he looks forward to an academic future. He earned a Bachelor's of Computer Science from Eastern Michigan University in 2004. During his undergraduate education he began working in the field of networking at UUNet as a trouble shooting software engineer and then Next Hop Technologies as an author and maintainer of internet routing protocol software.

His professional career continued at Arbor Networks developing distributed denial of service mitigation software and at Green Hills Software where he acted as a routing software developer, sales support engineer, and project manager. Concurrently, he earned a Master's of Computer Science from Eastern Michigan University in late 2011.

Joining Wayne State University in 2013 as a Ph.D. student, Tessler focused his interest in hard real-time systems under the supervision of Professor Nathan Fisher. Together they developed the BUNDLE analysis and scheduling algorithm, publishing several works in well respected conferences for real-time systems. In his final semester, Tessler began as a lecturer at Wayne State University marking his transition from a professional to an academic career. He is actively seeking a tenure-track professorship at a research university.