ABSTRACT

RAMAPRASAD, HARINI Analytically Bounding Data Cache Behavior for Real-Time Systems. (Under the direction of Associate Professor Frank Mueller).

This dissertation presents data cache analysis techniques that make it feasible to predict data cache behavior and to bound the worst-case execution time for a large class of real-time programs.

Data Caches are an increasingly important architectural feature in most modern computer systems. They help bridge the gap between processor speeds and memory access times. One inherent difficulty of using data caches in a *real-time system* is the unpredictability of memory accesses, which makes it difficult to calculate worst-case execution times of real-time tasks.

This dissertation presents an analytical framework that characterizes data cache behavior in the context of independent, periodic tasks with deadlines less than or equal to their periods, executing on a single, in-order processor. The framework presented has three major components.

1) The first component analytically derives **data cache reference patterns** for all scalar and non-scalar references in a task. Using these, it produces a safe and tight upper bound on the worst-case execution time of the task without considering interference from other tasks.

2) The second component calculates the worst-case execution time and response time of a task in the context of a multi-task, prioritized, preemptive environment. This component calculates **Data-Cache Related Preemption Delay** for tasks assuming that all tasks in the system are completely preemptive.

3) In the third component, tasks are allowed to have **critical sections** in which they access shared resources. In this context, two analysis techniques are presented. In the first one, a task executing in a critical section is not allowed to be preempted by any other task. In the second one, the framework incorporates Resource Sharing Policies to arbitrate accesses to shared resources, thereby improving responsiveness of high-priority tasks that do not use a particular resource.

In all the components presented in this dissertation, a direct-mapped data cache is assumed. Experimental results demonstrate the value of all the analysis techniques described above in the context of data cache usage in a hard real-time system.

Analytically Bounding Data Cache Behavior for Real-Time Systems

by
Harini Ramaprasad

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Tao Xie	Dr. Eric Rotenberg
Dr. Frank Mueller Chair of Advisory Committee	Dr. Vincent Freeh

BIOGRAPHY

Harini Ramaprasad was born on 6^{th} November 1979, in Bangalore, India. She received her Bachelor of Engineering degree in Computer Science from Bangalore University, India in September 2001. In fall of 2002, she came to the North Carolina State University to pursue graduate studies in Computer Science. She received her Master of Science degree in Computer Science from North Carolina State University in May 2006. She will receive the Doctor of Philosophy degree in Computer Science from North Carolina State University with the defense of this dissertation.

ACKNOWLEDGMENTS

This dissertation would not have been possible without guidance from my PhD advisor, Dr. Frank Mueller. I would like to thank him immensely for his support and patience. I would like to thank Dr. Vincent Freeh, Dr. Tao Xie and Dr. Eric Rotenberg for being on my advisory committee. I would like to thank my colleague, Sibin Mohan, for his support in the development of the static timing analyzer used in this dissertation. I would like to express my immense gratitude towards my friends Yatin Tawde and Meghana Velegar for all their support. Lastly, I would like to thank my husband, Gopalakrishnan Santhanaraman, and my parents, for their endless support to me through my doctoral degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In this chapter, the basic motivation for the work presented in this dissertation is provided, followed by a brief outline of its primary contributions.

1.1 Motivation

In recent times, embedded systems have become ubiquitous. They have numerous applications ranging from cell-phones and hand-held devices to aircraft control systems and space explorers. Several embedded systems have hard real-time requirements, which, in addition to logical constraints, introduce temporal constraints on the tasks (programs) executing on such systems . These temporal constraints are in the form of a *deadline* by which a task must complete its execution.

The process of determining whether a given set of tasks can be scheduled on a real-time system such that no task violates its temporal constraints is known as schedulability analysis. This analysis requires *a-priori* knowledge of the execution time of every task in the system.

The execution time of a task is not a constant and its calculation is not straightforward due to several reasons. First of all, programs do not follow the same execution path every time they are executed due to differences in input data. Furthermore, the execution time of a given execution path varies between executions due to effects of several architectural features of the system and due to the presence of other tasks in the system.

In order to guarantee adherence to the temporal constraints imposed on a real-time task, a *safe upper bound* on its execution time must be determined. Such an upper

bound is termed Worst-Case Execution Time (WCET) of the task. While calculating the WCET of a task, the effects of all architectural features and of other tasks in the system must be taken into account. In a situation where the effects are ambiguous, the worst-case behavior must be assumed in order to maintain safety of the system.

Caches are an invaluable architectural feature in today's higher-end processors. The savings they provide in terms of memory latency are immense. Hence, they have become indispensable. Nonetheless, caching has one inherent complexity, namely, the latency of a memory access becomes unpredictable.

An instruction cache may be analyzed to determine the latency of accessing an instruction depending on whether it is found in the instruction cache or not. However, data caches are much more difficult to analyze since a single memory access instruction (reference) could access different memory locations at different points in time. The simplest example of such a case is an access to an array element within a loop. In every iteration of the loop, a different element in the array and, hence, a different memory location, could be accessed. Thus, data cache analysis for the purpose of static timing analysis, while challenging, is an important problem.

While calculating the execution time of a single task is a necessary step in static analysis, it is far from being sufficient in the presence of multiple tasks executing in a prioritized manner. In this context, data caches introduce further complexity to static analysis.

Hypothesis: Data cache behavior of a large class of programs executing in a multi-tasking, prioritized environment can be predicted using advanced static analysis techniques.

1.2 Contributions of this Dissertation

This dissertation proposes analysis techniques to characterize data cache behavior in the context of hard real-time systems. The contributions of this dissertation may be categorized as follows.

1. Single task analysis: Here, an analysis technique that derives a safe and tight upper bound on the worst-case execution time of a single task using data cache reference patterns for all scalar and array memory references in the task is presented.

2. Preemptive tasks: Here, analysis techniques are presented to calculate safe and tight upper bounds on the response times of multiple tasks executing in a prioritized, fully-preemptive manner.

3. Tasks with critical sections: Here, analysis techniques that produce safe and tight upper bounds on the response times of tasks that may contain critical sections are presented.

The rest of this dissertation is organized as follows. Chapter 2 provides required background information and Chapter 3 presents the task model used in this dissertation. Chapter 4 presents techniques to analyze a single task. Chapter 5 introduces a multi-task execution environment. Chapters 6, 7 and 8 present analysis techniques for multiple tasks executing in a prioritized manner for fully-preemptive tasks, tasks with a non-preemptive region and tasks that execute in accordance with resource-sharing protocols, respectively. Chapter 9 discussed related work and Chapter 10 summarizes the contributions of this dissertation and discusses possibilities for future work.

Chapter 2

Background Information

This chapter provides background information required for the topic of this dissertation. Section 2.1 describes the basics of real-time systems and Section 2.2 introduces the concept of timing analysis in real-time systems. Section 2.3 describes memory hierarchies in most modern computer systems, with particular focus on cache memories.

2.1 Real Time Systems

A real-time computer system is one in which, in addition to logical constraints, tasks have temporal constraints to meet. In other words, for every task executing in a real-time environment, there is a specific *deadline* — a time before which the task must complete its execution [1].

2.1.1 Hard and Soft Real-Time Systems

There are two types of real-time systems, namely hard and soft real-time systems. In hard real-time systems, the temporal constraints that are placed on tasks are rigid [2]. If a task does not adhere to these and exceeds its deadline, the consequences are usually severe. Examples of a hard real-time system include aircraft control systems, space explorers, emergency life-support systems, etc. In such systems, completion of an operation after its deadline is considered useless and may lead to failure of the system.

On the other hand, in soft real-time systems, the temporal constraints are more relaxed. In such a system, the usefulness of results obtained from a task does not drop to zero immediately after the deadline, but declines at a slower rate. Hence, as long as a reasonable

quality of service is maintained, a task may miss a few deadlines. An example of a soft real-time system is a multimedia streaming application. Figure 2.1 depicts the usefulness of results with respect to the deadline of a task in hard and soft real-time systems.

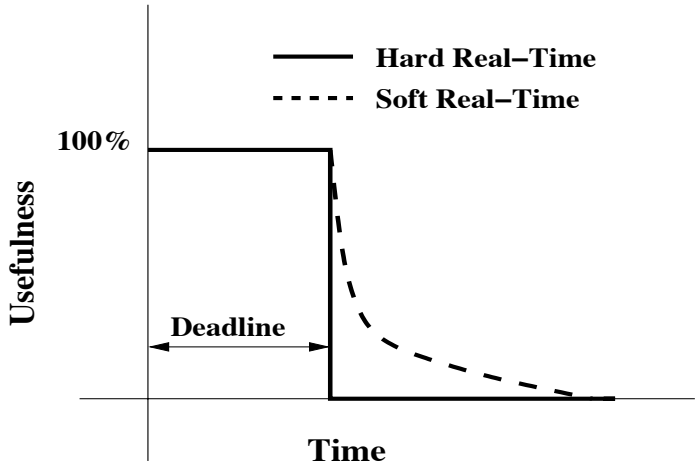


Figure 2.1: Hard versus Soft Real-Time Systems

2.1.2 Real-Time Tasks

There are two types of real-time tasks, namely, periodic tasks and aperiodic (sporadic) tasks. In the case of a periodic task, an *instance*, termed as a *job*, of the task is released at fixed intervals of time. The *release point* of a task represents the time at which it is *ready* for execution. On the other hand, an aperiodic task is one which has a *minimum* inter-arrival time between its instances, but not a fixed one. In this dissertation, the focus is on periodic, hard real-time systems.

Every periodic task possesses the following characteristics.

1. Phase: This represents the time between the start of the system and the release of the first instance of a task.

2. Period: This represents the inter-arrival time between two consecutive instances of a task.

3. Worst-Case Execution Time: This represents a guaranteed upper bound on the execution time of a task.

4. Relative Deadline: This represents the time by which every instance of a task must complete its execution, relative to its own time of release.

2.1.3 Schedulability Analysis

Schedulability analysis of a real-time system refers to the process of checking the feasibility of execution of a given set of tasks such that no task instance misses its deadline. In order to perform this analysis, the four basic characteristics of every task must be known *a-priori* [3].

2.2 Timing Analysis

Timing analysis refers to the process of determining the execution time of a given program. In the context of real-time systems, timing analysis is used to calculate the worst-case execution time of a task, a characteristic that must be known *a-priori* for conducting schedulability tests on a set of real-time tasks. There are two fundamental approaches to timing analysis, namely, dynamic timing analysis and static timing analysis.

2.2.1 Dynamic Timing Analysis

Dynamic timing analysis methods, also known as measurement-based timing methods, estimate the execution time of a given task either by actually executing the task or by simulating the execution of the task repeatedly using different sets of input data.

It has been demonstrated in earlier studies that dynamic analysis by actual execution of a task does not guarantee worst-case estimates [4]. Furthermore, exhaustive testing of the input space is impractical. In the context of a real-time system, specially in the case of a hard real-time system, temporal guarantees are vital to the safety of the system, hence making dynamic timing analysis unsuitable in general.

2.2.2 Static Timing Analysis

Static timing analysis is the process of determining the worst-case execution time of a task without actually executing or simulating the execution of the task. Analytical models of all the components of a system (both hardware/architectural components and software components) are constructed. All execution paths in the task being analyzed are

traversed and, using the models, the effects of all possible inputs on the control flow of a program are determined.

The result of such an analysis is a possibly conservative, yet *safe* upper bound on the execution time of the task. Due to this feature, static timing analysis is a method for determining the worst-case execution time of tasks in the context of a hard real-time system that raises the confidence in the temporal correctness of a system. In the rest of this dissertation, the focus is on static timing analysis.

2.2.3 Unpredictability in Static Timing Analysis

Static timing analysis typically consists of three different phases.

1. Low-level analysis: In this phase, execution times are calculated for all atomic instructions in the instruction set for the architecture under consideration.

2. Flow analysis: In this phase, the control-flow graph of the task being analyzed is traversed to construct paths of execution.

3. WCET calculation: In this phase, analytical models of the components of the system are used in conjunction with information gleaned from the low-level and flow analyses to determine the worst-case execution time of a task.

While the first two phases are fairly straightforward, the structure and control flow of a program may cause significant hurdles in the last phase of static timing analysis. Factors such as data-dependent control flow, indirect memory accesses via pointers, dynamic memory accesses, etc. may cause unpredictability in the process.

Furthermore, the correctness of static timing analysis relies on the correctness and precision of the analytical models of architectural components of the system. Several modern architectural features such as caches, pipelines, branch predictors, etc. are hard to model. One feature that is particularly hard to model is the *data cache*. While data caches are very useful to improve the average-case performance of a system, its worst-case behavior is not easily predictable. Hence, data cache analysis has hence been the focus of much research in recent years.

The static timing analyzer framework used in this dissertation is shown in Figure 2.2 [5, 6, 7, 8]. Source files constituting a program are first compiled using an enhanced GCC compiler to obtain information about the control flow and memory references in the

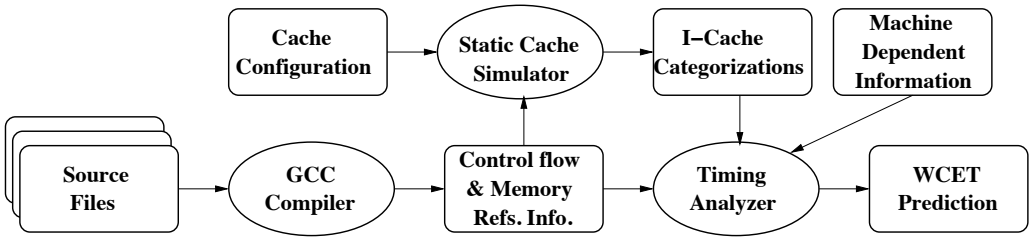


Figure 2.2: Static Timing Analysis Framework

program. This information is fed to a static cache simulator that simulates the *instruction cache* to produce instuction cache categorizations for the given program. The instruction cache categorizations, along with some machine-dependent information, are fed to the core timing analyzer that then calculates an upper bound for the worst-case exeution time of a task.

It may be observed that there is no data cache analysis incorporated in this framework. In Chapter 4, this issue is addressed, and a framework for data cache analysis is presented. The data cache analysis framework is then integrated into the static timing analyzer shown in Figure 2.2.

2.3 Cache Memory

While the processing power and speed of a processor double almost every 18 months, the same cannot be said about the speed of the memory that they need to access frequently. In order to bridge this gap between the processor speeds and memory speeds, most modern processors have a memory hierarchy in place. A typical memory hierarchy is shown in Figure 2.3. A register file, located in the processor core itself, is the fastest and smallest memory in the hierarchy. Disk memory is the slowest and largest memory and is located far away from the processor. Since faster technology is more expensive, the size of the memory is forced to be smaller as the memory gets faster. Hence, the amount of storage decreases moving from lower levels of memory to upper levels [9, 10, 11, 12].

The basic principle that enables such a memory hierarchy to bridge the gap between processor speeds and memory access times is *locality of reference*. In simple terms, this principle refers to the fact that, if a certain memory element is accessed at a certain time, it is very likely that the same memory element or consecutive memory elements in

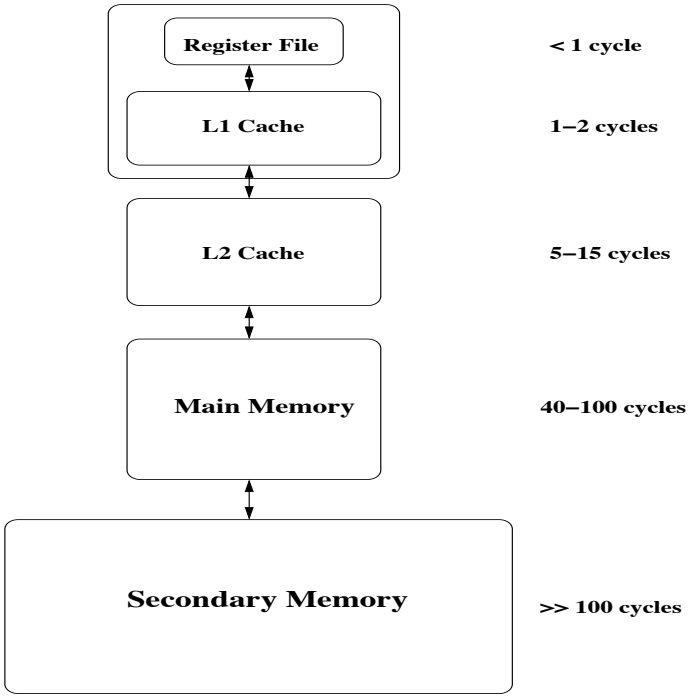


Figure 2.3: Memory Hierarchy in a Typical Computer System

the same memory line will be accessed in the near future. The former is termed *temporal locality* and the latter, *spatial locality*.

A cache is a small, fast area of memory that is located on or close to the processor chip. It stores frequently accessed memory lines for fast retrieval. When a program executes a memory access instruction, the first step is to check whether this requested memory line is already in the cache. If the requested memory line is located in cache, a *cache hit* occurs. Otherwise, a *cache miss* occurs.

The number of cycles required to retrieve the requested data from lower levels in the memory hierarchy is termed *miss penalty*. Cache misses are classified into three categories, namely, compulsary, capacity and conflict. Compulsary misses, also known as cold misses, are the ones that are incurred the first time a certain memory line is brought into the cache. Capacity misses occur when the cache is not large enough to hold a program's entire working set. Conflict misses occur if two memory lines map to the same cache line. The basics of mapping between memory and cache lines are discussed below.

2.3.1 Cache Organization

Every memory line in the main memory maps to a certain line in the cache memory. Since the main memory is much larger than cache, this mapping is many-to-one. There are three cache organizations based on the mapping function used.

1. Direct mapped: In a direct mapped cache, a certain memory line can map only to one specific line in the cache. The cache line is determined by performing a modulo operation on the address of the memory line.

2. Fully Associative: This is on the other end of the spectrum, where a certain memory line can be mapped to any line in cache. Hence, the mapping is not dependent on memory address, but rather on the usage of the cache.

3. Set associative (k-way): In this method, the cache is divided into sets. Each set consits of k cache lines. A modulo operation on the address of a memory line determines which cache set is to be used. The memory line may then be placed in any cache line within the identified set. A fully associative cache is a set associative cache with just one set and a direct mapped cache, a 1-way set associative cache.

Figure 2.4 depicts these three different cache organizations and shows which cache lines a particular memory line may be mapped to. This memory line is represented by a shaded rectangle, and the cache lines that it may be mapped to in the different organizations are also represented by shaded rectangles. The numbers above each rectangle representing a memory line or a cache line are line numbers. In the case of set-associative caches, the labels (prefixed by S) below indicate set numbers. In this example, the set associative cache has an associativity of 2, indicating a set size of two.

2.3.2 Replacement Policy

Multiple memory lines that map to the same cache line could replace each other in cache. In a direct mapped cache, this replacement is straightforward since every memory line maps to one specific cache line. In case of a set associative cache, when a set is full, any new memory line that maps to the same set has to replace *one* of the memory lines that already exist in the set. Now, there arises the question of which line to select for replacement. There are several replacement policies and some of the commonly used ones are described below.

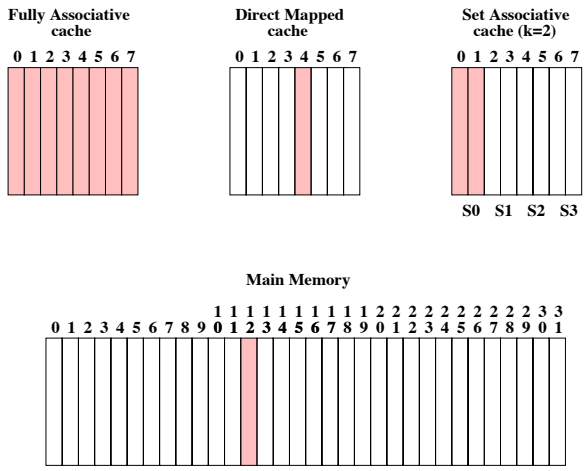


Figure 2.4: Possible Mappings Between Main and Cache Memories

1. First In First Out (FIFO): Here, the line which was brought into the cache the earliest is the first to be replaced.

2. Least Recently Used (LRU): This policy replaces the cache line that was used the longest time ago.

3. Random: As the name suggests, in this case, a cache line is chosen at random for replacement.

2.3.3 Write Policy

Memory accesses may be of two types: read and write. When a memory write is performed, there are two issues that need to be considered, namely, the issue of bringing the accessed memory line into cache and the issue of writing the modified data into main memory. There are four policies based on these issues.

1. Write Through: When a memory write is performed, the modified contents are written through to the cache and the main memory.

2. Write Back: When a memory write is performed, it is written only to the cache. It is written to main memory later when the cache line is replaced.

3. Write Allocate: The memory line is modified and then brought into the cache.

4. Write No Allocate: The memory line is modified, but it is not brought into the cache.

Systems specify a write policy for both a cache hit and a cache miss. The two most commonly used combinations of policies are the write through with no write allocate and the write back with write allocate.

2.3.4 Instruction and Data Caches

Instructions used to execute a program and data used by programs are both stored in memory. Since the way that instructions are interpreted and used is different from the way that data is used, typically, at the higest level of cache, there is a segregation between the part that holds instructions and the part that holds data. These are known as the *instruction cache* and the *data cache*, respectively.

Chapter 3

Task Model

In the work presented in this dissertation, the focus is on periodic, hard real-time tasks. Every task is assumed to have a deadline less than or equal to its period, an assumption that is reasonable in a majority of hard real-time systems.

A task is assumed to have two types of characteristics, namely *basic* and *derived*. A basic characteristic of a task is one that is defined for a specific *single* task, and a derived characteristic is one that is defined in the context of a given *set* of tasks.

As introduced in Section 2.1.2, every periodic real-time task has four basic characteristics. In addition to these four characteristics, a task may have more characteristics, both basic and dervied. The notations used in this dissertation for the basic and derived characteristics of a task are now presented.

A task is denoted by T_i, where i is a unique identifier assigned to the task. The jth instance of T_i is denoted as job $J_{i,j}$. The *basic* characteristics of T_i are represented by the 5-tuple (Φ_i, P_i, C_i, c_i, D_i).

1. Φ_i represents the *phase* of a task, *i.e.*, the time between the start of the system and the release of the first instance of the task (first job).

2. P_i represents the *period* of a task, *i.e.*, the inter-arrival time between consecutive instances of the task.

3. C_i represents the worst-case execution time ($WCET$), of a task, *i.e.*, the longest possible execution time for the task.

4. c_i represents the best-case execution time ($BCET$), of a task, *i.e.*, the shortest possible

execution time for the task.

5. D_i represents the *relative deadline* of a task, *i.e.*, the time by when the task must complete its execution relative to its time of release.

In the context of a specific task set, a task has a set of derived characteristics, represented by the 3 tuple (B_i, $\Re_i$, Δ_i).

1. B_i represents the *blocking time* of a task, *i.e.*, the time for which its execution might be interrupted due to a task with lower priority that holds a shared resource required by the task.

2. $\Re_i$ represents the *response time* of a task, *i.e.*, the time between the release of the task and its completion.

3. Δ_i represents the *data-cache related delay* incurred by a task due to interruptions by other tasks.

The lowest common multiple (LCM) of the periods of all tasks within a task set is known as the *hyperperiod* of the task set.

The work presented in this dissertation is divided into three major components. In the first component, data cache behavior is characterized with respect to a single task. In this component, only four basic characteristics of a task are used — phase, period, WCET and relative deadline. In the second component, the concept of preemptions among tasks is introduced. In this component, one basic characteristic and three derived characteristics are added to every task. The basic characteristic added is the best-case execution time of the task. The derived characteristics, in the context of a task set, are response time and data-cache related delay. In the final component, a task may have critical sections within its execution. In this context, every task has an additional derived characteristic, namely blocking time.

Chapter 4

Data Cache Analysis - Single Task

In this chapter, an analytical technique for statically characterizing data cache behavior in the context of a single real-time task is presented. An existing data cache analysis framework, known as the Cache Miss Equations framework [13] was enhanced to analyze a single task and calculate the number and positions of data cache misses that the task incurs.

First, background information about the Cache Miss Equations framework is provided. Next, the enhancements incorporated by the work presented in this chapter are discussed. Finally, experimental results are provided.

4.1 Cache Miss Equations Overview

The Cache Miss Equation (CME) framework proposed by Ghosh et al. [13] is a method to generate a set of linear Diophantine equations to characterize the behavior of a data cache in loop nest oriented code consisting of scalar and non-scalar (array) memory references.

4.1.1 Terminology

Iteration Space

Every iteration of a loop nest is represented as an entity known as an *iteration point*. For example, in a loop nest of depth 3, the iteration where the values of the induction variables are 1, 2 and 3, respectively, for each loop starting from the outermost one, would

be represented as the iteration point $\vec{i} = (1, 2, 3)$. The set of all iteration points for a given loop nest is known as its *iteration space*.

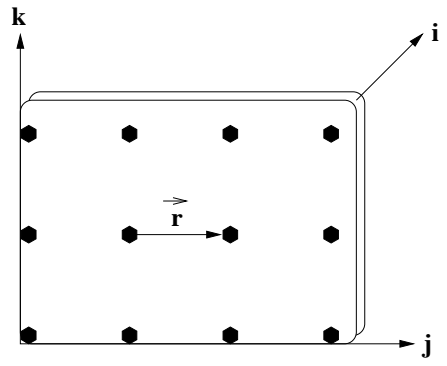
Reference and Access

A reference is a static memory read or write instruction in the program. A particular execution of a reference is termed as a memory access.

Reuse Vectors

In order to summarize data reuse among references in loop nest oriented code, the CME framework uses the concept of reuse vectors as defined by Wolf and Lam [14]. If a reference accesses the same memory line in two iterations $\vec{i}_1$ and $\vec{i}_2$, where $\vec{i}_2 > \vec{i}_1$, $\vec{r} = \vec{i}_2 - \vec{i}_1$ is called a reuse vector. For example, consider the matrix multiplication code shown in Figure 4.1(a).

```
for(i = 0; i < N; ++i)
    for(k = 0; k < N; ++k)
        for(j = 0; j < N; ++j)
            C[j][i] += A[k][i] * B[j][k]
```



(a) Matrix Multiplication Code (b) Iteration Space for Matrix Multiplication Code

Figure 4.1: Loop Nest and Iteration Space with one Reuse Vector $\vec{r} = (0, 0, 1)$ for $A[k][i]$

Here, the reference $B(j, k)$ has a reuse, which is represented by the reuse vector $(0, 1, 0)$. Reuse vectors are classified into four types as described below.

1. **Self-temporal reuse** A self-temporal reuse occurs when a reference accesses the same memory *element* in different iterations.

2. **Self-spatial reuse** A self-spatial reuse occurs when a reference accesses the same memory *line* in different iterations.

3. **Group-temporal reuse** A group-temporal reuse occurs when different references access the same memory element.

4. **Group-spatial reuse** A group-spatial reuse occurs when different references access the same memory line.

Consider the matrix multiplication code in Figure 4.1(a) again. The iteration space for this code is shown in Figure 4.1(b). An example reuse vector $\vec{r} = (0, 1, 0)$ is shown in the iteration space.

Perfectly Nested and Rectangular Loops

A perfectly nested loop is one in which all memory references are found in the inner most loop of the loop nest. A rectangular loop is one in which the upper bound of an inner loop does not depend on the current iteration of an outer loop.

4.1.2 Cache Miss Equations

CMEs are a set of equations whose solutions represent *potential* cache misses for references in a loop nest. They relate the iteration space of the loop nest, base addresses of arrays, array sizes and the cache parameters in a precise fashion. For every reference, and along every reuse vector for that reference, two kinds of CMEs are generated — *cold miss equations* and *replacement miss equations*. The term *along a reuse vector* means that, for a reference, only that reuse vector is assumed to be present, ignoring the presence of any other reuse vectors.

Cold Miss Equations

Solutions to cold miss equations represent potential cold data cache misses. These are misses that occur upon the first access to a memory line, hence making them *compulsary* by definition. Cold misses may occur in two cases, namely, when a reference reuses data from an iteration point that is *outside* the iteration space and when a reference reuses data that is mapped to a *different* cache line.

Replacement Miss Equations

Solutions to replacement equations account for the remaining misses, namely, capacity and conflict misses. For a given reference, replacement miss equations along a particular reuse vector represent interference with any other reference, including itself (self-conflict).

Solving CMEs directly is computationally complex. However, mathematical techniques for manipulating these equations are employed to make the process tractable [15, 16]. Solutions to each CME only represent *potential* cache misses. The effects of multiple CMEs are composed to find the *actual* miss points. A detailed description of the generation of CMEs and the algorithm to compute the actual misses may be found in [13].

4.1.3 CME Implementation Overview

In the work presented in this dissertation, an existing implementation of the CME framework is re-used and enhanced. The original implementation framework, named *Coyote*, is derived from work by Bermudo et al. [15]. The Coyote framework utilizes basic reuse vectors as suggested in [14]. The Coyote framework is extended, in the work presented in this dissertation, to take into account the precise shape of the iteration space. This extended Coyote framewok forms the *data cache analyzer* in Figure 4.7 of Section 2.2.2 and will be referred to as such in the remainder of this dissertation.

4.2 Conceptual Enhancements to the CME framework

The original CME framework imposes several restrictions on programs that it can analyze. Fundamental among these are as follows. First, the upper bounds of all loops must be known at compile-time. Second, array subscript expressions must be affine functions of the loop induction variables. Third, the program can contain only perfectly nested, rectangular loops. Fourth, the program cannot contain data-dependent conditionals. Several enhancements were introduced into the CME framework in order to relax some of the above restrictions. These enhancements are depicted by the block diagram shown in Figure 4.2.

Recent work by Vera et al. [17] relaxes the assumption about perfectly nested loops and allows sequential loop nests of equal depth by transforming arbitrary loop nests.

This part of the loop transformation is done in the blocks (d) and (e) of Figure 4.2. A disadvantage of this scheme is that it leads to changes in representation of reuse and iteration spaces. In order to overcome this disadvantage and yet allow arbitrary loop nests, the concept of *forced loop fusion* is introduced. This is represented by block (f) in Figure 4.2. These loop transformations are explained in detail in Section 4.3.

The enhanced framework permits non-rectangular loops in programs. Conceptually, a non-rectangular loop is represented by a condition posed on the upper bound of an inner loop, based on the current value of the induction variable of an outer loop. They are treated as such since any condition that is based solely on induction variables is analyzable statically.

Finally, programs with *data-dependent conditionals* that have no "else" part are permitted. If the condition is not satisfied, one part of the code is simply skipped. An upper bound on the number of misses incurred by a program with such a conditional is calculated. For a direct-mapped cache, the worst case is to assume that the condition is satisfied.

Since the data cache analyzer reuses the original Coyote framework (shown in block (g) of Figure 4.2), up to the CME generation stage, the equations are generated assuming that all references in the loop nest are executed at every point in the iteration space. However, several statically analyzable conditionals are introduced during forced loop fusion and while handling non-rectangular loop nests. Consequently, the reuse vectors generated are no longer correct and could lead to overly optimistic (unsafe) or overly pessimistic (unreal) results. To prevent timing violations and ensure timing safety, an extra analysis step is added to the actual miss calculation stage. This step is represented by block (h) of Figure 4.2.

4.3 Loop Transformations

The data cache analyzer performs several loop transformations on a program in order to make programs with arbitrary loop nests analyzable by the CME framework. In the first step, the depths of all loop nests in the program are equalized by introducing dummy loops with a single iteration where required. In the second step, forced loop fusion is applied to these sequential loop nests of equal depth. The concept of forced loop fusion is explained in Section 4.3.1.

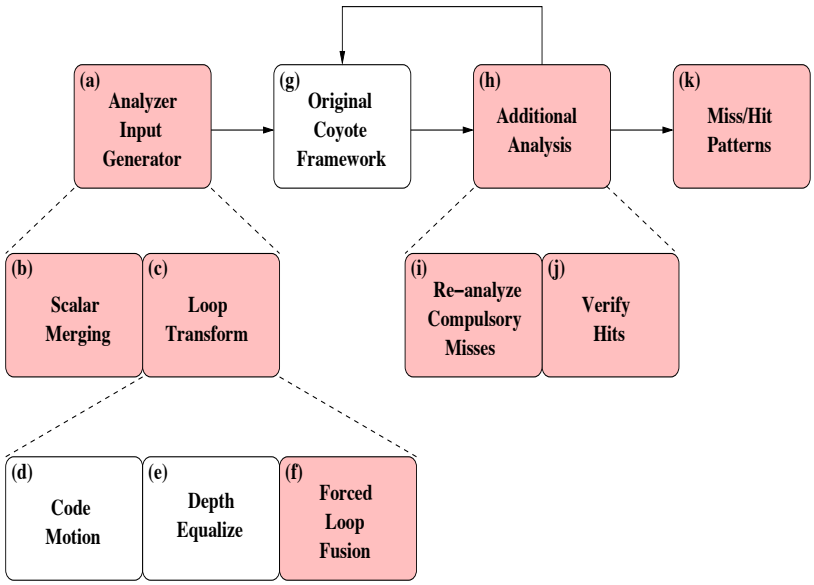


Figure 4.2: Data Cache Analyzer: Enhanced Coyote Framework

4.3.1 Forced Loop Fusion

Forced loop fusion is a technique whereby iteration spaces of several loop nests of equal depth are concatenated to form one single loop nest. The basic idea behind this technique is to concatenate iterations of all loops at the same depth. In order to maintain the correct order of memory accesses within loops, conditionals based on the loop induction variables are introduced as required.

The algorithm for performing forced loop fusion is described in Figure 4.3. Fusion starts from the outermost level and proceeds inwards. For every distinct level in the input loop nests, a corresponding level is introduced in the fused output loop nest. The number of iterations of the fused loop is the sum of the number of iterations of every loop at that level in the input loop nests.

As an example, consider the loop nests shown in Figure 4.4(a). Here, the input program has two distinct loop levels. Fusion is started at the outer loop level. Since there are two loops at that level, with M and P number of iterations respectively, the outer level of the fused loop nest would have a number of iterations equal to $M + P$. In order to maintain the correct order of memory accesses, one conditional is introduced for each reference in the innermost loop to specify when that reference is to be executed with respect to the recently fused loop level. The resulting loop nest after one level of fusion is shown

```
n iter = num iters in curr level
lb new = lower bound for new loop at curr level
ub new = upper bound for new loop at curr level
for each loop level in loop nests
    n iter = 0
    for each loop at curr level
        n iter += num iters in curr loop
    lb new = 1
    ub new = n iter
    remove loops at curr level from loop nests
    add new loop at curr level to loop nests with new index
    update subscripts of all references in loop nests as follows:
    for each reference
        for each subscript referring to curr loop level
            oldindex = (new index for level −
            num iters. in orig. loop represented by old index)
```

Figure 4.3: Forced Loop Fusion Pseudocode

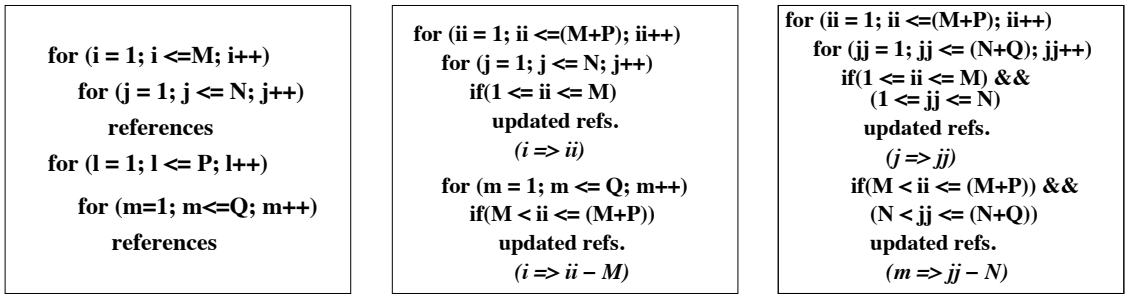
in Figure 4.4(b). The same process is repeated for the subsequent levels in the input loop nests until one single, perfectly nested loop is formed. The final fused output loop nest obtained in this example in shown in Figure 4.4(c).

4.4 Deriving Exact Data Cache Reference Patterns

The original CME work provides slightly pessimistic results for certain programs. In addition to the conceptual enhancements presented in Section 4.2, new approaches are proposed to overcome this pessimism.

4.4.1 Causes for Pessimism in CME Framework

Each individual CME only represents iteration points that could *potentially* suffer a data cache miss. These iteration points are then analyzed considering the combined effect of all reuse vectors for the reference, thus categorzing the reference as a miss or a hit in the data cache for that particular iteration point. The CME framework produces slightly pessimistic estimates for the number of misses that each reference in a loop nest incurs.



(a) Original Loop Nests (b) After Fusing Outermost Levels (c) Final Fused Loop Nest

Figure 4.4: Example Illustrating Forced Loop Fusion

There are several reasons for this.

First, the implementation of CMEs, as provided by Coyote, does not analyze all iteration points due to the complexity involved. Instead, a representative sample of the iteration space is considered for analysis and a *confidence* value is given as a feedback.

The second problem stems from the layout of array elements in cache lines. In the original CME framework by Ghosh et al. [13], all arrays are assumed to be *aligned* in memory lines and, hence, cache lines. This assumption might not always be true — the first element of an array may have a non-zero offset from the start of the cache line. The Coyote framework relaxes this assumption and takes *exact base addresses* into consideration during its analysis [15, 16]. However, even Coyote does not take arbitrary reuses into account.

If arrays are not cache-aligned and elements are accessed in non-sequential order, they may have reuses that the original CME framework does not detect. As an example, consider a *two-dimensional* array $a[1..10][1..10]$ that has a *column-major* layout. For the sake of simplicity, consider a data cache that is large enough to hold the array entirely. Let the size of each cache line be 32 bytes and the size of each array element be 4 bytes. The base address of the array is assumed to be such that it causes the the mapping shown in Figure 4.5.

Now, consider an iteration space of depth two that traverses the array in row-major order. The elements $a[1][1], a[1][2]$ and $a[1][3]$ are correctly categorized by the CME framework as cold misses since it is the first time those memory lines are accessed. Next, elements like $a[3][3]$ are also classified as cold misses since they are on a different memory line than previously accessed data. In a similar fashion, the CME framework also classifies access $a[5][2]$ as a cold miss. However, in reality, since $a[5][2]$ is on the same memory line

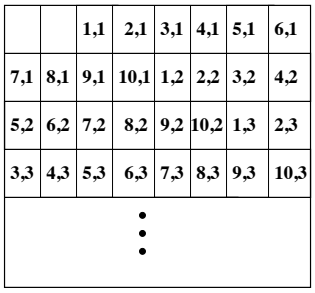


Figure 4.5: Sample Mapping of 2-D Column-Major Array in Cache

as $a[1][3]$, it has already been brought into the cache and should actually be classified as a *hit*. Ignoring such reuse leads to pessimism in the miss count.

A third reason for pessimism in the CME framework is that it only captures reuse between uniformly generated references. Reuse across variables is not captured. While this impacts array references only for layouts where one array ends and another starts in the same cache line, it severely impacts programs that have frequent references to scalar variables that share cache lines due to their placement in memory.

4.4.2 Enhancements for Removal of Pessimism

In the work presented in this dissertation, the stress is on deriving *exact* data cache reference patterns. This makes it mandatory for the data cache analyzer to consider *all* iteration points while computing actual miss points. This increases the complexity of computation. However, since the analysis is a *static* one that *pre-computes* all data cache reference patterns by code analysis, this *one-time* overhead is considered to be reasonable. Moreover, the improvement an exact pattern promises in accuracy of static timing analysis for a single task is a significant motivation for this approach in spite of the overhead.

The second problem described in Section 4.4.1 is not easy to resolve using reuse vectors. Hence, a different approach is taken and further analysis is performed on the iteration points that are classified as compulsory misses by the CME framework. This is represented by the block (i) in Figure 4.2. The analysis is as follows. First, a check is conducted to see if there exists any prior iteration that references an element in the same cache line as that of the reference under consideration. If so, a second check is conducted to see if this reference has been replaced since it was last brought into the data cache. In order to avoid traversing the iteration space to find such iterations, a back-tracking approach is

used. Elements that map to the same cache line as the reference under consideration are mapped back to the iteration space. This produces an iteration point that refers to these elements. Afterwards, the check to ensure that this iteration point is *earlier* in the iteration space than the current iteration is a simple process. Since the number of elements that can map to any cache line is a constant, the complexity involved in this process is generally affordable.

If a program has many scalars, the first access to each of them is treated as a miss by the original framework. Here, each variable would be considered separately and, hence, reuse vectors do not capture reuse between them. In order to overcome this limitation, scalars of equal size that are adjacent in memory are *merged* and treated as an array for the purposes of data cache analysis. This merging is performed as part of a pre-analysis phase as shown in block (b) of Figure 4.2. Hence, reuse between different scalars that map to the same cache line is captured by treating them as elements of a single array.

The result produced by the data cache analyzer, with the conceptual enhancements presented in Section 4.2 and the enhancements presented in this section, is a series of data cache miss/hit patterns - one for every reference in the program being analyzed.

4.5 Putting it All Together: An Example

In this section, a simple example illustrating data cache analysis approach is provided. Consider a direct-mapped, 1 KB data cache with a line size of 32 bytes. The input loop nests for the example task is shown in Figure 4.6(a). The input code consists of four memory references, named 1, 2, 3 and 4 in the order of the first execution of each reference. Characteristics of the two variables in the code, namely, A and B, are shown in Table 4.1.

Table 4.1: Characteristics of Variables

Ref	Dimensions	Base Address	Element Size
A	1..10, 1..10	151944	4
B	1..10, 1..10	153000	4

First, the input loop nests are pre-processed in accordance with the transformations described in Section 4.3.1. Recall that forced fusion concatenates loop bodies by extending the first iteration space with the second one. Loop bodies are conditionally executed depending on the iteration point in the fused space. The resulting loop nest is shown

```
for(i = 1; i <=10; i++)
    for(j = 1; j <= 5; j++)
       A[i][j] = 19 ;
    for(k = 1; k <= 10; k++)
       D[i][k] = A[i][k] + 7 ;
for(s = 1; s <= 10; s++)
    for(m = 1; m <= 5; m++)
       D[s][m] = 13 ;
```

(a) Original Loop Nests

```
for(i = 1; i <=20; i++)
  for(j = 1; j <= 20; j++)
    if( (i >=1) && (i <= 10)
      && (j >=0) && (j <= 5) )
        A[i][j] = 19 ;
    if( (i >=1) && (i <= 10)
      && (j >=6) && (j <= 15) )
        D[i][j−5] = A[i][j−5] + 7 ;
    if( (i >=11) && (i <= 20)
      && (j >=16) && (j <= 20) )
        D[i−10][j−15] = 13 ;
```

(b) Transformed Loop Nest

Figure 4.6: Loop Transformation

in Figure 4.6(b).

Using this transformed loop nest as input, cache miss equations are generated using the data cache analyzer (extended Coyote framework). The miss/hit patterns that are produced as a result are shown in Table 4.2. Column one shows the reference number. Column two indicates the number of misses the original Coyote framework produces for the given program and column three shows the results produced by the enhanced data cache analyzer.

Table 4.2: Coyote Output vs. Data Cache Analyzer Output (Hit: dot, Miss: M)

Ref	**Coyote misses**	**Data Cache Analyzer Output Miss/Hit patterns**
1	50	MMMMM.....M.....
2	100	.....MMMMM.....M.....M .....
3	100	MMMMMMMMMM.....M...M.....M .....
4	50	.....

The original Coyote framework cannot analyze loop nests with the structure shown in Figure 4.6(a). It is given the transformed loop nest shown in Figure 4.6(b), but has no knowledge of the conditionals introduced due to loop transformations. Hence, the Coyote

framework assumes that all the references are executed at every point in the fused iteration space and produces very pessimistic results.

On the other hand, the miss/hit patterns produced by the data cache analyzer are accurate. They not only convey information about the number of misses for every reference, but also about their positions in the access space of the reference.

4.6 Implications to the Static Timing Analyzer

The data cache analysis technique presented thus far in this chapter implies significant improvements to static timing analysis. The static timing analyzer framework described in Section 2.2.2 is now enhanced to include the data cache analyzer. The resulting framework is depicted in Figure 4.7. The shaded blocks represent the novel and enhanced modules of the framework.

Control flow and memory reference information is now fed to the input generator module, which is responsible for performing scalar merging, loop transformations, etc. as explained in Sections 4.3 and 4.4. The resulting single, perfectly nested loop is given to the data cache analyzer, which then produces data cache miss/hit patterns, also termed *data cache reference patterns*. These patterns indicate which data cache accesses are guaranteed

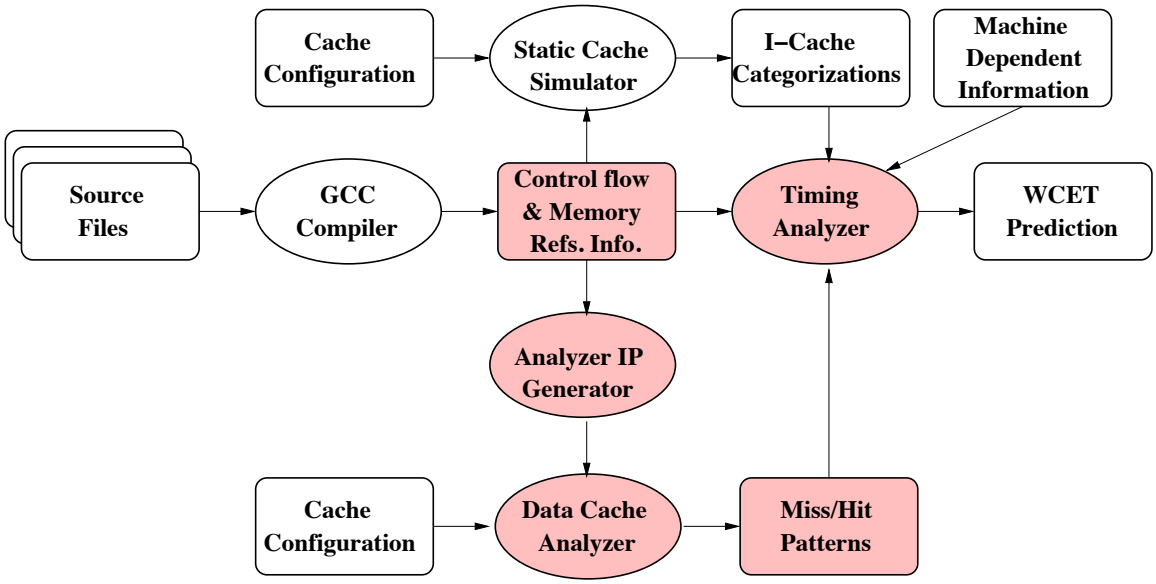


Figure 4.7: Static Timing Analysis Framework Enhanced with Data Cache Analyzer

to be hits and which others are not.

The data cache reference patterns indicate the *position* of each miss in a sequence of references. However, in this chapter, the aim is to obtain WCET estimates for a *single* task ignoring the interference from other tasks in the system. For this purpose, it is sufficient to use just the *number* of data cache misses that a reference incurs. Hence, in this chapter, only the number of misses is fed to the core timing analyzer. The timing analyzer considers the impact of these cache misses in the context of pipeline analysis and path traversal to obtain bounds on the WCET of the task.

In order to obtain *safe and tight* WCET bounds, static timing analysis considers one loop nest at a time starting with the inner-most nest. The times of the longest paths are then repeatedly determined as long as a change in the cache or processor states is observed. A *steady state* is said to be reached when two consecutive loop iterations result in the same WCET bound. The remaining loop iterations are then guaranteed to be bound by this fixpoint as well [18]. The overall bound for an inner loop can be used directly in the context of the outer loop in conjunction with adjustments due to caching effects between loop nests. This method assumes a consistent pattern for the worst-case cache categorization, even across loops. While this assumption is valid in the case of an instruction cache, it may not be valid for a data cache.

Consider n data cache misses for a reference, where n may exceed the upper bound on the number of iterations for an inner loop. Hence, misses extend beyond the iterations of the inner loop. Furthermore, these misses may be scattered over a subset of iterations of the outer loop, not necessarily following any regular pattern, as was observed in the experiments. To handle such misses, the next iteration of the outer loop needs to be considered when finding a fixpoint for the inner loop. This increases the number of iterations of the inner loop that need to be considered before reaching a steady state, thus increasing the complexity of static timing analysis. A method is now presented to solve this problem with a space and time complexity $O(r + 1)$, where r is the number of references in a loop nest.

To illustrate the solution, consider the code shown in Figure 4.8(a). For the sake of demonstration, assume that the number of misses predicted by the data cache analyzer for the refernce $A[i][j]$ is 13. The inner j loop is timed once considering the reference $A[i][j]$ to be a miss. This is termed the *miss time* for the loop. Next, the inner loop is timed considering the same reference to be a hit and this is termed the *hit time*. During timing analysis, information about the miss and hit times of the inner loop for specific iterations

```
for(i = 1; i < 10; ++i)
    for(j = 1; j <= 10; ++j
        A[i][j] = 19;
```

(a) Sample Loop Nests

i	Iterations with miss time for j loop	Iterations with Hit time for j loop
1	10	0
2	3	7
3..10	0	10

(b) Miss and Hit Time

Figure 4.8: Miss Pattern Crossing Loop Nests

of the outer loop are propagated. Table 4.8(b) shows these values for the example being considered.

This concept, when extended to a loop nest with several references, leads to an algorithm with complexity $O(r+1)$ since several permutations of miss/hit status of references need to be considered unlike just two timings in the current example.

4.7 Experimental Setup

Several experiments are conducted to demonstrate the data cache analysis technique introduced in this chapter. In all experiments, a direct-mapped cache of size 1 kilobyte is assumed unless otherwise mentioned. All but two of the benchmarks used in the experiments are taken from the DSPStone benchmark suite [19].

These benchmark programs are modified to replace pointer-based memory accesses with equivalent array accesses to make them statically analyzable. The concept of *abstract inlining* is used to inline all the functions in the benchmark due to implementation constraints. Since these changes do not affect the order of memory accesses in the benchmarks, they are acceptable for the purposes of data cache analysis. Some of the benchmarks in the DSPStone suite are not suitable since they have indirect memory accesses, which are currently not analyzable by the data cache analyzer.

A sorting benchmark, simple-srt-test, is taken from the CLAB benchmark suite. Lastly, a synthetic benchmark is contructed in order to assess the contributions of the data cache analyzer in comparison to those of the original Coyote framework.

Table 4.3: Comparison: Coyote/Data Cache Analyzer/Trace-Driven for 4KB Data Cache

Benchmark used	**Coyote framework**		**Data Cache Analyzer**		**Simulator**	
	Misses	**Hits**	**Misses**	**Hits**	**Misses**	**Hits**
convolution	400	400	**26**	374	26	374
dotproduct	8	0	**3**	5	1	7
fir	599	1192	**26**	573	26	573
lms	1207	9449	**27**	1071	27	1071
matrix1	4600	779400	**39**	4561	38	4562
nrealupdates	1200	2400	**52**	1148	50	1150
simple-srt-test	14	59986	**14**	29686	14	29686
looptest	39	161	**26**	174	26	174

4.8 Experimental Results

In this section, results from experiments conducted using the benchmarks described above are presented. The first set of experiments compares results obtained from the original Coyote framework and the enhanced data cache analyzer. Table 4.3 shows the number of misses and hits produced by the original framework and the enhanced analyzer, respectively.

For all except the last benchmark, arrays are assumed to be aligned on cache line boundaries for the sake of simplicity. For all benchmarks, it may be observed that there is a mismatch in the total number of accesses (hits+misses) between the original Coyote framework and the data cache analyzer.

As explained in Section 4.5, this is due to the fact that the original framework cannot analyze the benchmarks as they are. Thus, the benchmarks are transformed to a form accepted by the original framework. However, during this process, several conditionals based on the loop induction variables are introduced, which the Coyote framework is not aware of and cannot take into consideration. Hence, it assumes the entire fused iteration space with accesses in an unconditional fashion, thereby causing a mismatch in the total number of accesses.

It may be observed that Coyote does not catch even a single hit in reality in any except the last two benchmarks shown in Table 4.3. Hence, for these benchmarks, the very fact that the data cache analyzer is able to analyze them is an advantage that the original framework does not possess. For the simple-srt-test benchmark, the loop nest is non-rectangular. Since Coyote does not recognize non-rectangular benchmarks, it assumes

that the entire rectangular space is traversed.

For the sake of comparison on equal ground, a synthetic benchmark with a loop structure that is analyzable by Coyote is constructed. This is shown the last row in Table 4.3. For this benchmark, it may be observed that the data cache analyzer produces tighter estimates than the original Coyote framework. The reason for this is two-fold. First, arrays are not aligned on cache line boundary as assumed by Coyote, and, second, adjacent scalars that share a cache line are not recognized as such by Coyote (explained in Section 4.4).

In order to verify the correctness of the results obtained above, a cache simulation is performed for each of the benchmarks, using worst-case input. These results are also shown in Table 4.3. It may be observed that the data cache analyzer never underestimates the worst-case performance of a task. Table 4.4 shows the per-reference breakdown of the same results for one of the benchmarks. The reason for the small disparity between the results of the data cache simulator and the data cache analyzer is that the data cache analyzer only considers reuse within a variable. Reuse across multiple variables is not considered. As explained in Section 4.2, this problem is handled in the case of scalars since the disparity would be much more significant there.

simple-srt-test is a sorting benchmark taken from the CLAB suite. This benchmark contains data dependent conditionals and non-rectangular loops. It may be observed from results that the data cache analyzer produces an exact bound on the number of cache misses for this benchmark.

Table 4.4: Per-Reference Misses by Data Cache Analyzer vs. Trace-Driven for matrix1

Reference	**Data Cache Analyzer**	**Simulator**
1	13	13
2	13	13
3	13	12
4..10	0	0

The final set of experiments conducted demonstrates the fact that consideration of a data cache for purposes of timing analysis makes a significant difference to the WCET bound produced by the timing analyzer. The results in Table 4.5 show the worst case execution cycles (WCEC) when data references are considered as 1) always miss, 2) first N misses and 3) cold misses only. The second category, namely *first N misses*, uses the output produced by the data cache analyzer framework. The third category uses cold miss counts

from the trace-driven simulator to verify results.

From the results in Table 4.5, it may be seen that considering *every* reference as a miss would severely overestimate the WCET bounds. On the other hand, the estimate produced by the data cache analyzer is a tight upper bound on the number of data cache misses, thereby enabling tight WCET bounds. For a cache size of 4 KB, which is large enough to fit the entire data set for all benchmarks, the estimate comes very close to the estimate considering only cold misses as provided by the trace-driven simulator. For a smaller cache size which results in additional misses, the estimate produced by the data cache analyzer is tight.

Table 4.5: Timing Analysis for Different Data Cache Categorizations in *Cycles*

Benchmark	**Always Miss**	**First N Misses**		**Cold Misses**
		1K Cache	**4K Cache**	
convolution	8791	**5051**	**5051**	5051
dotproduct	530	**480**	**480**	460
fir	12797	**7097**	**7097**	7097
lms	18544	**11814**	**11814**	11814
matrix1	96168	**52378**	**50558**	50548
nrealupdates	23338	**12658**	**11858**	11838
simple-srt-test	668894	**372034**	**372034**	372034
looptest	6482	**4742**	**4742**	4742

Chapter 5

Data Cache Analysis - Multiple Tasks

In Chapter 4, a technique that characterizes data cache behavior for a *single* task is presented. In this technique, the data cache is assumed to be used solely by the task being analyzed, ignoring the effects of interference from other tasks. However, most practical real-time systems have multiple tasks.

In the next three chapters (Chapters 6, 7 and 8), data cache analysis techniques for task sets with multiple tasks executing in a prioritized fashion are presented.

5.1 Response Time Analysis

Response time analysis is used to determine schedulability of a task set [20, 21]. The response time of a task, as explained in Chapter 3, is the time between its release and its completion. The worst-case response time of a task includes a) the worst-case execution time of the task, b) the execution of higher-priority jobs within the response time of the task and c) the data-cache related delay experienced by the task due to interference from other tasks. In the context of tasks with critical sections, the worst-case response time includes an additional waiting time due to lower-priority tasks.

The response time of a task is calculated using an iterative approach as indicated in Equation 5.1.

$$\Re_i^{n+1} = C_i + B_i + \sum_{\forall k \in hp(i)} \lceil \frac{\Re_i^n}{P_k} \rceil \cdot C_k + \Delta_i \tag{5.1}$$

In some situations, there arises a need to calculate the response time of a particular job. Equation 5.2 is used in such a scenario.

$$\Re_{i,j}^{n+1} = C_i + B_{i,j} + \sum_{\forall (k,l) \in hp(i,j)} \lceil \frac{\Re_{i,j}^n}{P_k} \rceil \cdot C_{k,l} + \Delta_{i,j} \tag{5.2}$$

In Equations 5.1 and 5.2, the set *hp* denotes the set of tasks/jobs with a higher priority than the current task/job. For every task/job, the value of $\Re$ that converges this equation is its response time.

5.2 Experimental Framework

The next three chapters use a common experimental framework, described below. In all experiments, a 4KB, direct-mapped data cache with a hit penalty of 1 cycle and a miss penalty of 100 cycles is assumed. Several task sets are constructed using the DSPStone benchmarks ([19]) with different data set sizes. The benchmarks used, along with their stand-alone WCETs and BCETs (with base execution times calculated using the technique described in Chapter4), are shown in Table 5.1. A benchmark ID is given to each of the benchmarks. This ID will be referenced in experimental result tables. The prefixed numbers in some of the benchmarks indicate the number of iterations within the task. In all benchmarks, with the exceptions of matrix1 and dot-product, the number of iterations is 100 in cases where there is no prefix.

Table 5.1: Stand-Alone WCETs and BCETs of DSPStone Benchmarks

ID	Name	WCET	BCET	ID	Name	WCET	BCET
1	convolution	7491	7491	15	matrix1	59896	54015
2	200convolution	14191	14191	16	fir	9537	9537
3	300convolution	20891	20891	17	500fir	43937	43937
4	500convolution	34291	34291	18	600fir	54837	52537
5	600convolution	45291	40991	19	700fir	65937	61137
6	700convolution	55491	47691	20	800fir	77037	69737
7	800convolution	66191	54391	21	900fir	88137	78337
8	900convolution	76391	61091	22	1000fir	99237	86937
9	1000convolution	87091	67791	23	lms	14536	14536
10	n-real-updates	16738	16738	24	600lms	89636	79536
11	300n-real-updates	56538	47338	25	700lms	112636	92536
12	400n-real-updates	92238	62638	26	800lms	135636	105536
13	500n-real-updates	127538	77938	27	900lms	158636	118536
14	dot-product	750	750	28	1000lms	181636	131536

Chapter 6

Preemption Delay Analysis

In this chapter, data cache analysis techniques for systems with multiple tasks that execute in a prioritized, preemptive manner are presented. Here, all tasks are assumed to be *completely preemptive*.

In a prioritized, preemptive real-time system, every task is assigned a *priority*. At any given time, the task with the highest priority must be scheduled for execution. An implication of this requirement is that, when a certain task becomes ready for execution, it preempts the execution of the currently executing task if it has a higher priority.

When a task is preempted, a subset of its memory lines may be evicted from the cache due to execution of tasks with higher priority. Consequently, when the preempted task resumes its execution at a later point in time, it may incur additional delay in order to reload the evicted cache lines. This additional delay is termed *Cache Related Preemption Delay* (CRPD). Assuming that *all* the cache lines loaded by a preempted task are evicted by some task with a higher priority, although safe, is a very pessimistic assumption.

The fundamental steps involved in this calculation of a safe and tight estimate of the CRPD of a task are given below.

1. **Preemption delay:** Calculation of the delay incurred due to a certain preemption, given the preempted task, the set of possible preempting tasks and the time that preemption occurs.

2. **Number of preemptions:** Calculation of n_p, an upper bound on the number of times a task can be preempted during its execution in the context of a task set.

3. **Worst-case scenario:** Identification of the placement of the n_p preemption points in the iteration space of the preempted task such that the *worst* possible preemption delay is obtained.

The rest of this chapter focuses on analysis techniques for *data caches*. The same issues have been studied by Staschulat et al. for *instruction* caches [22], [23]. The work presented here is orthogonal to the work by Staschulat et al.

6.1 Methodology

In this section, a technique to calculate a *safe and tight* estimate of the data-cache related preemption delay (D-CRPD) for a task and, hence, estimates of the WCET and response time of the task, is presented. The method incorporates D-CRPD calculations into the WCET calculation itself. Furthermore, to make the preemption delay calculation as accurate as possible, the intersection of the set of cache lines that are useful to the preempted task on resuming execution and the set of cache lines that are potentially used by higher-priority tasks in the meantime is calculated.

The calculation of worst-case execution time with data-cache related preemption delay is performed in two phases. In the first phase, base execution times are calculated for every task in the system, independent of other tasks. In the second phase, preemption delay is calculated and added to the worst-case execution time of a task.

6.1.1 Phase 1: Calculation of Base Time and Data Cache Patterns

In order to calculate the base execution times for tasks, every task is analyzed individually using the data cache analyzer described in Chapter 4. This produces data cache reference patterns for each task, which are then used by the static timing analyzer to build **timing trees** for every task. The timing tree provides information about the timing of individual nodes (functions/loops) in a given task. It is to be noted that this base time does not include the D-CRPD and is calculated only *once* for every task.

6.1.2 Phase 2: Preemption Delay Calculation

In this phase, the data cache analyzer and the timing analyzer interact repeatedly to calculate the WCET of a task with D-CRPD included. The timing analyzer calculates

the execution time of the given program up to a preemption point. At this point, the data cache analyzer is invoked to calculate the number of additional data cache misses due to the preemption. The delay due to these misses is added to the remaining execution time of the task under consideration and the process is repeated for subsequent preemptions.

The data cache analyzer presented in Chapter 4 is capable of producing miss/hit patterns for every task. These patterns indicate not only the number of data cache misses, but also their positions. In Chapter 4, the analysis is for a single task and information on the *number* of misses is sufficient. In the current section, a method to use information about the positions of these misses to calculate the preemption delay at a given point is presented.

All data cache reference patterns for a task are merged maintaining the order of memory accesses. References that access the same cache line are connected together to form a chain that effectively indicates cache reuse. All chains for a task together form its *access chains*. An example with just three cache line chains is shown in Figure 6.1. Chains for different cache lines are shown using a different line style. Within each chain, a *miss* is represented by the letter 'M' and a *hit* is represented by a dot.

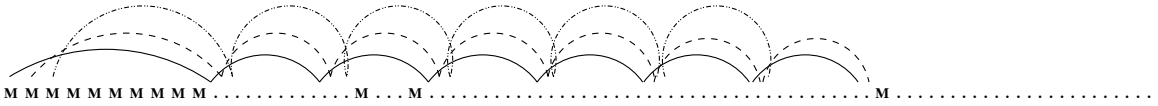


Figure 6.1: Access Chains for Cache Lines 1, 2 and 3

Each point in the access chains of a task is assigned a weight that indicates the number of *additional data cache misses* that would be incurred if a preemption were to occur just after that point. This weight is calculated as follows. First, the number of different cache-line chains that *cross* this point are counted. This effectively eliminates cache lines that are not used after the access point under consideration. Additionally, two checks are performed.

1. Chains in which the next access point on the chain is a *miss* in the pattern are not counted. The rationale behind this is as follows. If the point were a *miss* in the first place, it would be due to some intra-task interference. Hence, a preemption before that point will not cause any further delay with respect to the cache line that the chain represents.

2. Chains that correspond to cache lines that are not used by any higher-priority task are

not counted. This ensures that only the cache lines that could potentially be replaced after preempting the current task are considered.

Access chains for a task need to be constructed only **once**. However, since the assignment of weights for every point is dependent on higher-priority tasks in the task set, the weights are specific to a *job* in the context of a *task set* rather than a *task*.

6.2 An Upper Bound On Preemptions

In this section, a formula to calculate a guaranteed upper bound on the number of preemptions for a task is first presented. Next, the worst-case placement of these points in the iteration space of the preempted task is identified.

Consider a task T_i. An upper bound on the number of preemptions incurred by T_i is given by Equation 6.1.

$$n_p^i = \sum_{j \in hp(i)} (\lceil \frac{D_i}{P_j} \rceil) \tag{6.1}$$

Once again, the set $hp(i)$ denotes the set of tasks with a higher priority than T_i. Access chains are constructed as described in Section 6.1.2. From these chains, the sum of the n_p^i most expensive delays is calculated. This forms an *upper bound* on the worst-case preemption delay of the task under consideration. The preemption delay thus calculated is added to the WCET of the task. This method for calculating the number of preemptions is denoted as HJ-P in all experimental results presented later.

Equation 5.1 is used in this section to calculate the response time of a task. However, this section assumes that task do not have any critical sections. Hence, the blocking time factor is eliminated. In order to calculate the response time of a particular task, the WCET, with D-CRPD included, of every higher-priority task is required. Hence, the process starts with the calculation of the response time of the task with the highest priority and proceeds towards the task with the lowest priority. For every task, the D-CRPD is calculated using the method described above. The resulting execution time is used in Equation 5.1.

6.3 Preemption Delay Costs

As shown in Section 6.2, an upper bound on the worst-case preemption delay for a task may be obtained by adding together, the n_p most expensive preemption delays. In this process, no constraints are imposed on the interval between consecutive preemption points.

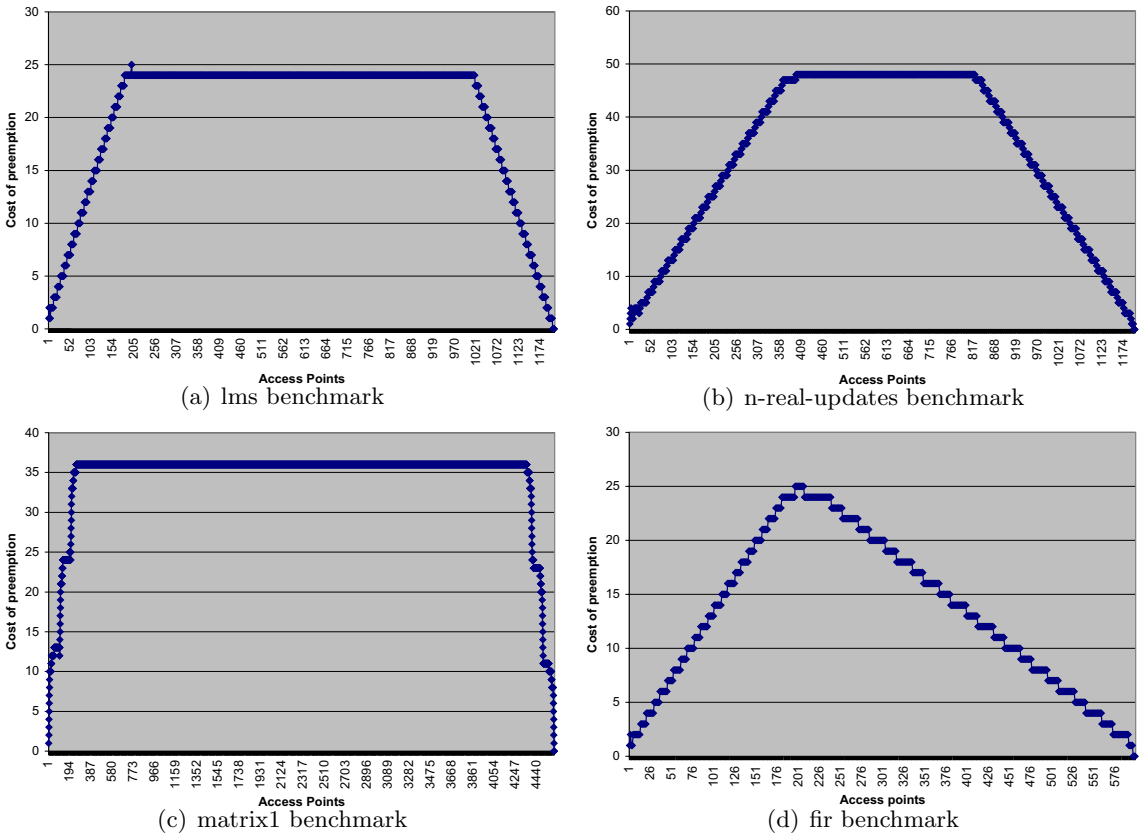
Table 6.1: Example Task Set Characteristics - Task Set 1

Benchmark	**Period (=deadline)**	**Stand alone WCET**
convolution	62500	7491
fir	125000	9537
lms	125000	14536
n-real-updates	250000	16738
matrix1	250000	54168

The reason for using this upper bound is an observation regarding the reuse of cache lines in a task. Consider a task set with characteristics as shown in Table 6.1. The distribution of preemption costs for the second, third, fourth and fifth tasks of this task set are shown in Figure 6.2. The X-axis shows the access space in order of memory accesses and the Y-axis shows the cost of preemption at a point in terms of the calculated *weight* at the point. The calculation of the weight at a given iteration point is decribed in Section 6.1.2.

From these graphs, it may be observed that, for the benchmarks in Figures 6.2(a), 6.2(b) and 6.2(c), a large number of access points have the highest preemption cost. Furthermore, they are all consecutive, indicating that a preemption at any of these points would result in the same preemption delay. Hence, picking the n_p most expensive points gives a reasonably tight bound on the worst-case preemption delay.

The distribution of these delays is a direct indication of the reuse of cache lines in the respective tasks. In most programs, ninety percent of the time is spent in ten percent of the code. Within this ten percent, there are repetitive reuse patterns, which implies temporal and spatial reuse. Hence, during the execution of this portion of the program, all data that is used in the portion is already in the data cache. Preemption at any point in this portion would result in more or less the same cache lines being evicted, hence causing the same preemption delay.



(a) lms benchmark (b) n-real-updates benchmark

(c) matrix1 benchmark (d) fir benchmark

Figure 6.2: Distribution of Preemption Costs Across the Iteration Space

Although the above behavior is observed in some cases, there are some benchmarks (Figure 6.2(d)) in which a gradual increase is observed in the preemption cost up to some point and then a decrease in the cost for successive access points. Hence, in the next section, methods to tighten the worst-case preemption delay bound are presented.

6.4 Tightening Preemption Delay Bounds

In Section 6.2, a method is described to calculate the WCET of a task with D-CRPD. However, in that section, simplified methods are used to calculate the maximum number of preemptions and the total preemption delay incurred. In this section, methods to calculate significantly tighter estimates of the number of preemptions and a more realistic, yet safe method to identify the worst-case placement of these preemption points are presented.

6.4.1 Preemption Delay Affects Critical Instant

Real-time systems theory often assumes that, in the case of a fixed-priority scheduling scheme, the worst-case response times for all tasks in a task set are exhibited when all tasks are released simultaneously. This instant of release of all tasks is known as the *critical instant* of the task set. While this is true in general, it does not necessarily hold true when preemption costs are taken into account.

Table 6.2: Task Set, Optional Phasing

	Φ	P	C	Δ
T_0	1	3	1	0
T_1	0.875	15	4.625	0.125
T_2	0.125	20	2.25	0.75
T_3	0	25	1	0.125

Consider the task set with characteristics shown in Table 6.2. The first column indicates the task ID, the second column indicates the phase Φ, the third column indicates the period P, the fourth column indicates the WCET C and the fifth column indicates the preemption delay Δ, respectively, for tasks T_0 to T_3. The scheduling policy used in this example is the static Rate Monotone (RM) policy.

Figure 6.3 depicts the schedule obtained for this task set when all tasks are released simultaneously (ignoring phase Φ_i). In contrast, Figure 6.4 depicts the schedule obtained when tasks are released according to the phasing indicated in the second column of Table 6.2. In both these figures, the hashed rectangles indicate normal execution of task and the black rectangles indicate preemption delay. It may be observed that, in Figure 6.4, the response time of T_3 (15 units) exceeds its response time in Figure 6.3 (14 units). This example demonstrates the fact that, when considering preemption costs, it is not safe to assume that the critical instant occurs when all tasks are released simulataneously. Furthermore, it is to be noted that there appears to be no straightforward method to *calculate* the critical instant for a task set in such a context.

6.4.2 Eliminating Infeasible Preemption Points

The formula used to calculate an upper bound on the number of preemptions for a task in Section 6.2 is based on the number of jobs of higher priority tasks that are released

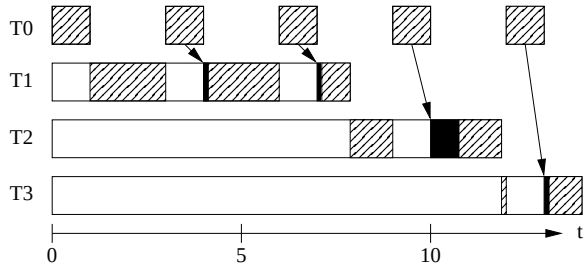


Figure 6.3: Preemption with Synchronous Release

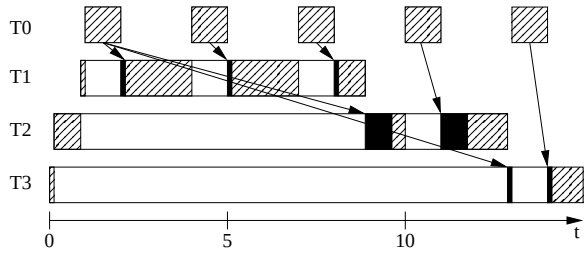


Figure 6.4: Preemption with Φ Phasing

in the period of the lower priority task. This leads to the consideration of several infeasible preemption points either because the lower priority job has not been scheduled at all and, hence, cannot be preempted, or because it has already finished executing. In this section, a method that considers the best and worst case execution times of higher priority jobs to eliminate these infeasible points is presented. As demonstrated in Section 6.4.1, the critical instant does not necessarily occur when all tasks are released at the same time. Furthermore, it is not straightforward to calculate the critical instant for a task set. Hence, WCET calculations are performed for *every job* in the hyperperiod of a task set, for a given phasing. The response time for each job is calculated using Equation 5.2, after eliminating the term for blocking time since tasks are assumed not to have critical sections.

The method used to eliminate infeasible preemption points is described in the remainder of this section. However, for the sake of simplicity, it is assumed that the preemption delays for all preemptions are zero. The actual calculation of preemption delay and the identification of the placement of preemption points in the iteration space of the preempted task are discussed in the next section.

An example illustrating the methodology to eliminate infeasible preemption points is now presented. The characteristics of the example task set are shown in Table 6.3. The

hyperperiod of this task set is *200* and all jobs within this hyperperiod are considered in the analysis. Once again, the task set is scheduled using the RM scheduling policy.

Table 6.3: Example Task Set Characteristics - Task Set 2

Task	**Period = deadline**	**WCET**	**BCET**
T_0	20	7	5
T_1	50	12	10
T_2	200	30	25

For the purposes of this analysis, the construction of a timeline for every task indicating release points for all higher priority jobs is required. Each of these release points is a potential preemption point for the task under consideration and the goal is to eliminate the infeasible ones. Figures 6.5 and 6.6 show the time lines for tasks T_1 and T_2, respectively. The arrows represent job releases and are numbered consecutively. The preemption points that get eliminated as a result of this analysis are circled. Best-case execution times are laid out above the time axis and worst-case execution times, below the time axis. In this example, black rectangles are used for jobs of task T_0, gray rectangles for those of task T_1 and the red rectangles, for those of T_2.

First, consider the timeline for task T_1 (Figure 6.5). In order to determine whether release point 1 is a feasible preemption point for $J_{1,0}$, two checks are performed. First, it is determined whether $J_{1,0}$ can get scheduled in the previous interval, *i.e.*, between points 0 and 1. Secondly, it is determined whether any execution of $J_{1,0}$ remains beyond point 1. For the first condition, the BCETs of all higher priority jobs (in this examples, $J_{0,0}$) are used. Since there is idle time after placing the BCET of $J_{0,0}$ (5 units), it is determined that $J_{1,0}$ could be scheduled before point 1. To check if any execution of $J_{1,0}$ remains beyond point 1, the sum of the WCETs of $J_{0,0}$ and $J_{1,0}$, 7 and 12 units respectively, are used. Since this does not exceed point 1, $J_{1,0}$ is guaranteed to finish within the current interval. Hence, it is concluded that no preemptions are possible for $J_{1,0}$.

Second, proceed to the next release of T_1, *i.e.*, $J_{1,1}$. During the interval between release points 3 and 4, in the best case, no higher priority job needs to be scheduled. Hence, $J_{1,1}$ can be scheduled in this interval. Next, it is determined that, in the worst case, $J_{1,1}$ is not guaranteed to finish before point 4. This leads to the conclusion that point 4 is a feasible preemption point for $J_{1,1}$. Proceeding this way, the maximum number of preemptions for

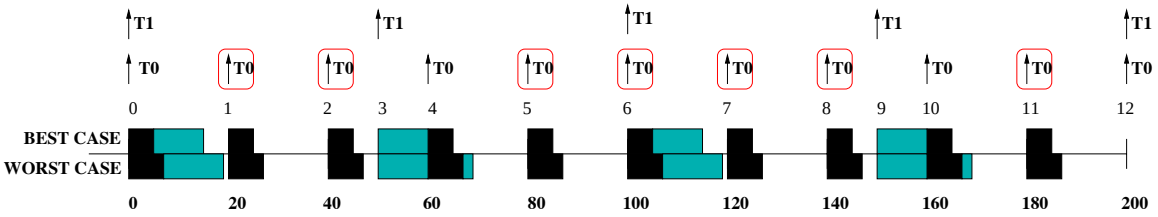


Figure 6.5: Timeline for Task T_1

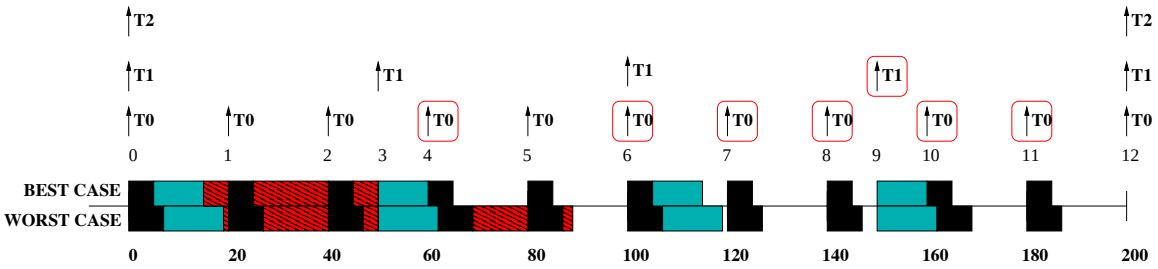


Figure 6.6: Timeline for Task T_2

$J_{1,1}$ is determined to be 1. The analysis is repeated for further releases of T_1 within the hyperperiod.

In the above example, it may be observed that the second release of T_1, namely $J_{1,1}$, has a higher number of preemptions than $J_{1,0}$, hence creating the possibility of a worse response time for $J_{1,1}$ by the addition of the preemption delay. Once again, this proves the claim that the critical instant does not necessarily occur when all tasks are released simultaneously.

The maximum number of preemptions for releases of task T_2 is calculated using the same analysis. The timeline for T_2 is shown in Figure 6.6. For this task, two higher priority tasks need to be considered, namely T_0 and T_1. Starting with $J_{2,0}$, it is determined that release point 1 is a feasible preemption point since $J_{2,0}$ can be scheduled before point 1 and is not guaranteed to finish before point 1. Similarly, it is determined that points 2 and 3 are feasible. However, when the interval between points 3 and 4 is considered, it is observed that, even in the best case, the execution of $J_{1,1}$ occupies the entire interval. Hence, there is no possibility of $J_{2,0}$ being scheduled in this interval. This leads to the elimination of point 4 as a feasible preemption point for $J_{2,0}$. In the worst-case, $J_{2,0}$ finishes execution at time 82. Hence, points 6, 7, 8, 9, 10 and 11 are also eliminated. At the end of the hyperperiod, the analysis determines that the maximum number of preemptions for $J_{2,0}$ is 4. On the

other hand, the method described in Section 6.2 produces a bound of 9 for the same job.

In summary, the method is as follows. Consider a set of tasks T_0, ..., T_n. Let $J_{i,0}$, ..., $J_{i,k}$ represent the jobs of task T_i. Assume that task T_0 has the highest priority and that task T_n has the lowest priority using a static priority scheme.

A timeline between 0 and the hyperperiod of the task set is constructed for every task T_i, and the releases of all higher priority jobs are marked on this timeline. The feasibility of a release point (say x) as a preemption point for $J_{i,j}$ is determined by performing two checks for the interval between release points x and $x-1$. First, it is checked whether $J_{i,j}$ has a possibility of being scheduled in this interval based on the BCETs of higher priority jobs. If not, point x is not a feasible preemption point. If yes, a further check if performed to determine if any portion of execution of $J_{i,j}$ remains beyond point x. If yes, point x is a feasible preemption point. For this, the WCETs of all jobs executing in this interval, including $J_{i,j}$, are considered in order of priority.

The above calculations are repeated for every interval between potential preemption points for $J_{i,j}$ until it is guaranteed to finish. *This analysis is performed for every job in the hyperperiod of the task set.*

6.4.3 Extension to a Dynamic Scheduling Policy

The analysis presented in Section 6.4 assumes a static priority scheme. The analysis may be extended to support dynamic priority schemes as follows. Instead of calculating priorities in the beginning of the analysis and assuming that they never change, job priorities are recalculated at the beginning of every interval between consecutive preemption points. This introduces the flexibility of being able to use different scheduling policies. The current implementation of this methodology supports the static Rate Monotone (RM) policy and the dynamic Earliest Deadline First (EDF) policy.

6.4.4 Calculation of the Preemption Delay

In Section 6.4.2, the method to eliminate infeasible preemption points is described in isolation, without details about calculation of the preemption delay incurred at every feasible point. In this section, the calculation of preemption delay at every point is described. Every preemption point determined to be feasible for a task is a point in time. This point in time needs to be translated into an execution point in the iteration space of the preempted

task. The preemption delay at this point may then be calculated using the access chains of the preempted task as explained in Section 6.1.2.

The static timing analyzer framework described in Section 2.2.2 provides best-case and worst-case execution time estimates for a program. Furthermore, given a certain interval of time, it can provide information about the iteration point at which the program could be at the end of the interval both in the best and the worst-case scenarios.

To explain the above with an example, consider the task set whose characteristics are shown in Table 6.3. On the timeline for task T_2 shown in Figure 6.6, point 1 is identified to be a feasible preemption point. In order to obtain the additional delay due to preemption at point 1, the iteration point reached by $J_{2,0}$ at the time of preemption must be identified.

Using the feasible preemption point analysis described in Section 6.4.2, the minimum and maximum execution times available for a task in every interval between preemption points is calculated. In the current example, the BCETs and WCETs of jobs $J_{0,0}$ and $J_{1,0}$ are used. In the first interval, after subtracting the time required for the higher priority jobs, 5 units of time remain in the best case and 1 unit of time remains in the worst case, for the execution of $J_{2,0}$. These provide an upper and lower bound, respectively, for the time available for $J_{2,0}$ in the interval.

The upper and lower bounds thus identified are each supplied as inputs to the static timing analyzer framework. The framework performs best and worst-case timing analysis of the preempted task to produce, for each input, two iteration points. One iteration point represents the latest possible point that could be reached (*i.e.*, cannot be exceeded) by the preempted task in the given time and is obtained from best-case timing analysis of the task. The second iteration point represents the earliest iteration point that is guaranteed to be reached in the given time and is obtained from worst-case timing analysis of the preempted task.

Considering the earliest and latest iteration points among the four iteration points obtained provides the range of iteration points that the preempted task could have reached when it is preempted. Now, the preemption delay at each point in this range is calculated and the maximum delay among those is considered to be the worst-case preemption delay due to preemption at the particular point. This delay is added to the remaining worst-case execution time of the task.

Upon resumption of the preempted task, execution should continue from the iteration where it had left off. However, this is imprecise since it is not known at what points the

preemption delay is actually incurred during the execution of the task. Hence, for future preemption points, determination of the iteration range where the task is supposed to be when it is preempted is not guaranteed.

In order to solve the above problem and provide safe estimates of the worst-case preemption delay at every point, the following solution is devised. When a task is preempted, the delay is calculated as indicated above. When the task later resumes execution, it enters a *preemption delay* phase for a time equal to the calculated delay. In this phase, the task prefetches all data cache items that contribute to the delay. Once done, the task resumes normal execution. If a task gets preempted during its preemption delay phase, it pessimistically starts the same preemption delay phase all over again once it resumes execution. This new phase ensures that all future delay calculations are accurate.

In the current example, assume that T_2 is a program that has a loop with 100 iterations. Using the lower bound of the time available for its execution (1 unit), assume the static timing analyzer determines that $J_{2,0}$ is guaranteed to reach at least iteration 4 and cannot proceed beyond iteration 7. Similarly, assume it determines that $J_{2,0}$ is guaranteed to reach at least iteration 9 in 5 units of time (upper bound of the time available for its execution) and does not proceed beyond iteration 13 in the same amount of time. Now, the delays for each iteration point in the entire range between 4 and 13 are calculated. Among these, the highest delay is chosen and added to the remaining WCET of $J_{2,0}$.

The algorithm used to implement the above method of calculating worst-case preemption delay at a given preemption point is summarized below. The static timing analyzer framework is invoked to perform worst-case partial timing on the minimum available execution time for the preempted task. This yields the beginning of the range of iteration points to be considered. Next, the timing analyzer is invoked to perform best-case partial timing on the maximum available execution time for the preempted task. This yields the end of the range of iteration points. The range thus identified is provided to a function that iterates through the access chains of the preempted task and calculates the highest delay in the given range.

In the method described thus far, it is assumed that, for every task, the values of its period, deadline and *phase* are known *a-priori*. For a given phasing of tasks, the method presented in Section 6.4.2 calculates the worst-case response times for all tasks. Instead, a modification to the method to calculate the worst-case response times for tasks regardless of the phasing of the tasks is now presented.

In the algorithm described above, for every preemption point, a *range* of iteration points where the preempted task could be when it is preempted is calculated. The maximum preemption delay in this range is assumed to be the preemption delay at the preemption point.

Instead, it is now assumed that the maximum delay in the *entire* iteration space is incurred at *every* preemption point. Further, in order to allow *any* phasing among tasks, extra preemptions are added for every job. Assume that the maximum possible phasing for any task is x units. Furthermore, for any given task, the maximum phasing is less than or equal to its own period. Under these conditions, Equation 6.2 gives the number of extra preemptions to add in case of a static-priority policy, and Equation 6.3 gives the number of extra preemptions in the case of a dynamic-priority scheduling policy.

$$pextra_x^{i,j} = \sum_{hp=1}^{i-1} (\lceil \frac{min(x, P_i)}{P_{hp}} \rceil) \tag{6.2}$$

$$pextra_x^{i,j} = \sum_{hp=1}^{n} (\left\{ \begin{array}{ll} \lceil \frac{min(x,P_i)}{P_{hp}} \rceil) & , if\ hp \neq i \\ 0 & , otherwise \end{array} \right\}) \tag{6.3}$$

However, it may be observed that, every time a lower-priority task gets preempted, at least one higher-priority task executes. The minimum amount of execution time of this higher-priority task may be safely assumed to be at least equal to the shortest best-case execution time (BCET) among all higher-priority tasks. Hence, the shortest BCET is subtracted from the maximum possible phase of the lower-priority task before adding any more extra preemptions, thereby tightening the bound on the number of extra preemptions. This calculation effectively produces an upper bound on the number of preemptions and on the worst case response times for a maximum phasing of x.

Consider the example used earlier in Section 6.4.1. The task-set characteristics for this example are shown in Table 6.2. Figure 6.3 shows response times of tasks when all tasks are released simultaneously. In order to calculate the response time of a task with a maximum phasing of 1 unit, this scenario is used. In accordance with Equation 6.2, one extra preemption needs to be added to each task. For example, for task T_1, the response time is calculated assuming three preemptions instead of two and considering the maximum preemption delay at each of these points. In this example, since it is assumed that a task has a constant preemption delay, Δ, for any preemption, that value is used as the maximum

preemption delay and a response time of 8 units is obtained instead of 7.875 units.

When a preemption takes place, it results in a context-switch at the operating system level. The operating system code that is executed in order to perform the context-switch may also use the data cache and, hence, alter the results of the feasible preemption point analysis. In the current experiments, this factor has not been considered. However, conceptually, this issue may be dealt with in the following manner.

First, the data cache lines that are used by the operating system code are identified. Since this code may not adhere to the constraints that the data cache analysis framework poses on the programs, it cannot be used. Hence, a predetermined area in the memory is allocated to hold the operating system code, thereby constraining the data cache lines that it may use.

Second, the effects that the execution of operating system code would have on the tasks being analyzed need to be considered. For this purpose, while calculating the preemption delay incurred by a task at a given preemption point using access chains, the cache lines allocated for the operating system code qre considered as potential candidates for eviction. In other words, the operating system code would be treated as the highest priority task in the system and would execute every time there is a preemption.

6.4.5 Analysis Algorithm

An algorithm briefly describing the methodology is shown in Figures 6.8 to 6.10. The implementation of the system uses an event hierarchy. Every event has a handler that performs all operations necessary on the occurrence of the particular event. Several event types exist, each with a priority. Every event has a type, a time of occurrence and information about the task and job that the event corresponds to. The events are ordered by time and, upon ties, by priority based on the type of event. The various events in the system, in order of priority, are BCEndExec, WCEndExec, BCPreemption WCPreemption, DeadlineCheck, JobRelease, BCStartExec, WCStartExec and DCacheRelatedDelayPhaseEnd. The algorithm in Figures 6.8 to 6.10 describes the actions that take place when a certain event is triggered. In the algorithm, the events are described in an order that follows the flow of the logic rather than priority.

The basic flow of operations in the analysis is as follows. Stand-alone WCETs and BCETs are calculated for each region of each task. JobRelease and DeadlineCheck events

are pre-created based on task periods and inserted into a global event list. Events in the event list are handled one at a time until there are no more events. The basic life-cycle of a job is described below. Upon release of a job, it is first determined when that job gets scheduled and whether any job that is currently executing gets preempted due to this release. This triggers a B/WCStartExec event which signifies the start of execution of a job. If a preemption occurs due to the job release, B/WCPreemption events are also triggered. The B/WCStartExec events schedule B/WCEndExec events or DCacheRelatedDelayPhaseEnd events as the case may be. Finally, a DeadlineCheck event is triggered and is responsible for checking if a certain job misses its deadline.

The creation dependencies between event types are represented by the state-transition diagram shown in Figure 6.7. An arrow from one event type to another indicates that the handler of the first event type may create an event of the second type. Events that do not have a creator in the diagram are created at the beginning outside any of the event types.

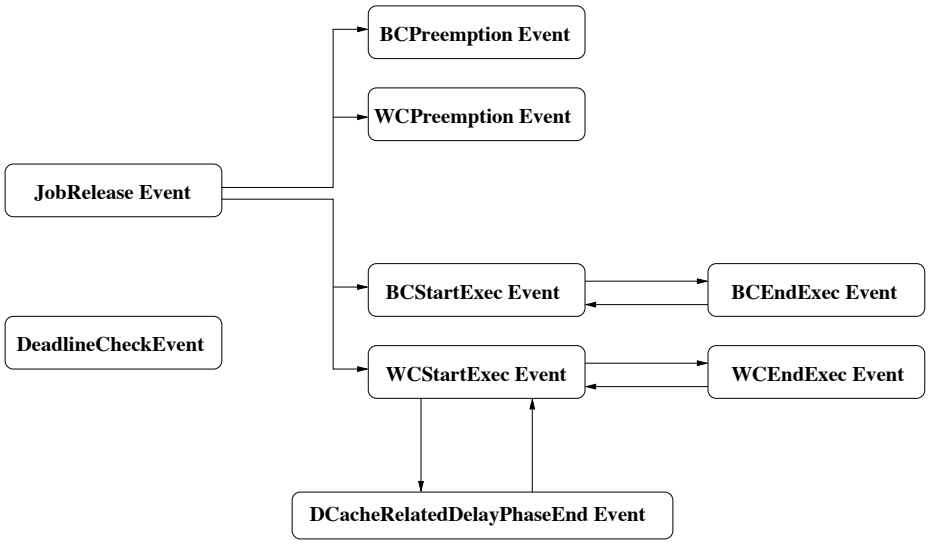


Figure 6.7: Creation Dependencies among Event Types

Structures global to all events are described below
bc_service_queue, wc_service_queue : queues of all released jobs that have not yet completed
Possible status : READY and IN_SERVICE
event_list : list of events ordered first by time and then by the priority of the type of event
Parameters for each event are described below
current_time : Time at which event occurs, *curr_job* : Job which the event corresponds to
Functions used in event handlers :
insertIntoB/WCServiceQueue(job, status)
inserts job with given status into priority-ordered service queue
removeFromB/WServiceQueue(job)
removes given job from service queue
insertIntoEventList(ev_job, ev_time, ev_type)
inserts an event of type ev_type into the event list for time ev_time and job ev_job
removeFromEventList(ev_job, ev_type)
removes events of ev_type corresponding to ev_job

JobRelease event: This event represents the release of a new job of a task
```
if (event_queue is empty) {
  insertIntoEventList(curr_job, current_time, B/WCStartExec)
} else {
  q_job ← head of b/wc_service_queue
  if (curr_job has higher priority than q_job) {
    insertIntoEventList(curr_job, current_time, B/WCStartExec)
    if (status of q_job is IN_SERVICE)
      insertIntoEventList(q_job, current_time, B/WCPreemption)
    else
      removeFromEventList(q_job, B/WCStartExec)
  }
}
insertIntoB/WCServiceQueue(curr_job, READY)
```

BCPreemption event : This event represents the preemption of a task in the best case
```
removeFromEventList(curr_job, BCEndExec)
update best case remaining time for curr_job
```

Figure 6.8: Algorithm for Calculation of WCET w/ Delay

WCPreemption event : This event represents the preemption of a task in the worst case
```
if (curr_job in DCache Related Delay Phase) {
  removeFromEventList(curr_job, DCacheRelatedDelayPhaseEnd)
} else {
  removeFromEventList(curr_job, WCEndExec)
  calculate preemption delay
}
update worst case remaining time for curr_job
```

BCStartExec event : This event represents the best-case start of execution
of the current job
```
set status of curr_job to IN_SERVICE in bc_service_queue
insertIntoEventList(curr_job, current_time + best-case remaining time of curr_job, BCEndExec)
```

B/WCEndExec event : This event represents the best/worst-case end of execution
of the current job
```
removeFromB/WCServiceQueue(curr_job)
update best/worst case remaining time for curr_job
if (b/wc_service_queue has more jobs in it) ) {
  retval ← first READY job ready_job in b/wc_service_queue
  if (retval == TRUE) {
    insertIntoEventList(ready_job, current_time, B/WCStartExec)
  }
}
```

WCStartExec event : This event represents the worst-case start of execution
of the current job
```
set status of curr_job to IN_SERVICE in wc_service_queue
if (curr_job in DCache Related Delay Phase) {
  insertIntoEventList(curr_job, current_time + preemption delay,
    DCacheRelatedDelayPhaseEnd)
} else {
  insertIntoEventList(curr_job, current_time + worst-case remaining time of curr_job, WCEndExec)
}
```

DeadlineCheck event : This event checks whether the given job has
exceeded its deadline
If yes, all associated structures are deleted from the event list and the bc and wc service queues

Figure 6.9: Algorithm (cont.) for Calculation of WCET w/ Delay

DCacheRelatedDelayPhaseEnd event : This event represents the end of the dcache related
  delay phase for the current job
update worst case remaining time for curr_job
insertIntoEventList(curr_job, current_time, WCStartExec)

Main Algorithm : This is the starting point of the analysis
for every task in the task-set {
    create JobRelease events for all jobs of task
    create DeadlineCheck events for all jobs of task
}
while (events in event list) {
  get highest priority event and handle it based on event type
}

Figure 6.10: Algorithm (cont.) for Calculation of WCET w/ Delay

6.4.6 Correctness of Analysis

The algorithm described in Section 6.4.5 calculates the worst-case response time of a job in the context of a given task set. Equation 5.2 shows the calculation of the worst-case response time of a job $J_{i,j}$. In the context of fully preemptive tasks, the response time of a job is the sum of three components, namely the base WCET of the task, the execution time of higher-priority jobs and the data-cache related delay incurred due to preemption by higher-priority jobs. The blocking time term in Equation 5.2 is not used. The formulation for each of these components for a job $J_{i,j}$ and a proof of their correctness are presented in this section.

Explanation of the new symbols used in the formulation is as follows. nr_i represents the number of regions within a task T_i. $C^r_{i,j}$ represents the WCET of region r of a job $J_{i,j}$. An added superscript of rem represents remaining WCET of job $J_{i,j}$ at a given time and a subscript of *base* represents its base WCET. $rel_{i,j}$ represents the time of release of job $J_{i,j}$, calculated as $(\phi_{i,j} + (j-1) \cdot P_i)$. I represents an interval between two consecutive releases of higher-priority jobs for $J_{i,j}$ and t_I^{rem} represents the time remaining before the end of an interval I. $\Delta^I_{i,j}$ represents the data-cache related delay experienced by job $J_{i,j}$ due to preemption by the release of a higher-priority job at the end of an interval I. $hp(i,j)$ represents the set of jobs that have a higher priority than job $J_{i,j}$.

Theorem 6.4.1 *The response time of a job $J_{i,j}$, calculated as the sum of the values produced by Equations 6.4, 6.8 and 6.9, is a safe upper bound on the worst-case response time of $J_{i,j}$ in the context of fully preemptive tasks.*

The correctness of Theorem 6.4.1 is proved using Lemmas 6.4.2, 6.4.3 and 6.4.4.

Lemma 6.4.2 *An upper bound on the execution time of a job $J_{i,j}$, without considering the effects of interference from other jobs, is given by Equation 6.4.*

$$Base\ WCET\ of\ J_{i,j} = \sum_{r=1}^{nr_i} C_{i,j}^{base,r} \tag{6.4}$$

The calculation of the base WCET of a job is performed using the static timing analyzer. The correctness of the static timing analyzer and, hence, that of Lemma 6.4.2 is assumed in this dissertation (see [5, 6, 7, 8] for details).

The execution time of higher-priority jobs within the response time of $J_{i,j}$ is calculated by counting the number of instances of every higher-priority task that may execute within the response time of $J_{i,j}$ and multiplying it by the execution time of the specific higher-priority job. Calculation of the execution time of higher-priority jobs is shown in Equation 6.5.

$$hpex_{i,j} = \sum_{(k,l)\in hp(i,j)} (\lceil \frac{\Re_{i,j}}{P_k} \rceil \cdot \sum_{r=1}^{nr_k} C_{k,l}^r) \tag{6.5}$$

Since the algorithm described in Section 6.4.5 calculates response times for every *job* in the task set, the relative phasing between jobs is known. Using this information, the calculation in Equation 6.5 is tightened. After the release of $J_{i,j}$, the time during which no other higher-priority job is released may be calculated using information about relative phasing as shown in Equation 6.6. The execution time remaining after the release of $J_{i,j}$ for any higher-priority job released *before* $J_{i,j}$ is calculated as shown in Equation 6.7.

$$at_{i,j} = \min_{(k,l)\in hp(i,j)} [\max(\lceil \frac{rel_{i,j} - \phi_{k,l}}{P_k} \rceil \cdot P_k, 0) + \phi_{k,l} - rel_{i,j}] \tag{6.6}$$

$$rem_{reli,j} = \sum_{(k,l)\in hp(i,j), rel_{k,l} < rel_{i,j}} C_{k,l}^{rem} \tag{6.7}$$

The difference between the times calculated in Equations 6.6 and 6.7 gives the time for which $J_{i,j}$ may execute without being preempted. Equation 6.8 shows the calculation for the new, tighter estimate on the execution time of higher-priority jobs within the response time of $J_{i,j}$, performed in accordance with the algorithm described in Section 6.4.5.

$$hpex_{i,j} = \sum_{(k,l)\in hp(i,j)} (\lceil \frac{\Re_{i,j} - \max((at_{i,j} - rem_{reli,j}), 0)}{P_k} \rceil \cdot \sum_{r=1}^{nr_k} C_{k,l}^r) \tag{6.8}$$

Lemma 6.4.3 *An upper bound on the execution time of higher-priority jobs within the response time of a job $J_{i,j}$, in the context of fully preemptive tasks, is given by Equation 6.8.*

Proof Assume that $\max((at_{i,j} - rem_{reli,j}), 0)$ is not subtracted from the iterative portion of Equation 6.8. It means that this time can be stretched due to *execution of higher-priority jobs* in between. By definition of $(at_{i,j} - rem_{reli,j})$, *all higher-priority jobs* released before $J_{i,j}$ *have completed execution* and no higher-priority jobs have been released yet after $J_{i,j}$. Contradiction. Hence, $\max((at_{i,j} - rem_{reli,j}), 0)$ can be subtracted from the iterative portion of Equation 6.8 without jeopardizing safety of the analysis. ∎

Every release of a higher-priority job is a potential preemption point for $J_{i,j}$. Consider an interval between two such consecutive releases. According to the algorithm described in Section 6.4.5, the job release at the end of this interval can be a feasible preemption point for $J_{i,j}$ if a) there is a possibility that $J_{i,j}$ is scheduled in the interval and b) there is a possibility that $J_{i,j}$ has not completed execution before the end of the interval. These conditions are mathematically represented and the preemption delay is given by Equations 6.9.

$$\begin{aligned} cond_a &= \sum_{(k,l)\in hp(i,j)} c_{k,l}^{rem} < t_I^{rem} \\ cond_b &= \sum_{(k,l)\in hp(i,j)} C_{k,l}^{rem} + C_{i,j}^{rem} > t_I^{rem} \\ PD_{i,j} &= \sum \Delta_{i,j}^I, \forall I \; s.t. \; (cond_a \wedge cond_b) \end{aligned} \tag{6.9}$$

Lemma 6.4.4 *An upper bound on the data-cache related delay experienced by job $J_{i,j}$ due to preemptions by higher-priority jobs, in the context of fully preemptive tasks, is given by Equation 6.9.*

Proof Consider the interval between two *consecutive* preemption points, p_{-1} and p. Assume that jobs J_{0,k_0}, ..., J_{i,k_i} have been released at some prior point and have not yet completed execution at p_{-1}. Further, assume that J_{i,k_i} is the job for which an upper bound on the number of preemptions is to be calculated.

Let x be the length of the interval between preemption points p_{-1} and p. There are three cases to consider.

1: $\sum_{j=0}^{i-1} c_{j,k_j} < x$, $\sum_{j=0}^{i} C_{j,k_j} > x$. Assume J_{i,k_i} cannot be preempted at p, *i.e.*, it cannot be running at time p. However, $\exists_{j=0..i-1} e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1}+\sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1}+\sum_{j=0}^{i} e_{j,k_j} > p$, *i.e.*, J_{i,k_i} is running at p. Contradiction. Hence, p is a feasible preemption point.

2: $\sum_{j=0}^{i-1} c_{j,k_j} < x$, $\sum_{j=0}^{i} C_{j,k_j} < x$. Assume J_{i,k_i} can be preempted at p, *i.e.*, it may be running at time p. Hence, $\exists_{j=0..i-1} e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} > p$. However, $\sum_{j=0}^{i} C_{j,k_j} < x$ implies $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} < p$. Contradiction. Hence, J_{i,k_i} cannot be running at p, and p is not a feasible preemption point.

3: $\sum_{j=0}^{i-1} c_{j,k_j} > x$. Assume J_{i,k_i} can be preempted at p, *i.e.*, it may be running at time p. Hence, $\exists_{j=0..i-1} e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} > p$. However, $\sum_{j=0}^{i-1} c_{j,k_j} > x$ implies $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} > p$. Contradiction. Hence, J_{i,k_i} cannot be running at p, and p is not a feasible preemption point. ∎

Proof Assume that the sum of the values produced by Equations 6.4, 6.8 and 6.9 is not a safe upper bound on the worst-case response time of a job. This implies that the value produced by at least one of the equations *is an underestimation of the specific component* represented by the equation. Lemmas 6.4.2, 6.4.3 and 6.4.4 *demonstrate the correctness of each component* of the response time of a job as a *safe upper bound*. Contradiction. Hence, the sum of the values produced by Equations 6.4, 6.8 and 6.9 *is a safe upper bound* on the worst-case response time of the job. ∎

6.4.7 Complexity of Analysis

Static data cache analysis to produce data cache reference patterns is performed only once for each task. Here, the iteration space of a task is iterated through, hence making the time and space complexity proportional to the number of data references, n_d in the task, namely $O(n_d)$.

The complexity of the algorithm presented above is $O(n_J * n_T * n_d)$ where n_J is the number of job releases in the hyperperiod of the task set and n_T is the number of tasks. The algorithm iterates over every interval between job release points. This introduces a complexity of n_J. Within each interval, the algorithm iterates over currently active jobs in order of priority. Since systems where a task has a deadline less that or equal to its period are considered, there can be at most *one* job of a particular task active at any time. Hence, iterating over active jobs adds a complexity of n_T. Finally, at every identified preemption point, maximum possible delay incurred by the preempted task is calculated using its access chains and information about the range of iteration points at which the preempted task is determined to be when it is preempted. This introduces a complexity of n_d since n_d is the length of the access chain of a task. However, in reality, this range is usually much smaller than n_d for a given preemption point since it is limited by the largest interval between two consecutive potential preemption points for a task.

6.5 Experimental Results

In all experiments, task sets that have a base utilization (utilization without considering data cache related delays) of 0.5, 0.6, 0.7 and 0.8 are used. Task sets of different sizes (2, 4, 6, 8) are constructed for each of these utilizations. For a utilization of 0.8, a task set consisting of 10 tasks is also constructed. In all these task sets, a static priority scheme is assumed.

An upper bound on the number of preemptions, n_p, for a task is calculated using three different methods to provide a comparison.

1. n_p is determined by using Equation 6.1. This is denoted as **HJ-P** in the results.

2. n_p is calculated by considering indirect preemption effects as proposed by Staschulat et al. This method uses the periods and response times of tasks [23]. This is denoted as **Stas-R**.

3. n_p is calculated using the range of execution times of higher priority jobs as proposed in Section 6.4. This method uses the periods, WCETs and BCETs of tasks to calculate feasible preemption points. Two methods are used for the actual calculation of preemption delay as described in Section 6.4.4. The method using the maximum delay in a *range* of iteration points (where the task is determined to be) is denoted as **OurFP-RangeMax** and the one using the maximum delay in the *entire* iteration space is denoted as **OurFP-ItSpMax**.

In the current set of experiments, the maximum phasing for a task, used in Equations 6.2 and 6.3, is assumed to be 1000 cycles. Although all the methods calculate an upper bound on the number of preemptions for a task, the first two methods do not provide information about the *placement* of the preemption points. Hence, in these cases, the n_p largest delays for a task are considered in order to obtain its D-CRPD.

Several experiments are performed to demonstrate the working of the new methods (OurFP-RangeMax and OurFP-ItSpMax) in comparison to prior methods (HJ-P and Stas-R). The results of these experiments are now presented and discussed.

Table 6.4: Task Set Characteristics: Benchmark IDs and Periods [cycles] for Task Sets

# Tasks	**2**	**4**	**6**	**8**	**10**
U = 0.5					
IDs	16, 19	1, 15, 18, 22	23, 3, 6, 11, 19, 26	2, 3, 4, 11, 15, 18, 7, 27	
Periods	50K, 200K	50K, 400K, 500K, 100K	400K, 500K, 1000K, 1000K, 2000K	100K, 400K, 500K, 800K, 1000K, 2000K, 2000K, 4000K	
U = 0.6					
IDs	21, 27	1, 15, 8, 27	3, 4, 6, 11, 19, 26	2, 5, 6, 11, 15, 18, 7, 27	
Periods	300K, 500K	50K, 400K, 500K, 1000K	100K, 400K, 500K, 1000K, 1000K, 2000K	100K, 400K, 500K, 800K, 1000K, 2000K, 2000K, 4000K	
U = 0.7					
IDs	27, 21	16, 9, 7, 27	3, 17, 8, 7, 20, 27	3, 5, 20, 11, 15, 19, 8, 26	
Periods	300K, 500K	50K, 400K, 500K, 1000K	100K, 400K, 500K, 1000K, 1000K, 2000K	100K, 400K, 500K, 800K, 1000K, 2000K, 2000K, 4000K	
U = 0.8					
IDs	27, 26	28, 13, 27, 19	21, 8, 20, 13, 25, 19	8, 26, 20, 15, 9, 11, 8, 21	10, 8, 15, 9, 5, 11, 20, 27, 22, 17
Periods	300K, 500K	500K, 500K, 1000K, 2000K	400K, 500K, 500K, 1000K, 1000K, 2000K	400K, 500K, 800K, 800K, 1000K, 2000K, 2000K, 4000K	100K, 625K, 625K, 625K, 1000K, 1000K, 1250K, 1250K, 2500K, 5000K

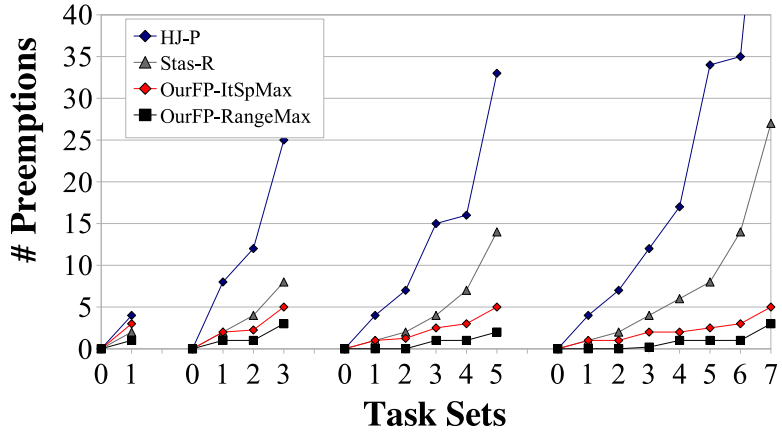
6.5.1 Response Time Analysis

In the first set of experiments, the task sets described in Table 6.4 are used. Response time analysis is performed using all the methods described in Section 6.5 for calculating the number of preemptions. Results obtained for the task sets with base utilization of 0.5 and 0.8 are shown in Figures 6.11 and 6.12 respectively. Results for utilizations of 0.6 and 0.7 exhibit similar trends and are hence omitted. Different graphs are used to present upper bounds on the three metrics studied, namely the number of preemptions, the WCET with preemption delay and the response times for tasks. In each graph, the x-axis represents the several task sets used. Tasks within each task set are numbered in decreasing order of priority.

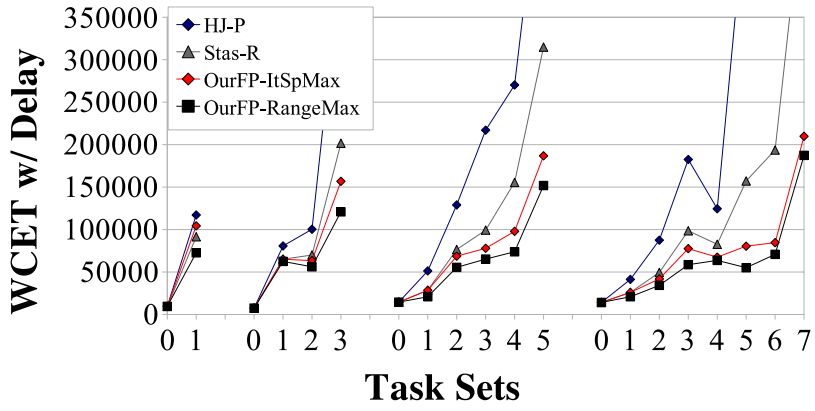
The method that uses only *feasible* preemption points consistently derives a much tighter bound on the number of preemptions for a given task compared to the two prior methods (HJ-P and Stas-R). Since the number of preemption points identified is smaller, bounds for the WCET with preemption delay and the response time for each task are also significantly tighter, as indicated in the results. Even the method that calculates an upper bound on the number of feasible preemption points for any possible phasing of tasks up to a given maximum (OurFP-ItSpMax) provides significantly tighter bounds than the two prior methods.

It may be observed from the graphs that, for some of the tasks, response times are not indicated for the first two methods of comparison (HJ-P and Stas-R). This means that the response time of the task exceeded its deadline, making the task set unschedulable. Results from the new methods (OurFP-RangeMax and OurFP-ItSpMax) show that these task sets are, in reality, schedulable. This underlines the potential benefits of the feasible preemption analysis techniques. For calculation of the response time, a fixed-point approach is used. Calculations are performed only as long as the deadline of the given task is not exceeded. In the Stas-R method, the value of the response time obtained in one iteration of this fixed-point approach is used to calculate the number of preemptions in the next iteration. Hence, if the response time of a task exceeds its deadline, the iterative calculation is stopped and the number of preemptions is not reported.

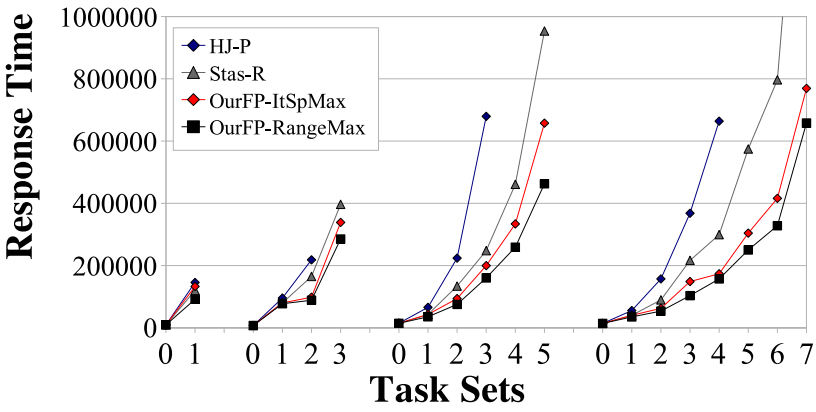
Widening gaps between results using the new methods (OurFP-RangeMax and OurFP-ItSpMax) and using the two prior methods (HJ-P and Stas-R) show an increase in the effectiveness of the new methods, especially for lower-priority tasks. Since tasks with a



(a) # Preemptions for U = 0.5

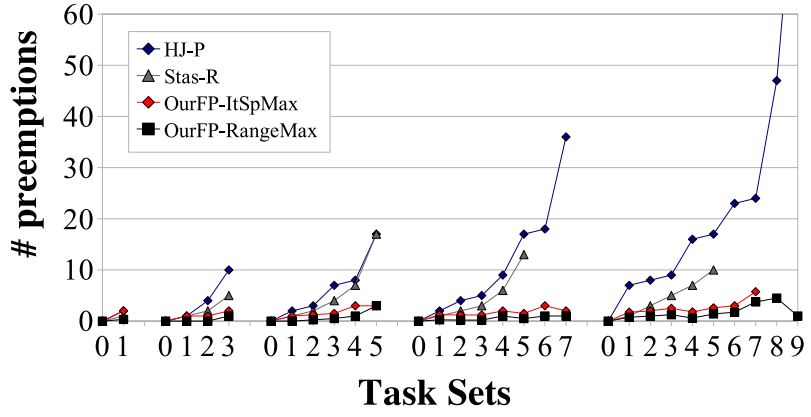


(b) WCET w/ Delay for U = 0.5

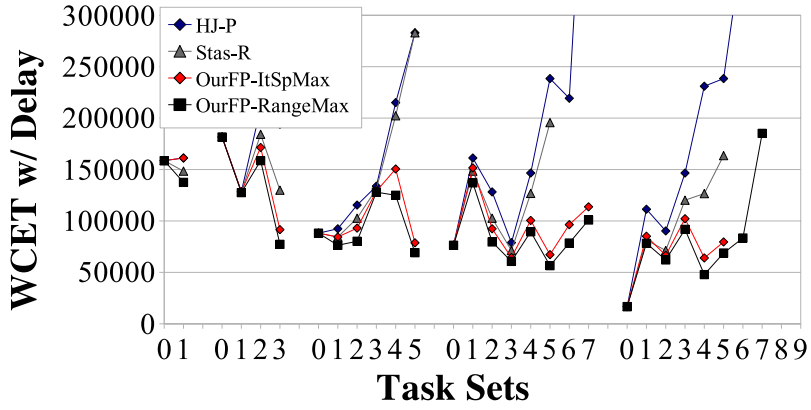


(c) Response Time for U = 0.5

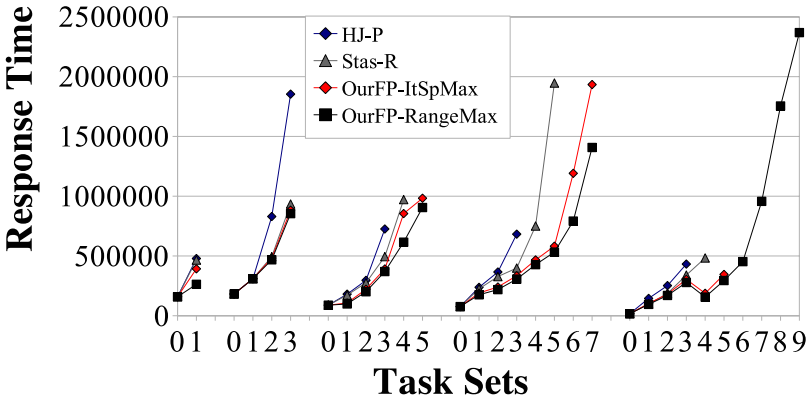
Figure 6.11: Results for U=0.5 using RM Policy



(a) # Preemptions for U = 0.8



(b) WCET w/ Delay for U = 0.8



(c) Response Time for U = 0.8

Figure 6.12: Results for U=0.8 using RM Policy

lower priority are less likely to be scheduled in the initial intervals, more preemption points are deemed infeasible by the new methods, hence producing tighter bounds. This feature of the new methods prevents the exponential increase in the number of preemptions observed in the method HJ-P for successive tasks.

Analysis times using the method OurFP-RangeMax are indicated in Table 6.5. The most significant factors affecting the analysis time of a task are the number of memory accesses within the task and the actual loop nest structure of the task. The actual task set characteristics, which determine the number of jobs of each task in the hyperperiod of the task set, also contribute to this time, albeit in a much less significant way.

Table 6.5: Analysis Times in Seconds

Utilization = 0.5		**Utilization = 0.8**	
Task Set Size	**Analysis Time**	**Task Set Size**	**Analysis Time**
2	33.65	2	558.44
4	177.80	4	626.47
6	175.20	6	436.49
8	417.85	8	419.33
		10	415.11

Between the two base utilizations whose results are shown in Figures 6.11 and 6.12, it may be observed that the 0.8 utilization shows a higher number of preemptions than the 0.5 utilization. This is due to the increased WCET in the case of higher utilization. Increased WCET means that the tasks have a greater number of feasible preemptions once they have started execution and, hence, the response times of tasks increase. It may be observed from the results that the WCET bound of a task does not depend significantly on its priority unlike the response time. This is due to the fact that the stand-alone, or base, WCET dominates the total preemption delay cost. Thus, the WCET with preemption delay does not necessarily increase monotonically with decreasing priority.

From the results, several observations can be made regarding the two prior methods (HJ-P and Stas-R). While both of them produce very similar results for the first two tasks in a task set, they start to exhibit differences for lower priority tasks. The Stas-R method consistently performs better than the HJ-P method since it correctly takes effects of indirect preemptions into account.

As already observed, the new method OurFP-RangeMax produces significantly

tighter bounds than the two prior methods, HJ-P and Stas-R, in *all* cases. Furthermore, the method that produces an upper bound for the number of preemptions for any phasing of tasks, namely OurFP-ItSpMax, also usually performs significantly better than both prior methods. However, in the case of the task with second-highest priority, it gives a higher number of preemptions (and, hence, WCET with delay and response time) than the Stas-R method. The reason for this is as follows. Since the task with second-highest priority has only one higher-priority task, the Stas-R and OurFP-ItSpMax methods actually calculate the same number of preemptions. However, in OurFP-ItSpMax, extra preemptions are added (and, hence, extra preemption delay) to account for any phasing less than or equal to a given maximum. This makes the number of preemptions higher than that produced by Stas-R.

To illustrate the variation in the maximum number of preemptions obtained by the new method OurFP-RangeMax between the various jobs of a task, results are provided for two of the task sets in Tables 6.6 and 6.7 respectively. As already observed from the graphs in Figures 6.11 and 6.12, OurFP-RangeMax always produces a *significantly* lower value for the number of preemptions than that produced by the two prior methods. Moving towards lower priority tasks, it may be observed that there is a difference between the number of preemptions for different jobs of the same task by noting that the minimum, maximum and average values obtained over all the jobs are different from each other. In the case of the task set with base utilization 0.8, it may be observed that the number of preemptions for certain tasks are not reported in the Stas-R method. This is for the same reason already explained with reference to the graphs in Figures 6.11 and 6.12, *i.e.*, the number of preemptions cannot be calculated since the response times of those tasks exceed their respective deadlines.

In these experiments, it is also observed that the number of preemptions obtained for the first job of every task (released at the same time as all higher priority jobs) is not always the maximum value obtained across all jobs. This proves the claim made in Section 6.4.1 about the critical instant *not* being the instant at which jobs of all tasks are released at the same time. It further underlines the necessity to perform such a feasible preemption point analysis for *every job* in the hyperperiod of a task set rather than once for every task.

Table 6.6: Number of Preemptions (# P) for Task Set with U = 0.5

Benchmark	**Period (cycles)**	**WCET (cycles)**	**BCET (cycles)**	**# Jobs**	**# P**				
					OurFP-RangeMax			**HJ-P**	**Stas-R**
					avg	min	max		
200convolution	100k	14191	14191	40	**0**	**0**	**0**	0	0
300convolution	400k	20891	20891	10	**0**	**0**	**0**	4	1
500convolution	500k	34291	34291	8	**0**	**0**	**0**	7	2
300n-real-updates	800k	56538	47338	5	**0.2**	**0**	**1**	12	4
matrix1	1000k	59896	54015	4	**1**	**1**	**1**	17	6
600fir	2000k	54837	52537	2	**1**	**1**	**1**	34	8
800convolution	2000k	66191	54391	2	**1**	**1**	**1**	35	14
900lms	4000k	158636	118536	1	**3**	**3**	**3**	71	27

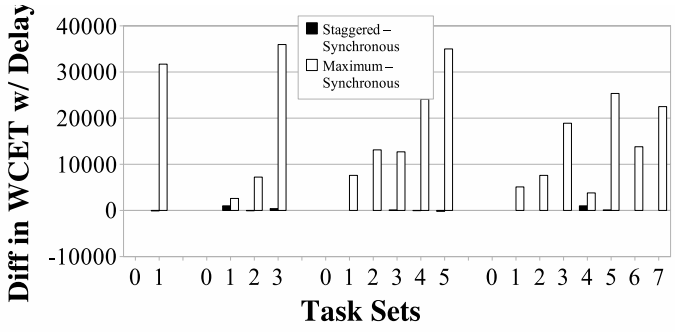
Table 6.7: Number of Preemptions (# P) for Task Set with U = 0.8

Benchmark	**Period (cycles)**	**WCET (cycles)**	**BCET (cycles)**	**# Jobs**	**# P**				
					OurFP-RangeMax			**HJ-P**	**Stas-R**
					avg	min	max		
n-real-updates	100k	16738	16838	50	**0**	**0**	**0**	0	0
900convolution	625k	76391	61091	8	**0.75**	**0**	**1**	7	1
matrix1	625k	59896	54015	8	**1**	**1**	**1**	8	3
1000convolution	625k	87091	67791	8	**1.25**	**1**	**2**	9	5
600convolution	1000k	45291	40991	5	**0.6**	**0**	**2**	16	7
300n-real-updates	1000k	56538	47338	5	**1.4**	**0**	**3**	17	10
800fir	1250k	77037	69737	4	**1.75**	**1**	**2**	23	
900lms	1250k	158636	118536	4	**3.75**	**3**	**5**	24	
1000fir	2500k	99237	86937	2	**4.5**	**3**	**6**	47	
500fir	5000k	43937	43937	1	**1**	**1**	**1**	94	

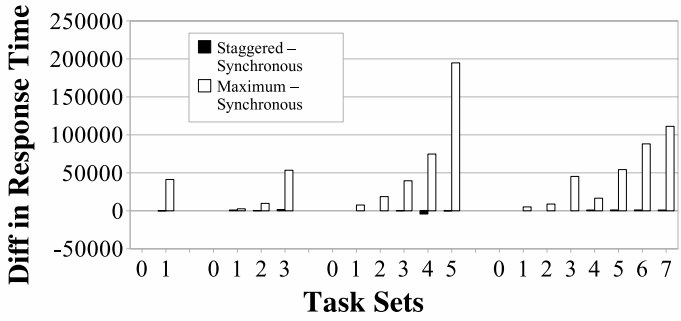
6.5.2 Task Sets with Staggered Releases

In the first set of experiments (presented in Section 6.5.1), it is assumed that all tasks in a task set are released simultaneously (synchronous release). However, since the analysis technique is capable of producing worst-case response time bounds for a task set with a particular phasing, experimental results for such a case are now presented. For this purpose, the task set characteristics from Table 6.4 are reused. However, in this set of experiments, the phasing of the tasks is changed. Tasks in every task set are released in reverse order of priority. Every task has a maximum phase of 1000 cycles or its own period,

whichever is smaller.



(a) Difference in WCET w/ Delay for U = 0.5

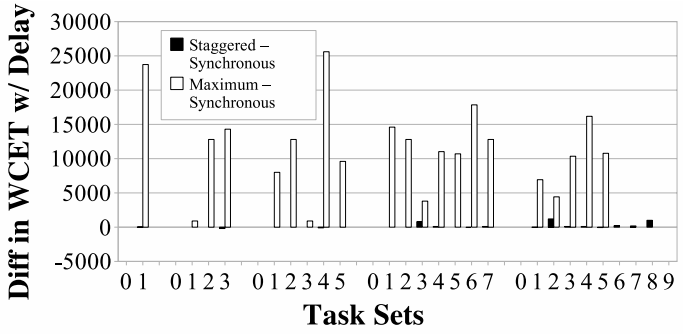


(b) Difference in Response Time for U = 0.5

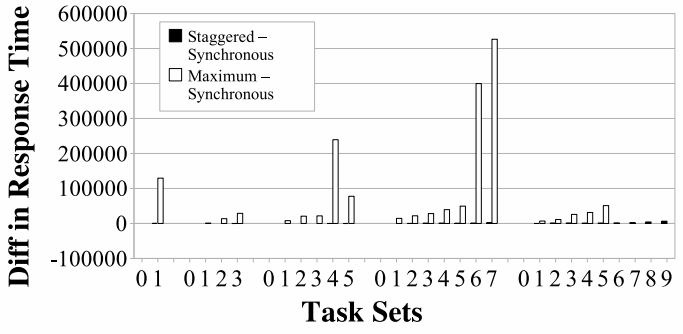
Figure 6.13: Results for U = 0.5 (Staggered - Synchronous)

In these experiments, the results of OurFP-RangeMax between the first set and the current set of experiments are similar. For this reason, the *differences* obtained in the bounds for the WCET and response times of task sets with synchronous and staggered releases are presented. The differences are shown in Figures 6.13 and 6.14. The graph also shows differences between maximum possible values (obtained using OurFP-ItSpMax) and synchronous release.

For most tasks, there are very small changes in the values of WCET and response times for the phased task sets compared to the difference between the maximum possible values and synchronous release. This is because, in OurFP-ItSpMax, extra preemptions are added to account for any possible phasing up to a given maximum (1000 cycles in this set of experiments). Furthermore, it is assumed that the maximum possible delay is incurred at every preemption point. Since the increase or decrease in WCET is entirely dependent



(a) Difference in WCET w/ Delay for U = 0.8



(b) Difference in Response Time for U = 0.8

Figure 6.14: Results for U = 0.8 (Staggered - Synchronous)

on the relative positioning of jobs at different points in the hyperperiod, it is difficult to determine the phasing of a task set that would result in the worst possible response-times for all tasks. This illustrates the merit of the method that calculates an upper bound on the number of preemptions *irrespective* of phasing (OurFP-ItSpMax).

6.5.3 Effects of WCET/BCET on # of Preemptions

In order to study the effects that the ratio of WCET of a task to its BCET has on the upper bound for the number of preemptions it incurs, a set of experiments is performed with *synthetic* task sets. The ratio of WCET to BCET for every task is varied, maintaining all other parameters. The characteristics of the task sets used in this set of experiments are shown in Table 6.8. Table 6.8 also indicates the number of preemptions obtained for the tasks using the $HJ - P$ bound and using the method by Staschulat et al. (Stas-R).

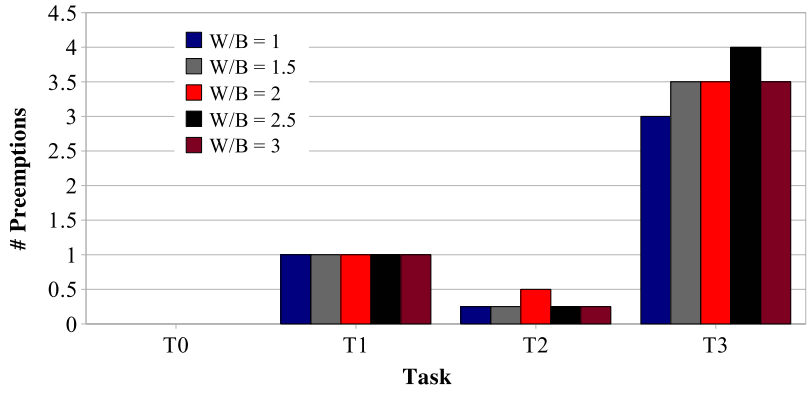
Since synthetic tasks are used, there is no actual code from which to construct access chains for calculation of preemption delay. Hence, a fixed value needs to be assumed for the preemption delay. Since the preemption delay is significantly less than the base WCET of a task, a delay value of 0 is assumed in this set of experiments for the sake of simplicity. The primary goal in this set of experiments is to show how the WCET/BCET ratio affects the number of preemptions.

Table 6.8: # Preemptions for Task Sets with U=0.5 and U=0.8

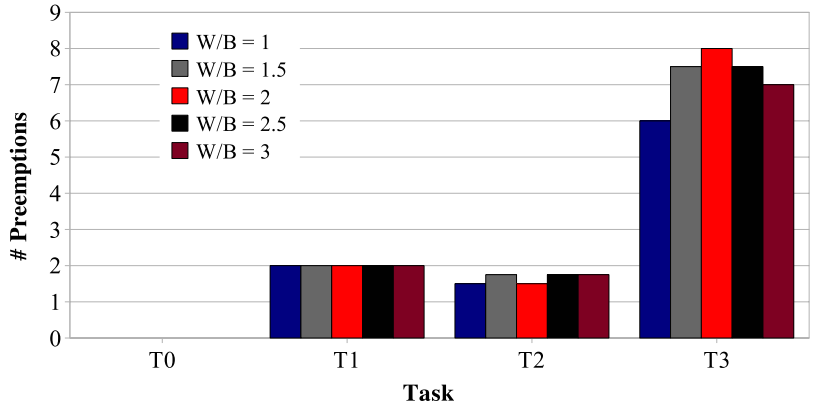
Task ID	**Period (cycles)**	**WCET (cycles)**	**# Preemptions**	
			HJ-P	**Stas-R**
U = 0.5				
0	10K	1K	0	0
1	80K	16K	8	2
2	100K	5K	12	4
3	200K	30K	25	8
U = 0.8				
0	10K	1.5K	0	0
1	80K	20K	8	3
2	100K	15K	12	6
3	200K	50K	25	19

The results obtained for the various ratios using the analysis presented in Section 6.4 (OurFP-RangeMax), for utilizations 0.5 and 0.8, are shown in Figures 6.15(a) and 6.15(b), respectively. For utilizations of 0.5 and 0.8, WCET/BCET ratios of 1, 1.5, 2, 2.5 and 3 are used. As before, in every case, the bounds obtained by the method that eliminates infeasible preemption points is significantly lower than those obtained by the two prior methods (HJ-P and Stas-R). As the ratio of WCET/BCET increases, the upper bound on the number of preemptions increases slightly for small ratios. After a ratio of around 3, the number of preemptions start to decrease once again. However, the number of preemptions does not go below the number obtained with ratio equal to 1. This is expected since the schedule with WCET/BCET ratio of 1 has the least amount of slack.

The maximum increase in the number of preemptions compared to the number with ratio equal to 1 is found to be approximately 30 percent. Hence, even if the BCET of a task is set to 0, the pessimism in the results obtained is not very significant. In fact, the results would still be tighter than those produced by the two prior methods. This is a useful



(a) U = 0.5



(b) U = 0.8

Figure 6.15: # Preemptions given by OurFP-RangeMax for Varying WCET/BCET

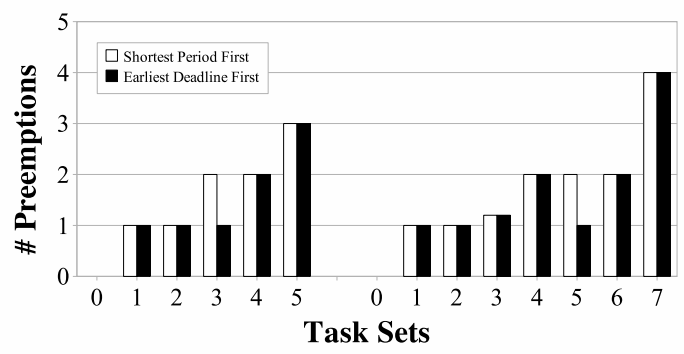
observation since several timing analyzers only provide WCET bounds for a task, but not BCET bounds. Even in such cases, the feasible preemption analysis would be applicable and useful to obtain reasonably tight bounds on the worst-case number of preemptions.

6.5.4 Static-Priority vs Dynamic-Priority Scheduling Policy

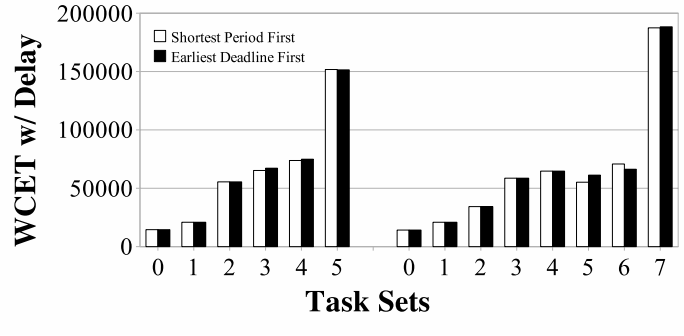
In all the above experiments, the static RM scheduling policy is used. However, the analysis framework is conceptually able to support dynamic-priority scheduling policies as well. In order to demonstrate this, a set of experiments is performed using the EDF scheduling policy. For this purpose, once again, the task sets whose characteristics are

shown in Table 6.4 are used. Tasks in every task set are released in reverse order of the lengths of their periods (*i.e.*, in reverse order of priority as determined by the RM scheduling policy). A phase difference of 10 cycles is used between successive tasks. Figures 6.16 and 6.17 compare results obtained using both the RM and the EDF scheduling policies for base utilizations of 0.5 and 0.8 respectively. Only the task sets that actually exhibit a difference in behavior between the two policies are presented.

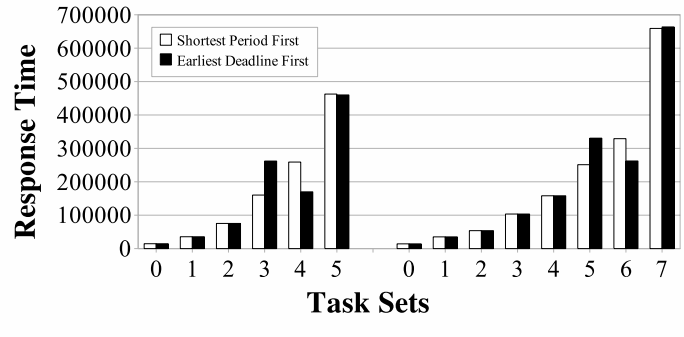
From the above results, it may be observed that, in some cases, the EDF policy increases the response times of tasks with shorter periods (higher priority according to the RM policy) in comparison to the RM policy. This is due to the fact that the relative deadlines of jobs alter their priorities. For the same reason, the EDF policy sometimes decreases the response times of tasks with longer periods compared to the RM policy. The experiments demonstrate the applicability of the feasible preemption analysis technique to systems with dynamic-priority scheduling policies.



(a) # Preemptions for U = 0.5

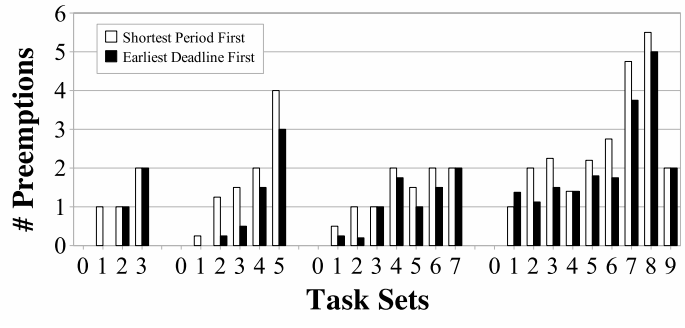


(b) WCET w/ Delay for U = 0.5

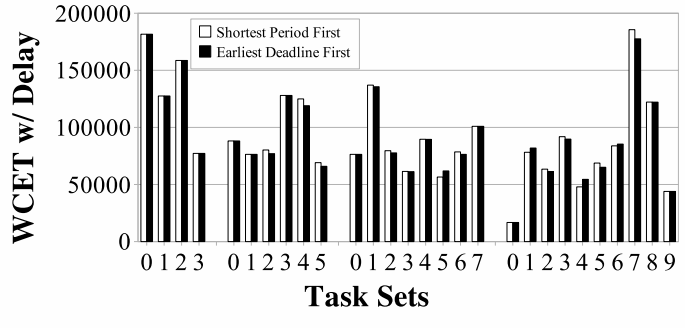


(c) Response Time for U = 0.5

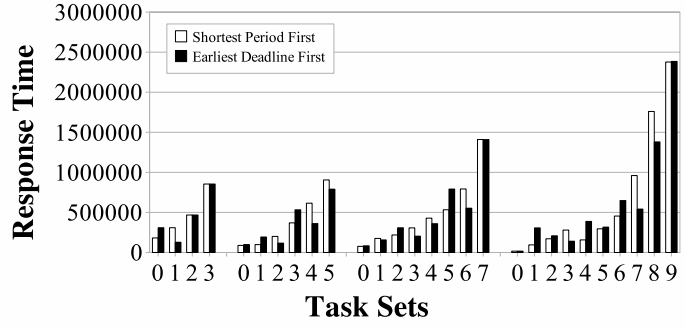
Figure 6.16: Comparison of Results for RM and EDF for U=0.5



(a) # Preemptions for U = 0.8



(b) WCET w/ Delay for U = 0.8



(c) Response Time for U = 0.8

Figure 6.17: Comparison of Results for RM and EDF for U=0.8

Chapter 7

Tasks with Non-Preemptive Regions

In Chapter 6, a framework to provide worst-case response time estimates for tasks in a multi-task, preemptive, hard real-time environment is presented. In such a system, every task has a priority. A task with a higher priority may preempt a task with a lower priority. The lower priority task then experiences a data-cache related preemption delay (D-CRPD) when it resumes execution, which increases its WCET and, hence, response time.

A fundamental assumption in the previous analysis is that all tasks are completely preemptive. In other words, a task may be interrupted by a task with higher priority *at any time* during its execution. However, this assumption may need to be relaxed for some tasks. A task may have a period in its execution during which it executes in a *critical section*. While a task is in a critical section, no other task may enter a critical section.

In this chapter, tasks are allowed to have a critical section within their execution. In order to maintain logical correctness of tasks, it is assumed that a task executing within a critical section cannot be preempted. In other words, every critical section is assumed to be a *non-preemptive region (NPR)*. A framework that statically analyzes task sets in such an environment and calculates worst-case response times for all tasks is now presented.

The complexity in the analysis of tasks with non-preemptive regions arises from the fact that the actual execution time of a task is usually unknown. Instead, a range of possible execution times bounded by the best and worst-case execution times of the task is

considered. Hence, if a higher-priority task is released when a lower-priority task is already in execution, an exact point of execution where the lower-priority task is guaranteed to be at the time cannot be identified. Thus, a situation might arise where the lower-priority task *could* be inside a non-preemptive region but is *not guaranteed* to be.

7.1 Methodology

A NPR is represented by the first and last iteration points of the range of consecutive iteration points during which a particular task may not be interrupted. Every task is hence effectively divided into multiple regions starting with a preemptive region and every alternate region representing a NPR. The static timing analyzer described in Section 2.2.2 is enhanced to calculate the worst-case and best-case execution times of every region based on the start and end iteration points of each NPR. For the sake of simplicity, the discussion in this section assumes that every task has at most one NPR. Hence, every task has three regions where regions 1 and 3 are preemptive and region 2 is a NPR. Examples with multiple NPRs are discussed later.

In the analysis described in Chapter 6, whenever an instance of a task is released, it is placed in a service queue and the scheduler is invoked. The scheduler chooses the task with the highest priority at the current time, preempting any lower priority task that might be executing at the time. However, in the new analysis, a task with higher priority may be required to wait if a lower-priority task is executing in its NPR. In order to calculate the worst-case response time for every task, several possible scenarios need to be considered.

Suppose that a task T_1 is released at time t and has three regions where the middle one is a NPR. At time $t + x$, a higher-priority task T_0 is released. At time $t + x$, there are three possible cases:

1. T_1 has finished executing its first region and started executing its NPR in both best and worst cases;

2. T_1 has not finished executing its first region or has already finished its NPR and entered its third region in both cases; or

3. T_1 has started executing its NPR in the best case, but not in the worst case.

Cases 1 and 2 are straightforward. In case 1, T_0 has to wait until T_1 finishes executing its NPR. In the best case, this time is equal to the best-case remaining execution time of T_1's

NPR. In the worst-case, it is equal to the worst-case remaining execution time of task T_1's NPR. In case 2, T_1 gets preempted and T_0 starts to execute immediately.

In case 3, it is not certain whether T_1 has started executing its NPR or not. Hence, the best and worst possible scenarios *for each task* are considered in order to determine its worst-case response time. For T_0, the worst case is to assume that T_1 has already started executing its NPR and add the worst-case remaining execution time of T_1's NPR to the response time of T_0. On the other hand, the best case for T_0 is to assume that T_1 has not yet started executing its NPR and, hence, may be preempted. The scenarios are reversed for T_1. Its best case is to assume that it has already started executing its NPR and, hence, is not preempted. Its worst case is to assume that it gets preempted by T_0 and add the associated preemption delay to its remaining execution time. By considering parallel execution scenarios for each task, safe response time estimates may be obtained.

7.1.1 Illustrative Examples

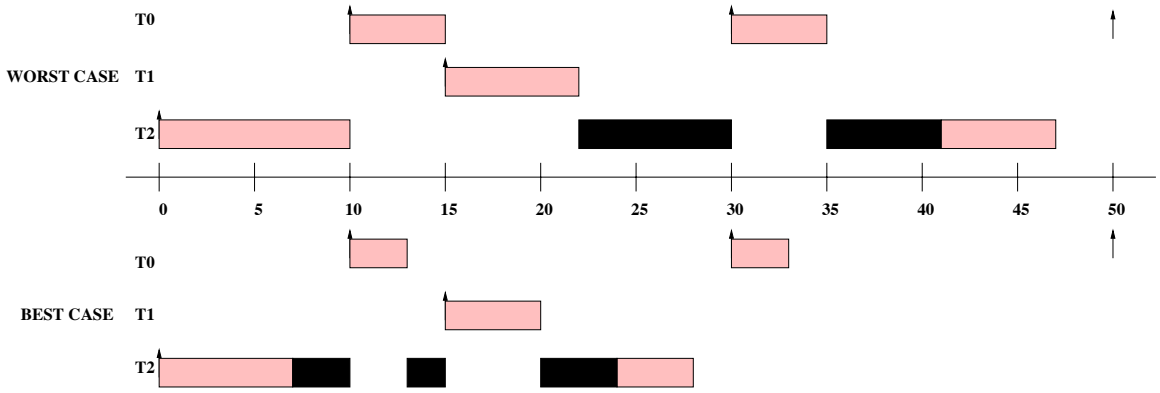
An illustrative example of the methodology to analyze tasks with non-preemptive regions is now presented. Consider the task set whose characteristics are specified in Table 7.1. The first column shows the task name. The second and third columns show the phase and period (equal to the deadline) of the each task. Assume that the Rate Monotonic (RM) scheduling policy is used for this task set and, hence, that the task with the shortest period has the highest priority. The fourth and fifth columns show the WCETs and BCETs of each of the three regions of a task.

Table 7.1: Task Set Characteristics - Task Set 1 [RM policy $\rightarrow T_0$ has Highest Priority]

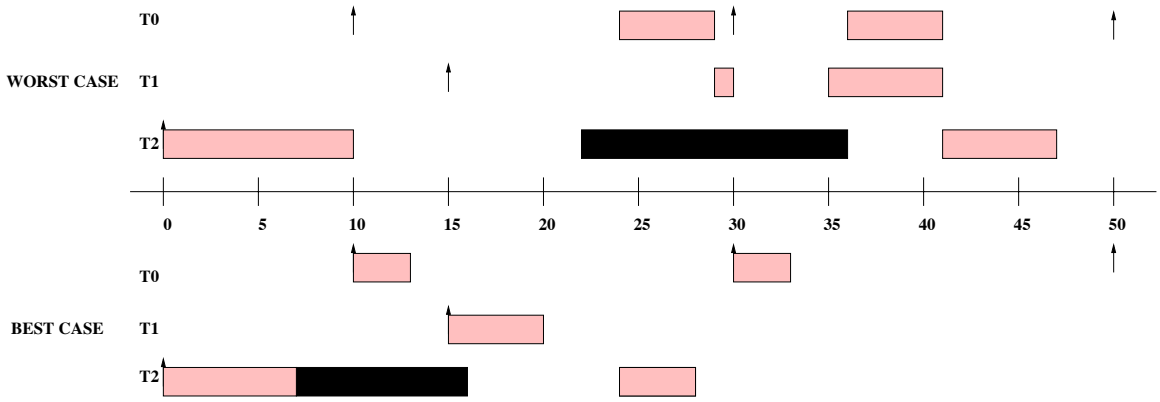
Task	**Phase**	**Period = deadline**	**WCET (r1/r2/r3)**	**BCET (r1/r2/r3)**
T_0	10	20	5/0/0	3/0/0
T_1	15	50	7/0/0	5/0/0
T_2	0	200	10/14/6	7/9/4

For ease of understanding, the same task set is also evaluated assuming that all three regions of every task are fully preemptive. Figure 7.1(a) shows the best and worst-case timelines for this case below and above the horizontal time axis respectively. The arrows show release points of the three tasks. The lightly shaded rectangles represent preemptive execution regions of tasks.

Now, a non-preemptive region is added to task T_2, as indicated in Table 7.1. Figure 7.1(b) shows the timeline for this situation. Here, the black rectangles represent non-preemptive regions of execution. For the sake of comparison, the second region of task T_2 is shown as a black rectangle in Figure 7.1(a) although it is fully preemptive in that example.



(a) Results with No NPR



(b) Results with NPR

Figure 7.1: Best and Worst Case Results for Task Set 1

In Figure 7.1(b), it may be observed that some execution regions overlap. This is because, at every release point, if there is some task that *could be* executing in its NPR but is *not guaranteed to be*, both best and worst case scenarios are considered for the higher and lower-priority tasks. In reality, only one of the scenarios takes place and there is no simultaneous execution of multiple tasks.

Every point on the timeline where an event occurs is now considered, and the best

or worst-case scenarios as the case may be examined.

Time 0: Job $J_{2,0}$ is released.

Best case:

$J_{2,0}$: It is the only job that is ready to execute and hence starts executing. It it scheduled to finish region 1 at time 7.

Worst-case:

$J_{2,0}$: It is the only job that is ready to execute and hence starts executing. It it scheduled to finish region 1 at time 10.

Time 7:

Best case:

$J_{2,0}$: It finishes executing region 1 and starts executing its NPR. It is scheduled to finish at time 16.

Time 10: Job $J_{0,0}$ is released.

Best case:

$J_{0,0}$: It is scheduled immediately since there is a chance that $J_{2,0}$ has not yet started executing its NPR. It is scheduled to finish region 1 at time 13.

$J_{2,0}$: It continues executing its NPR.

Worst case:

$J_{0,0}$: Since there is a chance that $J_{2,0}$ has started its NPR, $J_{0,0}$ has to wait for at most 14 units of time (worst-case remaining execution time of $J_{2,0}$'s NPR). It is scheduled to start at time 24.

$J_{2,0}$: It has finished executing region 1 . Since there is a chance that $J_{2,0}$ has not started its NPR, it gets preempted by $J_{0,0}$ and it now re-scheduled to start its NPR at time 15 (adding the WCET of $J_{0,0}$).

Time 13:

Best case:

$J_{0,0}$: Finishes executing region 1. Since it has no more regions, it is done.

Time 15: Job $J_{1,0}$ is released.

Best case:

$J_{1,0}$: It is scheduled immediately since there is a chance that $J_{2,0}$ has not yet started executing its NPR. It is scheduled to finish region 1 at time 20.

$J_{2,0}$: It continues executing its NPR.

Worst-case:

$J_{1,0}$: Since $J_{0,0}$, which has a higher priority that $J_{1,0}$ is waiting, $J_{1,0}$ is not scheduled now.
$J_{2,0}$: Since there is a chance that $J_{2,0}$ has not started its NPR, it gets preempted by $J_{1,0}$ and it re-scheduled to start its NPR at time 22 (adding the WCET of $J_{1,0}$).

Time 16:

Best case:

$J_{2,0}$: It finishes executing its NPR. Since there are higher priority tasks waiting, region 3 of $J_{2,0}$ is scheduled at time 24 (add BCETs of $J_{0,0}$ and $J_{1,0}$).

Time 20:

Best case:

$J_{1,0}$: It finishes executing region 1. Since it has no more regions, it is done.

Time 22:

Best case:

$J_{0,1}$: It starts executing region 1. It is scheduled to finish at time 33.

Worst case:

$J_{2,0}$: It starts executing its NPR. It is scheduled to finish this region at time 36.

Time 24:

Best case:

$J_{2,0}$: It starts executing region 3. It is scheduled to finish this region at time 28.

Worst case:

$J_{0,0}$: It starts executing region 1. It is scheduled to finish at time 29.

Time 28:

Best case: $J_{2,0}$: It finished executing region 3. It is now done.

Time 29:

$J_{0,0}$: It finishes executing region 1. Since it has no more regions, it is done.

$J_{1,0}$: It starts executing region 1. It is scheduled to finish at time 36.

Time 30: Job $J_{0,1}$ is released.

Best case:

$J_{0,1}$: It starts executing region 1. It is scheduled to finish at time 33.

Worst case:

$J_{0,1}$: Since $J_{2,0}$ is guaranteed to have started its NPR, $J_{0,1}$ has to wait until $J_{2,0}$ completes its NPR and is hence scheduled for time 36.

Time 36:

Worst case:

$J_{2,0}$: It finishes executing its NPR.

$J_{0,1}$: It starts executing region 1. It is scheduled to finish at time 41.

Time 41:

Worst case:

$J_{0,1}$: It finishes executing region 1. Since it has no more regions, it is done.

$J_{1,0}$: It resumes executing region 1. It is scheduled to finish at time 47.

Time 47:

Worst case:

$J_{1,0}$: It finishes executing region 1. Since it has no more regions, it is done.

$J_{2,0}$: It starts executing region 3. It is scheduled to finish at time 53.

Time 50: $J_{0,2}$ is released.

The analysis proceeds in a similar fashion up to the hyperperiod of the task set, namely 200. In this example, for the sake of simplicity, preemption delay calculations are not shown. The delay at every resumption point is assumed to be zero. Actual preemption delay calculations are performed in the same manner as that discussed in Section 6.4.4.

Table 7.2: Example Task Set Characteristics - Task Set 2

Task	**Phase**	**Period = deadline**	**WCET (r1/r2/r3)**	**BCET (r1/r2/r3)**
T_0	15	20	5/0/0	3/0/0
T_1	10	50	3/4/5	2/4/2
T_2	0	200	10/14/6	7/9/4

As a second example, consider the task set shown in Table 7.2. The structure of the table is the same as that of Table 7.1. The best and worst-case scenarios for the three tasks in this task set are shown in Figure 7.2. Until time 15, the analysis is similar to that in the first example and is hence omitted. However, in this example, $J_{1,0}$ is released before $J_{0,0}$. Also, both tasks T_1 and T_2 have NPRs. Hence, when $J_{0,0}$ is released at time 15, the analysis is different and is explained below.

Time 15: $J_{0,0}$ is released.

Best case:

$J_{0,0}$: Since there is a chance that neither $J_{1,0}$ nor $J_{2,0}$ has started its NPR, $J_{0,0}$ starts executing region 1. It is scheduled to finish at time 18.

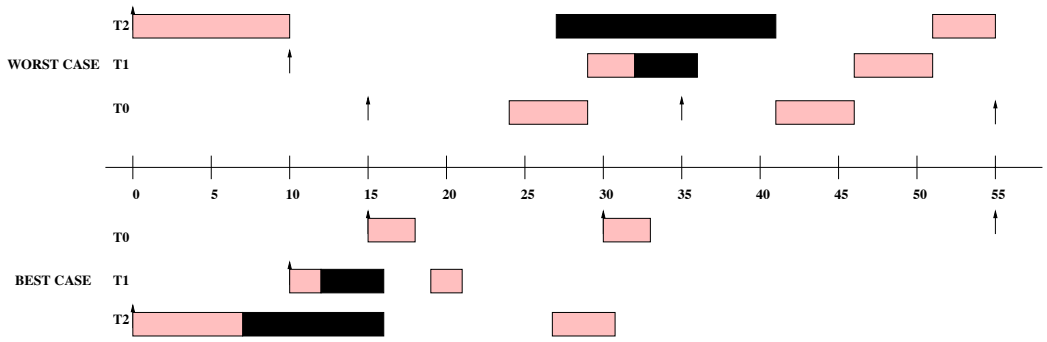


Figure 7.2: Best and Worst-Case Scenarios for Task Set 2

$J_{1,0}$: It continues executing its NPR.

$J_{2,0}$: It continues executing its NPR.

Worst case:

$J_{0,0}$: Since there is a chance that either $J_{2,0}$ or $J_{1,0}$ has started its NPR, $J_{0,0}$ has to wait. The worst-case remaining execution times of the NPRs of both $J_{2,0}$ and $J_{1,0}$ are calculated and the maximum of the two is considered as the worst-case waiting time of $J_{0,0}$. In this case, the waiting time is the maximum of 9 and 2 time units, which is 9 time units. Hence, $J_{0,0}$ is scheduled to start execution at time 24.

$J_{1,0}$: Since there is a chance that it has not started its NPR, it gets preempted by $J_{0,0}$. It is rescheduled to start its NPR at time 29 (adding the WCET of $J_{0,0}$).

$J_{2,0}$: Since there is a chance that it has not started its NPR, it gets preempted by $J_{0,0}$. It is rescheduled to start its NPR at time 27 (adding the WCET of $J_{0,0}$).

Time 16:

$J_{1,0}$: It finishes executing its NPR. Since there is a higher-priority task waiting, region 3 is scheduled to start at time 19 (add BCET of $J_{0,0}$).

$J_{2,0}$: It finishes executing its NPR. Since there are higher-priority task waiting, region 3 is scheduled to start at time 27 (add BCETs of $J_{0,0}$ and $J_{1,0}$).

Time 18:

Best case:

$J_{0,0}$: It finishes executing region 1. Since it has no more regions, it is done.

Time 19:

Best case:

$J_{1,0}$: It starts executing region 3. It is scheduled to finish at time 21.

Time 21:

Best case:

$J_{1,0}$: It finishes executing region 3. Since it has no more regions, it is done.

Time 24:

Worst case:

$J_{0,0}$: It starts executing region 1. It is scheduled to finish at time 29.

Time 27:

Best case:

$J_{2,0}$: It starts executing region 3. It is scheduled to finish at time 31.

Worst case:

$J_{02,0}$: It starts executing its NPR. It is scheduled to finish at time 41.

Time 29:

Worst case:

$J_{0,0}$: It finishes executing region 1. Since it has no more regions, it is done.

$J_{1,0}$: It starts executing region 1. It is scheduled to finish at time 32.

Time 31:

Best case:

$J_{2,0}$: It finishes executing region 3. Since it has no more regions, it is done.

Time 32:

Worst case:

$J_{1,0}$: It finishes executing region 1. It starts executing its NPR. It is scheduled to finish at time 36.

Time 35: $J_{0,1}$ is released.

Best case:

$J_{0,1}$: It starts executing region 1. It is scheduled to finish at time 38.

Worst case:

$J_{0,1}$: Since either $J_{2,0}$ or $J_{1,0}$ is guaranteed to be in its NPR, $J_{0,1}$ has to wait. The worst-case remaining execution times of the NPRs of both $J_{2,0}$ and $J_{1,0}$ are calculated and the maximum of the two is considered as the worst-case waiting time of $J_{0,1}$. In this case, the waiting time is the maximum of 6 and 1 time units, which is 6 time units. Hence, $J_{0,1}$ is scheduled to start execution at time 41.

Time 36:

Worst case:

$J_{1,0}$: It finishes executing its NPR.

Time 38:

Best case:

$J_{0,1}$: It finishes executing region 1. Since it has no more regions, it is done.

Time 41:

Worst case: $J_{2,0}$: It finishes executing its NPR.

$J_{0,1}$: It starts executing region 1. It is scheduled to finish at time 46.

Analysis proceeds further until the hyperperiod of the task set, namely 200.

7.2 Analysis Algorithm

An algorithm briefly describing the methodology is shown in Figures 7.3 to 7.6. The implementation of the system uses an event hierarchy similar to the one described in Section 6.4.5. However, the actual handling of some of the events is different. The events that are handled differently are described in Figures 7.3 to 7.6.

Structures global to all events are described below
bc_service_queue, wc_service_queue : queues of all released jobs that have not yet completed
Possible status : READY, WAITING_SCHED and IN_SERVICE
event_list : list of events ordered first by time and then by the priority of the type of event

Parameters for each event are described below
current_time : Time at which event occurs, *curr_job* : Job which the event corresponds to

Functions used in event handlers :
insertIntoB/WCServiceQueue(job, status)
inserts job with given status into priority-ordered service queue
removeFromB/WServiceQueue(job)
removes given job from service queue
checkIfNPRInWorstCase(job)
checks whether given job is guaranteed to have started execution of its NPR in the worst-case
checkIfNPRPossibleInBestCase(job)
checks whether given job has possibly started execution of its NPR in the best-case
calculateWaitTimeForJob(rel_job, exec_job)
calculates the maximum amount of time rel_job would get delayed due to exec_job's NPR
insertIntoEventList(ev_job, ev_time, ev_type)
inserts an event of type ev_type into the event list for time ev_time and job ev_job
removeFromEventList(ev_job, ev_type)
removes events of ev_type corresponding to ev_job

JobRelease event: This event represents the release of a new job of a task
preempt, suspend : boolean values initialized to FALSE
create_event: boolean value initialized to TRUE
curr_wait, wait_time, wc_start_exec : integers initialized to 0
status : enumeration with possible values of IN_SERVICE, WAITING_SCHED and READY

Best-case handling:
If (event_queue is empty) create_event ← TRUE
else {
for every job (q_job) in bc_service_queue starting from lowest priority job {
if (curr_job has higher priority than q_job) {
if (status of q_job is IN_SERVICE) {
if (q_job is in not in NPR in best case) preempt ← TRUE

Figure 7.3: Algorithm for NPR-Aware Calculation of WCET w/ Delay

```
        else if (q_job is in NPR even in worst case) {
          curr_wait ← best case remaining exec time
          if (wait_time ⟨ curr_wait) wait_time ← curr_wait
          status ← WAITING_SCHED
        }
      } else if (status of q_job is WAITING_SCHED) {
        removeFromEventList(q_job, BCStartExec)
        insertIntoEventList(q_job, current_time + BCET of curr_job, BCStartExec)
      } else if (status of q_job is READY) {
        removeFromEventList(q_job, BCStartExec)
        if (q_job just finished one sub part and is ready to start next one) preempt ← TRUE
      }
      if (preempt == TRUE) insertIntoEventList(q_job, current_time, BCPreemption)
    } else {
      create_event ← FALSE, status ← READY
      break from for loop
    }
  }
}
if (create_event == TRUE) insertIntoEventList(curr_job, current_time + wait_time, BCStartExec)
insertIntoBCServiceQueue(curr_job, status)
```

Worst-case handling:

```
If (event_queue is empty) {
  create_event ← TRUE
} else {
  for every job (q_job) in wc_service_queue starting from lowest priority job{
    if (curr_job has higher priority than q_job) {
      create_event = create_event AND TRUE
      npr ← checkIfNPRInWorstCase(q_job)
      npr_possible ← checkIfNPRPossibleInBestCase(q_job)
      if (npr_possible == TRUE) {
        curr_wait ← calculateWaitTimeForJob(curr_job, q_job)
        if (wait_time ⟨ curr_wait) wait_time ← curr_wait
        if (status of q_job is WAITING_SCHED) {
          reschedule WCStartExec event for q_job
          preempt ← suspend ← FALSE
        } else preempt ← suspend ← TRUE
```

Figure 7.4: Algorithm (cont.) for NPR-Aware Calculation of WCET w/ Delay

```
        create_event ← create_event AND TRUE
        status ← WAITING_SCHED
      } else if (npr == TRUE) {
        curr_wait ← max npr exec time remaining for q_job
        if (wait_time ⟨ curr_wait)
          wait_time ← curr_wait
        preempt ← suspend ← FALSE
        create_event ← create_event AND TRUE
        status ← WAITING_SCHED
      } else {
        preempt ← TRUE
        if ((status == WAITING_SCHED) AND (q_job has already waited for task in NPR))
          suspend ← TRUE
        else suspend ← FALSE
        create_event ← TRUE
      }
      if ( (preempt == TRUE) AND (suspend == TRUE) ) {
        insertIntoEventList(q_job, WCPreemptionAndSuspension)
      else if (preempt == TRUE) insertIntoEventList(q_job, WCPreemption)
    } else {
      create_event ← FALSE, status ← READY
      break from for loop
    }
  }
  if (create_event == TRUE)
    insertIntoEventList(curr_job, current_time + wait_time, WCStartExec)
}
insertIntoWCServiceQueue(curr_job, status)
```

WCPreemptionAndSuspension : This event represents preemption and suspension
of the current job

```
Perform functions of WCPreemption event
wc_start_time ← resumption time for curr_job
insertIntoEventList(curr_job, wc_start_time, WCStartExec)
set status of curr_job to WAITING_SCHED in wc_service_queue
```

Figure 7.5: Algorithm (cont.) for NPR-Aware Calculation of WCET w/ Delay

```
BCEndExec event : This event represents the best-case end of execution of the current region
create_event : boolean value initialized to TRUE, event_time : integer initialized to current_time
status : enumeration with possible values of IN_SERVICE, WAITING_SCHED and READY
removeFromBCServiceQueue(curr_job)
update best case remaining time of current job
if (curr_job has another region) {
  if (region currently ending is NPR) {
    if (higher-priority job waiting for curr_job's NPR to end) create_event ← FALSE
    else {
      delay ← sum of BCETs of waiting higher-priority tasks
      event_time ← current_time + delay
      status ← WAITING_SCHED
    }
  }  if (create_event == TRUE) insertIntoEventList(curr_job, event_time, BCStartExec)
  insertIntoBCServiceQueue(curr_job, status)
}
if ( (curr_job has no more regions) AND (bc_service_queue has more jobs in it) ) {
  retval ← first READY job ready_job in bc_service_queue
  if (retval == TRUE) insertIntoEventList(ready_job, current_time, BCStartExec)
}

WCEndExec event : This event represents the worst-case end of execution
  of the current region
create_event : boolean value initialized to TRUE
removeFromWCServiceQueue(curr_job)
update wc_remaining_time of current job
if (curr_job has another region) {
  if (region currently ending is NPR) {
    if (higher-priority job waiting for curr_job's NPR to end) create_event ← FALSE
    if (create_event == TRUE) insertIntoEventList(curr_job, current_time, WCStartExec)
  }
  insertIntoWCServiceQueue(curr_job, READY)
}
if ( (curr_job has no more regions) AND (wc_service_queue has more jobs in it) ) {
  retval ← first READY job ready_job in wc_service_queue
  if (retval == TRUE) insertIntoEventList(ready_job, current_time, WCStartExec)
}
```

Figure 7.6: Algorithm (cont.) for NPR-Aware Calculation of WCET w/ Delay

7.3 Correctness of Analysis

The algorithm described in Section 7.2 calculates the worst-case response time of a job in the context of a given task set. Equation 5.2 shows the calculation of the worst-case response time of a job $J_{i,j}$. As explained in Section 5.1, the response time of a job, in the context of tasks with critical sections, is the sum of four components, namely the base WCET of the task, the execution time of higher-priority jobs, the data-cache related delay incurred due to preemption by higher-priority jobs and the waiting time incurred due to lower-priority jobs executing in a NPR. The formulation for each of these components for a job $J_{i,j}$ and a proof of their correctness are presented in this section.

Explanation of the new symbols used in the formulation is as follows. $NPR^r_{i,j}$ is a boolean value that indicates whether region r of job $J_{i,j}$ is a NPR or not. $bst^r_{i,j}$ represents the earliest possible start time for region r of job $J_{i,j}$. $lp(i,j)$ represents the set of jobs that have a lower priority than job $J_{i,j}$.

Theorem 7.3.1 *The response time of a job $J_{i,j}$, calculated as the sum of the values produced by Equations 6.4, 7.5, 7.6 and 7.7, is a safe upper bound on the worst-case response time of $J_{i,j}$ in the context of tasks with non-preemptive regions.*

The correctness of Theorem 7.3.1 is proved using Lemma 6.4.2 (stated and proved in Section 6.4.6) and Lemmas 7.3.2, 7.3.3 and 7.3.4 (stated and proved below).

The base WCET of a job does not include the effects of interference from other jobs. Hence, Lemma 6.4.2 may be reused in this context.

The execution time of higher-priority jobs within the response time of $J_{i,j}$ is calculated by counting the number of instances of every higher-priority task that may execute within the response time of $J_{i,j}$ and multiplying it by the execution time of the specific higher-priority job as shown in Equation 7.1.

$$hpex_{i,j} = \sum_{(k,l)\in hp(i,j)} \lceil \frac{\Re_{i,j}}{P_k} \rceil \cdot \sum_{r=1}^{nr_k} C^r_{k,l} \tag{7.1}$$

Since the algorithm described in Section 7.2 calculates response times for every *job* in the task set, the relative phasing between jobs is known. Using this information, the calculation in Equation 7.1 is tightened. After the release of $J_{i,j}$, the time during which no other higher-priority job is released may be calculated using information about relative phasing as shown

in Equation 7.2. The execution time remaining after the release of $J_{i,j}$ for any higher-priority job released *before* $J_{i,j}$ is calculated as shown in Equation 7.3. (Note: Equations 7.2 and 7.3 are the same as Equations 6.6 and 6.7, respectively and are repeated for convenience.)

$$at_{i,j} = \min_{(k,l)\in hp(i,j)} [\max(\lceil \frac{rel_{i,j} - \phi_{k,l}}{P_k} \rceil \cdot P_k, 0) + \phi_{k,l} - rel_{i,j}] \tag{7.2}$$

$$rem_{rel_{i,j}} = \sum_{(k,l)\in hp(i,j), rel_{k,l} < rel_{i,j}} C^{rem}_{k,l} \tag{7.3}$$

The difference between the times calculated in Equations 7.2 and 7.3 gives the time for which $J_{i,j}$ may execute without being preempted. Execution time of all NPRs of $J_{i,j}$ is calculated as shown in Equation 7.4.

$$npr_{i,j} = \sum_{r=1}^{nr_i} (C^r_{i,j} \cdot NPR^r_{i,j}) \tag{7.4}$$

Equation 7.5 shows the calculation for the new, tighter estimate on the execution time of higher-priority jobs within the response time of $J_{i,j}$, performed in accordance with the algorithm described in Section 7.2.

$$hpex_{i,j} = \sum_{(k,l)\in hp(i,j)} \lceil \frac{\Re_{i,j} - \max((at_{i,j} - rem_{rel_{i,j}}), 0) - npr_{i,j}}{P_k} \rceil \cdot \sum_{r=1}^{nr_k} C^r_{k,l} \tag{7.5}$$

Lemma 7.3.2 *An upper bound on the execution time of higher-priority jobs within the response time of a job $J_{i,j}$, in the context of tasks with non-preemptive regions, is given by Equation 7.5.*

Proof Assume that $\max((at_{i,j} - rem_{rel_{i,j}}), 0)$ is not subtracted from the iterative portion of Equation 7.5. It means that this time can be stretched due to *execution of higher-priority jobs* in between. By definition of $(at_{i,j} - rem_{rel_{i,j}})$, *all higher-priority jobs* released before $J_{i,j}$ *have completed execution* and no higher-priority jobs have been released yet after $J_{i,j}$. Contradiction. Hence, $\max((at_{i,j} - rem_{rel_{i,j}}), 0)$ can be subtracted from the iterative portion of Equation 7.5 without jeopardizing safety of the analysis.

Assume that $npr_{i,j}$ is not subtracted from the iterative portion of Equation 7.5. It means that this time can be stretched due to *preemption* by higher-priority jobs. $npr_{i,j}$ represents the execution time of regions of $J_{i,j}$ that are *non-preemptive*. Contradiction. Hence, $npr_{i,j}$ can be subtracted from the iterative portion of Equation 7.5 without jeopardizing safety of the analysis. ∎

Every release of a higher-priority job is a potential preemption point for $J_{i,j}$. Consider an interval between two such consecutive releases. According to the algorithm described in Section 7.2, the job release at the end of this interval can be a feasible preemption point for $J_{i,j}$ if a) there is a possibility that $J_{i,j}$ is scheduled in the interval, b) there is a possibility that $J_{i,j}$ has not completed execution before the end of the interval and c) $J_{i,j}$ is not guaranteed to be executing in a NPR. These conditions are mathematically represented and the preemption delay is given by Equations 7.6.

$$\begin{aligned} cond_a &= \sum_{(k,l)\in hp(i,j)} c_{k,l}^{rem} < t_I^{rem} \\ cond_b &= \sum_{(k,l)\in hp(i,j)} C_{k,l}^{rem} + C_{i,j}^{rem} > t_I^{rem} \\ cond_c &= \neg(\exists r \; s.t. \sum_{cr=1}^{r-1} C_{i,j}^{rem,cr} < t_I^{rem} \wedge \sum_{cr=1}^{r} C_{i,j}^{rem,cr} > t_I^{rem} \wedge NPR_{i,j}^{r} = 1) \\ PD_{i,j} &= \sum \Delta_{i,j}^{I}, \forall I \; s.t. \; (cond_a \wedge cond_b \wedge cond_c) \end{aligned} \tag{7.6}$$

Lemma 7.3.3 *An upper bound on the data-cache related delay experienced by job $J_{i,j}$ due to preemptions by higher-priority jobs, in the context of tasks with non-preemptive regions, is given by Equation 7.6.*

Proof Assume the end of interval I is a feasible preemption point for $J_{i,j}$. Assume conditions a) and b) are satisfied. Conditions a) and b) have been proved necessary in Section 6.4.6. Assume condition c) is not satisfied. It means that a region of $J_{i,j}$ is *guaranteed* to be *executing in a NPR*. Hence, the end of interval I *cannot be a feasible preemption point* for $J_{i,j}$. Contradiction. This proves that condition c) is also necessary.

Assume the end of interval I is a not feasible preemption point for $J_{i,j}$. Assume conditions a) and b) are satisfied. It means that $J_{i,j}$ continues to execute even after a higher-priority job is released at the end of the interval. Conditions a) and b) have been proved sufficient when tasks are completely preemptive. Assume condition c) is also satisfied. It means that $J_{i,j}$ is guaranteed *not to be in a NPR*. This is a *violation of strict-priority scheduling*. Hence, the end of interval I is a feasible preemption point for $J_{i,j}$. This proves that condition a), b) and c) are sufficient. ∎

The calculation of the waiting time that $J_{i,j}$ experiences due to lower-priority jobs executing in their NPRs at the time of release of $J_{i,j}$, in accordance with the algorithm

described in 7.2, is given by Equation 7.7.

$$wait_{i,j} = \max_{(m,n)\in lp(i,j)} C^{rem,r}_{m,n}, \forall r \; s.t. \; NPR^r_{m,n} = 1 \wedge bst^r_{m,n} < rel_{i,j} \wedge bst^r_{m,n} + C^{rem,r}_{m,n} > rel_{i,j} \quad (7.7)$$

Lemma 7.3.4 *An upper bound on the waiting time that job $J_{i,j}$ experiences due to lower-priorty jobs executing in non-preemptive regions is given by Equation 7.7.*

Proof Assume that the worst-case waiting time for $J_{i,j}$ is zero. Assume there exists at least one region r of a lower-priority job such that r is a NPR and has possibly started, but is not guaranteed to have ended, before the release of $J_{i,j}$, thus satisfying the conditions in Equation 7.7. Due to the first assumption, $J_{i,j}$ *preempts* a job that could be *executing within a NPR.* Contradiction (Violation of NPR). Hence, the waiting time for $J_{i,j}$ is greater than zero when the conditions in Equation 7.7 are satisfied. Once $J_{i,j}$ starts to wait, it has to wait until the longest possible remaining execution among lower-priority jobs that may be in a NPR is completed (represented by the max function in Equation 7.7). ∎

Proof Assume that the sum of the values produced by Equations 6.4, 7.5, 7.6 and 7.7 is not a safe upper bound on the worst-case response time of a job. This implies that the value produced by at least one of the equations *is an underestimation of the specific component* represented by the equation. Lemmas 6.4.2, 7.3.2, 7.3.3 and 7.3.4 *demonstrate the correctness of each component* of the response time of a job as a *safe upper bound.* Contradiction. Hence, the sum of the values produced by Equations 6.4, 7.5, 7.6 and 7.7 *is a safe upper bound* on the worst-case response time of the job. ∎

7.4 Experimental Results

For all experiments, task sets that have a base utilization (utilization without considering data cache related delays) of 0.5, 0.6, 0.7 and 0.8 are constructed using benchmarks from the DSPStone benchmark suite. Task sets of different sizes (2, 4, 6, 8) are constructed for each of these utilizations. For a utilization of 0.8, a task set consisting of 10 tasks is also constructed.

In the first set of experiments, response time analysis is performed using the method presented in Section 7.1 to calculate the number of preemptions and the worst-case preemption delay. Due to the fact that the benchmarks used in experiments do not

Table 7.3: Characteristics of Regions of Tasks with NPR

ID	Region 1	Region 2 (NPR)	Region 3
	WCET / BCET	WCET / BCET	WCET / BCET
5N	39371 / 38271	5084 / 2184	836 / 536
6N	39371 / 38971	10924 / 6024	5196 / 2696
7N	46771 / 44471	14224 / 7224	5196 / 2696
8N	52371 / 48771	18824 / 9624	5196 / 2696
9N	61571 / 55471	15424 / 7224	10096 / 5096
11N	33494 / 31194	5337 / 3737	17707 / 12407
12N	52294 / 43194	9647 / 4847	30297 / 14597
13N	68444 / 53344	12987 / 5587	46107 / 19007
15N	28912 / 26172	22400 / 20760	8584 / 7083
17N	32302 / 32302	9045 / 9045	2590 / 2590
18N	45802 / 45402	5845 / 4545	3190 / 2590
19N	58652 / 55352	5845 / 4545	1440 / 1240
20N	56502 / 53602	11545 / 9045	8990 / 7090
21N	69352 / 63552	11545 / 9045	7240 / 5740
22N	70152 / 64052	17245 / 13545	11840 / 9340
24N	47756 / 45956	4649 / 3549	37231 / 30031
26N	66506 / 60406	5239 / 3939	63891 / 41191
27N	59256 / 54956	20639 / 15639	78741 / 47941

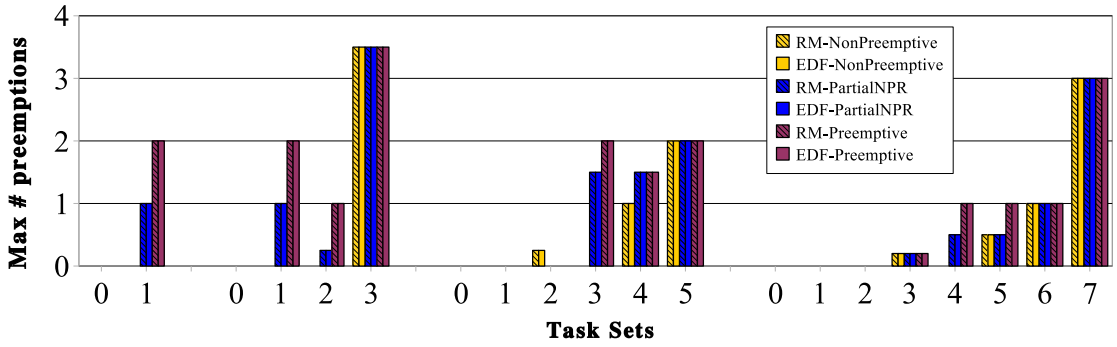
already have a NPR, an iteration range is chosen from the valid iteration range of a particular task and marked as non-preemptive. Table 7.3 shows execution times of each region as determined by the timing analyzer based on the chosen iteration ranges for a subset of the benchmarks. Since there are only have a fixed set of benchmarks, the same benchmark is used with and without NPRs in different task sets. The length of a task's NPR as a portion of its total execution time ranges from 4% to 37% in both the worst and the best cases.

The characteristics of task sets with base utilization 0.5 and 0.8 are shown in Table 7.4. The characteristics and results for utilizations 0.6 and 0.7 are similar and are hence omitted. The first column shows the tasks used in each task set. The IDs assigned to benchmarks in Table 5.1 are used to identify the tasks. If a task is chosen to have a NPR in a certain task set, the letter N is appended to its ID to indicate this fact. In this case, the WCETs and BCETs for the task are as shown in Table 7.3. Otherwise, they are as indicated in Table 5.1. The second column shows the phases of the tasks and the third column shows the periods (equal to the deadlines) of tasks. The phases of the tasks are chosen in a way to demonstrate interesting features of the analysis presented in Section 7.1.

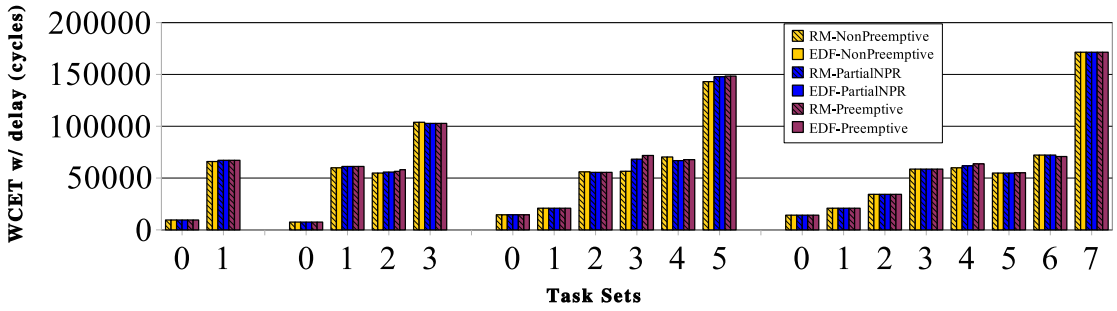
Table 7.4: Task Set Characteristics: Benchmark IDs, Phases[cycles] and Periods [cycles]

# Tasks	**2**	**4**	**6**	**8**	**10**
U = 0.5					
IDs	16, 19N	1, 15N, 18N, 22	23, 3, 6, 11N, 19, 26	2, 3, 4, 11, 15N, 18, 7, 27	
Phases	4K, 0	1K, 0, 10K, 0	32K, 32K, 32K, 0, 0, 0	0, 0, 0, 0, 0, 0, 0, 0	
Periods	50K, 200K	50K, 400K, 500K, 1000K	400K, 500K, 1000K, 1000K, 2000K	100K, 400K, 500K, 800K, 1000K, 2000K, 2000K, 4000K	
U = 0.8					
IDs	27, 26N	28, 13N, 27, 19	21, 8N, 20, 13, 25, 19	8, 26, 20, 15N, 9, 11, 8, 21	10, 8, 15, 9, 5, 11N, 20, 27, 22, 17
Phases	0, 0	54K, 0, 0, 0	49K, 0, 0, 0, 0, 0	27K, 27K, 27K, 0, 0, 0, 0, 0	32K, 32K, 32K, 32K, 32K, 0, 0, 0, 0, 0
Periods	300K, 500K	500K, 500K, 1000K, 2000K	400K, 500K, 500K, 1000K, 1000K, 2000K	400K, 500K, 800K, 800K, 1000K, 2000K, 2000K, 4000K	100K, 625K, 625K, 625K, 1000K, 1000K, 1250K, 1250K, 2500K, 5000K

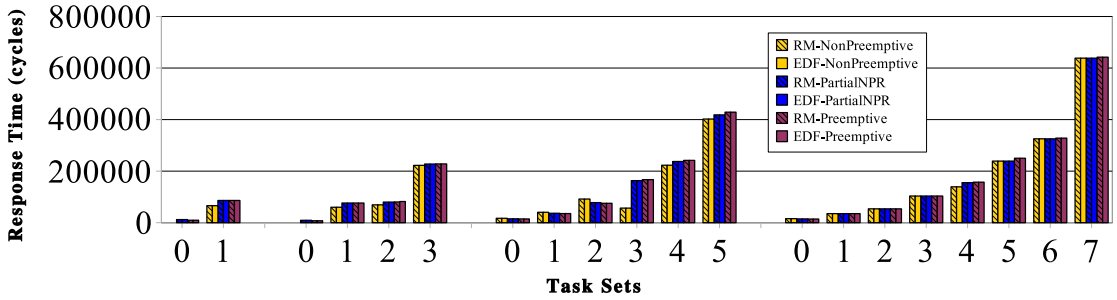
Results obtained for task sets in the above set of experiments are shown in Figures 7.7 and 7.8 for base utilizations of 0.5 and 0.8, respectively. Each graph shows the results of analysis of the same task sets using both the static Rate Monotonic (RM) scheduling policy and the dynamic Earliest Deadline First (EDF) scheduling policy. For each utilization, a separate graph shows the upper bound on the number of preemptions, the WCET with preemption delay and the response time respectively. These values form the y-axes in the graphs. In each case, the average values of these parameters over all jobs of a task are plotted. On the x-axis for each graph, the tasks used in each experiment are shown. Tasks



(a) # Preemptions for U = 0.5



(b) WCET w/ Delay for U = 0.5

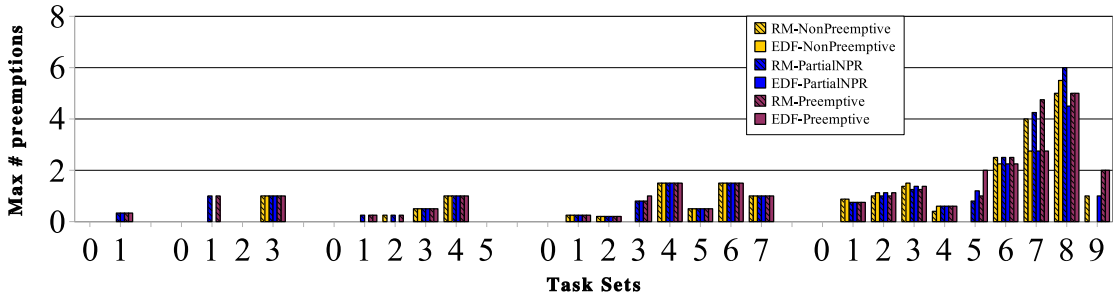


(c) Response Time for U = 0.5

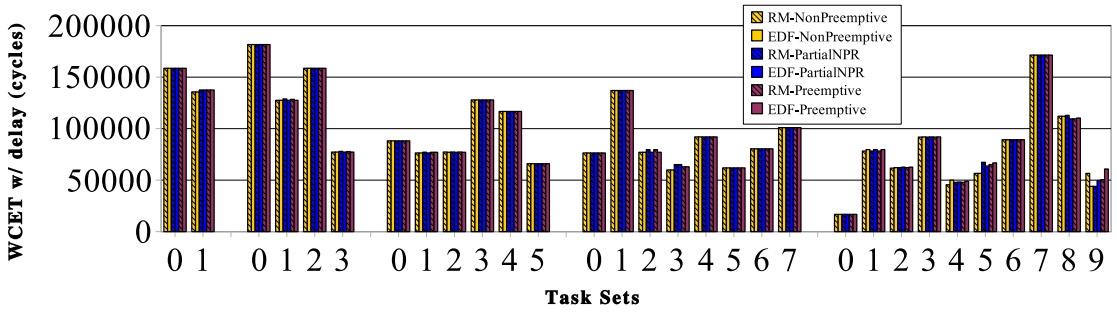
Figure 7.7: Results for U=0.5 under RM and EDF Scheduling

are grouped by task set and, within a task set, by task id starting from 0.

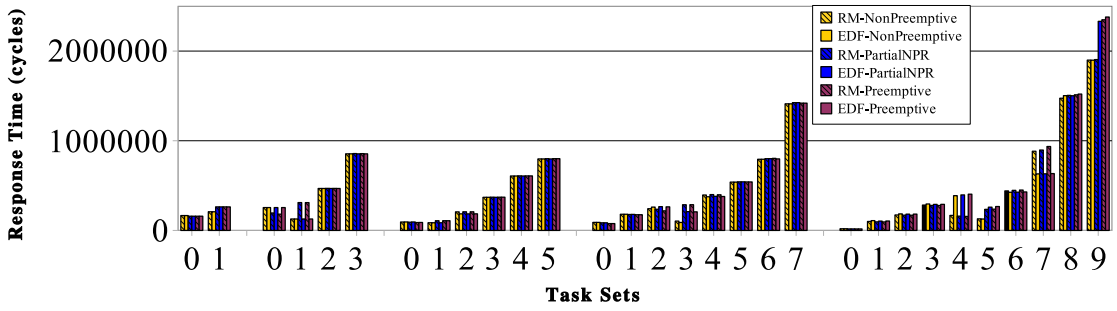
For each scheduling policy, results using three analysis techniques are presented. The first one is NPR unaware (Preemptive), in which all tasks are assumed to be completely preemptive, obtained using the analysis presented in Chapter 6. The second is a NPR-aware analysis, in which some tasks have a non-preemptive region in the middle (PartialNPR). The third analysis is a NPR-aware analysis, in which the tasks with a non-preemptive region



(a) # Preemptions for U = 0.8



(b) WCET w/ Delay for U = 0.8



(c) Response Time for U = 0.8

Figure 7.8: Results for U=0.8 under RM and EDF Scheduling

are assumed to be completely non-preemptive (NonPreemptive). The results for fully non-preemptive schedules are obtained using the algorithm described in Figures 7.3 to 7.6 and by setting the lengths of the first and third regions to zero. In these graphs, response time values for tasks that end up missing their deadlines are omitted.

At the outset, it is to be noted that, if a task is supposed to have a non-preemptive region, then forcing the task to be completely preemptive is unsafe since the results of the task could be incorrect (due to possible data races). Hence, the results of the NPR unaware

(Preemptive) analysis are unsafe as far as the tasks with NPR are concerned. It is purely for the sake of comparison that these results are presented here. On the other hand, making a task that is supposed to have a portion which is non-preemptive completely non-preemptive is conservative, yet safe.

From the graphs, several observations may be made. First of all, it may be observed that the results for the RM scheduling policy and the EDF scheduling policy are almost the same for most tasks. For RM and EDF to exhibit a difference in behavior, a task with a longer period needs to have an earlier deadline than one with shorter period somewhere in the execution timeline. This could happen in two situations, namely, when the shorter period does not divide the longer period and when there is phasing between the tasks. In most of the task sets used in the current set of experiments, neither case occurs as observable from the results. However, for a base utilization of 0.8, small differences may be observed between the two policies. Similar to observations made in Section 6.5, in some cases, the EDF policy increases the response times of tasks with shorter periods (higher priority according to the RM policy) in comparison to the RM policy. This is due to the fact that the relative deadlines of jobs alter their priorities. For the same reason, the EDF policy sometimes decreases the response times of tasks with longer periods compared to the RM policy.

For most of the task sets, it may be observed that the response time estimates obtained from the NonPreemptive analysis is shorter than those obtained from the Partial-NPR analysis. The reason for this is as follows. In the PartialNPR analysis, the following situation could occur. When a task is released, some task with a lower priority *might have* started its NPR, but *is not guaranteed* to have done so. In this situation, according to the analysis technique described in Section 7.1, the effects of contradicting worst-case scenarios for both tasks involved are considered. In other words, the worst possible scenario is assumed *for each task*. This is done in order to ensure safety of the response time estimates. In reality, however, only one of the scenarios can actually occur. In the case of the Non-Preemptive analysis, a task that has a NPR is assumed to be completely non-preemptive. Hence, a situation such as the one described above cannot occur.

On the other hand, in some task sets, the NonPreemptive analysis causes some high-priority tasks to miss their deadlines. This is because the waiting times for the high-priority tasks are now longer since the length of the non-preemptive region of a task extends to its entire execution time. This compensates, in part, for the pessimism that the Par-

tialNPR method introduces and may be observed by the fact that the actual differences between response times of tasks in the two cases are not significant.

Next, a sensitivity study is conducted using the example task set shown in Table 7.1. The same periods, phases and total execution times are maintained for all tasks. However, the length of the NPR in T_2 is varied in both the best and worst cases. Starting from a completely preemptive version of T_2, a NPR is added and extended from the middle outwards symmetrically in both directions until T_2 is completely non-preemptive. Table 7.5 shows the WCETs and BCETs of each region for different experiments. The average response times over all jobs of each task using the RM scheduling policy are shown in Figure 7.9. Response times are omitted from the graph if any job of a task misses its deadline. At one extreme, where T_2 is completely preemptive, it may be observed that the response time of T_0 is the same as its WCET since it executes to completion right after its initial release. At the other extreme, when T_2 is completely non-preemptive, it may be observed that T_0 misses its deadline due to increased waiting time. This sensitivity study demonstrates the improved schedulability of the PartialNPR analysis over the NonPreemptive analysis.

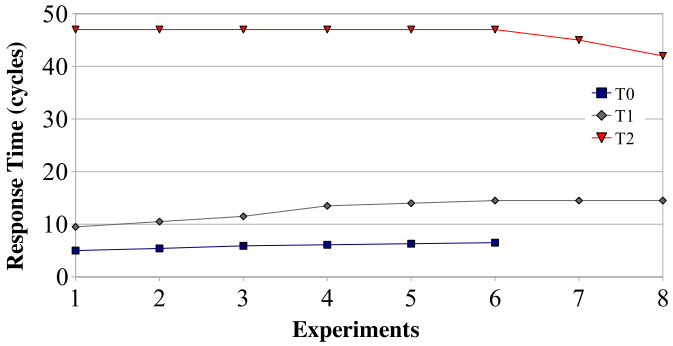


Figure 7.9: Response Times of Tasks

In summary, the techniques presented in Section 7.1 enable a study of the effects

Table 7.5: WCET/BCET Ratios for T_2

Expt. #	1	2	3	4	5	6	7	8
Region1	30/20	13/9	11/8	9/7	7/6	5/4	3/2	0/0
Region2:NPR	0/0	4/2	8/4	12/6	16/8	20/12	24/16	30/20
Region3	0/0	13/9	11/8	9/7	7/6	5/4	3/2	0/0

of having a non-preemptive region and the advantages of having partial NPRs compared to completely non-preemptive tasks in a task set. Assuming that a task is completely non-preemptive, though simpler to analyze, has the disadvantage that there is an increased probability of some high-priority task missing its deadline. On the other hand, a completely preemptive system might not be acceptable for certain kinds of tasks that inherently possess a region in which they should not be preempted in order to preserve correctness.

Chapter 8

Resource-Sharing Tasks

In Chapter 7, a methodology to analyze tasks with critical sections is presented. In that chapter, logical correctness of tasks is maintained by executing all critical sections as *non-preemptive regions (NPRs)*. Using that methodology, schedulability of task sets is improved in comparison to a fully non-preemptive scheduling policy by allowing (legal) preemptions outside the NPR. A fundamental assumption there is that a task executing in a NPR cannot be preempted by *any* higher-priority task for the entire duration of the NPR.

The need for a critical section typically arises due to access of shared resources by multiple tasks. While it is important to prevent two tasks from accessing a shared resource at the same time, it is not necessary to disallow preemptions altogether in such a critical section. In other words, although a shared resource has to be relinquished voluntarily by a task that has acquired it (making the *resource* non-preemptible), the task holding the resource may still be preempted. Several resource sharing policies have been proposed to control accesses to shared resources in the context of real-time systems. The fundamental aim of all these polices is to maintain correctness of all tasks while maximizing schedulability by reducing the waiting time for tasks that *do not use* a particular resource that has been acquired by some lower-priority task.

In this chapter, a framework that incorporates resource sharing policies within the process of estimation of worst-case response times of hard real-time tasks, in the presence of data caches, is presented.

8.1 Methodology

Currently, the Priority Inheritance Protocol (PIP) is used to manage accesses to shared resources [24]. Although the analysis technique being presented in this section can conceptually support different resource sharing policies with minor extensions to the algorithm, the discussion through the rest of this chapter is in the context of the Priority Inheritance Protocol for the sake of simplicity.

In PIP, when a task (say T_0) with a priority higher than the currently executing task (say T_1) is released, T_1 is preempted and T_0 is scheduled immediately. If T_0 later requests access to a shared resource, there are two possibilities. The resource could be available, in which case it is immediately granted to T_0. Alternatively, the resource could have been acquired by T_1 before it was preempted. In such a case, T_1 is now scheduled again and is allowed to execute until it relinquishes the required resource. For this duration of time, T_1 executes at the priority of T_0. In other words, T_1 *inherits* the priority of T_0 until the required resource is relinquished. The reason for this is to ensure that T_1 cannot be preempted by tasks with priority between those of T_0 and T_1, thus preventing a situation termed *priority inversion*.

Every task is split into multiple regions, namely regions that access some shared resource(s) and regions that do not. A total ordering is assumed among all shared resources in the system being analyzed. If a task needs multiple resources simultaneously, it requests them in accordance with the total order and releases them in the reverse order of request to avoid deadlocks in the system.

8.1.1 Motivating and Illustrative Examples

In this section, the methodology is illustrated using examples. In all examples, the deadline of a task is assumed to be equal to its period. Consider the task-set shown in Table 8.1. The first column indicates task names. The second and third columns show the phases and periods of tasks, respectively. The fourth and fifth columns show the worst-case and best-case execution times (WCET and BCET) of each of the regions of a task. In this example, every task has three regions, the second of which is the one in which resource requests are made. The sixth column indicates the name of the resource being used in the second region of a task.

Figure 8.1(a) shows the results obtained for this task set using a resource-sharing

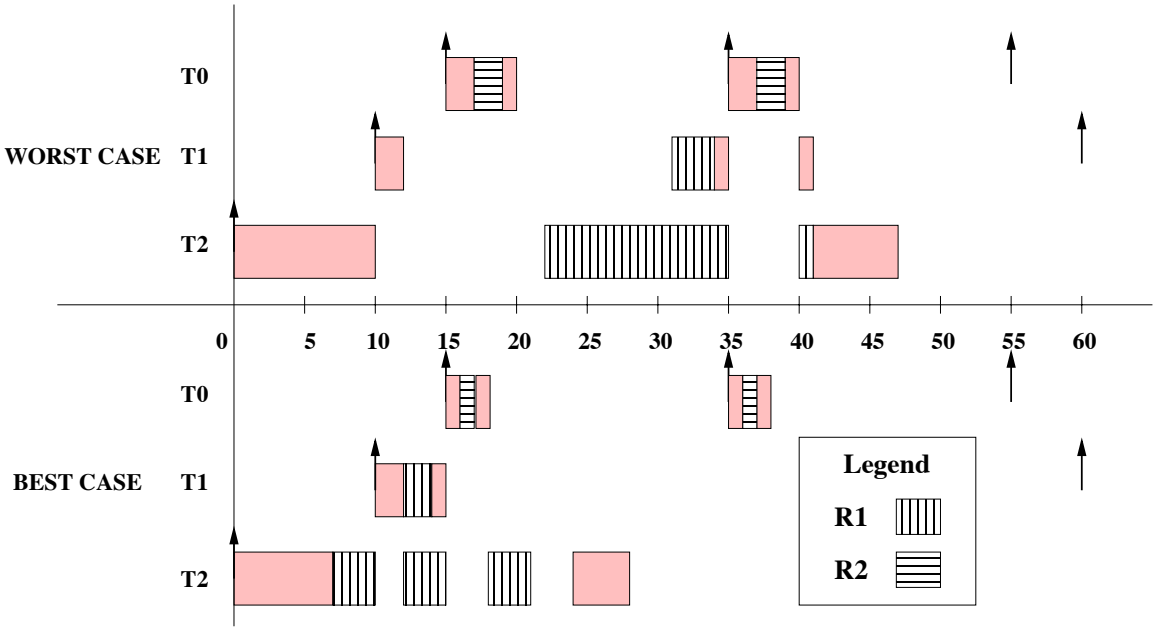
Table 8.1: Task Set Characteristics - Task Set 1 [RM policy → T_0 has Highest Priority]

Task	**Phase**	**Period = deadline**	**WCET** ($r1/r2/r3$)	**BCET** ($r1/r2/r3$)	**Resource used in region** $r2$
T_0	15	20	$2/\mathbf{2}/1$	$1/\mathbf{1}/1$	R_2
T_1	10	50	$2/\mathbf{3}/2$	$2/\mathbf{2}/1$	R_1
T_2	0	200	$10/\mathbf{14}/6$	$7/\mathbf{9}/4$	R_1

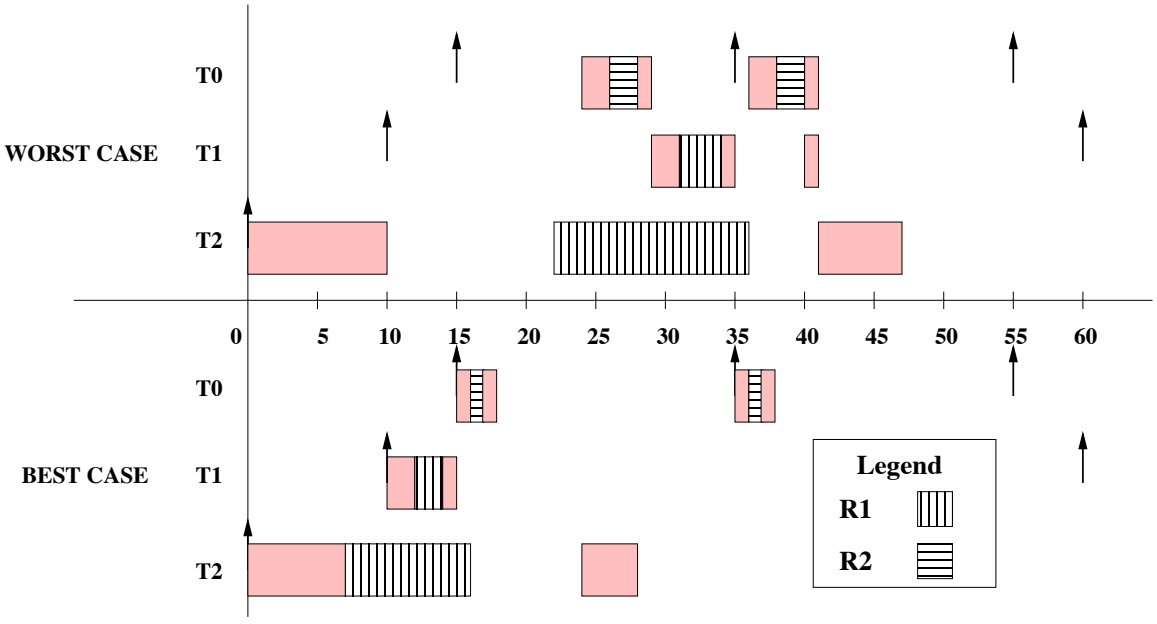
protocol, specifically the Priority Inheritence Protocol. This method will henceforth be refered to as ResourceSharingAnalysis. For the sake of comparison, for the same task-set, results obtained using the analysis technique described in Chapter 7 are also presented. That method is referred to as NPRAnalysis (equivalent to PartialNPR in the results presented in Section 7.4). It is to be noted that, in NPRAnalysis, a task executing in a critical section cannot be preempted by any other task. These results are shown in Figure 8.1(b). For the sake of simplicity, data cache related delays are assumed to be zero in these examples. Calculation of data cache related delays will be discussed in Section 8.2.

In both Figures 8.1(a) and 8.1(b), the x-axis represents time. Best-case scenarios and worst-case scenarios for *each individual task* are shown below and above the x-axis, respectively. It is important to note that the timelines do not indicate an actual schedule, but rather best and worst-case possibilities for each task. The arrows represent releases of tasks. Since the deadline of a task is assumed to be equal to the period of the task, a release of a task serves as the deadline for the previous release of the task. The shaded rectangles represent task execution in a preemptive region where no shared resource is accessed and the hatched rectangles represent task execution in a region where it accesses one or more shared resources. Different resources are depicted using different styles of hatching.

Instead of examining the entire timeline, the portions of the timeline that exhibit differences between NPRAnalysis and ResourceSharingAnalysis are examined. First, consider time 10. At this time, task T_2 is executing and the first instance of task T_1 is released. There is a possibility that T_2 has already finished executing region $r1$ and has entered region $r2$ (best case), but it is not guaranteed to be so (worst case). In NPRAnalysis (shown in Figure 8.1(b)), the assumption is that a task executing in its NPR is not preemptible by any other task. In this situation, the worst-case scenario for task T_1 is that it has to wait for task T_2 to finish executing its NPR. On the other hand, in ResourceSharingAnalysis (shown in Figure 8.1(a)), although a shared resource is not preemptible until a task volu-



(a) Results using ResourceSharingAnalysis



(b) Results using NPRAnalysis

Figure 8.1: Best and Worst Case Results for Task Set 1

tarily relinquishes it, the task itself can be preempted until some shared resource being held by it is required by a higher-priority task. Hence, in this situation, task T_1 preempts task T_2 in both the best and the worst cases.

While using ResourceSharingAnalysis (8.1(a)), at time 12, task T_1 finishes executing region $r1$ in the worst-case and enters region $r2$. On entering region $r2$, T_1 requests access to resource R_1. Since there is a possibility that T_2 has already acquired that resource (best case for T_2), the worst case for T_1 is to allow T_2 to execute and wait until the resource is relinquished. On the other hand, since there is a possibility that the resource R_1 has still not been acquired by T_2, the best case for T_1 is that it acquires R_1 immediately.

At time 15, an instance of task T_0 is released. Once again, in Figure 8.1(b), since there is a possibility that T_2 has already entered its NPR, the worst case for T_0 is for it to wait for the completion of T_2's NPR. Hence, it is scheduled to start at time 24. On the other hand, in Figure 8.1(a), since T_2 can be preempted, T_0 gets scheduled immediately in both the best and the worst cases. Furthermore, in ResourceSharingAnalysis, since T_2 never requests resource R_1, it can complete executing all its regions, resulting in a response time equal to its WCET. This example demonstrates the advantages of using a resource sharing policy as opposed to assuming that a task in a NPR is not preemptible at all.

As a second example, consider the task set whose characteristics are shown in Table 8.2. The first, second and third columns indicate the task name, phase and period of each task, respectively. The fourth and fifth columns show the WCET and BCET of each region of a task. In the sixth and seventh columns, worst-case and best-case resource request times are indicated. The eighth and ninth columns indicate the worst-case and best-case resource release times, respectively. The resource request and release times are relative to the start of the region in which they are used, namely region $r2$.

Table 8.2: Task Set Characteristics - Task Set 2 [T_0 has Highest Priority]

Task	**Phase**	**Period**	**WCET**	**BCET**	**Request**		**Release**	
			$(r1/\mathbf{r2}/r3)$	$(r1/\mathbf{r2}/r3)$	**WC**	**BC**	**WC**	**BC**
T_0	15	200	$3/\mathbf{6}/4$	$2/\mathbf{4}/2$	R_1: 0 R_2: 2	R_1: 0 R_2: 1	R_1: 6 R_2: 6	R_1: 4 R_2: 4
T_1	10	200	$5/\mathbf{8}/6$	$3/\mathbf{5}/4$	R_2: 0	R_2: 0	R_2: 8	R_2: 5
T_2	0	200	$7/\mathbf{10}/6$	$5/\mathbf{8}/4$	R_1: 0	R_1: 0	R_1: 10	R_1: 8

Figure 8.2 shows the results obtained using ResourceSharingAnalysis for the task

set shown in Table 8.2. In this example, nested resource usage occurs. Portions of the timeline shown in Figure 8.2 that illustrate the methodology being presented are discussed below.

Consider the events that occur at time 15. The first instance of task T_0 is released. Since T_0 has the highest priority, it gets scheduled immediately, preempting the currently executing task, T_1. T_0 completes execution of its first region, $r1$ (at time 12 in the best case and at time 13 in the worst case), and then requests the resource R_1. Since it is guaranteed that resource R_1 has been acquired by task T_2 in both the best and worst cases, T_0 gets blocked and T_2 gets scheduled. T_2 now executes at the priority of T_0 until it relinquishes resource R_1.

Later, task T_0 requests resource R_2 while still holding resource R_1. On the one hand, it is possible that R_2 has already been acquired by T_1, so that the worst case for T_0 is that it gets blocked for the remaining resource usage time of T_1. On the other hand, it is possible that R_2 has not yet been acquired by T_1, so that the best case for T_0 is that it obtains R_2 immediately.

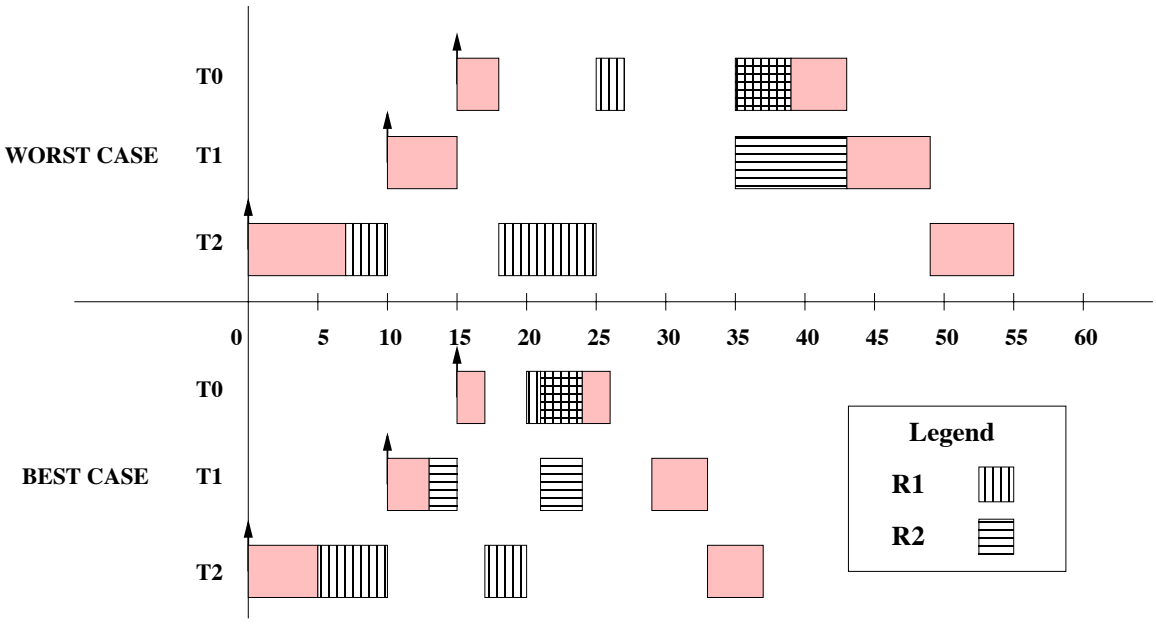


Figure 8.2: Best and Worst Case Results for Task Set 2

8.2 Data-Cache Related Delay

In the examples provided in Section 8.1.1, the data-cache related delays are assumed to be zero. In this section, calculation of data-cache related delays is described. As explained in Section 6.4.4, the data-cache related preemption delay (D-CRPD) of a task is calculated by identifying the range of iteration points at which a task is guaranteed to be within at the time that it is preempted. The delay at each point in this range is calculated using the access chains for the preempted task and the maximum among these is assumed to be the preemption delay at the given preemption point.

When a resource-sharing policy is used to control accesses to shared resources in a system, a situation may arise where a task that requests a resource is denied access to the resource because another lower-priority task has already acquired the same resource at an earlier point in time. In this situation, the task requesting the resource gets *blocked* and the lower-priority task is scheduled and allowed to execute until the required resource is relinquished by it. Since the task requesting the resource has already started its execution, there is a possibility that it loads some of its data into the data cache. When it gets blocked and lower-priority tasks are allowed to execute, some of these data cache lines may potentially be evicted from the data cache by the lower-priority tasks. Consequently, when the blocked task resumes execution at a later point in time, it experiences an additional delay to reload the evicted data cache lines into the data cache, similar to that experienced due to preemption. This delay is termed Data-Cache Related Blocking Delay (D-CRBD).

Calculations of the D-CRBD of a task are performed in a manner similar to that of D-CRPD. However, there are two distinctions. In the case of D-CRPD, the exact point of execution of the preempted task at the time of preemption is unknown. Instead, a range of iteration points where the task may be is identified. In the case of D-CRBD, since blocking occurs at the time when a resource is requested, the exact iteration point of the requesting task at the time is known.

The second distinction occurs in the identification of data cache lines that may be used by tasks that are responsible for causing the delay. In the case of D-CRPD, all cache lines used by all tasks with priority higher than the preempted task may potentially be candidates for eviction and, hence, need to be considered as such. On the other hand, in the case of D-CRBD, only the data cache lines used in specific resource-usage regions of specific tasks need to be considered.

The algorithm used to calculate the blocking time for a task that is blocked due to request for a particular resource is shown in Figure 8.3. In addition to the resource usage time remaining for the resource being requested by the task that currently holds it, nested resource usage must be taken into account. For example, assume that a task T_0 requests a resource R_1 and gets blocked on that account by task T_1. The blocking time for task T_0 includes the resource usage time remaining for R_1 by task T_1 and the blocking times that T_1 might in turn incur due to other resources that it requests while holding resource R_1. The union of data cache lines used in the regions thus identified forms the set of data cache lines that may potentially be used while task T_0 is blocked and, hence, may contribute to the D-CRBD experienced by task T_0.

```
function: calculateWaitTime(requesting_task, res, task_holding_resource, wait_time)
  wait_time ← wait_time + resource usage time remaining for task_holding_resource
  for (every resource other_res requested by task_holding_resource while holding res) {
    wait_time_res ← 0
    checkIfAvailableAndCalculateWaitTimeIfNot(task_holding_resource, other_res, wait_time_res)
    wait_time ← wait_time + wait_time_res
  }

function: checkIfAvailableAndCalculateWaitTimeIfNot(requesting_task res, wait_time)
  acquirers_bc ← tasks that have possibly acquired res in the best case
  acquirers_wc ← tasks that have possibly acquired res in the worst case
  if (requesting_task has not already waited for res) {
    if (either acquirers_bc or acquirers_wc contains tasks) {
      for (every task acquirer in acquirers_bc) {
        calculateWaitTime(requesting_task, acquirer, res, wait_time)
      }
      for (every task acquirer in acquirerswc) {
        calculateWaitTime(requesting_task, acquirer, res, wait_time)
      }
    }
  }
```

Figure 8.3: Algorithm to Calculate D-CRBD

8.3 Analysis Algorithm

An algorithm briefly describing the methodology to incorporate resource-sharing policies into the calculation of response times for real-time tasks in the presence of data caches is shown in Figures 8.4 to 8.7. The implementation of the system uses an event hierarchy similar to the one described in Sections 6.4.5 and 7.2. Six new event types are added, namely B/WCRequestResource, B/WCReleaseResource and B/WCBlocking events.

Structures global to all events are described below
bc_service_queue, wc_service_queue : queues of all released jobs that have not yet completed
Possible status : READY, WAITING_SCHED, WAITING_UNSCHED and IN_SERVICE
event_list : list of events ordered first by time and then by the priority of the type of event

Parameters for each event are described below
current_time : Time at which event occurs, *curr_job* : Job which the event corresponds to

Functions used in event handlers :
insertIntoB/WCServiceQueue(job, status)
inserts job with given status into priority-ordered service queue
removeFromB/WServiceQueue(job)
removes given job from service queue
insertIntoEventList(ev_job, ev_time, ev_type)
inserts an event of type ev_type into the event list for time ev_time and job ev_job
removeFromEventList(ev_job, ev_type)
removes events of ev_type corresponding to ev_job

JobRelease event: This event represents the release of a new job of a task
preempt, suspend : boolean values initialized to FALSE
create_event: boolean value initialized to TRUE
curr_wait, wait_time, wc_start_exec : integers initialized to 0
status : enumeration : IN_SERVICE, WAITING_SCHED, WAITING_UNSCHED and READY
Best-case and Worst-case handling:

```
If (event_queue is empty) create_event ← TRUE
else {
  for every job (q_job) in b/wc_service_queue starting from lowest priority job {
    if (curr_job has higher priority than q_job) {
      if (status of q_job is IN_SERVICE) {
        preempt ← TRUE
      } else if (status of q_job is WAITING_SCHED) {
        removeFromEventList(q_job, B/WCStartExec)
        insertIntoEventList(q_job, current_time + B/WCET of curr_job, B/WCStartExec)
      } else if (status of q_job is READY) {
        removeFromEventList(q_job, B/WCStartExec)
        if (q_job just finished one sub part and is ready to start next one) preempt ← TRUE
      }
      if (preempt == TRUE) insertIntoEventList(q_job, current_time, B/WCPreemption)
    } else {
```

Figure 8.4: Algorithm for Calculation of WCET w/ Delay for Resource-Sharing Tasks

B/WRequestResource events are responsible for carrying out the functions required to request for a resource in the best and worst case, respectively. B/WCReleaseResource events handle the release of a resource. B/WCBlocking events are responsible for carrying out the functions required when a task gets blocked due to denial of a requested resource. Once again, the actual handling of some of the events is different from that described in Sections 6.4.5 and 7.2. The events that are handled differently and the newly added events are described in Figures 8.4 to 8.7.

```
      create_event ← FALSE, status ← READY
      break from for loop
    }
  }
}
if (create_event == TRUE) insertIntoEventList(curr_job, current_time, B/WCStartExec)
insertIntoB/WCServiceQueue(curr_job, status)
```

BCPreemption event : This event represents the preemption of a task in the best case

```
removeFromEventList(curr_job, BCEndExec)
removeFromEventList(curr_job, BCRequestResource)
removeFromEventList(curr_job, BCReleaseResource)
update best case remaining time for curr_job
```

WCPreemption event : This event represents the preemption of a task
in the worst case

```
removeFromEventList(curr_job, WCStartExec)
removeFromEventList(curr_job, WCEndExec)
removeFromEventList(curr_job, WCRequestResource)
removeFromEventList(curr_job, WCReleaseResource)
if(curr_job has potentially started its NPR) {
  insertIntoEventList(curr_job, current_time + WCET of preempting task, WCStartExec)
  set status of curr_job to WAITING_SCHED in wc_service_queue
} else set status of curr_job to READY in wc_service_queue
calculate data-cache related delay for all regions of curr_job
  and update DCache Related Delay Phase
update worst case remaining time for curr_job
```

BCBlocking event : This event represents the blocking of a task in the best case

```
removeFromEventList(curr_job, BCEndExec)
removeFromEventList(curr_job, BCRequestResource)
removeFromEventList(curr_job, BCReleaseResource)
update best case remaining time for curr_job
```

WCBlocking event : This event represents the blocking of a task in the worst case

```
removeFromEventList(curr_job, WCEndExec)
removeFromEventList(curr_job, WCRequestResource)
removeFromEventList(curr_job, WCReleaseResource)
calculate data-cache related delay for all regions of curr_job
  and update DCache Related Delay Phase
update worst case remaining time for curr_job
```

Figure 8.5: Algorithm (cont.) for Calculation of WCET w/ Delay for Resource-Sharing Tasks

```
BCStartExec event : This event represents the best-case start of execution of the current region
set status of curr_job to IN_SERVICE in bc_service_queue
insertIntoEventList(curr_job, current_time + best-case remaining time of curr_job, BCEndExec)
if (current region uses resources) {
  for (each resource res required in current region in order of request) {
    event_time ← calculate best-case request time for res
    insertIntoEventList(curr_job, event_time, BCRequestResource)
  }
  for (each resource res required in current region in reverse order of request) {
    event_time ← calculate best-case release time for res
    insertIntoEventList(curr_job, event_time, BCReleaseResource)
  }
}
BCRequestResource event : This event represents the request for a resource in the best case
update best case remaining time for curr_job and request for resource required
if (resource is not acquired) insertIntoEventList(curr_job, current_time, BCBlocking)

BCReleaseResource event : This event represents the release of a resource
  in the best case
update best case remaining time for curr_job and release resource

WCStartExec event : This event represents the worst-case start of execution
  of the current region
set status of curr_job to IN_SERVICE in wc_service_queue
if (curr_job in DCache Related Delay Phase) {
  insertIntoEventList(curr_job, current_time + preemption delay, DCacheRelatedDelayPhaseEnd)
} else
  insertIntoEventList(curr_job, current_time + worst-case remaining time of curr_job, WCEndExec)
if (current region uses resources) {
  for (each resource res required in current region in order of request) {
    event_time ← calculate worst-case request time for res
    insertIntoEventList(curr_job, event_time, WCRequestResource)
  }
  for (each resource res required in current region in reverse order of request) {
    event_time ← calculate worst-case release time for res
    insertIntoEventList(curr_job, event_time, WCReleaseResource)
  }
}
```

Figure 8.6: Algorithm (cont.) for Calculation of WCET w/ Delay for Resource-Sharing Tasks

WCRequestResource event : This event represents the request for a resource in the worst case
update worst case remaining time for curr_job
for every job (q_job) in wc_service_queue {
  if (curr_job has higher priority than q_job) {
    if (status of q_job is WAITING_SCHED) {
      curr_start_time ← worst-case start time of q_job
      removeFromEventList(q_job, WCStartExec)
      insertIntoEventList(q_job, curr_start_time + worst-case time done of curr_job, WCStartExec)
    }
  }
}
if (request for resource denied) insertIntoEventList(curr_job, current_time, WCBlocking)

WCReleaseResource event : This event represents the release of a resource
  in the worst case
for every job (q_job) in wc_service_queue {
  if (curr_job has higher priority than q_job) {
    if (status of q_job is WAITING_SCHED) {
      curr_start_time ← worst-case start time of q_job
      removeFromEventList(q_job, WCStartExec)
      insertIntoEventList(q_job, curr_start_time + worst-case time done of curr_job, WCStartExec)
    }
  }
}
release resource

WCEndExec event : This event represents the worst-case end of the current region
create_event : boolean value initialized to TRUE
removeFromWCServiceQueue(curr_job)
for every job (q_job) in wc_service_queue {
  if (curr_job has higher priority than q_job) {
    if (status of q_job is WAITING_SCHED) {
      curr_start_time ← worst-case start time of q_job
      removeFromEventList(q_job, WCStartExec)
      insertIntoEventList(q_job, curr_start_time + worst-case time done of curr_job, WCStartExec)
    }
  }
}
Perform functions of NPR-Aware WCEndExec event

Figure 8.7: Algorithm (cont.) for Calculation of WCET w/ Delay for Resource-Sharing Tasks

8.4 Correctness of Analysis

The algorithm described in Section 8.3 calculates the worst-case response time of a job in the context of a given task set. Equation 5.2 shows the calculation of the worst-case response time of a job $J_{i,j}$. In the context of resource-sharing tasks, response time of a job is the sum of five components, namely the base WCET of the task, the execution time of higher-priority jobs, the D-CRPD incurred due to preemption by higher-priority jobs, the blocking time incurred due to shared resources and the D-CRBD incurred due to blocking by lower-priority jobs. The formulation for each of these components for a job $J_{i,j}$ and a proof of their correctness are presented in this section.

The new symbols introduced in this formulation are explained as follows. $\Delta_{i,j}^{R}$ represents the data-cache related delay experienced by job $J_{i,j}$ due to blocking by a lower-priority job using a resource R that is required by $J_{i,j}$. $\delta_{i,j}^{r}$ represents the delay incurred by a higher-priority job due to blocking by region r of a lower-priority job $J_{i,j}$. $Res_{i,j}$ represents the set of resources used by job $J_{i,j}$ and $Res_{i,j}^{r}$ represents the set of resources used by $J_{i,j}$ in a specific region r. $wst_{i,j}^{r}$ represents the latest possible start time for region r of job $J_{i,j}$. $bcreq_{i,j}^{R,r}$ and $wcreq_{i,j}^{R,r}$ represent the best and worst-case request times, respectively for resource R within region r of job $J_{i,j}$. Similarly, $bcrel_{i,j}^{R,r}$ and $wcrel_{i,j}^{R,r}$ represent the best and worst-case release times for R. Resource request and release times are relative to the start of the region in which they are used. $lp(i,j)$ represents the set of jobs that have a lower-priority than $J_{i,j}$. Due to the usage of resource-sharing protocols, the priority of a job may be different at different points of time. $cp_{i,j}^{t}$ represents the current priority of $J_{i,j}$ at time t and $chp(i,j)^{t}$ represents the set of jobs that have a higher priority than job $J_{i,j}$ at a time t.

Theorem 8.4.1 *The response time of a job $J_{i,j}$, calculated as the sum of the values produced by Equations 6.4, 8.5, 6.9, 8.8 and 8.10, is a safe upper bound on the worst-case response time of $J_{i,j}$ in the context of resource-sharing tasks.*

The correctness of the theorem is proved using Lemmas 6.4.2 and 6.4.4 (stated and proved in Section 6.4.6), and Lemmas 8.4.2 and 8.4.3 (stated and proved below).

The base WCET of a job does not include the effects of interference from other jobs. Hence, Lemma 6.4.2 may be reused in this context.

The execution time of higher-priority jobs within the response time of $J_{i,j}$ is cal-

culated by counting the number of instances of every higher-priority task that may execute within the response time of $J_{i,j}$ and multiplying it by the execution time of the specific higher-priority job. In the context of resource-sharing tasks, there may be some lower-priority executing at an inherited priority that is higher than $J_{i,j}$ and, hence, need to be considered as a higher-priority job. Calculation of the set of lower-priority jobs that need to be considered as higher-priority jobs is shown in Equation 8.1. Calculation of the execution time of higher-priority jobs is shown in Equation 8.2.

$$\begin{aligned} setlp_{i,j} &= \{(m,n)\}\ s.t.((m,n) \in (lp(i,j) \cap chp(i,j)^t)) \wedge (cp_{m,n}^{t-1} \neq cp_{m,n}^{t}), \\ &\quad \forall t\ s.t.\ (t > rel_{i,j}) \wedge (t < \Re_{i,j}) \end{aligned} \tag{8.1}$$

$$\begin{aligned} hpex_{i,j} &= \sum_{(k,l) \in hp(i,j)} (\lceil \frac{\Re_{i,j}}{P_k} \rceil \cdot \sum_{r=1}^{nr_k} C_{k,l}^{r}) \ + \\ &\quad (\sum_{(m,n) \in setlp_{i,j}} (\lceil \frac{\Re_{i,j}}{P_m} \rceil \cdot \sum_{r=1}^{nr_m} C_{m,n}^{rem,r}, \forall r\ s.t.\ Res_{m,n}^{r} \neq \emptyset)) \end{aligned} \tag{8.2}$$

Since the algorithm described in Section 8.3 calculates response times for every *job* in the task set, the relative phasing between jobs is known. Using this information, the calculation in Equation 8.2 is tightened. After the release of $J_{i,j}$, the time during which no other higher-priority job is released may be calculated using information about relative phasing as shown in Equation 8.3. The execution time remaining after the release of $J_{i,j}$ for any higher-priority job released *before* $J_{i,j}$ is calculated as shown in Equation 8.4. (Note: Equations 8.3 and 8.4 are the same as Equations 6.6 and 6.7, respectively and are repeated for convenience.)

$$at_{i,j} = \min_{(k,l) \in hp(i,j)} [\max(\lceil \frac{rel_{i,j} - \phi_{k,l}}{P_k} \rceil \cdot P_k, 0) + \phi_{k,l} - rel_{i,j}] \tag{8.3}$$

$$rem_{reli,j} = \sum_{(k,l) \in hp(i,j), rel_{k,l} < rel_{i,j}} C_{k,l}^{rem} \tag{8.4}$$

The difference between the times calculated in Equations 8.3 and 8.4 gives the time for which $J_{i,j}$ may execute without being preempted. Equation 8.5 shows the calculation for the new, tighter estimate on the execution time of higher-priority jobs within the response

time of $J_{i,j}$, performed in accordance with the algorithm described in Section 8.3.

$$
\begin{aligned}
hpex_{i,j} \quad = \quad & \sum_{(k,l)\in hp(i,j)} (\lceil \frac{\Re_{i,j} - \max((at_{i,j} - rem_{reli,j}), 0)}{P_k} \rceil \cdot \sum_{r=1}^{nr_k} C_{k,l}^{r}) \; + \\
& (\sum_{(m,n)\in setlp_{i,j}} (\lceil \frac{\Re_{i,j} - \max((at_{i,j} - rem_{reli,j}), 0)}{P_m} \rceil \\
& \cdot \sum_{r=1}^{nr_m} C_{m,n}^{rem,r}, \forall r \; s.t. \; Res_{m,n}^{r} \neq \emptyset)) \qquad (8.5)
\end{aligned}
$$

Lemma 8.4.2 *An upper bound on the execution time of higher-priority jobs within the response time of a job $J_{i,j}$, in the context of resource-sharing tasks, is given by Equation 8.5.*

Proof Assume that $\max((at_{i,j} - rem_{reli,j}), 0)$ is not subtracted from the iterative portion of Equation 8.5. It means that this time can be stretched due to *execution of higher-priority jobs* in between. By definition of $(at_{i,j} - rem_{reli,j})$, *all higher-priority jobs* released before $J_{i,j}$ *have completed execution* and no higher-priority jobs have been released yet after $J_{i,j}$. Contradiction. Hence, $\max((at_{i,j} - rem_{reli,j}), 0)$ can be subtracted from the iterative portion of Equation 8.5 without jeopardizing safety of the analysis. ∎

Resource-sharing tasks are fully preemptive as far as preemption delay calculations are concerned. Hence, Lemma 6.4.4 may be reused in this context.

Calculation of the set of regions within a lower-priority job that could block a higher-priority job $J_{i,j}$ requesting a resource R is shown in Equation 8.6.

$$
\begin{aligned}
setreg_{i,j}^{R} \quad = \quad & \{r\} \; s.t. R \in Res_{k,l}^{r} \wedge bst_{k,l}^{r} + bcreq_{k,l}^{R,r} < rel_{i,j} \\
& \wedge wst_{k,l}^{r} + wcrel_{k,l}^{R,r} > rel_{i,j} \qquad (8.6)
\end{aligned}
$$

The calculation of the blocking time that $J_{i,j}$ experiences due to denial of resource R, in accordance with the algorithm described in 8.3, is given by Equation 8.7 and the total blocking time for $J_{i,j}$ is given by Equation 8.8.

$$
B_{i,j}^{R} \quad = \quad \max_{(k,l)\in lp(i,j), R\in Res_{k,l}, r\in setreg_{i,j}^{R}} [C_{k,l}^{rem,r} + \sum_{R'\in Res_{k,l}^{r}, req(R')\geq req(R), rel(R')\leq rel(R)} B_{k,l}^{R'}] \qquad (8.7)
$$

$$
B_{i,j} \quad = \quad \sum_{R\in Res_{i,j}} B_{i,j}^{R} \qquad (8.8)
$$

Due to potential blocking by the regions identified in Equation 8.6, job $J_{i,j}$ experiences data-cache related delay. The calculation of the data-cache related that $J_{i,j}$ experiences due to denial of resource R is given by Equation 8.9 and the total data-cache related blocking delay for $J_{i,j}$ is given by Equation 8.10. Note that the formulae for blocking time and blocking delay are specific to the Priority Inheritance Protocol.

$$\Delta_{i,j}^{R} = \uplus_{(k,l)\in lp(i,j),R\in Res_{k,l},r\in setreg_{i,j}^{R}}[\delta_{k,l}^{r} + \sum_{R'\in Res_{k,l}^{r},req(R')\geq req(R),rel(R')\leq rel(R)} \Delta_{k,l}^{R'}] \tag{8.9}$$

$$\Delta_{i,j} = \sum_{R\in Res_{i,j}} \Delta_{i,j}^{R} \tag{8.10}$$

Lemma 8.4.3 *a) An upper bound on the blocking time that job $J_{i,j}$ experiences due to lower-priorty jobs holding resources required by $J_{i,j}$ is given by Equation 8.8.*

b) An upper bound on the data-cache related delay that job $J_{i,j}$ experiences due to all possible blocking scenarios identified using Equation 8.8 is given by Equation 8.10.

Proof Priority inheritance is transitive. If a job $J_{i,j}$ is blocked on resource R by a lower-priority job $J_{k,l}$, it is possible that $J_{k,l}$ in turn gets blocked on resource $R^{'}$ by $J_{m,n}$, which has a priority lower than $J_{k,l}$. By definition of the Priority Inheritance Protocol, $J_{m,n}$ transitively inherits the priority of $J_{i,j}$ and finishes using resource $R^{'}$. Then, it resumes its initial priority and $J_{k,l}$ executes at the priority of $J_{i,j}$ until it relinquishes R. This transitive property of PIP proves the recursive part of the calculation shown in Equations 8.7 and 8.9.

The correctness of the direct blocking time and blocking delay is now proved. Assume region r of $J_{k,l}$ directly blocks $J_{i,j}$ due to resource R.

a) $R \in Res_{k,l}$ is a necessary condition since the resource has to be used in region r in order to block.

b) Assume $bst_{k,l}^{r} + bcreq_{k,l}^{R,r} >= rel_{i,j}$. This implies that, even in the best case, resource R *has not yet been acquired* by the lower-priority job $J_{k,l}$ before the release of $J_{i,j}$. Hence, region r *of* $J_{k,l}$ *cannot directly block* $J_{i,j}$. Contradiction. Hence, $bst_{k,l}^{r} + bcreq_{k,l}^{R,r} < rel_{i,j}$.

c) Assume $wst_{k,l}^{r} + wcrel_{k,l}^{R,r} <= rel_{i,j}$. It means that, even in the worst case, resource R *has already been relinquished* by the lower-priority job $J_{k,l}$ before the release of $J_{i,j}$. Hence, region r *of* $J_{k,l}$ *cannot directly block* $J_{i,j}$. Contradiction. Hence, $wst_{k,l}^{r} + wcrel_{k,l}^{R,r} > rel_{i,j}$.

a), b) and c) demonstrate that the three conditions are necessary in order to ascertain whether a region can directly block a higher-priority job that requests a particular resource.

Assume region r of $J_{k,l}$ does not directly block $J_{i,j}$ due to resource R. Assume that all three conditions for region r in Equation 8.7 are satisfied. It means that there is a *possibility that resource R has been acquired* by $J_{k,l}$, but *no guarantee that it has been relinquished*, before the release of $J_{i,j}$. Hence, region r *of* $J_{k,l}$ *directly blocks* $J_{i,j}$. Contradiction. This proves that the three conditions specified in Equation 8.7 are sufficient to determine whether a region directly blocks a higher-priority job. Once the regions that could block job $J_{i,j}$ are identified, the data-cache related blocking delay calculation is a union of delays due to each region. ∎

Proof Assume that the sum of the values produced by Equations 6.4, 8.5, 6.9, 8.8 and 8.10 is not a safe upper bound on the worst-case response time of a job. This implies that the value produced by at least one of the equations *is an underestimation of the specific component* represented by the equation. Lemmas 6.4.2, 8.4.2, 6.4.4 and 8.4.3 *demonstrate the correctness of each component* of the response time of a job as a *safe upper bound*. Contradiction. Hence, the sum of the values produced by Equations 6.4, 8.5, 6.9, 8.8 and 8.10 *is a safe upper bound* on the worst-case response time of the job. ∎

8.5 Experimental Results

In all experiments, task sets that have a base utilization (utilization without considering data cache related delays) of 0.5 and 0.8 are used. Task sets of different sizes (2, 4, 6, 8) are constructed for both these utilizations. For a utilization of 0.8, a task set consisting of 10 tasks is also constructed.

The characteristics of the task sets constructed are shown in Table 8.3. The table indicates the task IDs, phases (cycles) and periods (cycles) of each task in the various task sets. The task IDs correspond to those assigned in Table 5.1. Task IDs that only have a single number indicate that the corresponding task does not use any shared resource. In contrast, IDs of tasks that use a shared resource are assigned a suffix of a dash followed by a number. This new ID is used to distinguish between different resource usage characteristics.

Table 8.4 shows the resource usage characteristics for tasks that use some shared

Table 8.3: Task Set Characteristics for Resource Sharing Tasks

# Tasks	**2**	**4**	**6**	**8**	**10**
			U = 0.5		
IDs	16, 19-1	1, 15-1, 18-1, 22	23-1, 3, 6, 11-1, 19, 26	2, 3, 4, 11-1, 15-1, 18, 7, 27	
Phases	45.4K, 0	40K, 47491, 0, 0	32K, 32K, 32K, 0, 0, 0	50K, 50K, 50K, 40K, 0, 0, 0, 0	
Periods	50K, 200K	50K, 400K, 500K, 1000K	100K, 400K, 500K, 1000K, 1000K, 2000K	100K, 400K, 500K, 800K, 1000K, 2000K, 2000K, 4000K	
			U = 0.8		
IDs	27-1, 26-1	28, 13-1, 27-2, 19	21-1, 8-1, 20, 13, 25, 19	8, 26, 20-1, 15-2, 9, 11, 8, 21	10-1, 8, 15, 9, 5, 11-2, 20-2, 27, 22, 17
Phases	60K, 0	100K, 70K, 0, 0	60K, 0, 0, 0, 0, 0	27K, 27K, 27K, 0, 0, 0, 0, 0	85.2K, 85.2K, 85.2K, 85.2K, 85.2K, 54K, 0, 0, 0, 0
Periods	300K, 500K	500K, 500K, 1000K, 2000K	400K, 500K, 500K, 1000K, 1000K, 2000K	400K, 500K, 800K, 800K, 1000K, 2000K, 2000K, 4000K	100K, 625K, 625K, 625K, 1000K, 1000K, 1250K, 1250K, 2500K, 5000K

resource. The first column indicates a task ID that corresponds to task IDs in Table 8.3. The second column shows the resource being used and the third and fourth columns indicate the iteration points at which the resource is requested and released, respectively. The format of the iteration point is as follows. Each pair of numbers within parantheses indicates one loop level, starting with the outermost level and proceeding inwards. Within each pair, the first number indicates the number of the loop in the current level (in case of sequential loop nests) and the second number indicates the iteration number within that loop.

Results for the task sets in Table 8.3, obtained using both the RM and the EDF scheduling policies, are shown in Figures 8.8 and 8.9, respectively. For each task set, results using two different analysis techniques are presented. The first technique is Resource-SharingAnalysis, which employs a resource-sharing protocol (specifically, the Priority In-

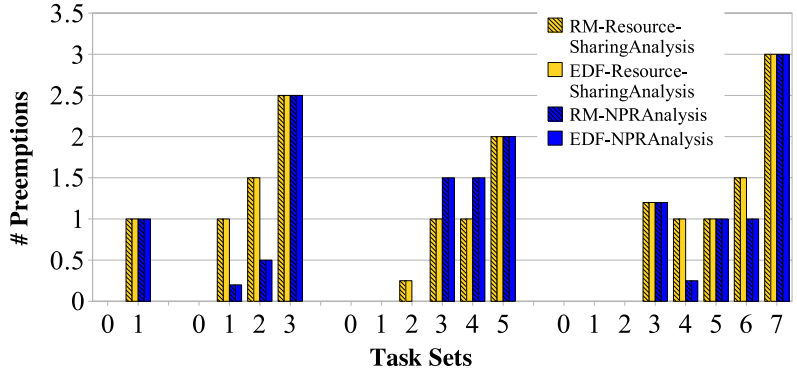
Table 8.4: Resource Usage Characteristics

Task ID	Resource	Request Iter	Release Iter
8-1	R_1	(1, 0), (2, 400)	(1, 0), (2, 600)
	R_2	(1, 0), (2, 500)	(1, 0), (2, 600)
	R_3	(1, 0), (2, 800)	(1, 0), (2, 850)
10-1	R_1	(1, 0), (2, 30)	(1, 0), (2, 50)
11-1	R_1	(1, 0), (2, 150)	(1, 0), (2, 250)
11-2	R_1	(1, 0), (2, 150)	(1, 0), (2, 200)
	R_2	(1, 0), (2, 170)	(1, 0), (2, 190)
13-1	R_2	(1, 0), (2, 300)	(1, 0), (2, 450)
15-1	R_1	(1, 0), (4, 5), (1, 5)	(1, 0), (4, 6, (1, 2)
	R_2	(1, 0), (4, 8), (1, 2)	(1, 0), (4, 9, (1, 8)
15-2	R_1	(1, 0), (4, 5), (1, 5)	(1, 0), (4, 8, (1, 8)
18-1	R_2	(1, 0), (2, 300)	(1, 0), (2, 400)
19-1	R_1	(1, 0), (2, 350)	(1, 0), (2, 500)
20-1	R_1	(1, 0), (2, 300)	(1, 0), (2, 400)
20-2	R_2	(1, 0), (2, 450)	(1, 0), (2, 500)
21-1	R_1	(1, 0), (2, 300)	(1, 0), (2, 400)
23-1	R_1	(1, 0), (2, 50)	(1, 0), (2, 75)
26-1	R_1	(1, 0), (2, 650)	(1, 0), (2, 750)
27-1	R_2	(1, 0), (2, 450)	(1, 0), (2, 650)
27-2	R_1	(1, 0), (2, 400)	(1, 0), (2, 800)
	R_2	(1, 0), (2, 650)	(1, 0), (2, 750)

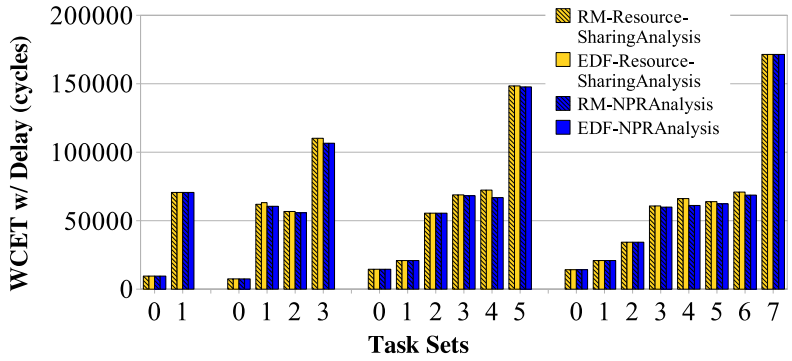
heritance Protocol) to control accesses to shared resources as described in this chapter. The second technique is NPRAnalysis (discussed in Chapter 7) and results obtained using this analysis are shown for the sake of comparison. In the case of NPRAnalysis, any region where a shared resource is used is assumed to be a non-preemptive region, *i.e.*, a region during which a task cannot be preempted by *any* other task.

The technique presented in this chapter extends from NPRs to resource-sharing protocols without loss of tightness. The method itself, bounding D-CRPD for resource-sharing tasks, is without precedence. Hence, no comparison with prior work can be presented.

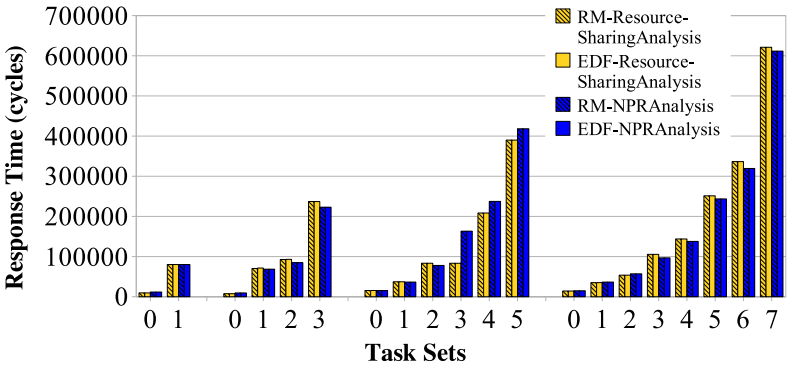
From the graphs, several observations may be made. First of all, RM and EDF exhibit little or no differences. In cases where they do exhibit differences (some task sets with utilization = 0.8), the behavior is as expected. The EDF policy sometimes increases the response times of tasks with shorter periods (higher priority according to the RM policy)



(a) # Preemptions for U = 0.5

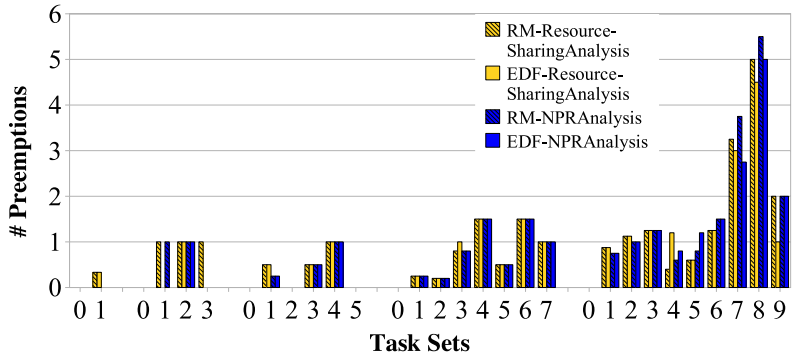


(b) WCET w/ Delay for U = 0.5

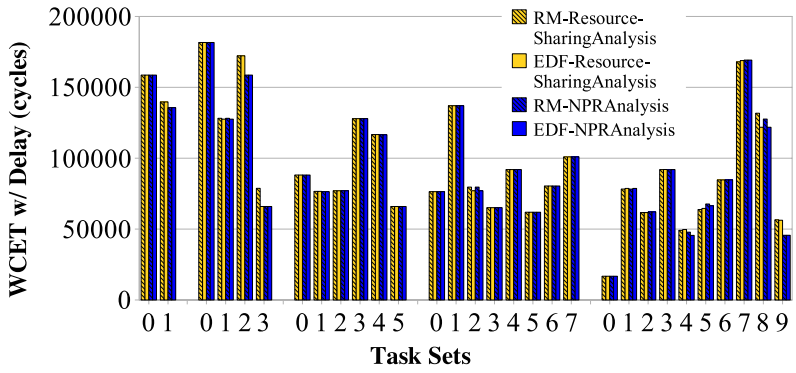


(c) Response Time for U = 0.5

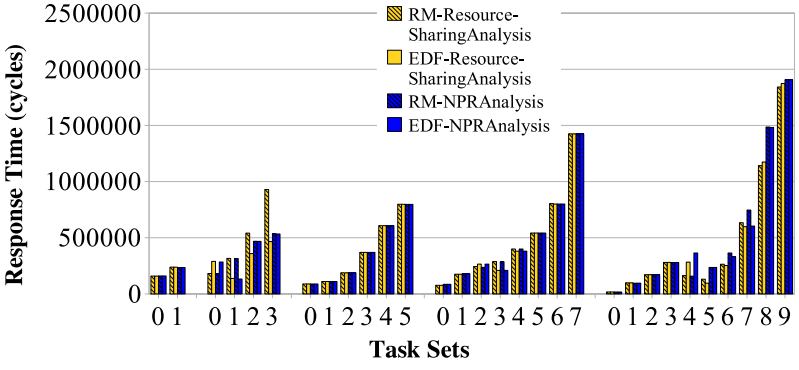
Figure 8.8: Results for U=0.5 using RM and EDF Policies



(a) # Preemptions for U = 0.8



(b) WCET w/ Delay for U = 0.8



(c) Response Time for U = 0.8

Figure 8.9: Results for U=0.8 using RM and EDF Policies

and sometimes decreases the response times of tasks with longer periods, compared to the RM policy. As explained in Section 6.5, this is due to the fact that the relative deadlines of jobs alter their priorities.

It may be observed from the graphs that tasks with a high priority sometimes have a higher response time in the case of NPRAnalysis compared to ResourceSharingAnalysis. This is expected since NPRAnalysis disallows preemptions altogether when a task is executing in a critical section and could thereby cause a delay in the start of execution of some higher-priority tasks. On the other hand, in the case of ResourceSharingAnalysis, a higher-priority does not have to wait unless (and until) it requires a resource that has been acquired by some lower-priority task.

For some higher-priority tasks, however, it may be observed that the response time is higher in the case of ResourceSharingAnalysis compared to NPRAnalysis. This is possible due to the fact that, in ResourceSharingAnalysis, a task may get blocked by a lower-priority task when it requests a resource and, consequently, may experience some data-cache related blocking delay. In the case of NPRAnalysis, although the start of execution of a higher-priority task may get *delayed* if a lower-priority task is executing in its critical section, a higher-priority task can never get *blocked* by a lower-priority task once it begins to execute. Hence, in NPRAnalysis, a task does not experience data-cache related blocking delay.

Another observation that may be made from the graphs is that some lower-priority tasks have a higher number of preemptions in the case of ResourceSharingAnalysis compared to NPRAnalysis while others have a lower number of preemptions. Although this may seem contradictory, both these results are valid. Some lower-priority task might be executing in a critical section when a higher-priority task is released. In this situation, NPRAnalysis disallows preemption of the lower-priority task whereas ResourceSharingAnalysis does not. Hence, the number of preemptions could be more in the case of ResourceSharingAnalysis. In some cases, due to relative positioning of jobs and the data-cache related delays experienced by tasks, lower-priority tasks could have a lower number of preemptions in ResourceSharingAnalysis compared to NPRAnalysis.

Based on the above observations, there is no clear answer to the question of whether using resource-sharing protocols is a better option than making critical sections completely non-preemptive when *data-cache related delays* are taken into account. The answer is dependent on the characteristics of the task set at hand. Analysis techniques, such as the ones presented in Chapter 7 and in this chapter, may be used to statically determine which

method is better suited for a given task set.

Chapter 9

Related Work

Data Cache characterization to predict the behavior of a data cache in the context of a single task has been the focus of much research in the past. While some of these approaches trade off accuracy of analysis for speed of analysis, others trade off speed of analysis for flexibility and the detail of the output information produced.

Trace-based simulators may be used to describe data cache behavior accurately for a specific input, but they are very slow and do not provide any information about the cause of the misses. Furthermore, they do not provide worst-case execution time guarantees since they work on specific inputs.

Several methods that statically bound data cache behavior have been proposed. Rawat [25] proposes a method to statically analyze data cache behavior using *interference graphs* on live-ranges of references in a program. This method does not allow global data and assumes that any execution path may potentially be the worst-case path, thus increasing the number of paths that must be explored exponentially.

Lim et al. [26] propose a method that takes data caching into account while computing the WCET for tasks. This method works for static memory references. For memory references whose addresses are unknown at compile-time, a hardware bit is set to zero, indicating that the reference must not be loaded into the data cache when the program is executed, effectively making the reference a data cache miss every time. Kim et al. [27] propose a method that classifies data references as static or dynamic. However, they do not deal with arrays or pointers.

Li et al. [28] use data flow analysis to analyze data cache behavior. White et al. [29] propose a method for direct-mapped caches. This work is based on static cache

simulation. These methods have high computational complexity due to the explosion of the data-flow state in the presence of arrays. Lundqvist et al. [30] present a study that shows to what extent data cache accesses are predictable and conclude that a majority of data cache accesses can be predicted.

Cache locking [31, 32] and cache partitioning [33] techniques have been used to make data cache behavior more predictable in real programs. In cache locking [31, 32], selected data is loaded into cache and locked in place so that it may not be replaced until the cache is explicitly unlocked. During the locked interval, since the cache contents are known, cache behavior is predictable. This approach has the disadvantage that locking and unlocking introduce some overheads. Furthermore, if the data is too large to fit into cache, it has to be completely unloaded from cache to make sure cache behavior is still predictable. This leads to performance loss. Recent work shows improvements in these methods for the case of instruction caches [34]. However, since data caches stride over large data sets, it is difficult to prevent loss in performance.

In cache partitioning, the cache is partitioned such that different portions may store different types of data [33] or store data used by different tasks in preemptive systems.

Recently, some analytical methods for predicting data cache behavior have been proposed. They include the Cache Miss Equations by Ghosh et al. [13], which has been built upon in this dissertation, a probabilistic method of analysis as proposed by Fraguella et al. [35] and another analytical method by Chatterjee et al. [36]. The basic idea behind all these methods is the same – to characterize data cache behavior by means of a set of mathematical equations. On solving these equations, information about data cache behavior may be obtained. Vera et al. have proposed analytical methods based on the Cache Miss Equations to predict data cache behavior [16, 17].

The CME framework [13], in its original form, and the probabilistic methods [35] can be used only for perfect loop-nests with no data dependent conditionals.

The formula-based method [36] is a method that models cache behavior exactly using Presburger formulae to specify cache misses. This method can also deal with multiple loop nests and conditionals. However, it has been applied only for small programs. The applicability in real programs has not been tested.

Vera et al [16, 17] build upon the cache miss equations to efficiently produce cache misses in loop-nest-oriented code. Their focus is on analysis speed and, for this, accuracy is traded off to a certain extent. In the work presented in this dissertation, the main focus

is accuracy in order to be able to supply the static timing analysis framework an accurate count of the data cache misses in a program.

Other techniques have been proposed to calculate preemption delay and analyze schedulability in a multi-task preemptive system. These techniques do not specifically analyze data cache behavior. Instead, they provide a more generic solution applicable to a cache including specific solutions for instruction caches.

Early on, Basumallick et al. conducted a survey of cache related issues in real-time systems [37]. This survey discusses some initial work related to the calculation of preemption delay. Busquets-Mataix et al. propose a method to incorporate the effect of instruction caches on response time analysis (RTA) [38]. They compare cached RTA with cached Rate Monotonic Analysis (RMA) and conclude that cached RTA outperforms cached RMA. Lee et al. propose and enhance a method to calculate an upper bound for cache related preemption delay in a real-time system [39, 40]. They use cache states at basic block boundaries and data flow analysis on the control flow graph of a task to analyze cache behavior and calculate preemption delay.

Another approach by Tomiyama et al. calculates cache related preemption delay for the program path that requires the maximum number of cache blocks [41]. This path is determined by an integer linear programming technique. In this paper, an empty cache is assumed at the beginning of every job and hence, each preemption is analyzed individually. Effects of multiple preemptions are not considered. Negi et al. combine the techniques proposed by Tomiyama et al. [41] and by Lee et al. [39, 40] to develop an enhanced framework [42]. Once again, multiple preemptions are not considered in their work since an empty cache is assumed at the beginning of a task.

The work by Lee et al. is enhanced by Staschulat et al. [22, 23]. The authors propose a complete framework for the calculation of response time for tasks in a given task set.

They address the three issues enumerated in the Section 6.1, namely calculation of the maximum number of preemption points, identification of their placement and calculation of the delay at each point. However, their focus is not on data caches, but on instruction caches.

In their work, Staschulat et al. discuss the concept of indirect preemptions [23]. Table 9.1 provides a sample task set with phase Φ, period P, WCET C and preemption delay Δ, respectively, for tasks T_1 to T_4. For simplicity, Δ is assumed to be fixed per task,

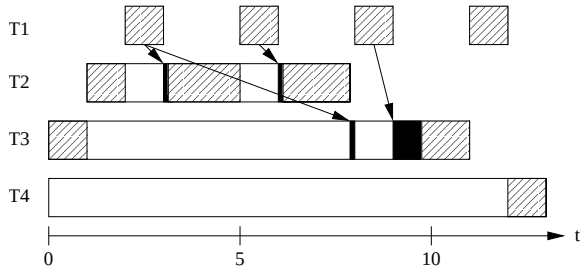


Figure 9.1: Preemption with Φ Phasing

i.e., incurred when inflicted by any higher priority task.

Table 9.1: Task Set, Optional Phasing

	Φ	P	C	Δ
T1	2	3	1	0
T2	1	15	4.625	0.125
T3	0	20	2.25	0.75
T4	0	25	1	0.125

In Figure 9.1, execution is depicted by shaded boxes and the preemption delay is depcited by black boxes. Staschulat et al. observe that several indirect preemptions affect lower priority tasks only once. For example, in the figure, T_2 is affected by the first two invocations of T_1. T_3 is actually only affected by the first and third invocations since, after being preempted once, it is not scheduled at all until T_2 completes execution. Furthermore, while incurring the delay due to preemption, T_3 is preempted again at time eight. Hence, the entire preemption cost is charged again when T_3 resumes at time nine. This results in a response time of R_3 of 11 units. It has been shown in Chapter 4 that considering indirect preemption along the lines of Staschulat et al. produces pessimistic results.

In more recent work [43], Staschulat et al. propose a timing framework that considers predictable and unpredictable (input-dependent) data cache accesses. For unpredictable accesses, a tight bound of their impact on predictable accesses and a worst-case estimate of the number of additional data cache misses is calculated. As such, their work considers any reused cache content to be replaced when a conflicting range of accesses for unpredictable data references exists, up to the number of cache blocks in either set. Alternatively, they handle cold misses for small arrays that entirely fit into cache and do not suffer replacements at all. In the work presented in this dissertation, no assumption is made about the

size of arrays. Furthermore, only predictable data accesses are analyzed. Notice that for array traversals exceeding cache size, their scheme breaks down as they assume that the entire cache has been replaced. As their schemes and those presented in this dissertation are complementary, it would be interesting to study the compatibility of these methods. However, such a study is beyond the scope of this dissertation.

There have been several pieces of work that provide schedulability analysis and tests for non-preemptive systems [44]. However, their fundamental assumption is that every task is completely non-preemptive. They do not allow any task to be partially or fully preemptive. This assumption simplifies analysis greatly but decreases schedulability of task sets. In order to increase schedulability, yet achieve lower analysis complexity, methods are proposed to "defer" preemptions to known points in time by splitting a job into several small sub-jobs and allowing preemptions only at the end of a sub-job [45, 46, 47]. Recent work by Bril et al. demonstrates flaws in these method [48, 49].

Chapter 10

Conclusion

This dissertation presents techniques that make static data cache analysis in the context of multiple tasks executing in a prioritized manner feasible for a large class of programs. The techniques presented are applicable to independent, periodic, hard-real time tasks executing on a single, in-order processor. The work presented consists of three major components.

10.1 Summary of Contributions

In the first component (Chapter 4), data cache behavior is analyzed with respect to a single task, without taking into account the effects that the execution of other tasks may have on it. In this component, an existing framework known as the Cache Miss Equations framework [13, 15] is enhanced to remove some restrictive assumptions and construct data cache reference patterns for scalar and non-scalar (array) references.

Experimental results using the enhanced framework indicate improvements in the tightness of worst-case cache behavior of one, sometimes even two orders of magnitude over the original CME approach. These results tightly and safely approximate results from trace-driven cache simulation under worst-case input. Subsequent bounds on the WCET by the timing analyzer underline the applicability of these results for end-to-end timing analysis. This work resulted in a publication [50].

The second component (Chapter 6) analyzes multiple tasks. Here, every task is assigned a unique priority and is assumed to be completely preemptive. Analysis techniques to calculate an upper bound on the number of preemptions that a task may undergo and

the worst-case preemption delay incurred at each preemption point are presented. Using these results, safe and tight upper bounds on the worst-case execution times and response times of tasks are calculated.

This component also demonstrates that the critical instant for a task set need not occur upon simultaneous release of all tasks when considering data cache related preemption delay. Hence, every job in the hyperperiod of a task set is analyzed individually.

Experimental results indicate significantly tighter bounds for (a) the number of preemptions, (b) the WCET and (c) the response time of a task compared to prior methods. The improvements are up to an order of magnitude over the prior methods. To the best of the author's knowledge, this work is novel in its contribution of a methodology to integrate data caches into preemption delay determination and in the consideration of critical instants for staggered releases of tasks. This work resulted in several publications [51, 52, 53].

The last component (Chapters 7, 8) consists of analytical techniques for tasks that may have critical sections within them. This component consists of two sub-parts. In the first sub-part, it is assumed that a task executing in a critical section cannot be preempted by any other task for the duration of the critical section. In other words, critical sections are treated as non-premptive regions. In the second sub-part, resource-sharing policies are employed to control accesses to shared resources that are used within critical sections. Here, although a resource, once acquired, has to be voluntarily relinquished by the acquiring task, the task itself may be preempted until a resource it holds is required by some higher-priority task.

Experimental results indicate that techniques in both the sub-parts demonstrate improved schedulability over completely non-preemptive systems. However, between the two methods, there is no clear answer to the question of which method is better since it depends on the characteristics of the task set being analyzed. Parts of this work resulted in a publication [54].

10.2 Future Work

This dissertation and several other works of research have proposed techniques to make caches more predictable in the context of real-time systems. However, there is still research needed before caches can be safely used in mission-critical real-time systems without requiring unnecessary pessimism in the analysis. Furthermore, several new areas

Block 1	~~**1**~~	**5**
Block 2	~~**2**~~	**8**
Block 3	~~**3**~~	**10**
Block 4	~~**4**~~	**2**

Result: M, M, M, M, M, M, M, M

(a) Condition is Not Satisfied

Block 1	~~**1**~~	**5**
Block 2	**2**	
Block 3	~~**3**~~	**10**
Block 4	~~**4**~~	**8**

Result: M, M, M, M, H, H, H, M, M, M, H

(b) Condition is Satisfied

Figure 10.1: Replacements using LRU

with real-time requirements require new methods of analysis.

10.2.1 Short-Term Goals

Throughout the work presented in this dissertation, a direct-mapped data cache has been assumed. However, set-associative caches are commonly used in practical systems. The extension of the data cache analysis to support set-associative caches is not trivial, specially in the presence of the data-dependent conditions that are allowed by the framework presented in Chapter 4.

In Section 4.2, it is assumed that an upper bound on the number of data cache misses in the presence of a data-dependent condition (with no "else" part) can be obtained by assuming that the condition *is satisfied*. While this is true in the case of direct-mapped caches, the same is not guaranteed for set-associative caches due to the presence of a replacement policy.

Consider the reference stream 1, 2, 3, 4, **4**, **3**, **2**, 5, 8, 10, 2 where each number denotes a memory line being accessed. The numbers in boldface indicate the lines that are only accessed if a certain data-dependent condition is satisfied. Assume the system that executes the program generating these references has a 4-way set-associative data cache with just one set. Further, assume that it uses the LRU (Least Recently Used) replacement policy (described in Section 2.3.2. Figure 10.1 shows the replacements that take place when the reference stream above is accessed. Figure 10.1(a) shows replacements that occur when the condition is not satisfied and Figure 10.1(b) shows those that occur when the condition is satisfied.

It may be observed from the figures that, due to the replacement policy used, the case where the condition *is not satisfied* actually has a higher number of data cache misses. This serves as a counter-example to the assumption made in Section 4.2 when it comes to set-associative caches. This also serves to show the need for research to extend the data cache analysis techniques presented in this dissertation to set-associative caches.

Data caches have some inherent difficulties that are a challenge to overcome. One such difficulty is calculating the WCET of a task in the presence of generic (if-then-else), data-dependent conditions. The cache hit/miss status of an access in a particular iteration depends on accesses in previous iterations. Assuming the worst order of accesses for every iteration in isolation need not necessarily result in overall worst-case behavior. Hence, such an assumption could jeopardize safety of the analysis. By imposing certain constraints that are reasonable in practice, analysis methods may be developed to characterize data cache behavior for programs with generic data-dependent conditionals and the development of such methods is an area of research to be explored.

10.2.2 Medium and Long-Term Goals

Cyber physical systems: Cyber physical systems are integrations of computations with physical processes that are finding an increasing number of applications in todays world. The usage of such systems in a real-time environment finds applications in assisted living, smart clothes, avionics, automotive networks, disaster response, etc. to mention a few. Dependability, efficiency and performance in computation, communication and control are primary requirements of such systems, introducing several new challenges in the areas of security and timing analysis [55].

Due to the inherent unpredictability introduced by data caches in timing analysis, several cyber-physical systems are tending towards usage of *scratchpad memory* — a portion of cache reserved for direct and private use by the CPU, *i.e.*, a software-managed cache — as a substitute [56, 57, 58, 59, 60]. A data cache has the advantage of being managed generically by hardware while scratchpad memory needs to be explicitly managed through software for each task. On the other hand, scratchpad memory has the advantage of increased predictability.

The study of the pros and cons of using data caches over using scratchpad memories and the development of new methods of analysis to make their combined usage viable in

cyber physical systems is an interesting area for future research. In order to make data cache analysis feasible for such systems, every component (computation, communication and control) of a task may be modeled as a separate sub-task with precedence constraints introduced to maintain correctness.

Multi-core systems: Starting from personal computers to the most advanced computing machines, there is movement towards multi-core architectures. Several embedded multi-core architectures are already in existence. Architects of such systems have several options for cache organization, each of which has its own advantages in terms of system performance, cost and simplicity. Adding real-time requirements to embedded multi-core systems brings to light a whole new set of challenges in program analysis. Cache characterization for data-sharing tasks now requires consideration of issues such as cache coherence and false sharing. The need for program analysis solutions for embedded multi-core architectures with real-time requirements creates an interesting avenue for research.

Bibliography

[1] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[2] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer, 1997.

[3] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Harbour. *A Practitioner's Handbook for Real-Time Analysis: Giude to Rate Monotonic Analysis for Real-Time Systems*. Kluwer, 1993.

[4] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, November 2001.

[5] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

[6] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[7] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.

[8] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[10] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.

[11] J. Handy. *The Cache Memory Book*. Academic Press, 1993.

[12] D. Patterson and J. Hennessy. *Computer Organization and Design – The Hardware / Software Interface*. Morgan Kaufmann, 1994.

[13] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[14] M. E. Wolf and M. Lam. A data locality optimizating algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[15] Nerina Bermudo and Xavier Vera. Coyote project documentation. Technical report, Mlardalen University, 2001.

[16] Xavier Vera, Josep Llosa, Antonio González, and Nerina Bermudo. A fast and accurate approach to analyze cache memory behavior (research note). *Lecture Notes in Computer Science*, 1900:194–198, 2000.

[17] Xavier Vera and Jingling Xue. Let's study whole-program cache behavior analytically. In *International Symposium on High Performance Computer Architecture*. IEEE, February 2002.

[18] C. Healy, M. Sjodin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):121–148, May 2000.

[19] V. Zivojnovic, J.M. Velarde, C. Schlager, and H. Meyr. Dspstone: A dsp-oriented benchmarking methodology. In *Signal Processing Applications and Technology*, 1994.

[20] John Lehoczky, Lui Sha, , and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium*, Santa Monica, California, December 1989.

[21] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[22] Jan Staschulat and Rolf Ernst. Multiple process execution in cache related preemption delay analysis. In *International Conference on Embedded Sofware*, 2004.

[23] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems*, 2005.

[24] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols - an approach to real-time synchronization. Technical report, Carnegie Mellon University, Departments of CS, ECE and Statistics, Pittsburgh, Pennsylvania, 1987.

[25] J. Rawat. Static analysis of cache analysis for real-time programming. Master's thesis, Iowa State University, 1993.

[26] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.

[27] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Embedded Technology and Applications Symposium*, June 1996.

[28] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.

[29] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, November 1999.

[30] Thomas Lundqvist and Per Stenström. Empirical bounds on data caching in high-performance real-time systems. Technical report, Chalmers University of Technology, 1999.

[31] Bjorn Lisper and Xavier Vera. Data cache locking for higher program predictability. In *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, March 06 2003.

[32] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, 2002.

[33] F. Mueller. Compiler support for software-based cache partitioning. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 137–145, June 1995.

[34] Isabelle Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *Euromicro Conference on Real-Time Systems*, 2006.

[35] Basilio B. Fraguela, Ramon Doallo, and Emilio L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[36] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–297, June 2001.

[37] Swagato Basumallick and Kelvin Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[38] J. V. Busquets-Matraix. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *EuroMicro Workshop on Real-Time Systems*, June 1996.

[39] 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 or cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[40] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Bounding

cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, November 2001.

[41] Hiroyuki Tomiyama and Nikil D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. *ACM International Symposium on Hardware Software Codesign*, 2000.

[42] Hemendra Sigh Negi, Tulika Mitra, and Abhaik Roychoudhury. Accurate estimation of cache-related preemption delay. *ACM International Symposium on Hardware Software Codesign*, October 2003.

[43] Jan Staschulat and Rolf Ernst. Worst case timing analysis of input dependent data cache behavior. In *Euromicro Conference on Real-Time Systems*, 2006.

[44] L. George, N. Rivierre, and M. Spuri. Pre-emptive and non-pre-emptive real-time uni-processor scheduling. Technical report, Institut National de Recherche et Informatique et en Automatique (INRIA), France, September 1996.

[45] A. Burns. Pre-emptive priority based scheduling: An appropriate engineering approach. *S. Son, editor, Advances in Real-Time Systems*, pages 225–248, 1994.

[46] A. Burns and A.J Wellings. Restricted tasking models. In *8th International Real-Time Ada Workshop*, pages 27–32, 1997.

[47] S. Lee, C.-G. Lee, M. Lee, S.L. Min, and C.-S. Kim. Limited preemptible scheduling to embrace cache memory in real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES), Lecture Notes in Computer Science (LNCS) 1474*, pages 51–64, 1998.

[48] R.J. Bril, J.J. Lukkien, and W.F.J. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. Cs-report 06-34, Technische Universiteit Eindhoven (TU/e), The Netherlands, December 2006.

[49] R.J. Bril. Existing worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption refuted. In *Work in Progress (WiP) session of the 18th Euromicro Conference on Real-Time Systems*, July 2006.

[50] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 148–157, March 2005.

[51] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 71–80, April 2006.

[52] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *IEEE Real-Time Systems Symposium*, pages 212–222, December 2006.

[53] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *To appear in ACM Transactions on Embedded Computing Systems*, 2007.

[54] H. Ramaprasad and F. Mueller. Bounding worst-case response time for tasks with non-preemptive regions. In *IEEE Real-Time Embedded Technology and Applications Symposium*, April 2008.

[55] F. Mueller. Challenges for cyber-physical systems: Security, timing analysis and soft error protection. In *National Workshop on High Confidence Software Platforms for Cyber-Physical Systems: Research Needs and Roadmap (HCSP-CPS)*, November 2007.

[56] P.R. Panda, N.D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *European Design and Test Conference*, 1997.

[57] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, pages 682–704, 2000.

[58] R. Banakar, S. Steinke, Bo-Sik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *ACM International Symposium on Hardware Software Codesign*, pages 73–78, May 2002.

[59] R. Barua O. Avissar and D. Stewart. An optimal memory allocation scheme for scratch-pad based embedded systems. *ACM Transactions on Embedded Computing Systems*, pages 6–26, November 2002.

[60] P. Marwedel L. Wehmeyer. Influence of memory hierarchies on predictability for time constrained embedded software. In *Design, Automation and Test in Europe*, March 2005.