공학박사학위논문

분산 실시간 임베디드 시스템을 위한 최악 성능 분석 기법

Worst-case Performance Analysis Techniques for Distributed Real-Time Embedded Systems

2018년 8월

서울대학교 대학원
컴퓨터공학부
최 준 철

Abstract

Worst-case Performance Analysis Techniques for Distributed Real-Time Embedded Systems

Junchul Choi
Department of Computer Science and Engineering
College of Engineering
The Graduate School
Seoul National University

For the design of embedded systems that support real-time applications, it is required to guarantee the satisfaction of real-time constraints. After applications are mapped to a candidate architecture, we check the feasibility of the architecture by estimating the performance. Fast estimation enables us to explore the wider design space of architecture selection and application mapping. More accurate estimation will reduce the system cost. It remains a challenging problem to tightly estimate the worst-case response time of an application in a distributed embedded system, especially when there are dependencies between tasks.

Multi-core systems become popular design elements not only for general-purpose systems but also for real-time embedded systems since the single-core systems reach performance limitation and do not provide scalable performance increase anymore. Multi-core systems that consist of many cores usually have shared resources such as shared L2 cache, memory, and peripheral devices. Since these resources are shared by applications executed on the multi-core systems, an application may experience non-deterministic resource access delay due to resource contention caused by simultaneous accesses from

several cores. In the worst-case response time (WCRT) analysis of multi-core systems with shared resources, non-deterministic arbitration delay due to resource contention should be considered conservatively.

Due to the continuous scaling in semiconductor manufacturing technology and their low-voltage/high-frequency operating condition, today's embedded systems operate under many reliability threats. In case of permanent faults a part of system can be permanently damaged out resulting in system crash, while some functions are temporarily out of order in the presence of transient faults. In deep submicron technology, it is known that transient faults are expected to represent the main source of errors in very-large-scale integration (VLSI) circuits. When a system is composed of several tasks with different reliability requirements, which is called a *mixed-criticality* system, tasks are typically *hardened* either by putting more hardware resources or by re-executing the defective part In order to fulfill the reliability requirements.

The behavior of the application may change in dynamic situations due to the increased complexity of applications. For instance, a video decoder switches to different behaviors according to the type of encoded input frame. A task may change its behavior according to its quality-of-service requirement. When the application has dynamic behaviors, it is required to analyze the worst-case response time of such applications.

In order to provide guarantee of real-time constraints, in this dissertation, we propose techniques that estimates the worst-case performance of an application in such distributed real-time systems. First of all, this dissertation proposes a baseline worst-case response time analysis technique, called hybrid performance analysis combining the response time analysis (RTA) technique and the scheduling time bound analysis (STBA) technique to compute a tighter bound faster. The proposed hybrid approach considers intra-graph and inter-graph interferences differently. The intra-graph interference is caused if a task is interfered or preempted by tasks in the same application, which can be estimated accurately by the STBA technique. On the other hand, the inter-graph in-

terference is caused by tasks in the other task graphs and it is estimated using a different analysis method based on the RTA.

Based on the baseline technique, this dissertation proposes three techniques that support various design aspects of the multi-core real-time embedded systems. The first technique is proposed for modeling the shared resource contention to find a tight upper bound of arbitration delay. While we compute the worst-case resource demand from each processing element based on the event stream model similarly to the reference technique, we improve the estimation accuracy significantly by accounting for the scheduling pattern of tasks. Then the proposed shared resource contention analysis is extended to support dependent tasks. To find a tight upper bound of arbitration delay, we derive a shared resource demand bound for each processing element, considering the task dependency. In addition, we study the industrial-strength engine management benchmark provided by Bosch GmbH, to declare the applicability of the proposed shared resource contention analysis.

To support fault-tolerant mixed criticality systems that has reliability constraints, this dissertation proposes a novel optimization technique with worst-case response time guarantees. Typically, in fault-tolerant multi-core systems, tasks can be *replicated* or *re-executed* in order to enhance the reliability. In addition, based on the policy of mixed-criticality scheduling, low-criticality tasks can be dropped at runtime. Such uncertainties caused by hardening and mixed-criticality scheduling make WCRT analysis very difficult. We improve the analysis in order to tighten the pessimism of WCRT estimates by considering the maximum number of faults to be *maximally* tolerated. Further, we improve the mixed-criticality scheduling by allowing partial dropping of low-criticality tasks. On top of those, we explore the design space of hardening, task-to-core mapping, and quality-of-service of the multi-core mixed-criticality systems.

The last problem addressed in this dissertation is to analyze the WCRT of an application with dynamic behaviors. It is assumed that an application is specified as a

multi-mode dataflow (MMDF) that switches to another behavior through mode transition. When a mode transition occurs during the execution of the MMDF application, the previous and the next modes may be executed at the same time, inducing a complex interference between the modes. Moreover, there may exist additional costs such as task migration cost in case a task migrates to other processor after the mode transition. The interference between modes and the task migration cost caused by mode transitions will affect the worst-case performance of the MMDF application.

Through extensive experiments with real-life benchmarks and synthetic examples, the performance of the proposed techniques is verified.

Keywords : Multi-core systems, Real-time systems, Performance analysis, Response time analysis, Worst-case response time, Partitioned scheduling, Task graph, Data dependency, Shared resource, Fault-tolerance, Mixed-criticality

Student Number : 2014-30320

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

For the design of embedded systems that support real-time applications, it is required to guarantee the satisfaction of real-time constraints. After applications are mapped to a candidate architecture, we check the feasibility of the architecture by estimating the performance. Fast estimation enables us to explore the wider design space of architecture selection and application mapping. More accurate estimation will reduce the system cost. The performance analysis problem addressed in this dissertation is to estimate the worst-case response time (WCRT) of an application that is executed on a distributed embedded system.

Despite a long history of research over two decades, it still remains a challenging problem to tightly estimate the WCRT of an application in a distributed embedded system based on a fixed priority scheduling policy. Since the response time of an application is affected by interference between applications as well as execution time variation of tasks, all possible execution scenarios should be considered to obtain the exact WCRT. There are some approaches proposed, such as a model checking approach [1] and an integer linear programming (ILP) based approach [2], to find the accurate WCRT. However, they require exponential time complexity. It is known that bounding the WCRT of an application with task dependencies and variable execution times is not trivial [3, 4, 5].

Analytical techniques have been extensively researched to obtain a tight upper bound of the WCRT with diverse assumptions on target architectures and applications. This dissertation assumes that an application is given as a directed acyclic graph (DAG) that represents data dependency between tasks and the execution time of a task may vary. It is assumed that each task has a fixed priority. In addition, we support an arbitrary mixture of preemptive and non-preemptive processing elements in the system. The first problem of this dissertation is to tightly estimate WCRTs of task graphs that are scheduled by a fixed priority preemptive or non-preemptive scheduler on distributed real-time embedded systems.

To analyze the WCRT of an application, this dissertation proposes a novel technique, called hybrid performance analysis (HPA), subsuming two analysis techniques: scheduling time bound analysis (STBA) [6] and response time analysis (RTA) [7]. The proposed technique is proven to be conservative and experimental results show that it provides a tighter bound of WCRT than the other state-of-the-art techniques. The proposed hybrid approach considers intra-graph and inter-graph interferences differently. The intra-graph interference is caused if a task is interfered or preempted by tasks in the same application. When an application modeled by a task graph runs on the multi-core distributed system, tasks in the same graph may interfere with each other. Since tasks in the same graph have an identical period and their relative starting offsets can be predicted by examining the dependency between tasks, intra-graph interference can be estimated accurately by the STBA technique. On the other hand, the inter-graph interference is caused by tasks in the other task graphs. Since applications may have different periods and dynamic starting offsets, inter-graph interference is not easy to analyze with the STBA technique. Therefore, the proposed hybrid approach uses a different analysis method based on RTA for inter-graph interference.

The second problem addressed in this dissertation is to analyze tight upper bound of the resource access delay due to resource contention from several cores. Multi-core sys-

tems become popular design elements not only for general-purpose systems but also for real-time embedded systems since the single-core systems reach performance limitation and do not provide scalable performance increase anymore. The automotive systems are representative of the emerging multi-core real-time systems. Multi-core electronic control units (ECUs) are increasingly used for complicated automotive applications such as In-Vehicle Infotainment (IVI) and Advanced Driver Assistance Systems (ADAS) whose executions can be parallelized to achieve higher performance than sequential execution on a single core. Multi-core systems that consist of many cores usually have shared resources such as shared L2 cache, memory, and peripheral devices. Since these resources are shared by applications executed on the multi-core systems, an application may experience non-deterministic resource access delay due to resource contention caused by simultaneous accesses from several cores. In this dissertation, we consider shared resources that are synchronized by hardware such as memory buses.

Even though schedulability analysis of a multi-core system gains extensive research focus recently, the inter-core interference due to shared resource contention and dependency between tasks have not been fully addressed in the previous work. Since resource contention cannot be estimated in the conventional worst-case execution time (WCET) analysis technique that considers each processor alone, it is very challenging to conservatively but accurately estimate the non-deterministic arbitration delay of shared resources.

Recently several techniques have been proposed for incorporating the shared resource contention in the schedulability analysis based on the event stream model that summarizes the resource access pattern of a task. [8][9][10]. While an earlier work considers the resource access request of each task independently [8], a recent technique estimates the worst-case arbitration delay more tightly by considering the fact that tasks cannot execute in parallel on a single processor [10]. They all assume that independent tasks are scheduled in a partitioned multi-core system, implying that each task corresponds to a sequentially executed application. On the other hand, we incorporate parallelized

execution of an application in the schedulability analysis.

This dissertation proposes an enhanced technique to find a tighter upper bound of arbitration delay for a shared resource access than the reference technique. While we compute the worst-case resource demand from each processing element based on the event stream model similarly to the reference technique, we improve the estimation accuracy significantly by accounting for the scheduling pattern of tasks. The estimated upper bound of arbitration delay is proven to be conservative and tighter than the bound obtained from the reference technique. Then we extend the proposed technique to support dependent tasks. We tightly estimate the maximum resource access delay by considering separation of task executions. Extensive experiments with synthetic examples and practical benchmarks show the performance of the proposed technique and prove the significance of considering data dependency of parallel applications for the modeling of shared resource contention delay. In addition, we study the industrial-strength engine management benchmark provided by Bosch GmbH, to declare the applicability of the proposed shared resource contention analysis.

Reliability of the application is another important issue to design a real-time embedded systems. Due to the continuous scaling in semiconductor manufacturing technology and their low-voltage/high-frequency operating condition, today's embedded systems operate under many reliability threats. There exist many kinds of faults. In device failures, a part of system can be permanently damaged out resulting in system crash. On the other hand, functional failures manifest themselves as corrupted data. Functional faults, in turn, based on the persistence of defects, faults are classified into two kinds: *permanent* and *transient* faults. In case of permanent faults the error lasts forever, while some functions are temporarily out of order in the presence of transient faults. In deep submicron technology, it is known that transient faults are expected to represent the main source of errors in very-large-scale integration (VLSI) circuits [11]. In this dissertation, we propose a design methodology of multi-core embedded systems with safety guarantee under transient

functional faults.

As can be seen in the existing safety standards such as IEC-61508 [12] or ISO-26262 [13], reliability requirements are defined as maximum tolerable probabilities of failures during operation. If a design does not fulfill the given reliability requirements, it is typically *hardened* either by putting more hardware resources (*replication*) or by re-executing the defective part (*re-execution*) [14, 15, 16, 17, 18, 19, 20, 21, 22]. That is, certain parts of a system can be redundantly executed on multiple processing elements, and the majority result out of redundantly calculated ones is taken as output to mitigate the effect of wrongly executed result. This approach is so-called *replication* or *majority voting*. On the other hand, as long as it is possible to detect the defects at the end of executions, we can re-execute the faulty task rather than using redundant processing elements. This approach is called *re-execution*. Either way, the resource usage or timing behavior of the design is considerably affected by the hardening decisions. Thus, it is compulsory to have a sophisticated analysis technique how such hardening decisions affect the design.

When a system is composed of several tasks with different reliability requirements, it is called a *mixed-criticality* system. Typical examples include avionic or automotive systems, where safety-critical functionalities such as flight or engine control coexist with non-safety-critical ones such as multimedia applications. In such systems, worst-case execution times (WCETs) of the constituent tasks are dependent upon the criticality levels of them [23]. Thus, the same task may be associated with two (or more) different WCETs. When critical tasks are executed within the lower WCET, both critical and non-critical tasks can be successfully scheduled without jeopardizing the time constraints. On the contrary, when a critical task exceeds its lower WCET, non-critical tasks are dropped and no longer guaranteed to be executed to ensure the schedulability of the critical tasks [24]. In the context of system hardening in fault-tolerant system design, re-executing safety-critical tasks has been interpreted as mixed-criticality scheduling [20, 22, 25]. That is, the original WCET of a safety-critical task is assumed to be its lower WCET, and its

execution time can reach to the higher WCET as it can be re-executed in case of fault detection.

Dropping all the non-critical tasks could be wasteful in mixed-criticality scheduling. From the perspective of maintaining service quality, it was proposed to minimize the number of dropped tasks while still keeping the schedulability of critical tasks [22, 25]. Instead of abandoning the entire execution of low-critical tasks, Huang et al. [20] proposed to degrade the service of them by enlarging their execution period. However, we observe that such alternating execution periods are not suitable in some application domains. For instances, it makes more sense to *reduce* the execution times in certain types of applications, such as computer-vision [26] or any-time algorithms [27]. Thus, this dissertation proposes to judiciously upper-bound the execution time of low-criticality tasks in critical mode.

Existing fault-tolerant system hardening techniques either assume independent tasks (without dependencies) [20, 15, 25] or simplify the WCRT analysis problem by having fixed execution time. Some works [17, 14] are free of WCRT analysis as they rely on static scheduling policy. But, in such inflexible designs, it is hard to apply the mixed-criticality scheduling with task dropping or service degradation. Kang et al. [21, 22] proposed a fault-tolerant mapping technique of task-graph or dataflow applications with WCRT guarantees. However, their WCRT estimates are too pessimistic since they assume worst-case scenario that all tasks always experience faults during executions.

In this dissertation, we tackle this pessimism in order to tighten the WCRT estimate and improve the resource efficiency in mixed-criticality multi-core embedded systems. The safety-critical applications may be hardened by conventional hardening techniques, such as re-execution and replication if necessary. On the other hand, non-critical applications can be dropped, completely or partially, in order to guarantee the schedulability of safety-critical ones. In the complete dropping, low criticality tasks that are decided to be dropped are detached from the scheduler immediately and completely as soon as a fault

is detected. On the contrary, in the partial dropping, they are still allowed to be scheduled within a given time budget.

The behavior of the application may change in dynamic situations [28] due to the increased complexity of applications. For instance, a video decoder switches to different behaviors according to the type of encoded input frame. A task may change its behavior according to its quality-of-service (QoS) requirements. The last problem addressed in this dissertation is to analyze the WCRT of an application with dynamic behaviors. It is assumed that an application specified as a multi-mode dataflow (MMDF) [29] that switches to another behavior through mode transition. When a mode transition occurs during the execution of the MMDF application, the previous and the next modes may be executed at the same time, inducing a complex interference between the modes. Moreover, there may exist additional costs such as task migration cost in case a task migrates to other processor after the mode transition. The interference between modes and the task migration cost caused by mode transitions will affect the worst-case performance of the MMDF application. This dissertation proposes a method that analyzes the worst-case response time of an MMDF application considering the MMDF mode trantision cost and interference from real-time tasks sharing the system. The performance of the proposed method is verified by experiments using benchmark applications specified as MMDF.

In summary, this dissertation covers several aspects of the distributed real-time embedded systems that should be considered in the worst-case performance analysis such as shared resource contention, fault tolerance, mixed criticality, and multi-mode applications. The proposed techniques are extended from the hybrid performance analysis, which is the baseline worst-case performance analysis technique.

1.2 Contribution

The contributions of this dissertation can be summarized as follows:

- We propose a novel WCRT estimation technique that combines the schedule time bound analysis for intra-graph interference and the response time analysis for inter-graph interference analyses, for tight and conservative estimation of the WCRT of each application separately.
  - The proposed technique supports a task graph model that represents data dependency between tasks and the execution time variation of a task. We support fixed-priority preemptive and non-preemptive partitioned scheduling.
  - We prove the conservativeness of the estimated WCRT from the proposed technique by showing that each task is always scheduled within the analyzed time bound mathematically. The convergence of the proposed technique is guaranteed since all upper bounds increase monotonically and all lower bounds decrease monotonically.
- We propose a novel modeling technique for shared resource contention to find a tight and conservative upper bound of shared resource access delay.
  - An enhanced technique is proposed to find a more accurate upper bound of arbitration delay for a shared resource access than the state-of-the-art technique when an application is modeled as a single independent task. The estimation accuracy is improved by accounting for the scheduling pattern of tasks. We prove that the estimated upper bound of arbitration delay is conservative and tighter than the reference technique.
  - We incorporate a task graph model that represents data dependency between tasks in the proposed shared resource contention analysis. We tightly estimate the maximum shared resource demand from each processing element

by identifying task clusters that have separated task executions.

- We propose a WCRT estimation technique for fault-tolerant mixed-criticality real-time systems.
  - We improve the tightness of the WCRT analysis of the mixed-criticality multi-core embedded systems by considering the probability distribution of fault occurrences.
  - To further improve the resource efficiency, we allow the non-critical applications to be partially dropped in the proposed mixed-criticality scheduling.
- We propose a WCRT estimation technique for an MMDF application that shares the system with real-time tasks.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 provides a review on the existing research and a brief overview of the limitations on the state-of-the-art techniques. The baseline performance analysis technique is explained in Chapter 3. How to analyze the shared resource contention is given in Chapter 4 and we introduce the optimization technique for fault-tolerant mixed-criticality systems in Chapter 5. WCRT analysis of an MMDF application is followed in Chapter 6. Finally, we summarize the proposed techniques and suggest some future works in Chapter 7.

Chapter 2

Background and Existing Research

2.1 Worst-case Performance Analysis

Partitioned scheduling and global scheduling are two representative policies when running multiple real-time applications on a multicore system. When a partitioned scheduling policy is used, tasks are statically mapped to cores and the task-to-core mapping is not changed at run-time. Hence the schedulability of a partitioned multi-core system can be analyzed by analyzing each core separately with the traditional schedulability analysis for a single-core system if shared resource contention does not exist. Many issues such as arbitrary deadline [30, 31], jitter of release times [32], and dynamic offset [33] have been investigated for schedulability analysis of fixed priority preemptive processors and several extensions have been proposed to support non-preemptive scheduling policy [34] and precedence constraints [35], and so on.

On the other hand, task-to-core mapping can be changed at run-time when global scheduling is used. The response time analysis of sporadic tasks for globally scheduled multi-core systems is researched in [36], and extended to support arbitrary deadline [37] and non-preemptive scheduling policy [38]. Even though global scheduling has received significant attention recently, partitioned scheduling is still regarded as a practical scheduling policy for multi-core embedded systems since a global scheduling system accompanies non-negligible task migration overhead, scheduling overhead, and shared

resource contention. In this dissertation, we are concerned about partitioned multi-core systems with shared resources.

The related work for WCRT estimation is mostly based on the response time analysis. Response time analysis (RTA) was first introduced for a single processor system based on preemptive scheduling of independent tasks that have fixed priorities, fixed execution times, and relative deadline constraints equal to their periods [7]. Extensive research efforts [30, 32] have been performed to release the restricted assumptions. Pioneered by K. Tindell et al. [31], a group of researchers extended the schedulability analysis technique to distributed systems; for example, supporting dynamic offset of tasks [33], communication scheduling [39], partitioned scheduling with shared resources [10], and earliest deadline first (EDF) scheduling [40].

There exist some researches that consider precedence constraints between tasks. A popular approach is to transform each node of synchronous dataflow (SDF) graphs into a set of independent periodic real-time tasks and apply existing real-time scheduling techniques [41, 42, 43, 44, 45]. In this approach, each real-time task is given a deadline that is conservatively estimated from the deadline constraint of the SDF application. To keep the data-dependency between nodes in the original graph, each transformed task is assigned a starting offset that is later than the deadlines of its predecessors. Note that the deadline of each task should be conservatively set to keep the data dependency between tasks regardless of varying execution times of tasks. Then it may incur deadline violation if the given mapping and scheduling result is applied. Thus this approach changes the order by transforming the graph into independent real-time tasks first and determining the mapping and scheduling of tasks next, which is different from the proposed approach. If the mapping and scheduling order is given a priori, this approach tends to produce longer response time and excessive buffer requirement as discussed with the experimental results later.

Another approach is to extend the response time analysis considering the task depen-

dency directly. An offset-based response time analysis technique for linear transactions has been proposed in [35], which is implemented in the MAST suite [46]. It supports both preemptive and non-preemptive processing elements. But it supports only chain-structured graphs where a task has a single input and/or a single output port. [47] extends the previous approach to consider the timing correlations between task activation in the tree-structured graphs. The authors in [48] propose a response time analysis technique for homogeneous synchronous dataflow (HSDF). They reduce the pessimism on the amount of interference by considering the precedence constraints and the maximum number of task executions during the busy window. The authors in [49] consider synchronously or asynchronously activated linear transactions, and optimize the WCRT of each sub-transaction, called task chain, by considering deferrable task executions due to the interleaved priority assignment. The aforementioned approaches all aim to minimize the pessimism of the response time analysis by considering the task dependency. [50] proposed an RTA-based approximation technique for a digraph real-time task model. Their task model is different from ours in that edges in a digraph represents *possible* execution flows between tasks and a single flow is taken among them at run-time. The minimum request interval should be defined between two connected tasks and the parent task execution should finish before the interval, while no interval exists in our model.

Distinguished from the above holistic RTA-based approaches, a compositional approach has been proposed and implemented in SymTA/S [51] where the RTA analysis is performed for each processing element separately and communication between processing components is abstracted by event streams that are characterized by period, jitter, and minimum distance between events. While the compositional approach achieves scalability, it sacrifices estimation accuracy by ignoring the release time constraints caused by data dependencies between tasks running on different processing elements.

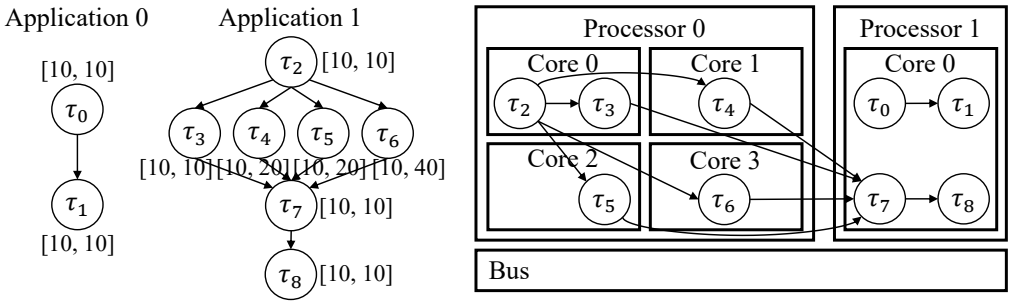
Recently, a holistic WCRT analysis approach, called scheduling time bound analysis (STBA), has been proposed [6]. It computes the conservative time bound for each task

within which the task will be scheduled, considering all possible scheduling patterns assuming that the starting offsets of applications are known and fixed. The STBA approach compares the time bounds of task instances to check the possibility of interference, expanding the task graphs up to the hyper-period of applications. In case the starting offset of an application varies dynamically, the schedule time bounds of each task become wider, which degrades the estimation accuracy significantly despite increased analysis time complexity.

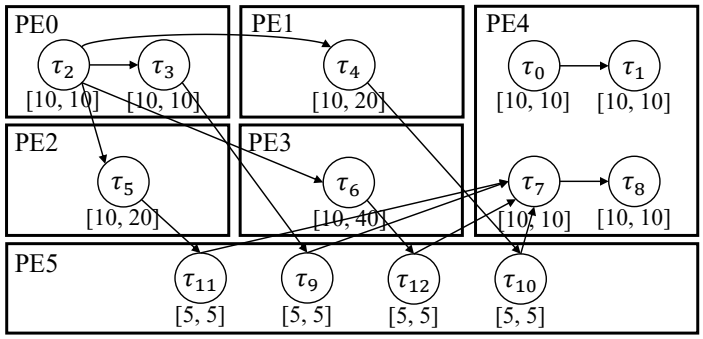
In the proposed technique, we adopt the idea of the STBA method to analyze the schedule time bounds of tasks in the same task graph. Instead of applying the STBA idea to multiple applications with dynamic offset, however, we adopt the response time analysis technique to consider the interference from other task graphs to overcome the limitations of the STBA technique.

We show the proposed approach with a simple example with two task graphs in Figure 2.1 (a). Application 0 consists of two tasks τ_0 and τ_1 which are executed sequentially, and Application 1 consists of 7 tasks from τ_2 to τ_8. The execution time bound is specified as [BCET, WCET] on each task where BCET and WCET denote the best-case execution time and the worst-case execution time, respectively. The example assumes that a system has two processors; Processor 0 has four cores and processor 1 has a single core. The processors are connected by a non-preemptive bus. It is assumed that task-to-core mapping is given as Figure 2.1 (a). The data produced from τ_3, τ_4, τ_5, and τ_6 should be transferred to τ_7 through the bus. Since the data transfer on the bus can be regarded as a task whose execution time is the transfer time, we model all cores and buses as processing elements (PEs). Figure 2.1 (b) shows the transformed analysis model corresponding to Figure 2.1 (a). In the transformed model, four tasks τ_9, τ_{10}, τ_{11}, and τ_{12} representing the data transfer from τ_3, τ_4, τ_5, and τ_6 respectively, are added.

Due to the varying execution time of tasks τ_4, τ_5, and τ_6, the data transfer tasks may contend on the bus, which incurs intra-graph interference. The worst-case intra-



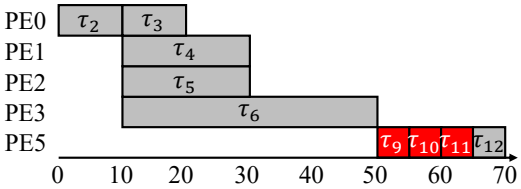
(a) Two example applications modeled by task graphs and task-to-core mapping onto an example distributed real-time system



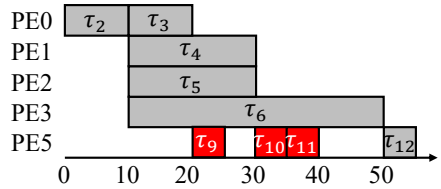
(b) Transformed analysis model with five processing elements (PEs) and additional tasks τ_9, τ_{10}, τ_{11}, and τ_{12} that represent data transfer on the bus

Figure 2.1: An example distributed real-time system and the analysis model

graph interference to the task τ_{12} can be naively estimated as 15 since there is no direct topological dependency between the tasks mapped on the bus ($PE5$). Then, the worst-case finish time of τ_{12} becomes $70(= 10+40+15+5)$ as displayed in Figure 2.2 (a). In reality, however, tasks τ_9, τ_{10}, and τ_{11} always finish before the release time of τ_{12} if τ_{12} is released at 50. STBA approach is effective for this case. It analyzes schedule time bounds considering the possible interference from other job instance. The table in Figure 2.2 (c) summarizes the time ranges that tasks in Application 1 are released or finished. Since τ_2 has a fixed execution time, 10, τ_3, τ_4, τ_5, and τ_6 are all released at 10 which is the finish time of their predecessor task τ_2. Unlike τ_3, τ_4, τ_5, and τ_6 that have fixed release times, τ_9, τ_{10}, τ_{11}, and τ_{12} are released at the varying finish time of their parents, τ_3, τ_4, τ_5, and τ_6 respectively since the parents have varying execution times. For the minimum finish time computation of a task, STBA method considers the best-case scenario that has



(a) Overestimated contention delay for τ_{12}



(b) Exact worst-case finish time of τ_{12}

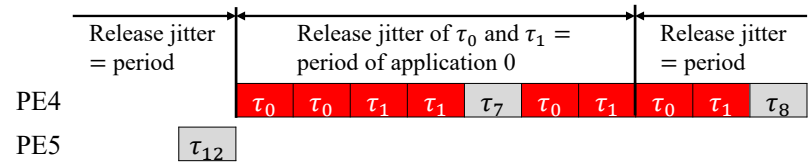
	Release time bound	Finish time bound		Release time bound	Finish time bound
τ_3	(10, 10)	(20, 20)	τ_9	(20, 20)	(25, 25)
τ_4	(10, 10)	(20, 30)	τ_{10}	(20, 30)	(30, 40)
τ_5	(10, 10)	(20, 30)	τ_{11}	(20, 30)	(30, 45)
τ_6	(10, 10)	(20, 50)	τ_{12}	(20, 50)	(30, 55)

(c) Release time bounds and finish time bounds

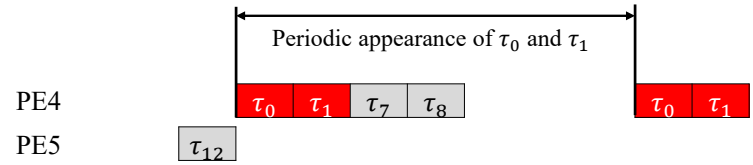
Figure 2.2: An example of intra-graph interference estimation

minimum interference from the other tasks while it has the minimum execution time. For maximum finish time computation, STBA method considers the worst-case scenario that experiences the maximum interference from the other tasks and it takes the maximum execution time. Since the finish time bounds of τ_9, τ_{10}, and τ_{11} are no larger than 50, the actual worst-case finish time of τ_{12} is 55 as shown in Figure 2.2 (b). This example shows that we can compute intra-graph interference more accurately if we adopt STBA approach that uses the schedule time bounds of tasks with a given mapping information.

For the worst-case response time estimation of Application 1, the inter-graph interference from Application 0 that has a dynamic offset should also be considered. Unfortunately, in the STBA technique, a dynamic offset is modeled as a release jitter that can be as large as the period. If an application has a large jitter, multiple jobs of the same task can be released in sequence, which is illustrated in Figure 2.3 (a) where two jobs of τ_0 and τ_1 preempt τ_7 and τ_8, respectively. In the proposed technique, we separately use an RTA-based technique to account for the inter-graph interference as shown in Figure 2.3 (b) where τ_0 and τ_1 preempt τ_7 and τ_8 only once.



(a) Overestimation when dynamic offset is modeled by a large release jitter



(b) Actual worst-case response time of τ_8

Figure 2.3: An example of inter-graph interference estimation

2.2 Shared Resource Contention Analysis

When the cores in a multi-core system share resources such as cache, memory, and peripheral devices, the system needs a protocol to synchronize and protect the resource accesses. Two representative priority-based protocols are a suspension-based locking protocol, Multiprocessor Priority Ceiling Protocol (MPCP) [8], and a spin-based locking protocol, Multi-core Locking Protocol (MLP) [9]. A suspension-based protocol shows good performance when the section for synchronization is long so that paying context switching cost is more beneficial than waiting until the resource is released. An empirical evaluation of the performance differences between spin-based and suspension-based synchronization alternatives for multiprocessor systems is presented in [52]. [53] further considered two first-in-first-out(FIFO)-based locking protocols, FIFO Multiprocessor Locking Protocol (FMLP$^+$) and Distributed FIFO Locking Protocol (DFLP), and compares the performance of four protocols with linear-programming-based blocking analysis. Since we focus on the shared resources such as shared memory which has relatively short access duration, we choose a spin-based locking protocol in our system model.

Various approaches have been proposed to take account of interference on the shared

resources. [54] proposed an approach that quantifies the slowdown due to shared resource contention caused in the Commercial Off-The-Shelf (COTS) processors. They measured the slowdown by executing resource stressing benchmarks simultaneously with a task of interest. Although the approach has an advantage that the benchmarks can be used in any COTS architectures, it does not guarantee the conservativeness since it is based on the measurement.

A group of approaches aims to design a software protocol or a resource protocol to make real-time systems more predictable. [55] suggested four spin locks with different queuing policies for AUTOSAR spin lock API and empirically compared spin locks with experiments. A framework of OS-level techniques for COTS architectures is proposed to regard a multi-core platform as a set of equivalent single-core systems [56]. This approach allows per-core schedulability analysis in isolation. [57] presents cache management policy for mixed criticality systems to make the worst-case execution time computations more predictable. The response time analysis based on the Time Division Multiple Access (TDMA) protocol proposed in [58] considers shared resources easily. TDMA protocol simplifies the timing analysis since the response time of a resource access can be computed without considering the accesses from other cores, but may not be efficient in the view of resource utilization. These approaches commonly have a restriction that a system should be designed to be applicable for their approaches.

Recently, several approaches have been proposed to consider the shared resource contention in various multi-core system models. [59] analyzed the worst-case latency of shared DRAM in COTS architectures by considering DDR3 SDRAM architecture design in detail. The worst-case resource access delay was computed by considering the cache states for the systems with a memory hierarchy in [60] where a memory access is passed from local memory such as cache and scratchpad memory to global memory connected by memory bus. A cache hierarchy that consists of a private L1 cache and a shared L2 cache was considered in [61]. Both approaches of [60] and [61] suffer from huge time

complexity since all data references should be analyzed considering the cache states and replacement policy.

On the other hand, the approaches in [8] and [9] use the event stream model to express resource access patterns at a high level, assuming a fixed priority preemptive scheduling policy and a fixed priority non-preemptive policy, respectively. The former work was extended to improve the estimation accuracy in [10]. Since it does not consider the scheduling pattern of tasks, it still produces a loose bound over a certain interval of time window. [62] proposes an algorithm that computes the maximum amount of resource request during any time window of size Δt according to the idea presented in [10]. However, their technique assumes that the tasks are scheduled by a non-preemptive scheduling policy and task deadline may not be greater than period. Its extension was presented [63] under the same assumption. Differently from this work, we make no assumption on the deadline constraint and assume a preemptive scheduling policy.

We briefly discuss the limitation of the reference technique [10] and present the key idea of the proposed technique. In the system model of interest, resource accesses are arbitrated by a non-preemptive scheduler with priorities of the issuing tasks. Since interference from lower priority tasks can be computed easily, the resource demand from a set of higher priority tasks mapped on other PEs should be considered to compute the resource access delay that a task experiences during its execution.

For a set of all higher priority tasks in another PE, the upper bound of resource demand can be conservatively computed by summing up every resource demand of each task. Figure 2.4 shows the task execution profiles that produce the worst-case resource demands for two tasks τ_0 and τ_1 where τ_0 and τ_1 generate 2 and 3 resource accesses in one job execution, respectively. The graph below the execution profiles in Figure 2.4 shows the resource demand function $D^q_{T,s}(\Delta t)$ that gives the maximum resource demand from tasks in a set T. For instance, τ_0 generates 3 memory accesses in any time window of size 4. The reference technique computes the resource demand from PE_0 as sum of all re-

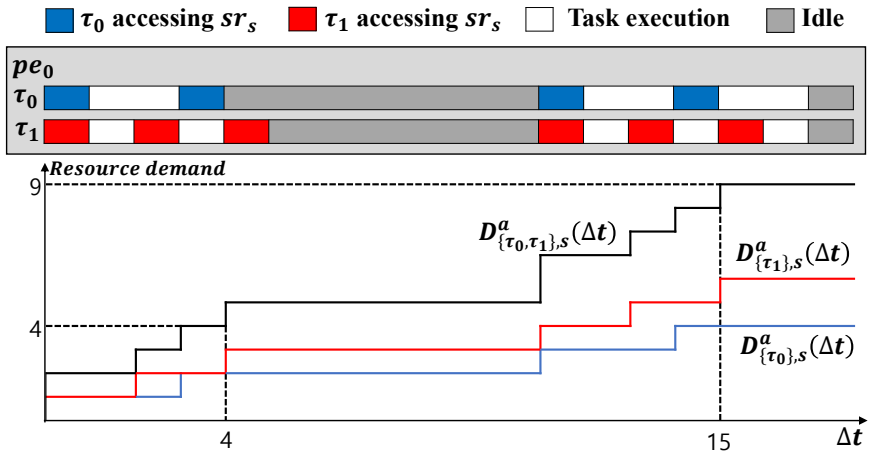


Figure 2.4: Execution profile and resource demand curve of two tasks in PE_0 assuming tasks can be executed in parallel

source demand functions of tasks in that PE, τ_0 and τ_1, as $D^a_{\{\tau_0,\tau_1\},s}(\Delta t)$ in Figure 2.4. The problem is that summing all resource demands results in a loose bound since it assumes all tasks can run in parallel on a single processor. For instance, $D^a_{\{\tau_0,\tau_1\},s}(4) = 5$ which means that maximum 5 resource accesses are issued from PE_0 during any time window of size 4. However, more than 3 resource accesses are impossible in actual schedule. To compensate this drawback, the reference technique [10] defines another resource request bound function that distributes a time window Δt to higher priority tasks in the same PE. If each task executes fully in the given time budget and generates resource demands maximally, the total resource demands are maximized for the time distribution. Then they find out the worst-case time distribution of the time window to maximize the resource demand for a given time window Δt. This approach allows to avoid concurrent execution of two tasks, but the resource demand bound is still overestimated since both functions assume impossible scheduling scenario of tasks. In this dissertation, we propose an enhanced technique that considers the possible task schedules, especially the maximum execution time and the maximum number of task instances in a given time window Δt.

A few approaches have been proposed to consider task dependency to some ex-

tent. The technique proposed in [64] divides a task into several super-blocks that have different resource access patterns. Though super-blocks can be regarded as dependent sub-tasks, they assume simple scheduling policy that is static time slot assignment. There is a technique that analyzes the worst-case cache access scenario of parallelized application modeled by Message Sequence Graph (MSG) [61]. This approach has huge time complexity since all cache references are analyzed considering the cache states and replacement policy. In addition, it assumes that the cache access behaviors are known and finite. Compared with those approaches, we use more general models, an event stream model for resource access and a task graph model for dependent tasks, in order to support a wide range of resource access patterns and parallelized execution of an application.

2.3 Fault Tolerant Mixed Criticality Systems

In the past decade, mixed-criticality systems have been actively studied in real-time systems domain. Vestal [65] proposed a fixed-priority preemptive mixed-criticality scheduling, whose response time analysis is proposed by Baruah et al. [66]. This analysis technique was further extended considering variable initiation period [67] and processor speed drop [68]. Jan et al. [69] proposed an earliest deadline first (EDF) based mixed-criticality scheduling, where low-criticality tasks are scheduled only within slacks generated by the scheduling result of high-criticality tasks. All of the above-mentioned techniques supported only independent task models without execution dependencies and no fault-tolerance was considered.

Recent research in fault-tolerant mixed-criticality scheduling is being actively pursued. Thekkilakattil et al. [70] proposed a response time analysis technique under the assumption that high-criticality tasks re-execute when faults detected. An Integer Linear Programming (ILP) based approach was proposed by Thekkilakattil et al. [71] to minimize resource usage, assuming transient faults are mitigated by re-execution. Al-bayati et al. [72] classified the system states into four modes based on whether faults occur

(or not) and whether tasks overrun (or not). For each of those four states, they provided an efficient analysis. A typical problem of such mixed-criticality scheduling techniques is that the schedulability of the system can only be achieved at the cost of excessive sacrifice of low-criticality tasks' execution. In order to mitigate such side effects, Lin et al. [73] presented a heuristic algorithm that maximizes successful (re-)executions of low criticality tasks in EDF scheduling. Huang et al. [20] considered service degradation, instead of completely dropping the low-criticality tasks, to improve the quality-of-service. Hu et al. [74] allowed to defer the dropping of low-criticality tasks, if possible, by monitoring runtime behavior of the system. Zeng et al. [25] also proposed a multi-core mixed-criticality scheduling, where different cores are allowed to start dropping at different moments. The above-mentioned approaches, however, are limited in that they only considered independent tasks, where response-time analysis is simpler.

There have been many approaches proposed to assure reliable execution despite the presence of transient faults. Traditionally, fault tolerance was achieved by replicating the hardware components [75]. However, in such a naive hardware duplication approach, the system cost becomes prohibitively big due to the continuous increase of fault occurrence frequency. On the other hand, hardening techniques such as check-pointing and roll-back [76, 24] or re-execution/replication [15, 19, 14, 18, 17, 16, 21, 20] have been proposed to tolerate transient faults by paying additional time overhead in a given limited resources. Some approaches [77] focused on the fault-tolerant mapping optimization problem of selecting the optimal fault-tolerant techniques for real-time applications modeled as task graphs, but they assume a static non-preemptive scheduling approach that fixes the start times of processes. Although this approach simplifies the guarantee of the worst-case performance, the run-time system utilization may be degraded.

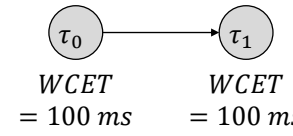
Kang et al. [21] proposed a fault-tolerant mapping technique of task graphs that considers re-execution, active replication, and passive replication as hardening policies. They also extended their technique to consider the effects of task dropping and proposed

a static mapping optimization with WCRT guarantees [22]. Although their approach is general in supporting the task graph model and many scheduling policies, the design cost could be unnecessarily big due to the pessimistically estimated WCRT as will be shown in what follows.

Suppose an application is specified by a task graph where arcs represent execution dependency between tasks. We assume that faults may occur multiple times during the execution and they are independent. Then, the success probability can be computed as the *product* of the success probabilities of all the constituent tasks. To be more concrete, let us take an example of a simple task graph that is depicted in Figure 2.5(a). It is with two tasks, τ_0 and τ_1, and τ_1 is only executed after the execution of τ_0. As their executions are serialized by the dependency, its WCRT is upper-bounded by $wcet(\tau_0) + wcet(\tau_1)$. This application is ill-executed if τ_0 *or* τ_1 is faulty. That is, assuming faults are independent events, the failure probability of the application is $1-(1-p(\tau_0))\cdot(1-p(\tau_1)) = p(\tau_0) + p(\tau_1) - p(\tau_0)\cdot p(\tau_1)$[1] where $p(\tau)$ denotes the failure probability of task τ. In this work, we employ Poisson distribution for fault occurrences since it is popularly used for multi-core systems [78, 79]. In this model, as will be elaborated in Section 5.2, the probability that we have n faults for t time units can be formulated as $\frac{(t\cdot\lambda)^n\cdot e^{-t\cdot\lambda}}{n!}$ where λ denotes the probability of an individual fault event. In this example, the WCET of τ_0 and τ_1 is 100 ms and the expected number of faults per 1 ms is given as 10^{-6}. Then, the minimum probability that the task successfully executes is $\frac{(100\cdot\lambda)^0\cdot e^{-100\cdot\lambda}}{0!} = e^{-10^{-4}}$ ($n = 0$ and $t = 100$). In other words, the upper bound of failure probability is $1-e^{-10^{-4}}$, i.e., $p(\tau_0) = p(\tau_1) = 1-e^{-10^{-4}} \approx 10^{-4}$ under the Poisson distribution model. Suppose that the given reliability constraint is 1.0×10^{-6}; then we must harden the task graph by re-executing some of the constituent tasks. For the rest of this chapter, WCRT and failure probability mean the upper bounds of the actual values for simple description.

Figure 2.5(b) shows three hardening examples and their WCRT and failure proba-

[1]The failure probability $1-(1-p(\tau_0))\cdot(1-p(\tau_1)) = p(\tau_0)+p(\tau_1)-p(\tau_0)\cdot p(\tau_1)$ can be approximated as $p(\tau_0)+p(\tau_1)$ if $p(\tau_0)$ and $p(\tau_1)$ are sufficiently small.



(a) A simple task graph

	τ_0 re-exec.	τ_1 re-exec.	WCRT	Failure probability
(i)	1	none	$300\ ms$	$10^{-4} + 10^{-8} - 10^{-12}$
(ii)	none	1	$300\ ms$	$10^{-4} + 10^{-8} - 10^{-12}$
(iii)	1	1	$400\ ms$	$2 \times 10^{-8} - 10^{-16}$

(b) WCRT and failure probability analysis

Figure 2.5: A motivational example that illustrates the pessimism of the previous WCRT analysis technique.

bility analysis. In case (i), τ_0 is decided to be re-executed once more. Then, τ_0 only fails if both of the two consecutive executions fail, i.e., $p(\tau_0) \approx 10^{-4} \times 10^{-4} = 10^{-8}$. As τ_1 is without any re-execution, $p(\tau_1) = 10^{-4}$ and the failure probability of the entire task graph becomes $1 - (1 - 10^{-8}) \cdot (1 - 10^{-4}) = 10^{-4} + 10^{-8} - 10^{-12}$, still larger than the given reliability constraint, 1.0×10^{-6}. Similarly, case (ii) is not a viable solution in terms of reliability. Thus, in Kang's approach [22], both τ_0 and τ_1 must be re-executed as shown in case (iii); it fulfills the reliability requirement, $2.0 \times 10^{-8} - 10^{-16} < 1.0 \times 10^{-6}$.

In terms of WCRT, case (i) or (ii) in Figure 2.5(b) results in 300 ms as at most one instance of either τ_0 or τ_1 is additionally executed. But, in case (iii), both τ_0 and τ_1 can be re-executed, resulting in the WCRT of 400 ms. We argue that this WCRT estimate is overestimated as it is blind to the probabilistic distribution of faults over the entire WCRT. The expected number of faults over the WCRT of 400 ms is $10^{-6} \times 400 = 4 \cdot 10^{-4}$. Then, under Poisson distribution, the probability that we have more than one fault within 400 ms becomes $1 - e^{-(4 \cdot 10^{-4})} - (4 \cdot 10^{-4}) \cdot e^{-(4 \cdot 10^{-4})} \approx 8.0 \times 10^{-8}$. Since this probability is smaller than the reliability constraint, the case (iii) that has more than one fault needs not to be considered in the WCRT analysis. To overcome this pessimism, in this dissertation, we propose an enhanced analysis technique that tightens WCRT estimates, considering the number of possibly occurring faults during the actual response time of a task graph.

2.4 Multi-mode Dataflow

There exists a research that propose MMDF bandwidth optimization scheduling scheme considering mode transition [29]. However, the technique in [29] cannot analyze the real-time inter-application interference between multiple applications because they finds a static schedule of MMDF assuming it is executed alone in the system. Traditionally in the real-time system researches, it is assumed that currently executed tasks are replaces with a new set of independent tasks when the mode change of the system occurs [80]. This model is applicable only when all applications are specified as independent tasks. Conversely, there is a previous work [81] that suggests the worst-case performance analysis when a dataflow application shares the system with real-time tasks. This technique can analyze the worst-case performance due to interference of various applications, but does not support multi-mode applications.

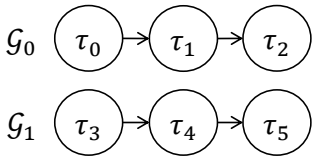
Chapter 3

Worst-case Performance Analysis

3.1 Problem Definition

We formally describe the application model and the system model assumed in this chapter. An input application, $\mathcal{G}_i$, is represented by an acyclic task graph as illustrated in Figure 3.1 (a). In a task graph, $\mathcal{G} = \{\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G}\}$, $\mathcal{V}_\mathcal{G}$ represents a set of tasks and $\mathcal{E}_\mathcal{G} = \{(\tau_i, \tau_j) | \tau_i \in \mathcal{V}_\mathcal{G}, \tau_j \in \mathcal{V}_\mathcal{G}\}$ a set of edges to represent execution dependencies between tasks. If a task has more than one input edge, it is released after all predecessor tasks are completed. Note that an application may have multiple graphs or multiple source nodes. In these cases, we add a single dummy task to make a parent task of all source tasks in our model. An application $\mathcal{G}$ can be initiated periodically or sporadically, characterized by a tuple $(T_\mathcal{G}, J_\mathcal{G})$ where $T_\mathcal{G}$ and $J_\mathcal{G}$ represent the period and the maximum jitter, respectively. For sporadic activation, $T_\mathcal{G}$ denotes the minimum initiation interval. Task graph $\mathcal{G}$ is given a relative deadline $D_\mathcal{G}$ to meet once activated. We assume that $D_\mathcal{G}$ and $J_\mathcal{G}$ are not greater than $T_\mathcal{G}$ in the baseline technique. This assumption will be removed in Section 3.5. The task graph that task τ_i belongs to is denoted by $\mathcal{G}_{\tau_i}$.

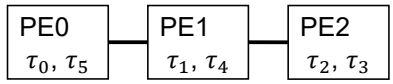
A system consists of a set of processing elements (PEs) as shown in Figure 3.1 (c). Tasks are mapped onto processing elements and we assume that task mapping is given and fixed. The processing element that task τ_i is mapped to is denoted by M_i. For each task τ_i, the varying execution time is represented as a tuple $[C_i^l, C_i^u]$ indicating the lower and



(a) Task graphs

Task Graph	$T_{\mathcal{G}}, J_{\mathcal{G}}$
$\mathcal{G}_0$	100, 0
$\mathcal{G}_1$	100, 0

(b) Activation parameters



Exec. Time [min, max]	τ_0[10, 20], τ_1[5, 20] τ_2[10, 20], τ_3[20, 20] τ_4[10, 20], τ_5[10, 20]					
Mapping	M_0	M_1	M_2	M_3	M_4	M_5
	PE0	PE1	PE2	PE2	PE1	PE0
Priority	$PR_0 > PR_3 > PR_1 > PR_4 > PR_2 > PR_5$					
Scheduling	$\{\text{PE0}, \text{PE2}\} \subset \mathcal{P}, \{\text{PE1}\} \subset \mathcal{N}$					

(c) Mapping and task information

Figure 3.1: An example system and application model

the upper bound on the mapped PE. Note that a communication network can be modeled as a separate PE. For instance, the PE graph of Figure 3.1 (c) represents a system that consists of two processors (PE0 and PE2) connected to a bus (PE1). Tasks mapped to a communication network deliver messages between two computation tasks; for example τ_1 indicates message communication between two computation tasks, τ_0 and τ_2.

We assume that the scheduling policy of a PE can be either a fixed-priority preemptive scheduling or a fixed-priority non-preemptive scheduling. $\mathcal{P}$ denotes a set of PEs that have a preemptive scheduling policy, and $\mathcal{N}$ denotes a set of PEs with a non-preemptive scheduling policy. A PE belongs to either $\mathcal{P}$ or $\mathcal{N}$. In Figure 3.1, PE0 and PE2 use preemptive scheduling while PE1 serves the communication tasks in a non-preemptive fashion; a higher-priority message cannot preempt the current message delivery. The priority of the task τ_i is denoted by PR_i. We assume that all tasks mapped to each PE have distinct priorities to make the scheduling order deterministic. We also assume that applications are ordered in priorities and so priorities of task graphs are not inter-mixed, meaning that priorities of all tasks in a graph is either higher or lower than those of all tasks in another graph: $\forall_{i \neq j}(\min_{\tau_s \in \mathcal{G}_i} PR_s > \max_{\tau_d \in \mathcal{G}_j} PR_d$ or $\max_{\tau_s \in \mathcal{G}_i} PR_s < \min_{\tau_d \in \mathcal{G}_j} PR_d)$. This assumption is made to simplify the proof in this chapter while the proposed technique can be extended to a general case without this assumption.

Table 3.1: Notations used in problem definition

Notation	Description
$\mathcal{G}_i$	a task graph with its index i
$T_{\mathcal{G}}$	an initiation interval of the task graph $\mathcal{G}$
$J_{\mathcal{G}}$	the maximum initiation jitter of the task graph $\mathcal{G}$
$D_{\mathcal{G}}$	a deadline of the task graph $\mathcal{G}$
τ_i	a task with its index i
M_i	the mapped processing element of a task τ_i
C_i^l	the best-case execution time of a task τ_i
C_i^u	the worst-case execution time of a task τ_i
PR_i	the priority of the task τ_i
$\mathcal{P}$	the set of preemptive processing elements
$\mathcal{N}$	the set of non-preemptive processing elements
$\mathcal{R}_{\mathcal{G}}$	the worst-case response time of the task graph $\mathcal{G}$

The WCRT of task graph $\mathcal{G}$, denoted by $\mathcal{R}_{\mathcal{G}}$, is defined as the time difference between the latest finish time and the earliest release time among tasks in the task graph. Table 3.1 summarizes the notations described above.

3.2 Overall Analysis Flow

Before explaining the proposed technique in detail, we show how the analysis is performed with illustrative examples and present the overall flow of the proposed technique in this section.

The proposed approach consists of two analyses: intra-graph and inter-graph interference analyses. At first, we consider the intra-graph interference. Since it is assumed that a deadline $D_{\mathcal{G}}$ is not greater than a period $T_{\mathcal{G}}$, we can analyze only one job instance per task; we use "task" to refer to a job instance in this section. The release times of tasks are defined relatively to the graph initiation time, which is assumed zero. We compute the amount of intra-graph interference by comparing the schedule time bounds of tasks. To conservatively compute every possible interference, we compute three pairs of time bound information for each task: release time bound (RB_i^l, RB_i^u), start time bound (SB_i^l,

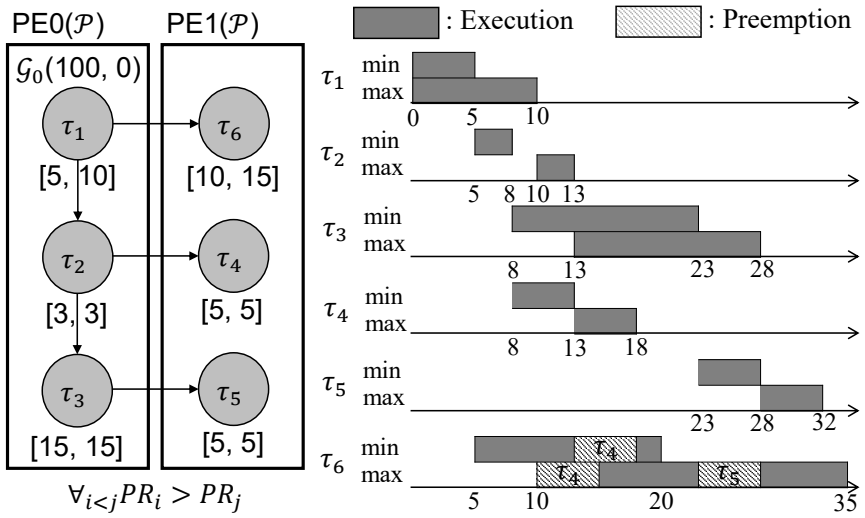


Figure 3.2: Intra-graph interference analysis example

SB_i^u), and finish time bound (FB_i^l, FB_i^u) for task τ_i.

In Figure 3.2, we plot the best-case schedule and the worst-case schedule for a given task graph $\mathcal{G}_0$ and task mapping information. The tasks in PE0 have no interference since all tasks in PE0 are executed sequentially. For tasks in PE1, however, we have to consider interference from higher priority tasks by comparing the time bounds. Task τ_5 is not preempted by task τ_4 since task τ_4 always finishes earlier than the release time of task τ_5. For the best-case schedule of task τ_6, we consider the minimum interference. Task τ_6 is always preempted by task τ_4 since task τ_4 always appears during the execution. On the other hand, we consider the maximum interference for the worst-case schedule of task τ_6. Task τ_6 can be preempted by both tasks τ_4 and τ_5 since their release times can be earlier than the finish time of task τ_6. In this way, we derive all time bounds of tasks, considering intra-graph interference.

Next, we consider the inter-graph interference and revise the computed time bounds accordingly. Suppose there is a higher priority task τ_0 that belongs to another task graph as shown in Figure 3.3. If we apply the response time analysis independently to the dependent tasks, the number of preemptions may be over-estimated as displayed in Fig-

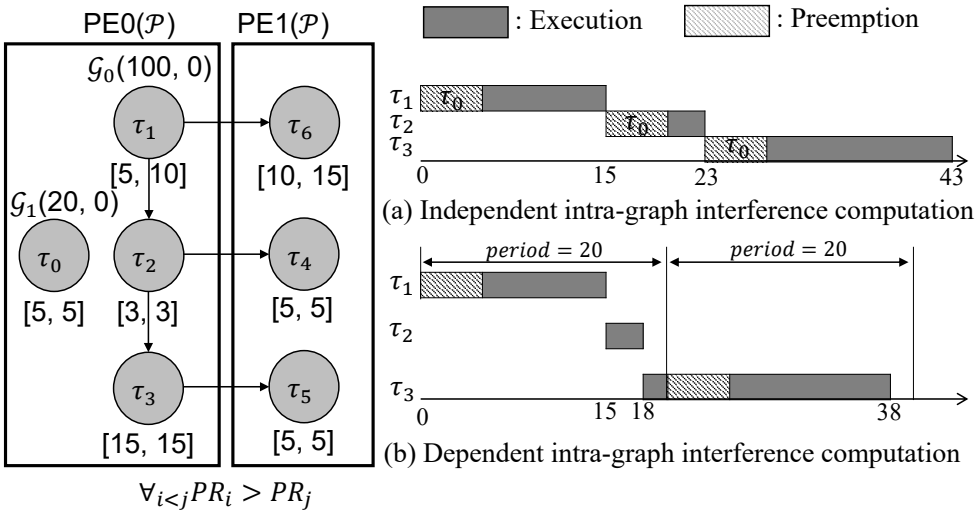


Figure 3.3: Inter-graph interference analysis example

ure 3.3 (a) where τ_1, τ_2, and τ_3 are all preempted by τ_0 independently. For more accurate estimation, we need to consider the possible schedule scenario of task τ_0 considering its period. Figure 3.3 (b) shows how the schedule of task τ_0 is considered. When τ_0 preempts τ_1, τ_2 knows that the next request of τ_0 will occur after 5 time units from the finish time of τ_1 so that τ_0 does not preempt τ_2. Since the next request of τ_0 occurs 2 time units from the finish time of τ_2, τ_3 can be preempted by τ_0. Hence the maximum interference from task τ_0 is computed to 10 as shown in Figure 3.3 (b). In summary, for the inter-graph interference analysis, we compute the time difference from the finish time of a predecessor task to release time of the next job instance of the higher priority task.

Figure 3.4 shows the overall flow of the proposed technique. We compute three pairs of time bound information for each task: release time bound (RB_i^l, RB_i^u), start time bound (SB_i^l, SB_i^u), and finish time bound (FB_i^l, FB_i^u), in the *Time Bound Computation* module considering the intra-graph interference and the inter-graph interference as explained above. How to compute the intra-graph interference and the inter-graph interference will be explained in subsections 3.3.1 and 3.3.2, respectively. During the time bound computation of each task, optimization techniques in Section 3.4 are used for tighter estimation.

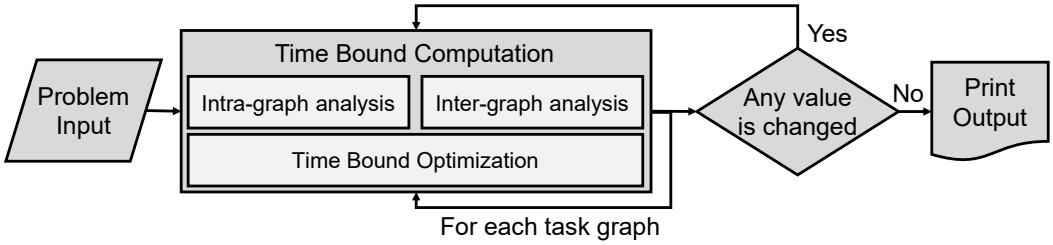


Figure 3.4: The overall flow of the proposed HPA technique

Time bound computation is performed for each task graph individually and we repeat this process until all time bounds are converged since time bounds of task graphs affect each other.

3.3 Proposed Analysis Technique: Hybrid Performance Analysis

In this section, we explain the key techniques of the proposed analysis and discuss how to achieve a safe and tight bound of the WCRT. At first, how to compute the schedule time bounds is explained in subsection 3.3.1 considering only the interference from tasks in the same task graph. Secondly, we refine the time bounds by incorporating the interference from the other task graphs in subsection 3.3.2. Table 3.2 summarizes the symbols and notations used in this section.

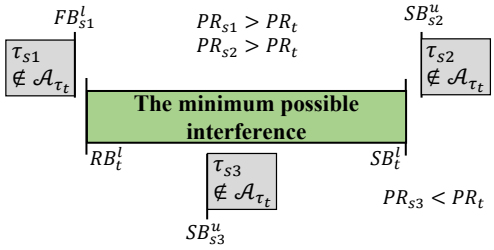
3.3.1 Time Bound Computation: Intra-graph Interference Analysis

The minimum (RB_t^l) and maximum (RB_t^u) release time bounds are computed as follows:

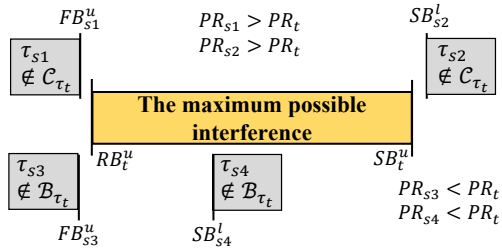
$$RB_t^l = \begin{cases} 0, & \text{if } \tau_t \text{ is a source task} \\ \max_{\tau_p \in pred(\tau_t)} FB_p^l, & \text{otherwise} \end{cases} \tag{3.1}$$

Table 3.2: Notations for the proposed analysis technique

Notation	Description
$r(\mathcal{G}_{\tau_t})$	the actual release time of a task graph which task τ_t belongs to
$r(\tau_t)$	the actual release time of task τ_t
$s(\tau_t)$	the actual start time of task τ_t
$f(\tau_t)$	the actual finish time of task τ_t
RB_t^l	the lower bound of the release time of task τ_t
RB_t^u	the upper bound of the release time of task τ_t
SB_t^l	the lower bound of the start time of task τ_t
SB_t^u	the upper bound of the start time of task τ_t
FB_t^l	the lower bound of the finish time of task τ_t
FB_t^u	the upper bound of the finish time of task τ_t
$pred(\tau_t)$	a set of immediate predecessors of task τ_t
$succ(\tau_t)$	a set of immediate successors of task τ_t
$ancestor(\tau_t)$	a set of ancestors of task τ_t
$descendant(\tau_t)$	a set of descendants of task τ_t
$Delay_t^l$	the maximum intra-graph blocking delay to τ_t
$Delay_t^h$	the maximum intra-graph preemption to τ_t between RB_t^u and SB_t^u
$Preempt_t^B$	the minimum intra-graph preemption to τ_t between SB_t^l and FB_t^l
$Preempt_t^W$	the maximum intra-graph preemption to τ_t between SB_t^u and FB_t^u
$\overline{Delay_t^l}$	the maximum inter-graph blocking delay to τ_t
$\overline{Delay_t^h}$	the maximum inter-graph preemption to τ_t between RB_t^u and SB_t^u
$\overline{Preempt_t^W}$	the maximum inter-graph preemption to τ_t between SB_t^u and FB_t^u
Ψ_t	the request time difference that makes the worst-case interference
$\phi_{t,i}^r$	the minimum distance from RB_t^u to the next request of τ_i
$\phi_{t,i}^s$	the minimum distance from SB_t^u to the next request of τ_i
$\phi_{t,i}^f$	the minimum distance from FB_t^u to the next request of τ_i
$\mathcal{A}_{\tau_t}$	a set of tasks who may contribute to the best-case intra-graph interference between RB_t^l and SB_t^l
$\mathcal{B}_{\tau_t}$	a set of lower priority tasks who may contribute to the worst-case intra-graph interference between RB_t^u and SB_t^u
$\mathcal{C}_{\tau_t}$	a set of higher priority tasks who may contribute to the worst-case intra-graph interference between RB_t^u and SB_t^u
$\mathcal{D}_{\tau_t}$	a set of higher priority tasks who may contribute to the best-case intra-graph interference between SB_t^l and FB_t^l
$\mathcal{E}_{\tau_t}$	a set of higher priority tasks who may contribute to the worst-case intra-graph interference between SB_t^u and FB_t^u
$\mathcal{F}_{\tau_t}$	a set of tasks who belong to the task graph of τ_t and may start earlier than RB_t^u
$\mathcal{Y}_{\tau_t}$	a set of all lower priority tasks not belonging to the task graph of τ_t
$\mathcal{Z}_{\tau_t}$	a set of all higher priority tasks not belonging to the task graph of τ_t



(a) Tasks that cannot contribute to the minimum possible interference from RB^l_t to SB^l_t



(b) Tasks that cannot contribute to the maximum possible interference from RB^u_t to SB^u_t

Figure 3.5: The tasks that cannot contribute to the start time bound

$$RB^u_t = \begin{cases} J_{\mathcal{G}_{\tau_t}}, & \text{if } \tau_t \text{ is a source task} \\ \max_{\tau_p \in pred(\tau_t)} FB^u_p, & \text{otherwise} \end{cases} \tag{3.2}$$

The earliest and the latest release times of a non-source task are defined as the maximum value among the earliest and the latest finish times of predecessors, respectively since it becomes executable only after all predecessor tasks are finished.

For task τ_t to start, it should be already released and the processor must be available: the start time of τ_t is not smaller than the release time and the maximum time among finish times of tasks that have higher priority, start earlier, and finish after task τ_t is released. Then the earliest start time SB^l_t is formulated as follows:

$$SB^l_t = \max(RB^l_t, \max_{\tau_s \in \mathcal{A}_{\tau_t}} FB^l_s) \tag{3.3}$$

where set $\mathcal{A}_{\tau_t}$ for the preemptive scheduling policy is defined as $\mathcal{A}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t, RB^l_t < FB^l_s, SB^u_s \leq SB^l_t\}$ and for the non-preemptive scheduling policy as $\mathcal{A}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, (PR_s > PR_t, RB^l_t < FB^l_s, SB^u_s \leq SB^l_t)$ or $(PR_s < PR_t, SB^u_s < RB^l_t < FB^l_s)\}$. Figure 3.5 (a) illustrates the tasks that are not included in set $\mathcal{A}_{\tau_t}$. A higher or lower priority task τ_{s1} will not interfere τ_t if it finishes earlier than RB^l_t. A higher priority task τ_{s2} that may start after SB^l_t is also ignored and a lower priority task τ_{s3} cannot block τ_t. On the other hand, if a higher priority task τ_s always starts before SB^l_t and the

earliest finish time of task τ_s is greater than RB_t^l, task τ_t should wait for the completion of task τ_s. In case a non-preemptive scheduling is used, a lower priority task that starts before RB_t^l is included.

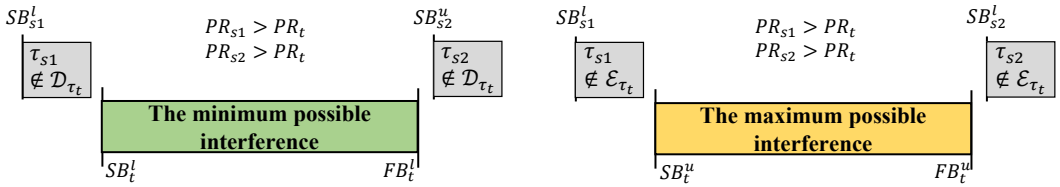
To estimate the maximum start time SB_t^u for conservative estimation, we should consider all possible preemptions.

$$SB_t^u = RB_t^u + Delay_t^l + Delay_t^h \tag{3.4}$$

where $Delay_t^l$ and $Delay_t^h$ denote the amounts of preemption between the release time and the start time by lower and higher priority tasks respectively. For the preemptive scheduling policy, $Delay_t^l$ is zero. In case a lower priority task is running when τ_t is released, τ_t should wait until the current lower priority task finishes in the non-preemptive scheduling policy, which is accounted as follows:

$$Delay_t^l = \begin{cases} 0, \text{if } \forall_{\tau_p \in pred(\tau_t)} M_p = M_t \\ \max\limits_{\tau_s \in \mathcal{B}_{\tau_t}} \min(C_s^u, FB_s^u - RB_t^u), \text{otherwise} \end{cases} \tag{3.5}$$

where $\mathcal{B}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s < PR_t, SB_s^l < RB_t^u < FB_s^u\}$. Set $\mathcal{B}_{\tau_t}$ includes lower priority tasks in the same task graph that may start earlier than τ_t and delay the start time of τ_t. In Figure 3.5 (b), a lower priority task τ_{s3} is not included in set $\mathcal{B}_{\tau_t}$ since it always finish earlier than RB_t^u. A lower priority task τ_{s4} that always starts later than RB_t^u also cannot block task τ_t. Note that partial blocking is considered in the formula by $FB_s^u - RB_t^u$, which can be smaller than C_s^u. For a task to be blocked by a lower priority task, a lower priority task should start before and is executing at the release time of target task. In case every predecessor is mapped to the same PE, the target task is released right after the latest predecessor finished its execution and no lower priority task can start between the predecessor and the task so that the blocking term $Delay_t^l$ is zero.



(a) Tasks that cannot contribute to the minimum possible interference from SB^l_t to FB^l_t

(b) Tasks that cannot contribute to the maximum possible interference from SB^u_t to FB^u_t

Figure 3.6: The tasks that cannot contribute to the finish time bound

$Delay^h_t$ is commonly formulated for both scheduling policies as follows:

$$Delay^h_t = \sum_{\tau_s \in \mathcal{C}_{\tau_t}} \min(C^u_s, FB^u_s - RB^u_t) \tag{3.6}$$

where $\mathcal{C}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t, SB^l_s \leq SB^u_t, RB^u_t < FB^u_s\}$. Set $\mathcal{C}_{\tau_t}$ includes higher priority tasks in the same task graph that can possibly delay the start time of τ_t. In Figure 3.5 (b), tasks τ_{s1} and τ_{s2} cannot make interference to task τ_t between RB^u_t and SB^u_t since they either always finish earlier than RB^u_t or start later than SB^u_t. Partial preemption is considered similarly to $Delay^l_t$ formulation.

The minimum finish time FB^l_t is formulated as follows:

$$FB^l_t = SB^l_t + C^l_t + Preempt^B_t \tag{3.7}$$

where $Preempt^B_t$ represents the unavoidable preemption delay that is zero for the non-preemptive scheduling policy. For the preemptive scheduling policy, $Preempt^B_t$ becomes

$$Preempt^B_t = \sum_{\tau_s \in \mathcal{D}_{\tau_t}} C^l_s \tag{3.8}$$

where $\mathcal{D}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t, SB^l_t \leq SB^l_s \leq SB^u_s \leq FB^l_t\}$. $\mathcal{D}_{\tau_t}$ is a set of higher priority tasks which always start to execute and preempt τ_t during τ_t is running. Tasks τ_{s1} and τ_{s2} in Figure 3.6 (a) are not included in set $\mathcal{D}_{\tau_t}$ since task τ_{s1} starts earlier

than SB_t^l and task τ_{s2} starts after FB_t^l.

The maximum finish time FB_t^u is formulated as follows:

$$FB_t^u = SB_t^u + C_t^u + Preempt_t^W \tag{3.9}$$

where $Preempt_t^W$ represents the worst-case preemption delay that is zero for the non-preemptive scheduling policy. $Preempt_t^W$ for the preemptive scheduling policy is formulated as follows:

$$Preempt_t^W = \sum_{\tau_s \in \mathcal{E}_{\tau_t}} C_s^u \tag{3.10}$$

where $\mathcal{E}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t, SB_t^u < SB_s^l \leq FB_t^u\}$, indicating a set of higher priority tasks which can appear during the execution of τ_t. Figure 3.6 (b) shows two tasks τ_{s1} and τ_{s2} that are not in $\mathcal{E}_{\tau_t}$. Since task τ_{s2} always starts after FB_t^u, it is obvious that task τ_{s2} cannot interfere task τ_t. In case of τ_{s1}, it may seem to make interference to τ_t if $SB_t^u \leq SB_{s1}^u < FB_t^u$. However, we do not consider this case in the finish time bound computation since it is already considered in the start time bound computation as a possible interference between RB_t^u and SB_t^u.

After determining all time bounds of tasks, we compute the WCRT of each task graph $\mathcal{G}$ as follows:

$$\mathcal{R}_{\mathcal{G}} = \max_{\tau_s \in \mathcal{G}} FB_s^u \tag{3.11}$$

Theorem 3.3.1. *The HPA technique guarantees the conservativeness of every schedule time bound when there is no inter-graph interference.*

Proof. See subsection 3.6.1 for the proof. □

3.3.2 Time Bound Computation: Inter-graph Interference Analysis

In this subsection, we explain how the schedule time bounds are adjusted to incorporate the interference from the other task graphs.

At first, we formulate the blocking delay by a lower priority task of other task graphs under the non-preemptive scheduling policy:

$$\overline{Delay_t^l} = \begin{cases} 0, & \text{if } \forall_{\tau_p \in pred(\tau_t)}(M_p = M_t) \\ \max\limits_{\tau_s \in \mathcal{Y}_{\tau_t}} C_s^u, & \text{otherwise} \end{cases} \tag{3.12}$$

where $\mathcal{Y}_{\tau_t} = \{\tau_s | \tau_s \notin \mathcal{G}_{\tau_t}, M_s = M_t, PR_s < PR_t\}$: set $\mathcal{Y}_{\tau_t}$ includes all lower priority tasks in the other task graphs. Since there is no constraint assumed between starting offsets of task graphs, we take the maximum WCET for conservative estimation. In case every predecessor is mapped to the same PE, $Delay_t^l$ is zero. Since only one lower priority task can block τ_t under non-preemptive scheduling policy, SB_t^u is modified to find the worst-case blocking delay as the maximum between $Delay_t^l$ and $\overline{Delay_t^l}$.

$$SB_t^u = RB_t^u + \max\left(Delay_t^l, \overline{Delay_t^l}\right) + Delay_t^h \tag{3.13}$$

Second, we perform the response time analysis that computes the maximum number of appearances during a given time interval in order to incorporate interference from a higher priority task in other task graphs. We regard each higher priority task in other task graphs as an independent task for conservative estimation. Under a preemptive scheduling policy, the maximum number of preemptions by a task τ_s that has a period $T_{\mathcal{G}_{\tau_s}}$ and a release time bound (RB_s^l, RB_s^u) will occur when RB_s^u is aligned with the release time of τ_t and the second request appears after the shortest interval of $T_{\mathcal{G}_{\tau_s}} + (RB_s^u - RB_s^l)$, followed by subsequent requests that appear periodically from the second request, which is induced by the critical instant theorem.

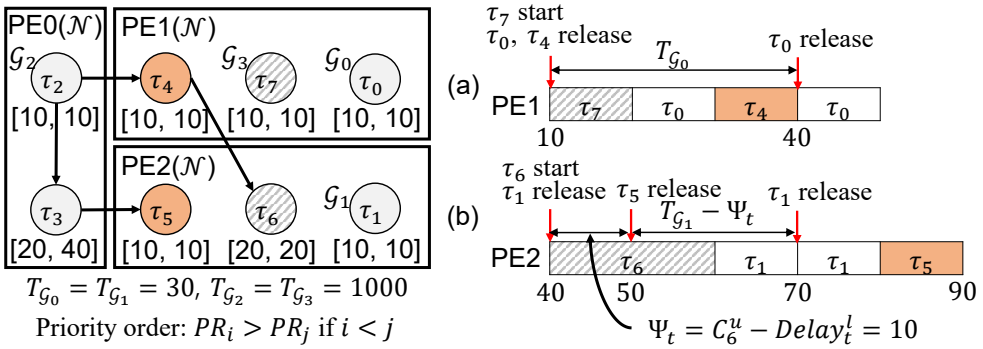


Figure 3.7: An example that earlier release of a higher priority task than the release time of a target task leads to the maximum interference

On the other hand, under a non-preemptive scheduling policy, earlier release of a higher priority task than the release of the target task may result in the worst-case interference since the higher priority task may be blocked by a lower priority task until the release time of the target task. In the example of Figure 3.7, two processing elements, $PE1$ and $PE2$, have similar task profiles: a target task (τ_4 or τ_5) is interfered by a higher priority task (τ_0 or τ_1) and a lower priority task (τ_7 or τ_6). On $PE1$ where all three tasks are independent, the worst-case interference occurs when all releases are aligned at the same time and τ_7 that is released just before other tasks blocks both τ_0 and the target task τ_4. It coincides with the observation of conventional response time analysis with independent tasks.

In case there are lower priority tasks in the same graph, release times of tasks may not be aligned as illustrated in Figure 3.7 (b) where task τ_6 is in the same graph of the target task τ_5 and may block the target task. The maximum number of preemptions by τ_1 occurs when the release of τ_1 coincides with the start of τ_6, not with the release of the target task. The figure shows the worst-case scenario that produces the upper bounds of task scheduling time of $\mathcal{G}_2$; RB_6^u is the same as FB_4^u, RB_5^u is the same as FB_3^u, and then FB_5^u becomes 90 because τ_1 makes two preemptions onto τ_5.

As shown in this example, it is necessary to find out the release time difference

between a higher priority task and the target task that incurs the maximum number of preemptions for conservative estimation when there are dependent tasks with the target task. This time difference is called *period shift* and denoted as Ψ_t [82]. Note that period shift Ψ_t does not actually change the periodic behavior of higher priority tasks. It defines the shifted amount of release time of a higher priority task to make the worst-case interference to the target task. Obviously, Ψ_t is zero for a preemptive scheduling policy. For a non-preemptive scheduling policy, we find the maximum possible blocking time of a higher priority task by a task that belongs to the same task graph with the target task and set it as the period shift. If the target task and its all predecessors are mapped onto the same PE, the maximum execution time among predecessors is selected as Ψ_t since the higher priority task can be blocked only by a predecessor task and task priorities are not inter-mixed among task graphs. Otherwise, we have to consider all candidate tasks that may block the target task in the same task graph.

Note that a lower priority task affects the blocking delay computation of $Delay_t^l$ and period shift computation. When a lower priority task τ_s blocks τ_t by α amount of time, the maximum preemption by a preempting task occurs when it is released right after the start time of τ_s, which is equal to $r(\tau_t) - C_s^u + \alpha$. Let A be a set of preempting tasks of τ_t and $\mathcal{R}_{\tau_t}^{\alpha}$ be the worst-case response time of τ_t when τ_s blocks τ_t by α amount of time. Then the conventional response time analysis tells that $\mathcal{R}_{\tau_t}^{\alpha}$ becomes

$$\mathcal{R}_{\tau_t}^{\alpha} = \sum_{\tau_i \in A} \left\lceil \frac{\mathcal{R}_{\tau_t}^{\alpha} + C_s^u - \alpha}{T_{\mathcal{G}_{\tau_i}}} \right\rceil \cdot C_i^u + \alpha + C_t^u$$

where $C_s^u - \alpha$ is the period shift when the blocking time is α. The following lemma says that the response time becomes the worst when α is the maximum blocking time as formulated in equation (3.5).

Lemma 3.3.1. *Suppose τ_s is a task that may block both preempting tasks of other task graphs and the target task τ_t. Let $\mathcal{R}_{\tau_t}^{\alpha}$ be the worst-case response time of τ_t when τ_s blocks*

τ_t *by* α *amount of time. Then,* $\mathcal{R}_{\tau_t}^{\alpha} < \mathcal{R}_{\tau_t}^{\beta}$ *if* $0 \leq \alpha < \beta \leq C_s^u$.

Proof. See subsection 3.6.2. □

For the example of Figure 3.7, $\tau_t = \tau_5$ and $\tau_s = \tau_6$. Then inter-interference is maximized when τ_6 blocks τ_t maximally as 10 and period shift is $C_6^u - 10 = 10$. Thus we compute the period shift Ψ_t as the maximum time difference between the start time of a candidate task τ_s and the release time of the target task τ_t when τ_s maximally blocks τ_t as $C_s^u - \min(C_s^u, \max(0, FB_s^u - RB_t^u))$. Among candidate tasks, we find out the task that gives the maximum difference. In summary the period shift of the target task, Ψ_t, is formulated as follows:

$$\Psi_t = \begin{cases} 0, & \text{if } M_t \in \mathcal{P} \\ \max\limits_{\tau_p \in pred(\tau_t)} C_p^u, & \text{else if } \forall_{\tau_p \in pred(\tau_t)}(M_p = M_t) \\ \max\limits_{\tau_s \in \mathcal{F}_{\tau_t}} (C_s^u - \min(C_s^u, \max(0, FB_s^u - RB_t^u))), & \text{otherwise} \end{cases} \tag{3.14}$$

where $\mathcal{F}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, SB_s^l < RB_t^u\}$. Since $\mathcal{F}_{\tau_t}$ includes all tasks that may be executed before the target task, the maximum time difference between the start time of the candidate task and the release time of the target task is computed as $C_s^u - \min(C_s^u, \max(0, FB_s^u - RB_t^u))$ where $\max(0, FB_s^u - RB_t^u)$ indicates the maximum blocking time.

So far it is assumed that a higher priority task may appear anytime in the computation of the maximum interference. Now consider a sequence of task executions that have dependencies. Applying response time analysis independently to each task in the sequence may over-estimate the number of requests during the same time interval. In Figure 3.8 (b), preemption by τ_0 is considered independently for τ_1 and τ_2, which results in excessive counting of preemptions. On the other hand, observing that the next request appears 3 time units after the release time of τ_2, the interference can be tightly estimated as shown in Figure 3.8 (c). Hence the relative distance to the next request appearance needs to be considered for tight estimation when response time analysis is applied for de-

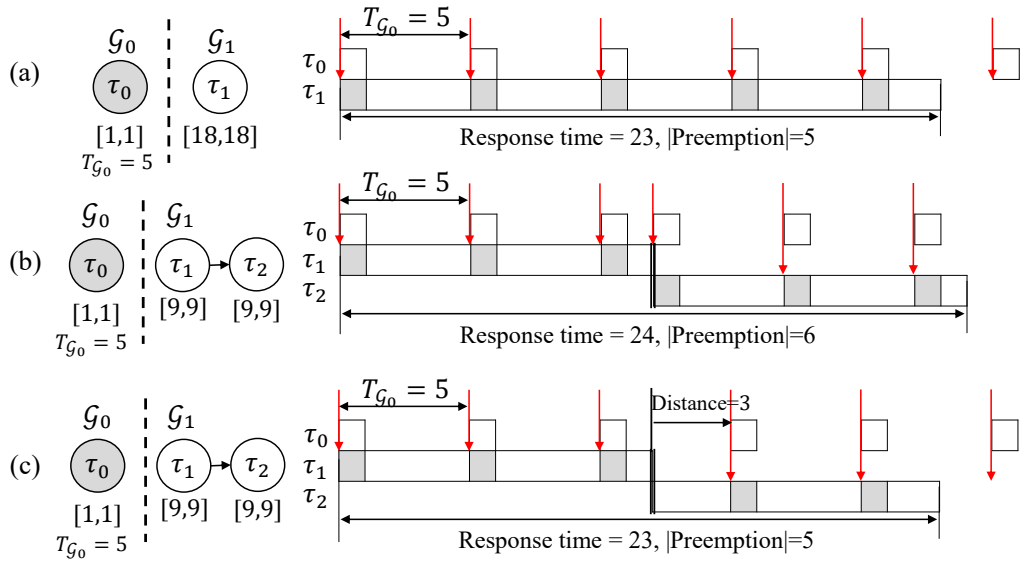


Figure 3.8: Computation of the maximum number of preemptions from a higher priority task τ_0 to (a) a single task τ_1, (b) to two dependent tasks, τ_1 and τ_2 individually, and (c) to two dependent tasks jointly

pendent tasks. To this end, we define three distance values: request phase $\phi^r_{t,i}$, start phase $\phi^s_{t,i}$, and finish phase $\phi^f_{t,i}$, meaning the relative distances from RB^u_t, SB^u_t, and FB^u_t to the next release of preempting task τ_i, respectively.

For computing the amount of interference from a higher priority task τ_i to τ_t during time interval $[RB^u_t, SB^u_t]$, we compute the minimum distance from RB^u_t to the next appearance of τ_i which is denoted as $\phi^r_{t,i}$. If τ_t is a source task, $\phi^r_{t,i}$ for each higher priority task τ_i in the other task graph is initialized to $-(\Psi_t + RB^u_i - RB^l_i)$. Note that the request phase of the source task is set to negative to account for the effect of period shift and release jitter together. If τ_t is a non-source task, $\phi^r_{t,i}$ depends on predecessors. If τ_t has one predecessor task as Figure 3.8 (c), $\phi^r_{t,i}$ can be easily induced from its predecessor τ_p: the distance to the next appearance of τ_i from FB^u_p, which is denoted as $\phi^f_{p,i}$, becomes $\phi^r_{t,i}$ since $FB^u_p = RB^u_t$.

On the other hand, if τ_t has multiple predecessor tasks mapped on the same PE, we have to compute the minimum distance considering $\phi^f_{p,i}$ of all predecessors for conser-

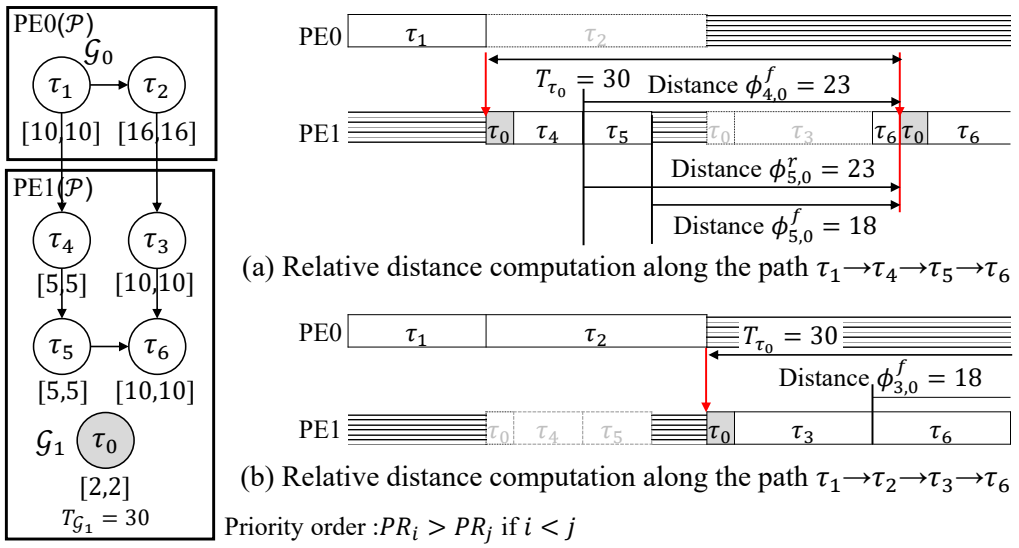


Figure 3.9: Relative distance computation to the next invocation of a higher priority task when τ_t has multiple predecessors: an example scenario

vative computation. Figure 3.9 illustrates the case where task τ_6 has two predecessors τ_3 and τ_5 mapped onto the same PE. Since there may exist multiple paths to τ_t in the dependency graph, predecessor tasks may see the next request time of τ_i differently from each other. In Figure 3.9, there are two paths to target task τ_6: $\tau_1 \rightarrow \tau_4 \rightarrow \tau_5 \rightarrow \tau_6$ and $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \tau_6$. In case of the first path as shown in Figure 3.9 (a), τ_0 makes maximum preemption when it is released at the same time as the release time of τ_4. Then the next invocation will be at $FB^u_4 + \phi^f_{4,0} = 40$ which is also equal to $FB^u_5 + \phi^f_{5,0}$ assuming that τ_1 starts its execution at time 0. On the other hand, in case of the second path in Figure 3.9 (b), τ_0 makes maximum preemption when its release time is equal to the release time of τ_3 which is 26. The next appearance will be $FB^u_3 + \phi^f_{3,0} = 56$, which is later than the first case. For conservative estimation, when computing $\phi^r_{6,0}$ based on $\phi^f_{3,0}$ and $\phi^f_{5,0}$, we choose the earliest invocation, which is $\phi^f_{5,0} + FB^u_5$, and compute the time difference from RB^u_6.

Note that the computed distance can be negative. To avoid excessive over-estimation, we set the lower bound of the negative distance as $-(\Psi_t + RB^u_i - RB^l_i)$, which corresponds

to the upper bound of interference ignoring the task dependency. In case there is a predecessor mapped to a different processing element, $\phi^r_{t,i}$ is again set to $-(\Psi_t + RB^u_i - RB^l_i)$ for conservative estimation since the finish time of the predecessor is not known. In summary, we formulate $\phi^r_{t,i}$ as follows;

$$\phi^r_{t,i} = \begin{cases} -(\Psi_t + RB^u_i - RB^l_i), & \text{if } \tau_t \text{ is a source task or} \\ & \exists_{\tau_p \in pred(\tau_t)}(M_p \neq M_t) \\ \max \begin{pmatrix} -(\Psi_t + RB^u_i - RB^l_i), \\ \min\limits_{\tau_p \in pred(\tau_t)} (\phi^f_{p,i} + FB^u_p) - RB^u_t \end{pmatrix}, & \text{otherwise} \end{cases} \tag{3.15}$$

For the higher priority tasks in the other task graphs, we perform response time analysis to compute the maximum preemption delay that can appear during a time interval $[RB^u_t, SB^u_t]$. Since we know that τ_s will be released after $\phi^r_{t,s}$ from RB^u_t, interval $[RB^u_t + \phi^r_{t,s}, SB^u_t]$ is considered in the response time analysis. Thus the amount of interference can be formulated as follows.

$$\overline{Delay^h_t} = \sum_{\tau_s \in \mathcal{Z}_{\tau_t}} \left\lceil \frac{\max(0, SB^u_t - RB^u_t + 1 - \phi^r_{t,s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s \tag{3.16}$$

where $\mathcal{Z}_{\tau_t} = \{\tau_s | \tau_s \notin \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t\}$ is a set of all higher priority tasks in the other task graphs. "+1" is required in this computation to account for the case when a higher priority task arrives at the same time when the target task is about to start.

Now we adjust SB^u_t after considering all possible interferences from the other applications as follows:

$$SB^u_t = RB^u_t + max\left(Delay^l_t, \overline{Delay^l_t}\right) + Delay^h_t + \overline{Delay^h_t} \tag{3.17}$$

The relative distance from SB^u_t to the next appearance of τ_i, which is denoted as $\phi^s_{t,i}$,

for each higher priority task $\tau_i \notin \mathcal{G}_{\tau_t}$ is computed based on $\phi^r_{t,i}$ and SB^u_t as follows:

$$\phi^s_{t,i} = \left((\phi^r_{t,i} + RB^u_t) - SB^u_t\right) \bmod T_{\mathcal{G}_{\tau_i}} \tag{3.18}$$

In order to find $\phi^s_{t,i}$ which is the distance from SB^u_t to the earliest future invocation of τ_i after SB^u_t, we use modulo operation. For instance, suppose $\phi^r_{t,i} = 2, RB^u_t = 0, SB^u_t = 10$, and $T_{\mathcal{G}_i} = 5$. Then τ_i appearances at 2 and 7 are considered in equation (3.16). The next τ_i appearance after $SB^u_t = 10$, which is at 12, is computed by the distance $\phi^s_{t,i}$ from SB^u_t: $\phi^s_{t,i} = ((2+0) - 10) \bmod 5 = 2$. Then the response time analysis is used for computing the amount of interference during the interval $[SB^u_t, FB^u_t)$ as follows.

$$\overline{Preempt^W_t} = \sum_{\tau_s \in Z_{\tau_t}} \left\lceil \frac{\max(0, FB^u_t - SB^u_t - \phi^s_{t,s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s \tag{3.19}$$

$\overline{Preempt^W_t}$ is zero when a non-preemptive scheduling policy is used. FB^u_t is refined to incorporate the interference from all other task graphs during $[SB^u_t, FB^u_t]$ as follows:

$$FB^u_t = SB^u_t + C^u_t + Preempt^W_t + \overline{Preempt^W_t} \tag{3.20}$$

Similarly, the distance from FB^u_t, denoted as $\phi^f_{t,i}$, is formulated based on $\phi^s_{t,i}$ and FB^u_t as follows:

$$\phi^f_{t,i} = \begin{cases} (\phi^s_{t,i} + SB^u_t) - FB^u_t, & M_t \in \mathcal{N} \\ \left((\phi^s_{t,i} + SB^u_t) - FB^u_t\right) \bmod T_{\mathcal{G}_{\tau_i}}, & \text{otherwise} \end{cases} \tag{3.21}$$

Since there is no preemption during the execution of τ_t when $M_t \in \mathcal{N}$, $\phi^f_{t,i}$ refers to the same invocation of the preempting task as $\phi^s_{t,i}$.

Theorem 3.3.2. *The inter-graph interference is conservatively considered in schedule time bound, which means that $\overline{Delay^h_t}$ and $\overline{Preempt^W_t}$ that use phases $\phi^r_{t,i}$, $\phi^s_{t,i}$, and $\phi^f_{t,i}$ are*

conservative.

Proof. See subsection 3.6.2. □

3.3.3 Convergence of The HPA Technique

Since time bounds are dependent on each other, the proposed technique computes the schedule time bounds of all tasks iteratively until every time bound is converged or any WCRT becomes larger than the deadline. Since iteration is performed with non-linear equations, it is very challenging to prove the convergence formally. To tackle this challenge, after a predefined number of iterations, we restrict the upper bounds RB_i^u, SB_i^u, and FB_i^u to increase monotonically by discarding new bound value if it is smaller than the previous value. Similarly, the lower bounds RB_i^l, SB_i^l, and FB_i^l are also restricted to decrease monotonically. Since the schedule time bounds cannot grow infinitely, all time bounds are eventually converged.

3.4 Optimization Techniques

In this section, we describe two optimization techniques to tighten the time bounds by considering the dependency relation. If two tasks have a dependency, one task cannot interfere the other even if the time bounds of two tasks intersect. In the *Exclusion Set Management* technique, we manage the set of tasks that cannot preempt the target task by tracing the dependency relation among tasks. In the *Duplicate Preemption Elimination* technique, on the other hand, we avoid redundant preemption where a single preemption by a higher priority task is considered multiple times in two dependent tasks. When a higher priority task may preempt both tasks that have a dependency, it is observed that preempting the descendant task results in the longer response time than the case of preempting the ancestor task.

3.4.1 Exclusion Set Management

The exclusion technique manages for each task τ_i a set $\mathcal{EX}_{\tau_i}$ which includes tasks that are guaranteed to have no possibility of preempting τ_i. It is obvious that successor tasks belong to this set. If τ_i always preempts one of the predecessors of τ_s, τ_s cannot preempt τ_i since it will always be scheduled after τ_i. In addition, if τ_j is excluded by τ_i, then all $\tau_s \in \mathcal{EX}_{\tau_j}$ are also excluded by τ_i. In summary, the exclusion set $\mathcal{EX}_{\tau_i}$ becomes

$$\mathcal{EX}_{\tau_i} = \left\{ \tau_s \;\middle|\; \begin{array}{c} \tau_s \in \mathit{descendant}(\tau_i) \; or \\ \tau_i \in \bigcup_{\tau_p \in \mathit{ancestor}(\tau_s)} (\mathcal{A}_{\tau_p} \cup \mathcal{D}_{\tau_p}) \; or \\ \tau_s \in \bigcup_{\tau_j \in \mathcal{EX}_{\tau_i}} \mathcal{EX}_{\tau_j} \end{array} \right\} \tag{3.22}$$

where $\mathit{ancestor}(\tau_s)$ is a set of ancestors of τ_s and $\mathit{descendant}(\tau_i)$ is a set of descendants of τ_i. Since there is a cyclic dependency in equation (3.22), iterative computation is required for $\mathcal{EX}_{\tau_i}$, initially defined by $\mathit{descendant}(\tau_i)$. After time bound computation, it is updated based on $\mathcal{A}_{\tau_i}$ and $\mathcal{D}_{\tau_i}$. Sets $\mathcal{A}_{\tau_t}$, $\mathcal{B}_{\tau_t}$, $\mathcal{C}_{\tau_t}$, $\mathcal{D}_{\tau_t}$, $\mathcal{E}_{\tau_t}$, and $\mathcal{F}_{\tau_t}$ are modified to have an additional condition $\tau_s \notin \mathcal{EX}_{\tau_t}$. It is obvious that the exclusion technique does not affect the conservativeness of the proposed technique.

3.4.2 Graph Reconstruction

Since the distance $\phi^r_{t,i}$ is computed as the minimum expected request time of τ_i among predecessors for conservative estimation, the number of appearances is likely to be overestimated when there are multiple predecessors. The example of Figure 3.10 shows a case where multiple predecessors may induce overestimation on the WCRT. WCRT estimation by the proposed technique is displayed in Figure 3.10 (b). After τ_1 finishes with one preemption by τ_0, three tasks τ_2, τ_3 and τ_4 are released at the same time. τ_2 is not interfered by τ_0 since the minimum distance $\phi^r_{2,0}$ is larger than its execution time, and $\phi^f_{2,0}$ points the second τ_0 appearance. τ_3 and τ_4 are executed in order with one preemption by the second τ_0 appearance. $\phi^f_{3,0}$ and $\phi^f_{4,0}$ points the third τ_0 ap-

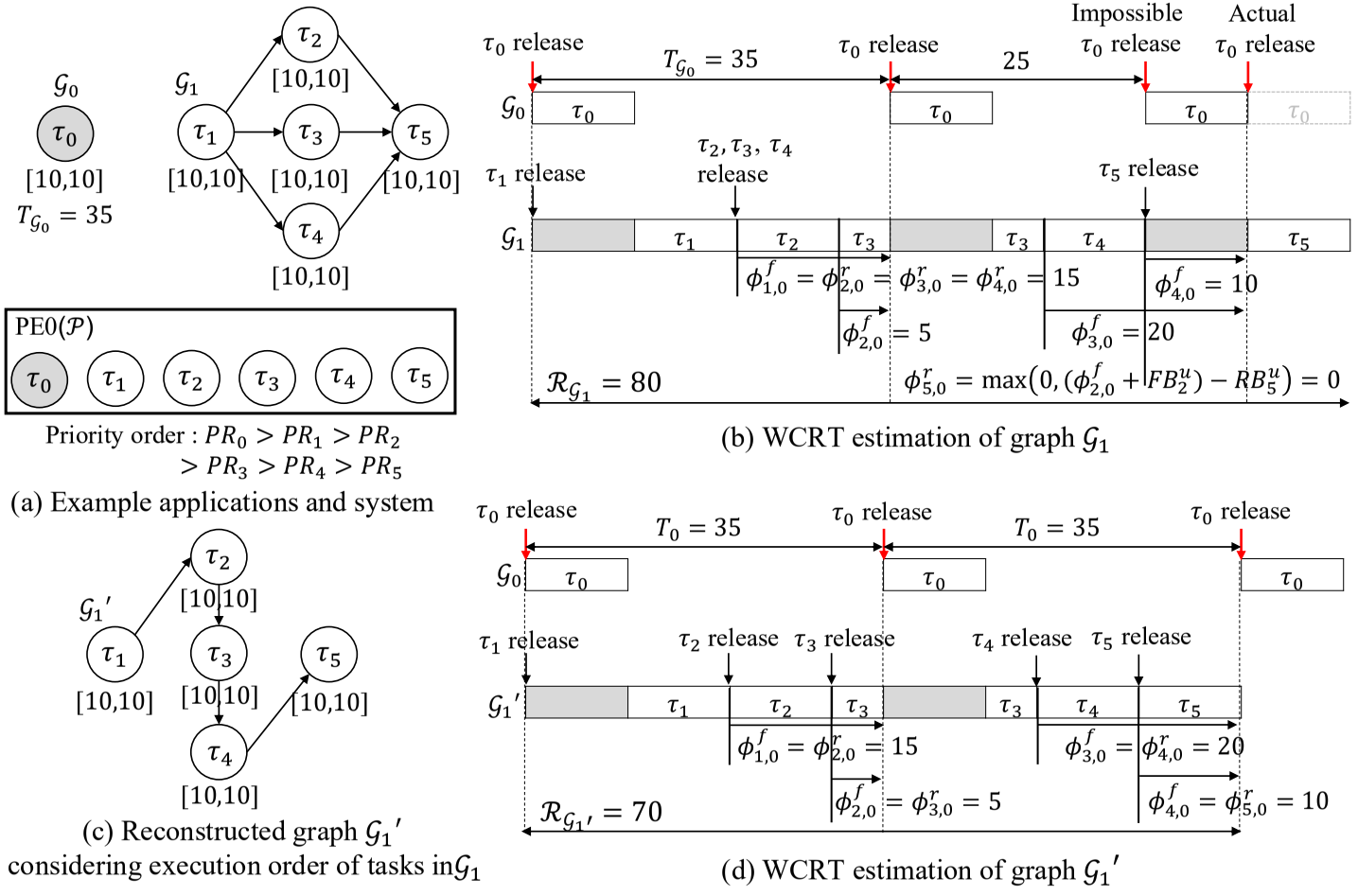


Figure 3.10: Overestimation when computing the distance from multiple predecessors

Algorithm 1 Graph reconstruction algorithm that reconstruct a task graph $\mathcal{G}$

1: $O \leftarrow \emptyset$; $T \leftarrow \mathcal{V}_{\mathcal{G}}$
2: **repeat**
3: $C \leftarrow \{\tau_s | \tau_s \in T, T \cap pred(\tau_s) = \emptyset\}$
4: $\tau_c \leftarrow$pick one task from C
5: $O \leftarrow \{\tau_s | \tau_s \in C, M_s = M_c, pred(\tau_s) = pred(\tau_c)\}$
6: $T \leftarrow T - O$
7: **repeat**
8: $O \leftarrow O \cup \{\tau_s | \tau_s \in T, M_s = M_c, pred(\tau_s) \subset O\}$
9: **until** O is not changed
10: $T \leftarrow T - O$
11: Sort O by scheduling order
12: $\mathcal{E}_{\mathcal{G}} \leftarrow \mathcal{E}_{\mathcal{G}} \cup \{(O[i], O[i+1]) \mid 0 \leq i < |O| - 1\}$
13: **until** $T = \emptyset$
14: $\mathcal{E}_{\mathcal{G}} \leftarrow \mathcal{E}_{\mathcal{G}} - \{(\tau_s, \tau_d) | (\tau_s, \tau_d) \in E_G, \exists_{\tau_c}(\tau_c \in succ(\tau_s), \text{there is a path from } \tau_c \text{ to } \tau_d)\}$

pearance. When we consider τ_5 as target task, we choose the minimum distance among predecessors τ_2, τ_3, and τ_4 which see the next appearance of τ_0 from their finish time as 35, 70, and 70 respectively. Hence τ_5 considers the minimum value as $\max(-(0+0-0), \min(35-60, 70-60, 70-60)) = 0$ from equation (3.15) where 60 is RB_5^u, which is earlier than actual appearance. Actually, tasks in the graph $\mathcal{G}_1$ are executed in order according to their priorities as depicted in the graph $\mathcal{G}_1'$ in Figure 3.10 (c). If dependency edges are reconstructed as Figure 3.10 (c), all tasks have only one predecessor so that the minimum distance to the real-time task is directly inherited from the predecessor. Hence for the target task τ_5, it can see the next appearance of τ_0 from its release time as $\max(-(0+0-0), 70-60) = 10$ which is exact appearance of τ_0. Since τ_0 appears after τ_5 finishes, the estimated WCRT of the reconstructed graph becomes 70. It is observed that finding an equivalent graph that has smaller number of dependency edges than the original graph is beneficial to reduce this kind of overestimation. Thus we propose an algorithm that reconstructs the graph by removing unnecessary dependency edges.

Algorithm 1 presents the pseudo code for graph reconstruction. Figure 3.11 shows an example of how a graph is reconstructed by the proposed algorithm. A set T contains tasks that are not yet visited and is initialized by $\mathcal{V}_{\mathcal{G}}$ (line 1). The aim of the algorithm

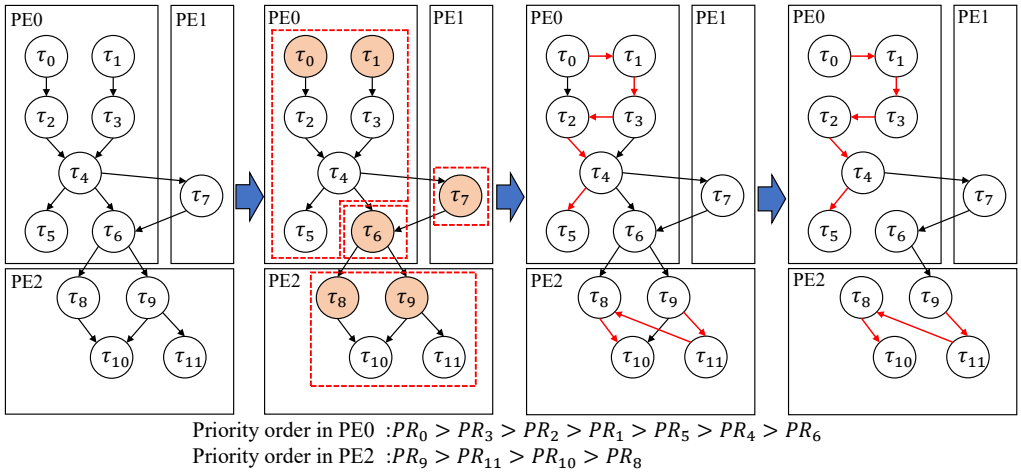


Figure 3.11: Graph reconstruction progress for an example task graph

is to repeatedly collect a set of tasks that are always executed in order among tasks in a set T into a set O. For this, we traverse a graph in the breadth-first search manner. At first, we select a seed task of set O, which is a source task in the first round of repetition (lines 3-4). And we initialize set O with the tasks that have the same predecessor set as the seed task and mapped onto the same processing element (lines 5). At this point, set O consists of tasks that are released at the same time on the same processing element. In Figure 3.11, source tasks τ_0 and τ_1 are put into set O. Then we traverse the successors of tasks in set O and add them into set O if the predecessor tasks all belong to set O and mapped to the same processing element (lines 7-9). Since the releases of newly added tasks are solely determined by tasks in set O, the schedule order of tasks in set O is fixed. In Figure 3.11, τ_2, τ_3, τ_4, and τ_5 are added into O. Task τ_6 cannot be assigned to the same ordered set because its release time is affected by τ_7 which is mapped to another processing element. When there is no such a task any more (line 9), we obtain a set of tasks O whose execution order is always fixed. Then, tasks in set O are sorted in the topological and priority order, which is the execution order (line 11). Edges are added according to the scheduling order for task set O (line 12). After removing the tasks in set O from set T, we repeat this process until all tasks are assigned to ordered sets (line

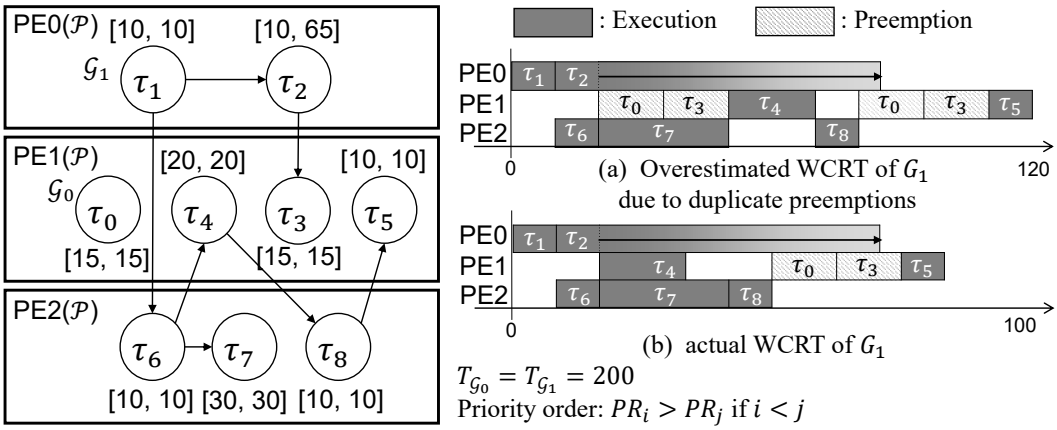


Figure 3.12: An example of duplicate preemption elimination

13). Subsequently, ordered sets $\{\tau_6\}$, $\{\tau_7\}$, and $\{\tau_9, \tau_{11}, \tau_8, \tau_{10}\}$ are constructed. Finally, unnecessary edges are removed from edge set $\mathcal{E}_{\mathcal{G}}$ (line 14). An edge $\varepsilon = (\tau_s, \tau_d)$ can be removed if there is another path from τ_s to τ_d.

Note that after removing unnecessary edges from the graph, we get a new graph in which each task has only one predecessor in the ordered set as illustrated in Figure 3.11. Thus there will be less overestimation for the computation of the next request of a preempting task to the target task. Graph reconstruction algorithm is applied before computing the schedule time bounds.

Theorem 3.4.1. *The original graph and the reconstructed graph are equivalent in terms of task execution order.*

Proof. See subsection 3.6.3. □

3.4.3 Duplicate Preemption Elimination

In our baseline technique, preemptions may occur redundantly; Figure 3.12 (a) shows an elaborated example that experiences two types of duplicate preemptions. The first type of duplicate preemption may occur between tasks in the same task graph in case a higher priority task has large release time variation. In the scheduling time bound

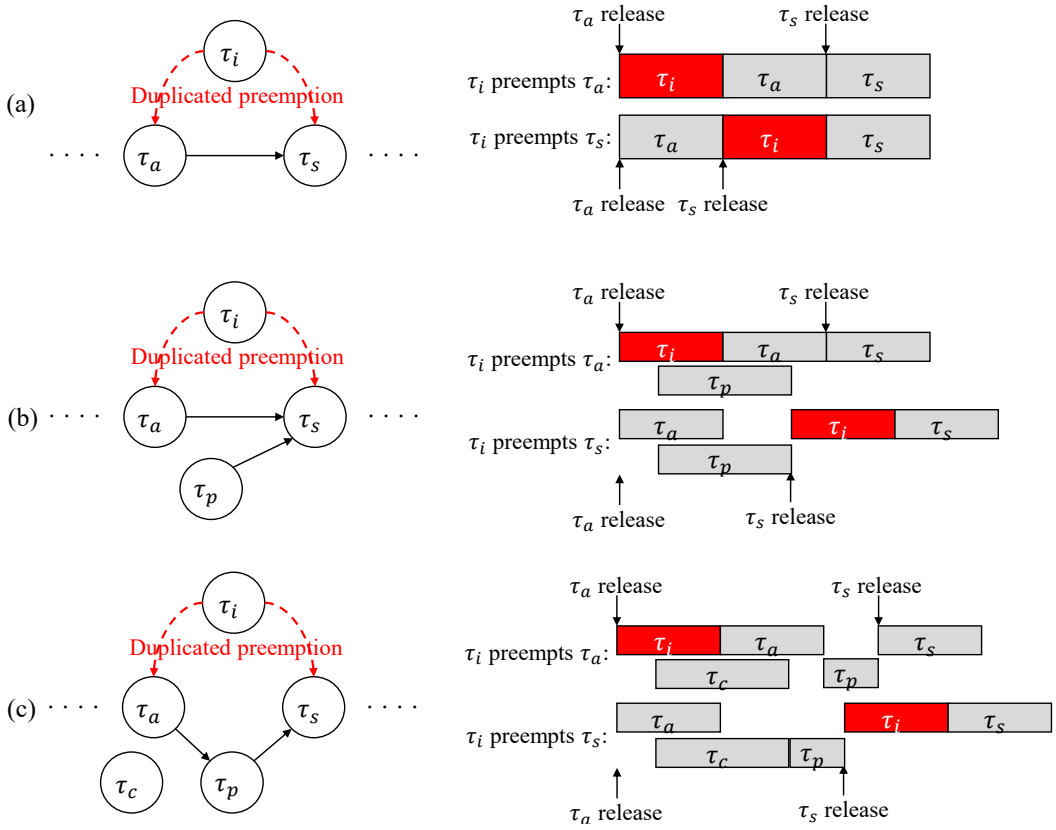


Figure 3.13: Later preemption on the dependency path gives worse response time than earlier preemption

analysis, we detect the preemption possibility by checking if a higher priority task can be released during task execution. In Figure 3.12 (a), τ_3 can preempt both τ_4 and τ_5 because its release time varies widely between 20 and 75.

The second type of duplicate preemption occurs between tasks in different task graphs. In the phase adjustment technique, it is assumed that a preempting task preempts a predecessor task first. In the example of Figure 3.12, τ_0 preempts τ_4 and phase adjustment is performed afterwards. The request phase of τ_5 to τ_0 is reset to $-(\Psi_5 + RB_0^u - RB_0^l)$ since τ_8 is assigned to a different processing element, according to equation (3.15). Then, τ_5 experiences another preemption by τ_0. As a result, the WCRT is over-estimated as illustrated in Figure 3.12 (a) that contains both types of duplicate preemptions.

To avoid duplicate preemptions, an optimization technique is devised to check the duplicated preemptions on the dependency path and to remove duplicate preemptions on the ancestors. The proposed technique is based on an abstruse fact that a later preemption gives worse response time than an earlier preemption when there are duplicate preemptions. Figure 3.13 shows the simpler examples where preempting the target task τ_s shows a larger response time than preempting its ancestor task τ_a.

In Figure 3.13 (a), since two tasks τ_a and τ_s are linearly dependent and mapped on a same PE, the finish time of τ_s remains the same when τ_i preempts either τ_a or τ_s. In case τ_s has a predecessor task that is mapped to another PE, preempting τ_s results in larger response time than preempting τ_a in Figure 3.13 (b). If τ_i does not preempt τ_a, τ_s is released after τ_p finishes its execution, making an idle time between the finish time of τ_a and the release time of τ_s. Thus the finish time of τ_s becomes larger if τ_i preempts τ_s. Finally, consider the case in Figure 3.13 (c) where task τ_p that belongs to the path from τ_a to τ_s is mapped on the other processing element. When the finish time of τ_a is reduced by moving the preemption of τ_i to τ_s, the release time of child task τ_p may not be reduced as much as parent task τ_a, due to τ_c. Then similarly to the case of Figure 3.13 (b), preempting τ_s gives longer finish time. Theorem 3.4.2 summarizes that a later preemption on the dependency path gives a worse response time than an earlier preemption.

Theorem 3.4.2. *If a common preemptor τ_p can preempt either an ancestor task τ_a or the target task τ_t, then the finish time of τ_t is no smaller when τ_p preempts τ_t rather than τ_a.*

Proof. See subsection 3.6.4. □

According to Theorem 3.4.2, we can safely reduce the overestimation from duplicate preemption by finding a duplicate preemption on the ancestors of the target task and removing the preemption.

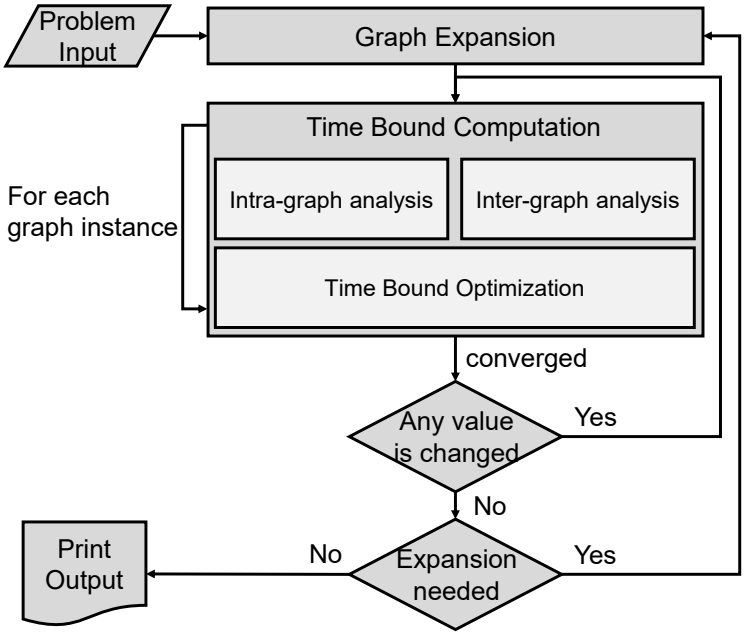


Figure 3.14: The overall flow of the proposed HPA technique for supporting arbitrary deadlines

3.5 Supporting Arbitrary Deadline

If the deadline of a task graph is greater than the period, multiple invocations of the task graph may be executed in a pipelined fashion. Then, we need to consider the intra-graph interference from the overlapped execution of those invocations.

Figure 3.14 shows how the overall flow changes when arbitrary deadlines are allowed in the task model. Initially, we create $\lceil \frac{D_{\mathcal{G}}}{T_{\mathcal{G}}} \rceil + 1$ graph instances for each task graph $\mathcal{G}$ in the *Graph Expansion* module. Let the i-th instance of task τ_t be $\tau_{t(i)}$. Then the release time bounds of a task $\tau_{t(i)}$ are computed as follows:

$$RB^{l}_{t(i)} = \begin{cases} i \times T_{\mathcal{G}_{\tau_t}}, & \text{if } \tau_t \text{ is a source task} \\ \max_{\tau_{p(i)} \in pred(\tau_{t(i)})} FB^{l}_{p(i)}, & \text{otherwise} \end{cases} \tag{3.23}$$

$$RB^{u}_{t(i)} = \begin{cases} i \times T_{\mathcal{G}_{\tau_t}} + J_{\mathcal{G}_{\tau_t}}, & \text{if } \tau_t \text{ is a source task} \\ \max_{\tau_{p(i)} \in pred(\tau_{t(i)})} FB^{u}_{p(i)}, & \text{otherwise} \end{cases} \tag{3.24}$$

In case τ_t is a source task, its release time depends on the graph activation time. Otherwise, it is released when all predecessors finish. If the previous instance does not finish until the release time of the next instance, the next instance should wait for the finish of the previous instance. We handle this by giving higher priorities to earlier instances than later instances: $PR_{t(i)} > PR_{t(j)}$ iff $i < j$. For conservative estimation, we increase the time bounds of each task as the instance index increases as follows.

$$\begin{aligned} RB^l_{t(i)} &\leq (RB^l_{t(i-1)} + T_{\mathcal{G}_{\tau_t}}), \quad RB^u_{t(i)} \geq (RB^u_{t(i-1)} + T_{\mathcal{G}_{\tau_t}}) \\ SB^l_{t(i)} &\leq (SB^l_{t(i-1)} + T_{\mathcal{G}_{\tau_t}}), \quad SB^u_{t(i)} \geq (SB^u_{t(i-1)} + T_{\mathcal{G}_{\tau_t}}) \\ FB^l_{t(i)} &\leq (FB^l_{t(i-1)} + T_{\mathcal{G}_{\tau_t}}), \quad FB^u_{t(i)} \geq (FB^u_{t(i-1)} + T_{\mathcal{G}_{\tau_t}}) \end{aligned} \tag{3.25}$$

Note that the first instance $\tau_{t(0)}$ has no restriction and equation (3.25) holds from the second instance. This restriction guarantees the convergence of the proposed technique while it may incur over-estimation that is not negligible. Further optimization to reduce the over-estimation is left as a future work.

We apply the STBA analysis technique to compute the interference among graph instances of the same task graph. While we use the same intra-graph interference formula without modification, we should consider the interference from task instances that have overlapped time bounds with the target task.

In the inter-graph interference analysis, we consider the worst-case appearance of a task in another task graph by period shift Ψ_t. For easy proof of the conservativeness, we make the period shift Ψ_t of a task be the same for all graph instances by choosing the maximum value among all of instances $\tau_{t(i)}$: $\Psi_t = \max_{\forall_i} \Psi_{t(i)}$ where $\Psi_{t(i)}$ is period shift value computed for $\tau_{t(i)}$ using equation (3.14). Then Ψ_t which is the maximum among all $\Psi_{t(i)}$ is used in equation (3.15). We reset the request phase $\phi^r_{t,i}$ to Ψ_t for all source task instances in equation (3.15).

After the analysis of the currently expanded graph instance set is completed, the relative schedule time bounds of the last task instance are recorded as the schedule time

bounds of a task τ_t:

$$\begin{aligned} RB_t^l &= RB_{t(last(t))}^l - last(t) \times T_{\mathcal{G}_{\tau_t}} \quad RB_t^u = RB_{t(last(t))}^u - last(t) \times T_{\mathcal{G}_{\tau_t}} \\ SB_t^l &= SB_{t(last(t))}^l - last(t) \times T_{\mathcal{G}_{\tau_t}} \quad SB_t^u = SB_{t(last(t))}^u - last(t) \times T_{\mathcal{G}_{\tau_t}} \\ FB_t^l &= FB_{t(last(t))}^l - last(t) \times T_{\mathcal{G}_{\tau_t}} \quad FB_t^u = FB_{t(last(t))}^u - last(t) \times T_{\mathcal{G}_{\tau_t}} \end{aligned} \tag{3.26}$$

where $last(t)$ is the last index of currently created τ_t job instances. We check whether the schedule time bound by equation (3.26) is changed in the "*Expansion needed*" module in Figure 3.14. If it is changed, we create one more graph instance of the corresponding task graph and perform the analysis again. We repeat graph expansion and WCRT analysis until all schedule time bounds are converged.

Theorem 3.5.1. *The computed worse-case response times are conservative for the arbitrary deadline model.*

Proof. See subsection 3.6.5. □

3.6 Proof of Conservativeness

In this section, we prove lemmas and theorems presented in sections 3.3, 3.4, and 3.5. As listed in Table 3.2, $r(\tau_t)$, $s(\tau_t)$, and $f(\tau_t)$ denote the actual release time, start time, and finish time of task τ_t, respectively.

3.6.1 Proof of Theorem 3.3.1

We prove that the schedule time bounds computed in the proposed technique are all conservative. We make several definitions and lemmas in this subsection.

Definition 3.6.1. *In our task graph model, the release time of a task is the maximum*

finish time of its immediate predecessors, which is summarized as

$$r(\tau_t) = \begin{cases} r(\mathcal{G}_{\tau_t}), & \text{if } \tau_t \text{ is a source task} \\ \max_{\tau_p \in pred(\tau_t)} f(\tau_p), & \text{otherwise} \end{cases}$$

where $r(\mathcal{G}_{\tau_t})$ denotes the release time of the task graph and $pred(\tau_t)$ is the immediate predecessor set of task τ_t.

Lemma 3.6.1. *RB_t^l and RB_t^u are conservative, or $RB_t^l \leq r(\tau_t) \leq RB_t^u$.*

Proof. If τ_t is a source task, $RB_t^l = 0 \leq r(\tau_t) \leq J_{\mathcal{G}_{\tau_t}} = RB_t^u$ since $0 \leq r(\mathcal{G}_{\tau_t}) \leq J_{\mathcal{G}_{\tau_t}}$.

For a non-source task τ_t, assume that it holds for all predecessor tasks of τ_t. Since $RB_t^l = \max_{\tau_p \in pred(\tau_t)} FB_p^l \leq \max_{\tau_p \in pred(\tau_t)} f(\tau_p) = r(\tau_t) \leq \max_{\tau_p \in pred(\tau_t)} FB_p^u = RB_t^u$, $RB_t^l \leq r(\tau_t) \leq RB_t^u$.

By induction, Lemma 3.6.1 holds. □

Definition 3.6.2. *The start time is formally defined as*

$$s(\tau_t) = \max(r(\tau_t), \max_{\tau_s \in \Gamma_{\tau_t}} f(\tau_s))$$

where $\Gamma_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t, r(\tau_t) < f(\tau_s), s(\tau_s) \leq s(\tau_t)\}$ for the preemptive scheduling policy, and $\Gamma_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, (PR_s > PR_t, r(\tau_t) < f(\tau_s), s(\tau_s) \leq s(\tau_t))$ or $(PR_s < PR_t, s(\tau_s) < r(\tau_t) < f(\tau_s))\}$ for the non-preemptive scheduling policy.

Lemma 3.6.2. *SB_t^l is conservative, or $SB_t^l \leq s(\tau_t)$.*

Proof. Since $RB_t^l \leq r(\tau_t)$, we need to prove $\max_{\tau_s \in \mathcal{A}_{\tau_t}} FB_s^l \leq s(\tau_t)$.

$$\begin{aligned} \max_{\tau_s \in \mathcal{A}_{\tau_t}} FB_s^l &\leq \max(\max_{\tau_s \in \Gamma_{\tau_t}} FB_s^l, \max_{\tau_s \in \mathcal{A}_{\tau_t} - \Gamma_{\tau_t}} FB_s^l) \\ &\leq \max(\max_{\tau_s \in \Gamma_{\tau_t}} f(\tau_s) \max_{\tau_s \in \mathcal{A}_{\tau_t} - \Gamma_{\tau_t}} f(\tau_s)). \end{aligned}$$

Proof will be completed by showing that $\max_{\tau_s \in \mathcal{A}_{\tau_t} - \Gamma_{\tau_t}} f(\tau_s) \leq r(\tau_t)$ since

$$\max(\max_{\tau_s \in \Gamma_{\tau_t}} f(\tau_s), \max_{\tau_s \in \mathcal{A}_{\tau_t} - \Gamma_{\tau_t}} f(\tau_s)) \leq \max(\max_{\tau_s \in \Gamma_{\tau_t}} f(\tau_s), r(\tau_t)) = s(\tau_t)$$

We define $\mathcal{A}^{\mathcal{P}}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t, RB^l_t < FB^l_s, SB^u_s \leq SB^l_t\}$ and $\mathcal{A}^{\mathcal{N}}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s < PR_t, SB^u_s < RB^l_t < FB^l_s\}$. Then $\mathcal{A}_{\tau_t} = \mathcal{A}^{\mathcal{P}}_{\tau_t}$ for preemptive scheduling policy and $\mathcal{A}_{\tau_t} = \mathcal{A}^{\mathcal{P}}_{\tau_t} \cup \mathcal{A}^{\mathcal{N}}_{\tau_t}$ for non-preemptive scheduling policy. Similarly, We define $\Gamma^{\mathcal{P}}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t, r(\tau_t) < f(\tau_s), s(\tau_s) \leq s(\tau_t)\}$ and $\Gamma^{\mathcal{N}}_{\tau_t} = \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s < PR_t, s(\tau_s) < r(\tau_t) < f(\tau_s)\}$.

1. **Preemptive case**

$$\begin{aligned}
\mathcal{A}_{\tau_t} - \Gamma_{\tau_t} &= \mathcal{A}^{\mathcal{P}}_{\tau_t} \cap (\Gamma^{\mathcal{P}}_{\tau_t})^c \\
&= \mathcal{A}^{\mathcal{P}}_{\tau_t} \cap (\{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t\}^c \\
&\qquad \cup \{\tau_s | r(\tau_t) < f(\tau_s)\}^c \cup \{\tau_s | s(\tau_s) \leq s(\tau_t)\}^c) \\
&= \emptyset \cup (\mathcal{A}^{\mathcal{P}}_{\tau_t} \cap \{\tau_s | r(\tau_t) \geq f(\tau_s)\}) \cup \emptyset \\
&\subseteq \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, r(\tau_t) \geq f(\tau_s)\}
\end{aligned}$$

2. **non-preemptive case**

$$\begin{aligned}
\mathcal{A}_{\tau_t} - \Gamma_{\tau_t} &= (\mathcal{A}^{\mathcal{P}}_{\tau_t} \cup \mathcal{A}^{\mathcal{N}}_{\tau_t}) - (\Gamma^{\mathcal{P}}_{\tau_t} \cup \Gamma^{\mathcal{N}}_{\tau_t}) = (\mathcal{A}^{\mathcal{P}}_{\tau_t} - \Gamma^{\mathcal{P}}_{\tau_t}) \cup (\mathcal{A}^{\mathcal{N}}_{\tau_t} - \Gamma^{\mathcal{N}}_{\tau_t}) \\
&= (\mathcal{A}^{\mathcal{P}}_{\tau_t} \cap (\Gamma^{\mathcal{P}}_{\tau_t})^c) \cup (\mathcal{A}^{\mathcal{N}}_{\tau_t} \cap (\Gamma^{\mathcal{N}}_{\tau_t})^c) \\
&= (\mathcal{A}^{\mathcal{P}}_{\tau_t} \cap (\Gamma^{\mathcal{P}}_{\tau_t})^c) \cup \\
&\qquad (\mathcal{A}^{\mathcal{N}}_{\tau_t} \cap (\{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s < PR_t\}^c \\
&\qquad\qquad \cup \{\tau_s | s(\tau_s) < r(\tau_t)\}^c \cup \{\tau_s | r(\tau_t) < f(\tau_s)\}^c) \\
&= (\mathcal{A}^{\mathcal{P}}_{\tau_t} \cap (\Gamma^{\mathcal{P}}_{\tau_t})^c) \cup (\emptyset \cup \emptyset \cup (\mathcal{A}^{\mathcal{N}}_{\tau_t} \cap \{\tau_s | r(\tau_t) \geq f(\tau_s)\})) \\
&\subseteq \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, r(\tau_t) \geq f(\tau_s)\}
\end{aligned}$$

From (1) and (2), $\mathcal{A}_{\tau_t} - \Gamma_{\tau_t} \subseteq \{\tau_s | \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, r(\tau_t) \geq f(\tau_s)\}$ for both preemptive and non-preemptive cases. Therefore, $\max_{\tau_s \in \mathcal{A}_{\tau_t} - \Gamma_{\tau_t}} f(\tau_s) \leq r(\tau_t)$ □

Definition 3.6.3. *We define* $LP_t^{intra} = \{\tau_s \mid \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s < PR_t\}$*, a set of lower priority tasks that may interfere* τ_t*, and* $HP_t^{intra} = \{\tau_s \mid \tau_s \in \mathcal{G}_{\tau_t}, M_s = M_t, PR_s > PR_t\}$*, a set of higher priority tasks that may preempt* τ_t*.*

Definition 3.6.4. $P_s[a,b]$ *is defined as the execution amount of* τ_s *in time window* $[b,a]$*. Then following conditions hold:*

1. $\sum_{\tau_s \in HP_t^{intra}} P_s[a,b] \leq a-b$
2. $P_s[a,b] = 0$ *if* $f(\tau_s) \leq b$ *or* $s(\tau_s) \geq a$
3. $P_s[a,b] \leq \min(C_s^u, \min(f(\tau_s), a) - \max(s(\tau_s), b))$

Lemma 3.6.3. SB_t^u *is conservative, or* $s(\tau_t) \leq SB_t^u$*.*

Proof. We will consider the interference by lower priority tasks and higher priority tasks separately.

1. For the interference by a lower priority task we prove that $s(\tau_t) \leq RB_t^u + Delay_t^l$. For a preemptive scheduling policy, it is impossible for a lower priority task to interfere τ_t so that the interference amount is zero. For a non-preemptive scheduling policy, if all predecessors are mapped to the same processing element, there is no time interval between the finish time of the latest predecessor task τ_p and the release time of τ_t. Thus no lower priority task can start after τ_p finishes and before τ_t is released and the interference amount is zero. Otherwise, at most one lower priority task τ_s can delay the execution of τ_t if τ_s starts before and finishes after τ_t is released, or $s(\tau_s) < r(\tau_t) < f(\tau_s)$. We compute the actual start time without the preemptions from higher priority tasks as

$$s(\tau_t) = r(\tau_t) + \max_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t) < f(\tau_s)} (f(\tau_s) - r(\tau_t))$$

$\forall_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t) < f(\tau_s)} (f(\tau_s) - r(\tau_t) \leq C_s^u)$ holds since $f(\tau_s) - s(\tau_s) \leq C_s^u$ for task τ_s under the non-preemptive policy. Then

$$
\begin{aligned}
s(\tau_t) &= r(\tau_t) + \max_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t) < f(\tau_s)} \min(C_s^u, f(\tau_s) - r(\tau_t)) \\
&= r(\tau_t) + \max(0, \max_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t)} \min(C_s^u, f(\tau_s) - r(\tau_t))) \\
&= \max(r(\tau_t), \max_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t)} \min(r(\tau_t) + C_s^u, f(\tau_s))) \\
&\leq \max(RB_t^u, \max_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t)} \min(RB_t^u + C_s^u, f(\tau_s))) \\
&= RB_t^u + \max(0, \max_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t)} \min(C_s^u, f(\tau_s) - RB_t^u)) \\
&= RB_t^u + \max_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t), RB_t^u < f(\tau_s)} \min(C_s^u, f(\tau_s) - RB_t^u) \\
&\leq RB_t^u + \max_{\tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t), RB_t^u < FB_s^u} \min(C_s^u, FB_s^u - RB_t^u) \\
&\leq RB_t^u + \max_{\tau_s \in \mathcal{B}_{\tau_t}} \min(C_s^u, FB_s^u - RB_t^u) = RB_t^u + Delay_t^l
\end{aligned}
$$

The last inequality holds since $\{\tau_s | s(\tau_s) < r(\tau_t)\} \subseteq \{\tau_s | SB_s^l < RB_t^u\}$ and $\{\tau_s \mid \tau_s \in LP_t^{intra}, s(\tau_s) < r(\tau_t), RB_t^u < FB_s^u\} \subseteq \{\tau_s | \tau_s \in LP_t^{intra}, SB_s^l < RB_t^u < FB_s^u\} = \mathcal{B}_{\tau_t}$.

2. Consider the interference from higher priority tasks. We compute the actual start time without the lower priority blocking as

$$
\begin{aligned}
s(\tau_t) &= r(\tau_t) + \sum_{\tau_s \in HP_t^{intra}} P_s[s(\tau_t), r(\tau_t)] \\
&\leq r(\tau_t) + \sum_{\tau_s \in HP_t^{intra}} \{P_s[\max(s(\tau_t), RB_t^u), RB_t^u] + P_s[RB_t^u, r(\tau_t)]\}
\end{aligned}
$$

Since $\sum_{\tau_s \in HP_t^{intra}} P_s[RB_t^u, r(\tau_t)] \leq RB_t^u - r(\tau_t)$ from Definition 3.6.4.1,

$$
s(\tau_t) \leq RB_t^u + \sum_{\tau_s \in HP_t^{intra}} P_s[\max(s(\tau_t), RB_t^u), RB_t^u]
$$

In case $s(\tau_t) \leq RB_t^u$, $s(\tau_t) \leq RB_t^u \leq RB_t^u + Delay_t^l + Delay_t^h = SB_t^u$. So we consider only the case $s(\tau_t) > RB_t^u$.

$$s(\tau_t) \leq RB_t^u + \sum_{\tau_s \in HP_t^{intra}} P_s[s(\tau_t), RB_t^u]$$

$P_s[s(\tau_t), RB_t^u]$ is not zero when $s(\tau_s) < s(\tau_t), RB_t^u < f(\tau_s)$ according to Definition 3.6.4.2.

$$\begin{aligned} s(\tau_t) &\leq RB_t^u + \sum_{\tau_s \in HP_t^{intra}, s(\tau_s) < s(\tau_t), RB_t^u < f(\tau_s)} P_s[s(\tau_t), RB_t^u] \\ &\leq RB_t^u + \sum_{\tau_s \in C_{\tau_t}} P_s[s(\tau_t), RB_t^u] \end{aligned}$$

According to Definition 3.6.4.3,

$$\begin{aligned} s(\tau_t) &\leq RB_t^u + \sum_{\tau_s \in C_{\tau_t}} \min(C_s^u, \min(f(\tau_s), s(\tau_t)) - \max(s(\tau_s), RB_t^u)) \\ &\leq RB_t^u + \sum_{\tau_s \in C_{\tau_t}} \min(C_s^u, f(\tau_s) - RB_t^u) \\ &\leq RB_t^u + \sum_{\tau_s \in C_{\tau_t}} \min(C_s^u, FB_s^u - RB_t^u) = RB_t^u + Delay_t^h \end{aligned}$$

Merging two cases 1 and 2, $s(\tau_t) \leq RB_t^u + Delay_t^l + Delay_t^h = SB_t^u$. □

Lemma 3.6.4. *FB_t^l is conservative, or $FB_t^l \leq f(\tau_t)$.*

Proof. Since there is no preemption delay involved under a non-preemptive scheduling policy, we consider the preemptive scheduling policy only.

$$f(\tau_t) \geq s(\tau_t) + C_t^l + \sum_{\tau_s \in HP_t^{intra}, s(\tau_t) \leq s(\tau_s) \leq f(\tau_t)} C_s^l$$

where the last term accounts for tasks that start to execute during τ_t is running, and preempt τ_t. Since $s(\tau_t) + \sum_{\tau_s \in HP_t^{intra}, s(\tau_t) \leq s(\tau_s) \leq f(\tau_t)} C_s^l \geq SB_t^l + \sum_{\tau_s \in HP_t^{intra}, SB_t^l \leq s(\tau_s) \leq f(\tau_t)} C_s^l$,

$$
\begin{aligned}
f(\tau_t) &\geq s(\tau_t) + C_t^l + \sum_{\tau_s \in HP_t^{intra}, s(\tau_t) \leq s(\tau_s) \leq f(\tau_t)} C_s^l \\
&\geq SB_t^l + C_t^l + \sum_{\tau_s \in HP_t^{intra}, SB_t^l \leq s(\tau_s) \leq f(\tau_t)} C_s^l \\
&\geq SB_t^l + C_t^l + \sum_{\tau_s \in \mathcal{D}_{\tau_t}} C_s^l = SB_t^l + C_t^l + Preempt_t^l = FB_t^l
\end{aligned}
$$

The last inequality holds since $\{\tau_s \mid \tau_s \in HP_t^{intra}, SB_t^l \leq s(\tau_s) \leq f(\tau_t)\} \supseteq \{\tau_s \mid \tau_s \in HP_t^{intra}, SB_t^l \leq SB_s^l \leq SB_s^u \leq FB_t^l\} = \mathcal{D}_{\tau_t}$. □

Lemma 3.6.5. *FB_t^u is conservative, or $f(\tau_t) \leq FB_t^u$.*

Proof. Since there is no preemption delay involved under a non-preemptive scheduling policy, we consider the preemptive scheduling policy only. The actual finish time is defined as

$$
\begin{aligned}
f(\tau_t) &= r(\tau_t) + P_t[f(\tau_t), r(\tau_t)] + \sum_{\tau_s \in HP_t^{intra}} P_s[f(\tau_t), r(\tau_t)] \\
&\leq r(\tau_t) + C_t^u + \sum_{\tau_s \in HP_t^{intra}} \{P_s[\max(f(\tau_t), RB_t^u), RB_t^u] + P_s[RB_t^u, r(\tau_t)]\}
\end{aligned}
$$

We consider only the case $f(\tau_t) > SB_t^u \geq RB_t^u$ since if $f(\tau_t) \leq SB_t^u$ then $f(\tau_t) \leq SB_t^u + C_t^u + Preempt_t^W = FB_t^u$.

According to Definition 3.6.4.1, $RB_t^u - r(\tau_t) \geq \sum_{\tau_s \in HP_t^{intra}} P_s[RB_t^u, r(\tau_t)]$.

$$
\begin{aligned}
f(\tau_t) &\leq RB_t^u + C_t^u + \sum_{\tau_s \in HP_t^{intra}} P_s[f(\tau_t), RB_t^u] \\
&= RB_t^u + C_t^u + \sum_{\substack{\tau_s \in HP_t^{intra}, \\ SB_s^l < SB_t^u}} P_s[f(\tau_t), RB_t^u] + \sum_{\substack{\tau_s \in HP_t^{intra}, \\ SB_t^u \leq SB_s^l}} P_s[f(\tau_t), RB_t^u]
\end{aligned}
$$

Now we prove that $\sum_{\tau_s \in HP_t^{intra}, SB_s^l < SB_t^u} P_s[f(\tau_t), RB_t^u] \leq Delay_t^h$ and $\sum_{\tau_s \in HP_t^{intra}, SB_t^u \leq SB_s^l} P_s[f(\tau_t), RB_t^u] \leq Preempt_t^W$.

1. By Definition 3.6.4.2 and 3.6.4.3,

$$\sum_{\tau_s \in HP_t^{intra}, SB_s^l < SB_t^u} P_s[f(\tau_t), RB_t^u]$$

$$\leq \sum_{\substack{\tau_s \in HP_t^{intra}, SB_s^l < SB_t^u, \\ s(\tau_s) < f(\tau_t), RB_t^u < f(\tau_s)}} \min(C_s^u, \min(f(\tau_s), f(\tau_t)) - \max(s(\tau_s), RB_t^u))$$

$$\leq \sum_{\tau_s \in \mathcal{C}_{\tau_t}} \min(C_s^u, FB_s^u - RB_t^u) = Delay_t^h$$

2. $P_s[SB_t^u, RB_t^u] = 0$ if $SB_t^u \leq s(\tau_s)$ by Definition 3.6.4.2.

$$\sum_{\tau_s \in HP_t^{intra}, SB_t^u \leq SB_s^l} P_s[f(\tau_t), RB_t^u]$$

$$= \sum_{\tau_s \in HP_t^{intra}, SB_t^u \leq SB_s^l} \{P_s[f(\tau_t), SB_t^u] + P_s[SB_t^u, RB_t^u]\}$$

$$= \sum_{\tau_s \in HP_t^{intra}, SB_t^u \leq SB_s^l} P_s[f(\tau_t), SB_t^u]$$

By Definition 3.6.4.2 and 3.6.4.3,

$$\leq \sum_{\substack{\tau_s \in HP_t^{intra}, SB_t^u \leq SB_s^l, \\ s(\tau_s) < f(\tau_t), SB_t^u < f(\tau_s)}} \min(C_s^u, \min(f(\tau_s), f(\tau_t)) - \max(s(\tau_s), SB_t^u))$$

$$\leq \sum_{\tau_s \in \mathcal{E}_{\tau_t}} C_s^u = Preempt_t^W$$

From 1 and 2, $f(\tau_t) \leq (RB_t^u + Delay_t^h) + C_t^u + Preempt_t^W = FB_t^u$. □

Now we restate the theorem and prove it.

Theorem 3.3.1. *The HPA technique guarantees the conservativeness of every schedule time bound when there is no inter-graph interference.*

Proof. Theorem 3.3.1 is proven by lemmas 3.6.1, 3.6.2, 3.6.3, 3.6.4, and 3.6.5. □

3.6.2 Proof of Theorem 3.3.2

We first prove Lemma 3.3.1 and later define two more lemmas for the proof of Theorem 3.3.2.

Lemma 3.3.1. *Suppose τ_s is a task that may block both preempting tasks of other task graphs and the target task τ_t of the same task graph. Let $\mathcal{R}_{\tau_t}^{\alpha}$ be the worst-case response time of τ_t when τ_s blocks τ_t by α amount of time. Then, $\mathcal{R}_{\tau_t}^{\alpha} < \mathcal{R}_{\tau_t}^{\beta}$ if $0 \leq \alpha < \beta \leq C_s^u$.*

Proof. When τ_s blocks τ_t by α amount of time, the maximum preemption by a preempting task occurs when it is released right after the start time of τ_s, which is equal to $r(\tau_t) - C_s^u + \alpha$. Let A be a set of preempting tasks of τ_t. Then the conventional response time analysis tells that $\mathcal{R}_{\tau_t}^{\alpha}$ and $\mathcal{R}_{\tau_t}^{\beta}$ become

$$\mathcal{R}_{\tau_t}^{\alpha}(i) = \sum_{\tau_i \in A} \left\lceil \frac{\mathcal{R}_{\tau_t}^{\alpha}(i-1) + C_s^u - \alpha}{T_{\mathcal{G}_{\tau_i}}} \right\rceil \cdot C_i^u + \alpha + C_t^u$$

$$\mathcal{R}_{\tau_t}^{\beta}(i) = \sum_{\tau_i \in A} \left\lceil \frac{\mathcal{R}_{\tau_t}^{\beta}(i-1) + C_s^u - \beta}{T_{\mathcal{G}_{\tau_i}}} \right\rceil \cdot C_i^u + \beta + C_t^u$$

where $\mathcal{R}_{\tau_t}^{\alpha}(i)$ and $\mathcal{R}_{\tau_t}^{\beta}(i)$ are computed iteratively, and initial values $\mathcal{R}_{\tau_t}^{\alpha}(0)$ and $\mathcal{R}_{\tau_t}^{\beta}(0)$ are $\alpha + C_t^u$ and $\beta + C_t^u$, respectively. The iteration terminates when it converges. For $i = 0$, $\mathcal{R}_{\tau_t}^{\beta}(0) - \mathcal{R}_{\tau_t}^{\alpha}(0)$ is $\beta - \alpha$. For $i = 1$, The first terms in $\mathcal{R}_{\tau_t}^{\alpha}(1)$ and $\mathcal{R}_{\tau_t}^{\beta}(1)$ are equal since $\mathcal{R}_{\tau_t}^{\beta}(0) + C_s^u - \beta = \mathcal{R}_{\tau_t}^{\alpha}(0) + \beta - \alpha + C_s^u - \beta = \mathcal{R}_{\tau_t}^{\alpha}(0) + C_s^u - \alpha$. Hence $\mathcal{R}_{\tau_t}^{\beta}(1) - \mathcal{R}_{\tau_t}^{\alpha}(1) = \beta - \alpha$. For $i > 1$, we can derive $\mathcal{R}_{\tau_t}^{\beta}(i) - \mathcal{R}_{\tau_t}^{\alpha}(i) = \beta - \alpha$ in the same way of the case $i = 1$, which means that $\mathcal{R}_{\tau_t}^{\alpha}(i)$ and $\mathcal{R}_{\tau_t}^{\beta}(i)$ increase by the same amount at each iteration. Hence $\mathcal{R}_{\tau_t}^{\beta}$ is always larger than $\mathcal{R}_{\tau_t}^{\alpha}$. □

Lemma 3.6.6. *Period shift formulated as equation* (3.14) *is conservative, which means that $\sum_{\tau_i \in Z_{\tau_t}} \left\lceil \frac{\Delta t + \Psi_t + RB_i^u - RB_i^l}{T_{\mathcal{G}_{\tau_i}}} \right\rceil \cdot C_i^u$ is the upper bound of the inter-graph interference to τ_t during time interval $[RB_t^u, RB_t^u + \Delta t]$.*

Proof. We prove it by contradiction. Assume that $\left\lceil \frac{\Delta t + \Psi_t + RB_i^u - RB_i^l}{T_{\mathcal{G}_{\tau_i}}} \right\rceil$ is underestimated.

Since RB_i^l and RB_i^u are conservative, the assumption only holds when Ψ_t is underestimated. It means that an appearance of a higher priority task τ_i earlier than $RB_t^u - \Psi_t$ makes the worst interference to the target task.

1. **If target task is under preemptive scheduling policy,** $\Psi_t = 0$. If higher priority task $\tau_i \in \mathcal{G}_{\tau_t}$ is released at $RB_t^u - \delta$ where $\delta > 0$, at most δ amount of higher priority task execution is in interval $[RB_t^u - \delta, RB_t^u]$, which cannot be included in the inter-graph interference to τ_t in $[RB_t^u, RB_t^u + \Delta t]$. Hence the inter-interference decreases, which is contradictory.

2. **Else if all predecessors of target task are on the same PE,** $\Psi_t = \max_{\tau_p \in pred(\tau_t)} C_p^u$. In this case, one of predecessors $\tau_s \in pred(\tau_t)$ is executed right before the release of τ_t. Assume some task $\tau_i \in \mathcal{Z}_{\tau_t}$ appears earlier than $RB_t^u - \max_{\tau_p \in pred(\tau_t)} C_p^u$. Then at least one τ_i execution will start before τ_s instead of being delayed by τ_s since τ_i has a higher priority than all tasks in $\mathcal{G}_{\tau_t}$. Since RB_t^u is conservative, the τ_i execution start before τ_s cannot delay $FB_s^u = RB_t^u$, so it cannot be included in the inter-interference to τ_t in $[RB_t^u, RB_t^u + \Delta t]$. Hence the inter-interference decreases, which is contradictory to the assumption.

3. **Otherwise**. Suppose there exists a worst-case scheduling scenario that a lower priority task τ_s starts earlier than $RB_t^u - \Psi_t$ and blocks higher priority tasks in $\mathcal{Z}_{\tau_t}$. It means that τ_s is not considered in Ψ_t computation ($\tau_s \notin \mathcal{F}_{\tau_t}$), or larger period shift value should be considered for $\tau_s \in \mathcal{F}_{\tau_t}$. If $\tau_s \in \mathcal{G}_t$ and $SB_s^l \geq RB_t^u$, it cannot start and block tasks in $\mathcal{Z}_{\tau_t}$ before RB_t^u, which is contradictory to the assumption. In case $\tau_s \notin \mathcal{G}_t$ or $\tau_s \in \mathcal{G}_t \wedge SB_s^l < RB_t^u$, the larger period shift means lower blocking delay since the sum of the blocking delay and the period shift is equal to the execution time of τ_s. By Lemma 3.3.1, it decreases the response time, which is contradictory.

From 1, 2 and 3, the conservativeness is proven. □

Lemma 3.6.7. $\forall\tau_t, \forall\tau_s \in \mathcal{Z}_{\tau_t}, \left\lceil \frac{\max(0, SB_t^u - RB_t^u + 1 - \phi_{t,s}^r)}{T_{\mathcal{G}_{\tau_s}}} \right\rceil = \left\lceil \frac{\max(0, SB_t^u - RB_t^u - \phi_{t,s}^r)}{T_{\mathcal{G}_{\tau_s}}} \right\rceil$

Proof. We prove it by contradiction. Assume there exists a task $\tau_s \in \mathcal{Z}_{\tau_t}$ that does not satisfy the above lemma. It is only possible when $SB_t^u - RB_t^u - \phi_{t,s}^r$ is a multiple of $T_{\mathcal{G}_{\tau_s}}$. Then $\left\lceil \frac{\max(0, SB_t^u - RB_t^u + 1 - \phi_{t,s}^r)}{T_{\mathcal{G}_{\tau_s}}} \right\rceil - \left\lceil \frac{\max(0, SB_t^u - RB_t^u - \phi_{t,s}^r)}{T_{\mathcal{G}_{\tau_s}}} \right\rceil = 1$.

Consider equation (3.17) that computes SB_t^u. Since it is a fixed-point iteration function with an initial value RB_t^u and is a non-decreasing function, we can simply represent the equation as $x_{i+1} = f(x_i)$ where x_i is the intermediate result of SB_t^u, $x_0 = RB_t^u$, and $x_{i+1} \geq x_i$. Then $SB_t^u = f(SB_t^u) = f(SB_t^u - 1)$ should hold since SB_t^u is a fixed-point result of the equation. By the assumption, $\overline{Delay_t^h}$ of $f(SB_t^u - 1)$ is smaller than $\overline{Delay_t^h}$ of $f(SB_t^u)$ by at least C_s^u. Since other terms in equation (3.17) are also non-decreasing, $f(SB_t^u) - f(SB_t^u - 1) > C_s^u$, which is contradictory to the assumption. □

Now we restate the theorem 3.3.2 and prove it.

Theorem 3.3.2. *The inter-graph interference is conservatively considered in schedule time bound, which means that $\overline{Delay_t^h}$ and $\overline{Preempt_t^W}$ that use phases $\phi_{t,i}^r$, $\phi_{t,i}^s$, and $\phi_{t,i}^f$ are conservative.*

Proof. Consider the longest path from source task to any non-source task in the graph $\mathcal{G}$ and let L be the task set that includes tasks in the longest path. $\tau_{L(i)}$ indicates the i-th task in the longest path in the scheduling order. The worst-case response time is conservative only if the inter-graph interference is conservatively estimated for every longest path to non-source task.

If we partition the task set L into subsets of chained tasks, the overall interference onto L is not greater than the sum of interference on each subset. Hence we prove that the proposed technique computes the upper bound of inter-interference on a subset $\{\tau_{L(k)} | i \leq k \leq j\}$ that are mapped to the same processing element and have no predecessor task mapped onto other processing elements except the first task $\tau_{L(i)}$ of the subset.

We prove it by induction that the interference is conservatively computed for (1) a subset of one task, (2) a subset of two tasks, and (3) a subset of arbitrary number of tasks.
(1) A subset of one task $\{\tau_{L(i)}\}$: The only task $\tau_{L(i)}$ is the source task or has a predecessor that is mapped to other processing element. Then, according to equation (3.15), the relative distance of task τ_s from $RB^u_{L(i)}$ are $-(\Psi_{L(i)} + RB^u_s - RB^l_s)$. For non-preemptive scheduling policy, the amount of inter-graph interference on $\tau_{L(i)}$ is computed by equation (3.16) as

$$\begin{aligned}\overline{Delay^h_{L(i)}} &= \sum_{\tau_s \in Z_{\tau_{L(i)}}} \left\lceil \frac{\max(0, SB^u_{L(i)} - RB^u_{L(i)} + 1 - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s \\ &= \sum_{\tau_s \in Z_{\tau_{L(i)}}} \left\lceil \frac{\max(0, SB^u_{L(i)} - RB^u_{L(i)} + \Psi_{L(i)} + RB^u_s - RB^l_s)}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s\end{aligned}$$

"+1" can be removed according to lemma 3.6.7. By lemma 3.6.6, $\overline{Delay^h_{L(i)}}$ is the upper bound of the inter-graph interference on $\tau_{L(i)}$ during time interval $[RB^u_{L(i)}, SB^u_{L(i)}]$. For preemptive scheduling policy, the amount of interference is $\overline{Delay^h_{L(i)}} + \overline{Preempt^W_{L(i)}}$. We first derive $\phi^s_{L(i),s}$ of equation (3.18).

$$\begin{aligned}\phi^s_{L(i),s} &= ((\phi^r_{L(i),s} + RB^u_{L(i)}) - SB^u_{L(i)}) \bmod T_{\mathcal{G}_{\tau_s}} \\ &= \left\lceil \frac{\max(0, SB^u_{L(i)} - (RB^u_{L(i)} + \phi^r_{L(i),s}))}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot T_{\mathcal{G}_{\tau_s}} + (\phi^r_{L(i),s} + RB^u_{L(i)}) - SB^u_{L(i)}\end{aligned}$$

Then, we replace $\phi^s_{L(i),s}$ in $\overline{Preempt^W_{L(i)}}$ of equation (3.19) with the above equation.

$$\begin{aligned}&\sum_{\tau_s \in Z_{\tau_{L(i)}}} \left\lceil \frac{\max(0, FB^u_{L(i)} - SB^u_{L(i)} - \phi^s_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s \\ &= \sum_{\tau_s \in Z_{\tau_{L(i)}}} \max\left(0, \left\lceil \frac{FB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s}}{T_{\mathcal{G}_{\tau_s}}} - \left\lceil \frac{\max(0, SB^u_{L(i)} - (RB^u_{L(i)} + \phi^r_{L(i),s}))}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \right\rceil\right) \cdot C^u_s\end{aligned}$$

Since $\lceil\rceil$ is an integer term and $FB^u_{L(i)} \geq SB^u_{L(i)}$, the above equation becomes

$$\sum_{\tau_s \in Z_{\tau_{L(i)}}} \left(\left\lceil \frac{\max(0, FB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil - \left\lceil \frac{\max(0, SB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \right) \cdot C^u_s$$
$$= \sum_{\tau_s \in Z_{\tau_{L(i)}}} \left\lceil \frac{\max(0, FB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s - \overline{Delay^h_{L(i)}}$$

By lemma 3.6.6, $\overline{Delay^h_{L(i)}} + \overline{Preempt^W_{L(i)}}$ is the upper bound of the inter-graph interference on τ_t during time interval $[RB^u_{L(i)}, FB^u_{L(i)}]$.

(2) A subset of two tasks $\{\tau_{L(i)}, \tau_{L(i+1)}\}$: According to the subset definition, $\tau_{L(i+1)}$ is released right after $\tau_{L(i)}$ finishes on the same processing element so that there is no idle time between two tasks $\tau_{L(i)}$ and $\tau_{L(i+1)}$. It means that two tasks can be clustered as a single task even though the proposed technique computes the interference of each task separately.

For non-preemptive scheduling policy, the sum of inter-graph interferences on two tasks is $\overline{Delay^h_{L(i)}} + \overline{Delay^h_{L(i+1)}}$. First we derive the terms of $\phi^r_{L(i+1),s}$ into terms of $\phi^r_{L(i),s}$. Since $FB^u_{L(i)} = RB^u_{L(i+1)}$ and the relative distance $\phi^r_{L(i+1),s}$ is selected as the minimum among all predecessors of $\tau_{L(i+1)}$, we derive an inequality of $\phi^r_{L(i+1),s}$ from equation (3.18) and equation (3.21) as follows.

$$\begin{aligned}
\phi^r_{L(i+1),s} &\leq (\phi^f_{L(i),s} + FB^u_{L(i)}) - RB^u_{L(i+1)} \\
&= ((\phi^s_{L(i),s} + SB^u_{L(i)}) - FB^u_{L(i)} + FB^u_{L(i)}) - RB^u_{L(i+1)} = \phi^s_{L(i),s} + SB^u_{L(i)} - RB^u_{L(i+1)} \\
&= \left\lceil \frac{\max(0, SB^u_{L(i)} - (RB^u_{L(i)} + \phi^r_{L(i),s}))}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot T_{\mathcal{G}_{\tau_s}} + (\phi^r_{L(i),s} + RB^u_{L(i)}) - RB^u_{L(i+1)}
\end{aligned}$$

Then, we replace $\phi^r_{L(i+1),s}$ in $\overline{Delay^h_{L(i+1)}}$ of equation (3.16) with the above equation. Note

that $\mathcal{Z}_{\tau_{L(i+1)}} = \mathcal{Z}_{\tau_{L(i)}}$ since priorities of task graphs are not inter-mixed.

$$
\begin{aligned}
&\sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i+1)}}} \left\lceil \frac{\max(0, SB^u_{L(i+1)} - RB^u_{L(i+1)} - \phi^r_{L(i+1),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s \\
&\geq \sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i)}}} \max\left(0, \left\lceil \frac{SB^u_{L(i+1)} - RB^u_{L(i)} - \phi^r_{L(i),s}}{T_{\mathcal{G}_{\tau_s}}} - \left\lceil \frac{\max(0, SB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \right\rceil\right) \cdot C^u_s \\
&= \sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i)}}} \left(\left\lceil \frac{\max(0, SB^u_{L(i+1)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil - \left\lceil \frac{\max(0, SB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \right) \cdot C^u_s \\
&= \sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i)}}} \left\lceil \frac{\max(0, SB^u_{L(i+1)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s - \overline{Delay^h_{L(i)}}
\end{aligned}
$$

By lemma 3.6.6, the sum of interferences on two tasks $\overline{Delay^h_{L(i)}} + \overline{Delay^h_{L(i+1)}}$ is the upper bound of the inter-graph interference during time interval $[RB^u_{L(i)}, SB^u_{L(i+1)}]$.

For preemptive scheduling policy, the sum of inter-graph interferences on two tasks becomes

$$
\begin{aligned}
&\sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i)}}} \left\lceil \frac{\max(0, FB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s \\
&\qquad + \sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i+1)}}} \left\lceil \frac{\max(0, FB^u_{L(i+1)} - RB^u_{L(i+1)} - \phi^r_{L(i+1),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s
\end{aligned}
$$

Similarly, we derive the terms of $\phi^r_{L(i+1),s}$ into terms of $\phi^r_{L(i),s}$.

$$
\begin{aligned}
\phi^r_{L(i+1),s} &\leq (\phi^f_{L(i),s} + FB^u_{L(i)}) - RB^u_{L(i+1)} \\
&= \phi^f_{L(i),s} = \left((\phi^s_{L(i),s} + SB^u_{L(i)}) - FB^u_{L(i)}\right) \bmod T_s \\
&= \left\lceil \frac{\max(0, FB^u_{L(i)} - (SB^u_{L(i)} + \phi^s_{L(i),s}))}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot T_{\mathcal{G}_{\tau_s}} + (\phi^s_{L(i),s} + SB^u_{L(i)}) - FB^u_{L(i)}
\end{aligned}
$$

On the other hand, $\phi^s_{L(i),s}$ is derived as

$$\phi^s_{L(i),s} = \left\lceil \frac{\max(0, SB^u_{L(i)} - (RB^u_{L(i)} + \phi^r_{L(i),s}))}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot T_{\mathcal{G}_{\tau_s}} + (\phi^r_{L(i),s} + RB^u_{L(i)}) - SB^u_{L(i)}$$

Therefore we get the following inequality for $\phi^r_{L(i+1),s}$ from above two equations.

$$\begin{aligned}
\phi^r_{L(i+1),s} &\leq \left\lceil \frac{\max(0, SB^u_{L(i)} - (RB^u_{L(i)} + \phi^r_{L(i),s}))}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot T_{\mathcal{G}_{\tau_s}} + (\phi^r_{L(i),s} + RB^u_{L(i)}) - FB^u_{L(i)} \\
&\quad + \max\left(0, \left\lceil \frac{FB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s}}{T_{\mathcal{G}_{\tau_s}}} - \left\lceil \frac{\max(0, SB^u_{L(i)} - (RB^u_{L(i)} + \phi^r_{L(i),s}))}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \right\rceil \right) \cdot T_{\mathcal{G}_{\tau_s}} \\
&= \left\lceil \frac{\max(0, FB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot T_{\mathcal{G}_{\tau_s}} + (\phi^r_{L(i),s} + RB^u_{L(i)}) - FB^u_{L(i)}
\end{aligned}$$

Using above inequality, we derive the inter-graph interference on task $\tau_{L(i+1)}$.

$$\begin{aligned}
&\sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i+1)}}} \left\lceil \frac{\max(0, FB^u_{L(i+1)} - RB^u_{L(i+1)} - \phi^r_{L(i+1),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s \\
&\geq \sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i)}}} \max\left(0, \left\lceil \frac{FB^u_{L(i+1)} - RB^u_{L(i)} - \phi^r_{L(i),s}}{T_{\mathcal{G}_{\tau_s}}} - \left\lceil \frac{\max(0, FB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \right\rceil \right) \cdot C^u_s \\
&= \sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i)}}} \left(\left\lceil \frac{\max(0, FB^u_{L(i+1)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil - \left\lceil \frac{\max(0, FB^u_{L(i)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \right) \cdot C^u_s
\end{aligned}$$

since $RB^u_{L(i+1)} = FB^u_{L(i)}$. Finally, the sum of the inter-graph interferences on two tasks is larger than or equal to $\sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i)}}} \left\lceil \frac{\max(0, FB^u_{L(i+1)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_{\mathcal{G}_{\tau_s}}} \right\rceil \cdot C^u_s$, which is the upper bound of the inter-graph interference during time interval $[RB^u_{L(i)}, FB^u_{L(i+1)}]$ by lemma 3.6.6.

(3) A subset of arbitrary number of tasks $\{\tau_{L(k)} | i \leq k \leq j\}$: The total sum of interference can be induced to $\sum_{\tau_s \in \mathcal{Z}_{\tau_{L(i)}}} \left\lceil \frac{\max(0, FB^u_{L(j)} - RB^u_{L(i)} - \phi^r_{L(i),s})}{T_s} \right\rceil \cdot C^u_s$ by repeating the above derivation until we reach at the final task of the subset, $\tau_{L(j)}$. Hence the interference on the subset is conservatively computed.

From the inductive steps (1), (2), and (3), the computed inter-graph interference onto L is proven to be conservative since the interference is computed as the sum of

interference of each task in L. □

3.6.3 Proof of Theorem 3.4.1

We restate the theorem and prove it.

Theorem 3.4.1. *The original graph and the reconstructed graph are equivalent in terms of task execution order.*

Proof. Task execution order is determined by dependency edges. In Algorithm 1, the edges between different ordered sets are preserved. Only the edges inside of an ordered set are changed. We prove by contradiction that addition or removal of an edge does not change the task execution order of the original graph.

1. **An edge is added**: Suppose that addition of edges change the task execution order by adding a new path from τ_c to τ_d while τ_d may be released earlier or during τ_c execution in the original graph. It means that there is a path from τ_d to τ_c or there is no direct dependency between τ_c and τ_d in the original graph. The former case is impossible since the edges are added according to the topological order. For the latter case, note that the execution order of τ_c and τ_d of the original graph is fixed for a sub-graph O that runs on one processing element. Since edges are added according to the scheduling order (line 11), it is not possible to add edges in the reverse execution order between two tasks.

2. **An edges is removed**: Suppose that removal of an edge changes the execution order by deleting a path from τ_c to τ_d of the original graph. Since we remove an edge only when there is another path between two tasks (line 14), there should exist at least one path between τ_c and τ_d that determines the execution order between two tasks. The assumption thus fails.

From (1) and (2), task execution order of the original graph is proven to be preserved in the reconstructed graph. □

3.6.4 Proof of Theorem 3.4.2

We prove that the optimization technique of duplicate preemption elimination preserves the conservativeness of the HPA technique. It is a rather long proof. We make several definitions, lemmas, and theorems in this subsection.

Definition 3.6.5. *$pe(\tau_t)[x,y]$ is defined as the sum of execution time of tasks of which priority is higher than τ_t from time x to time y. Then $0 \leq pe(\tau_t)[x,y] \leq y-x$ and $pe(\tau_t)[x,z] = pe(\tau_t)[x,y] + pe(\tau_t)[y,z]$ are hold.*

Lemma 3.6.8. *If the maximum release time bound of τ_t is reduced by Δ, or $r(\tau_t) \leq RB_t^u - \Delta$, then it always holds that the maximum finish time bound of τ_t is reduced by $\Delta - pe(\tau_t)[RB_t^u - \Delta, RB_t^u]$, or $f(\tau_t) \leq FB_t^u - \Delta + pe(\tau_t)[RB_t^u - \Delta, RB_t^u]$.*

Proof. We prove it by contradiction. Suppose that $f(\tau_t)$ exists such that $f(\tau_t) > FB_t^u - \Delta + pe(\tau_t)[RB_t^u - \Delta, RB_t^u]$. It is obvious that $FB_t^u \geq RB_t^u + C_t^u + pe(\tau_t)[RB_t^u, FB_t^u]$ and $f(\tau_t) \leq r(\tau_t) + C_t^u + pe(\tau_t)[r(\tau_t), f(\tau_t)]$ since $pe(\tau_t)[x,y]$ is the actual preempting time while C_t^u and FB_t^u are defined as upper bounds. From two inequalities for $f(\tau_t)$, we have

$$\begin{aligned} f(\tau_t) &> FB_t^u - \Delta + pe(\tau_t)[RB_t^u - \Delta, RB_t^u] \\ &\geq RB_t^u + C_t^u + pe(\tau_t)[RB_t^u, FB_t^u] - \Delta + pe(\tau_t)[RB_t^u - \Delta, RB_t^u] \\ &= RB_t^u + C_t^u - \Delta + pe(\tau_t)[RB_t^u - \Delta, FB_t^u] \end{aligned}$$

and

$$\begin{aligned} f(\tau_t) &\leq r(\tau_t) + C_t^u + pe(\tau_t)[r(\tau_t), f(\tau_t)] \\ &= r(\tau_t) + C_t^u + pe(\tau_t)[r(\tau_t), RB_t^u - \Delta] + pe(\tau_t)[RB_t^u - \Delta, f(\tau_t)] \end{aligned}$$

Thus,

$$
\begin{aligned}
& pe(\tau_t)[r(\tau_t), RB_t^u - \Delta] \\
& \geq f(\tau_t) - r(\tau_t) - C_t^u - pe(\tau_t)[RB_t^u - \Delta, f(\tau_t)] \\
& > RB_t^u - \Delta - r(\tau_t) + pe(\tau_t)[RB_t^u - \Delta, FB_t^u] - pe(\tau_t)[RB_t^u - \Delta, f(\tau_t)] \\
& \geq RB_t^u - \Delta - r(\tau_t)
\end{aligned}
$$

Last inequality comes from $pe(\tau_t)[RB_t^u - \Delta, FB_t^u] \geq pe(\tau_t)[RB_t^u - \Delta, f(\tau_t)]$.

$pe(\tau_t)[r(\tau_t), RB_t^u - \Delta] > RB_t^u - \Delta - r(\tau_t)$ is contradiction since $pe(\tau_t)[r(\tau_t), RB_t^u - \Delta] \leq RB_t^u - \Delta - r(\tau_t)$ according to Definition 3.6.5. Hence $f(\tau_t) \leq FB_t^u - \Delta + pe(\tau_t)[RB_t^u - \Delta, RB_t^u]$. □

Lemma 3.6.8 explains how much the maximum finish time can be reduced if the maximum release time decreases. Since the release time depends on the finish times of predecessor tasks, Lemma 3.6.8 presents the effect of the finish time of predecessor tasks onto the finish time of the target task. This lemma will be generalized below in Lemma 3.6.9. Hereafter, $pe(\tau_t, \Delta)$ denotes $pe(\tau_t)[RB_t^u - \Delta, RB_t^u]$ for brevity.

Now we will examine the effect of the early finish time of an ancestor task to the finish time of the target task. If we apply lemma 3.6.8 repeatedly, we can compute how much the finish time reduction of an ancestor task affects the finish time of the target task.

Lemma 3.6.9. *If* $\forall_{\tau_a \in pred(\tau_t)}(f(\tau_a) \leq FB_a^u - \Delta_{\tau_a} + pe(\tau_a, \Delta_{\tau_a}))$, *then* $f(\tau_t) \leq FB_t^u - \Delta_{\tau_t} + pe(\tau_t, \Delta_{\tau_t})$ *where*

$$\Delta_{\tau_m} = \max_{\tau_i \in pred(\tau_m)}(FB_i^u) - \max_{\tau_i \in pred(\tau_m)}(FB_i^u - \Delta_{\tau_i} + pe(\tau_i, \Delta_{\tau_i}))$$

Note that $\Delta_{\tau_m} = 0$ *if there is no predecessor.*

Proof. We prove it by induction. First, $f(\tau_a) \leq FB_a^u - \Delta_{\tau_a} + pe(\tau_a, \Delta_{\tau_a}) = FB_a^u$ holds for

every source task τ_a since $\Delta_{\tau_a} = 0$. Second, for non-source task τ_m,

$$r(\tau_m) = \max_{\tau_i \in pred(\tau_m)} f(\tau_i) \leq \max_{\tau_i \in pred(\tau_m)} (FB_i^u - \Delta_{\tau_i} + pe(\tau_i, \Delta_{\tau_i}))$$

assuming that $f(\tau_i) \leq FB_i^u - \Delta_{\tau_i} + pe(\tau_i, \Delta_{\tau_i})$ for all predecessor tasks of τ_m in the induction process. From definition of Δ_{τ_m}, we have

$$\max_{\tau_i \in pred(\tau_m)} (FB_i^u - \Delta_{\tau_i} + pe(\tau_i, \Delta_{\tau_i})) = \max_{\tau_i \in pred(\tau_m)} (FB_i^u) - \Delta_{\tau_m}$$

Since $RB_m^u = \max_{\tau_i \in pred(\tau_m)} FB_i^u$, $r(\tau_m) \leq \max_{\tau_i \in pred(\tau_m)} (FB_i^u) - \Delta_{\tau_m} = RB_m^u - \Delta_{\tau_m}$, which induces $f(\tau_m) \leq FB_m^u - \Delta_{\tau_m} + pe(\tau_m, \Delta_{\tau_m})$ according to Lemma 3.6.8. □

For the brevity of further formulation, we define a new notation for the reduced finish time bound, $\overleftarrow{FB}_t^u$, after reducing the finish time of an ancestor task τ_a by Δ as following:

Definition 3.6.6. *The reduced finish time bound $\overleftarrow{FB}^u_{t,\Delta_{\tau_a} \leftarrow \Delta}$ after reducing the finish time of a task τ_a by Δ is*

$$\overleftarrow{FB}^u_{t,\Delta_{\tau_a} \leftarrow \Delta} = \begin{cases} FB_t^u - \Delta, & \text{if } \tau_t = \tau_a \\ FB_t^u - \Delta_{\tau_t} + pe(\tau_t, \Delta_{\tau_t}), & \text{else if } \tau_t \in descendant(\tau_a) \\ FB_t^u, & \text{otherwise} \end{cases}$$

where $\Delta_{\tau_m} = \max_{\tau_i \in pred(\tau_m)} FB_i^u - \max_{\tau_i \in pred(\tau_m)} \overleftarrow{FB}^u_{i,\Delta_{\tau_a} \leftarrow \Delta}$.

Lemma 3.6.10. *After reducing the finish time of a task τ_a by Δ, $\Delta_{\tau_t} \leq \Delta$ holds for all $\tau_t \in descendent(\tau_a)$.*

Proof. We prove it by induction. First, consider a set of descendant tasks $\{\tau_t \mid \tau_t \in descendent(\tau_a), \forall_{\tau_i \in pred(\tau_t)} (\tau_i \notin descendent(\tau_a))\}$. Then $\Delta_{\tau_t} \leq \Delta$ holds since $\overleftarrow{FB}^u_{i,\Delta_{\tau_a} \leftarrow \Delta}$ of every predecessor task τ_i is either FB_i^u or $FB_i^u - \Delta$.

Second, assuming $\forall_{\tau_i \in pred(\tau_t), \tau_i \in descendent(\tau_a)} \Delta_{\tau_i} \leq \Delta$, we prove that $\Delta_{\tau_t} \leq \Delta$ holds for remaining descendant tasks $\{\tau_t \mid \tau_t \in descendent(\tau_a), \exists_{\tau_i \in pred(\tau_t)}(\tau_i \in descendent(\tau_a))\}$. For a predecessor task τ_i who is also descendant of τ_a,

$$\overleftarrow{FB}^u_{i,\Delta_{\tau_a} \leftarrow \Delta} = FB^u_i - \Delta_{\tau_i} + pe(\tau_i, \Delta_{\tau_i}) \geq FB^u_i - \Delta$$

since $\Delta_{\tau_i} \leq \Delta$ and $pe(\tau_i, \Delta_{\tau_i}) \geq 0$. Finally, $\Delta_{\tau_t} = \max_{\tau_i \in pred(\tau_t)} FB^u_i - \max_{\tau_i \in pred(\tau_t)} \overleftarrow{FB}^u_{i,\Delta_{\tau_a} \leftarrow \Delta} \leq \Delta$ since $\overleftarrow{FB}^u_{i,\Delta_{\tau_a} \leftarrow \Delta} \geq FB^u_i - \Delta$ holds for every predecessor task τ_i. □

Lemma 3.6.10 implies that the contribution of the maximum finish time reduction of an ancestor task diminishes as it propagates to the child tasks.

Now we restate the key theorem in the proposed duplicate preemption elimination technique.

Theorem 3.4.2. *If a common preemptor τ_p can preempt an ancestor task τ_a and the target task τ_t, then $f(\tau_t)$ is no smaller when it preempts τ_t rather than τ_a.*

Proof. Since τ_p can preempt τ_a and τ_t, it preempt τ_a completely but τ_t partially in case $FB^u_p - RB^u_t < C^u_p$. Otherwise, it preempt both tasks completely. Let δ be the preemption time of τ_p to τ_t. Then $\delta \leq C^u_p$.

$\overleftarrow{FB}^u_{t,\Delta_{\tau_t} \leftarrow \delta}$ and $\overleftarrow{FB}^u_{t,\Delta_{\tau_a} \leftarrow C^u_p}$ are the reduced finish time bound when τ_p does not preempt τ_t but preempts τ_a and the reduced finish time bound when τ_p preempts τ_t, respectively. We prove that $\overleftarrow{FB}^u_{t,\Delta_{\tau_a} \leftarrow C^u_p} \geq \overleftarrow{FB}^u_{t,\Delta_{\tau_t} \leftarrow \delta} = FB^u_t - \delta$.

1. If $\Delta_{\tau_t} \leq C^u_p - \delta$ then the time interval $[RB^u_t - \Delta_{\tau_t}, RB^u_t]$ is always filled with execution of τ_p or other higher priority tasks since δ amount of τ_p execution appears after RB^u_t. It means that $pe[\tau_t, \Delta_{\tau_t}] = \Delta_{\tau_t}$. Then,

$$\overleftarrow{FB}^u_{t,\Delta_{\tau_a} \leftarrow C^u_p} = FB^u_t - \Delta_{\tau_t} + pe[\tau_t, \Delta_{\tau_t}] = FB^u_t \geq FB^u_t - \delta = \overleftarrow{FB}^u_{t,\Delta_{\tau_t} \leftarrow \delta}.$$

2. If $\Delta_{\tau_t} > C_p^u - \delta$ then the time interval $[RB_t^u - \Delta_{\tau_t}, RB_t^u]$ consists of at least $C_p^u - \delta$ amount of execution of τ_p or other higher priority tasks. It means that $pe[\tau_t, \Delta_{\tau_t}] \geq C_p^u - \delta$. Since $\Delta_{\tau_t} \leq C_p^u$ according to Lemma 3.6.10,

$$\overleftarrow{FB}^u_{t,\Delta_{\tau_a} \leftarrow C_p^u} = FB_t^u - \Delta_{\tau_t} + pe[\tau_t, \Delta_{\tau_t}] \geq FB_t^u - C_p^u + C_p^u - \delta = FB^u - \delta = \overleftarrow{FB}^u_{t,\Delta_{\tau_t} \leftarrow \delta}.$$

From 1 and 2, it is confirmed that preempting τ_t provides no smaller maximum finish time of τ_t than preempting τ_a, or $\overleftarrow{FB}^u_{t,\Delta_{\tau_a} \leftarrow C_p^u} \geq \overleftarrow{FB}^u_{t,\Delta_{\tau_t} \leftarrow \delta}$. □

3.6.5 Proof of Theorem 3.5.1

We restate the theorem and prove it.

Theorem 3.5.1. *The computed worse-case response times are conservative for the arbitrary deadline model.*

Proof. We prove it by contradiction. Assume that the estimated worst-case response time is smaller than the exact worst-case response time, which means that there exists a task τ_t FB_t^u is under-estimated. We prove that both intra-interference and inter-interference on the task instance τ_t is conservatively computed in the proposed technique for the arbitrary deadline model.

1. Suppose that intra-interference is under-estimated. Since Theorem 3.3.1 holds under the condition that *all interfering task instances are considered*, the assumption is only possible when task instances are not sufficiently expanded. It means that there exists an uncreated interfering task instance $\tau_{s(i)}$ which will increase

FB^u_t if it is created and considered during analysis. Let the index of lastly created graph instance be $last$. Since the graph expansion is terminated when all time bounds are converged, the schedule time bounds before and after adding $last$ graph instance are equal. It means that $FB^u_{t(last-1)} + T_{\mathcal{G}_{\tau_t}} = FB^u_{t(last)}$ and $\tau_{s(last)}$ could not affect $FB^u_{t(last-1)}$. Then $FB^u_{t(last-1)} \leq SB^l_{s(last)}$ holds. Since $last < i$ and the analysis is converged, $SB^l_{s(last)} + T_{\mathcal{G}_{\tau_t}} \leq SB^l_{s(i)}$. On the other hand, the assumption implies that $\tau_{s(i)}$ can affect FB^u_t, or $SB^l_{s(i)} < FB^u_{t(last)}$. In summary, $FB^u_{t(last-1)} + T_{\mathcal{G}_{\tau_t}} \leq SB^l_{s(last)} + T_{\mathcal{G}_{\tau_t}} \leq SB^l_{s(i)} < FB^u_{t(last)}$. which is contradictory to the fact that $FB^u_{t(last-1)} + T_{\mathcal{G}_{\tau_t}} = FB^u_{t(last)}$.

2. Suppose that inter-interference is under-estimated. By Lemma 3.3.2, the under-estimation only happens when the period shift is under-estimated. The period shift Ψ_t may be under-estimated only when at least one candidate task of $\mathcal{F}_{\tau_t}$ is missed due to the lack of graph expansion. Let the index of the last graph instance be $last$ in the initial graph expansion. Then $last = \lceil \frac{D_\mathcal{G}}{T_\mathcal{G}} \rceil + 1$. The last τ_t instance, $\tau_{t(last)}$ can check all task instances released after $RB^u_{t(last)} - \lceil \frac{D_\mathcal{G}}{T_\mathcal{G}} \rceil \cdot T_\mathcal{G} \leq RB^u_{t(last)} - D_\mathcal{G}$. If there is a missed candidate task, it should be released before $RB^u_t - D_\mathcal{G}$ and it can block the task that finishes at $RB^u_{t(last)}$. It is impossible since the task cannot violate the deadline. Hence $\Psi_{t(last)}$ is conservative. Since we use the maximum period shift among all $\Psi_{t(i)}$s in the proposed technique and $\Psi_{t(last)}$ is conservatively computed. So, the inter-interference cannot be under-estimated.

From 1 and 2, we prove that the proposed technique for the arbitrary deadline model computes a conservative WCRT. □

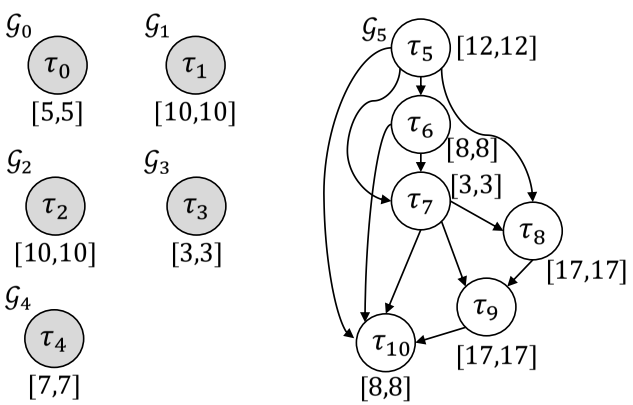
3.7 Experiments

In experiment, we compare the proposed technique with four existing tools: MAST suite [46], pyCPA [83], pyCPA_TC [49]. and STBA [6]. pyCPA is a freely available

compositional performance analysis tool similar to SymTA/S. pyCPA_TC is the implementation of [49] based on pyCPA. For MAST suite, we choose the best among the experimental results from three options "offset_based_optimized", "offset_based_slanted", and "offset_based_approximate_with_precedence_relations". The STBA method is implemented following the formulas in [6]. In addition, we compare the proposed technique with naive RTA approach that analyzes the WCRT of each task independently with the RTA and finds the longest path as the WCRT of the application.

3.7.1 Comparison with Transformation Approaches

With this experiment, we will show that the transformation approach [45] requires more processing elements to make all applications schedulable than the proposed technique due to the loose WCRT estimation. Figure 3.15 (a) shows a simple example that consists of a target graph application $\mathcal{G}_5$ and five applications each of which consists of one task. The worst-case response times of all applications are analyzed by the proposed technique and summarized in Figure 3.15 (b), which confirms that all applications are schedulable. The approach introduced in [45] transforms the graph into independent real-time tasks with starting offsets and deadlines: each task is assigned a deadline in proportion to the execution time and its offset is assigned to be later than the deadlines of all predecessors. Figure 3.15 (c) summarizes the offsets and deadlines of all transformed tasks in $\mathcal{G}'_5$. Since all transformed tasks should be schedulable independently, we have to consider the worst-case interference from all higher priority tasks, but none of the tasks is schedulable. Hence the transformation approach requires at least two processing elements for all applications to be schedulable. In addition, this approach in general leads to a longer response time since the starting offset is fixed. The latency of the graph $\mathcal{G}'_5$ is always larger than 94, while $\mathcal{G}_5$ can terminate at 65 with our approach. This example shows that a transformation approach may require more resources than the proposed technique.



Mapping : $\forall_{0 \le i \le 10} M_i = \text{PE0}$

Priority : $PR_i > PR_j \quad if\ i < j$

Period & deadline : $\forall_{0 \le i \le 5} T_{\mathcal{G}_i} = D_{\mathcal{G}_i} = 100$

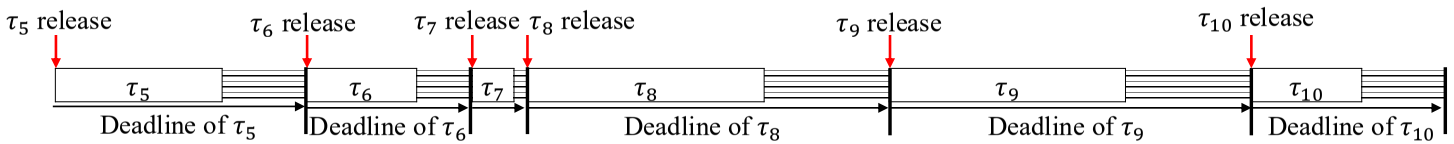
(a) Example graphs

Schedule Time Bounds: (RB^l, RB^u), (SB^l, SB^u), (FB^l, FB^u)		
τ_0	τ_1	τ_2
(0,0), (0,0), (5,5)	(0,0), (0,5), (10,15)	(0,0), (0,15), (10,25)
τ_3	τ_4	
(0,0), (0,25), (3,28)	(0,0), (0,28), (7,35)	
τ_5	τ_6	τ_7
(0,0), (0,35), (12,47)	(12,47), (12,47), (20,55)	(20,55), (20,55), (23,58)
τ_8	τ_9	τ_{10}
(23,58), (23,58), (40,75)	(40,75), (40,75), (57,92)	(57,92), (57,92), (65,100)

(b) Analysis result from the proposed technique

Graph Transformation: Offset, Deadline		
τ_5	τ_6	τ_7
0, 18	18, 30	30, 34
τ_8	τ_9	τ_{10}
34, 60	60, 86	86, 100

(c) Graph transformation result of $\mathcal{G}_5{}'$ from the technique in [3]



(d) Releases and deadlines of transformed graph $\mathcal{G}_5{}'$ in (c)

Figure 3.15: An example system for comparison with the transformation approach

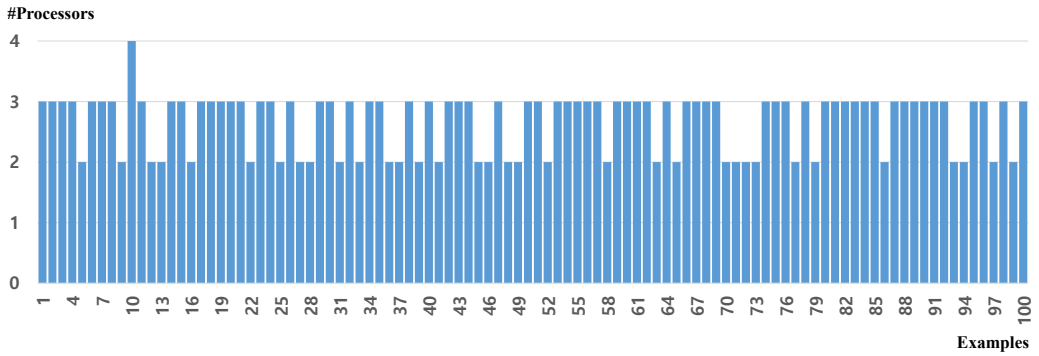


Figure 3.16: Required number of processing elements for a system in each synthetic example to be scheduled by the transformation approach

For more elaborate comparison with [45], we generate synthetic examples with randomly selected values determined by uniform distribution according to the following rules. A target graph $\mathcal{G}$ is randomly generated with 20 tasks. We generate 20 applications each of which includes single higher priority task. The C_i^l and C_i^u of each task are randomly selected in the range of [500, 1000] and [C_i^l,$C_i^l \times 1.5$], respectively. When we make a synthetic example, we have to make the example schedulable. The periods and deadlines are initially assigned as WCET scheduling latency for each applications. Until the system becomes schedulable, which can be checked by schedulability analysis, the period and the deadline of each application are multiplied by 2 with 50% probability. With a generated example, we first estimate the worst-case response time of the target graph, assuming that all tasks are running on one processing element. Then we give the estimated WCRT value as the latency constraint of the technique [45]: target graph is transformed into real-time tasks that the end-to-end deadline is the estimated WCRT. With transformed real-time tasks, we analyze how many processing elements are required to meet all tasks' deadlines. With 1000 synthetic examples, as shown in Figure 3.16, the experiment confirms that the transformation approach requires on average 2.67, maximum 4 processing elements to achieve the same latency performance as the proposed technique.

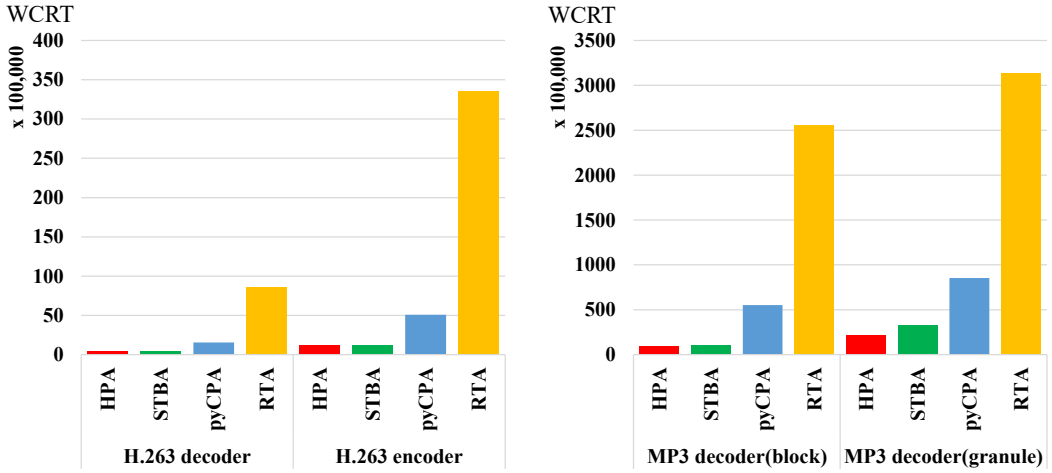


Figure 3.17: Estimated WCRTs for four benchmarks: H.263 decoder, H.263 encoder, MP3 decoder (block level), and MP3 decoder (granule level)

3.7.2 Experiment with Real-life Benchmarks

Four real-life applications are used for comparison: H.263 decoder, H.263 encoder, MP3 decoder (block level), and MP3 decoder (granule level). They are obtained from SDF3 [84] with profiled information and transformed to the task graph model in Section 3.1. Task mapping is performed to minimize the latency on the system with eight preemptive cores. The application priority is assigned in the increasing order of latency. The periods and deadlines are set arbitrarily to make the system schedulable.

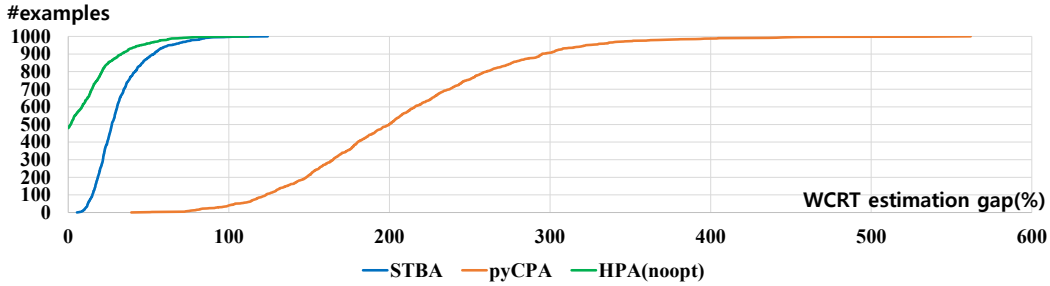
Figure 3.17 shows the WCRTs of each benchmark estimated by the proposed technique and the other techniques. HPA gives tighter bounds than the other approaches in all benchmarks except for the case of H.263 decoder. Since other applications cannot interfere H.263 decoder which has the highest graph-level priority, both HPA and STBA produce the same WCRT estimation by considering intra-graph interference only with the same schedule time bound computation. The WCRT from STBA becomes loose for the other applications that are interfered by different applications since it conservatively transforms a dynamic offset into a large static offset. The naive RTA approach shows the worst estimation since it does not consider the dependent relation of tasks.

3.7.3 Experiment with Synthetic Examples

For more experiments, synthetic examples are randomly generated according to the following rules. An example has 3-5 processing elements and 3-5 task graphs in which the total number of tasks is between 30 and 50. C_i^l and C_i^u of each task are randomly selected in the range of [500, 1000] and [C_i^l,$C_i^l \times 1.5$], respectively. The deadline is assigned equally to the period. Three different configurations are prepared with different graph topology and scheduling policy. "DAG_MIX" examples have DAG applications and allows a mixture of preemptive and non-preemptive scheduling policy. An application in "LIN_MIX" examples is restricted to be a linear graph. In a linear graph, each task has at most one parent and one child. In "LIN_FPP" configuration, examples have only linear graphs and preemptive scheduling policy.

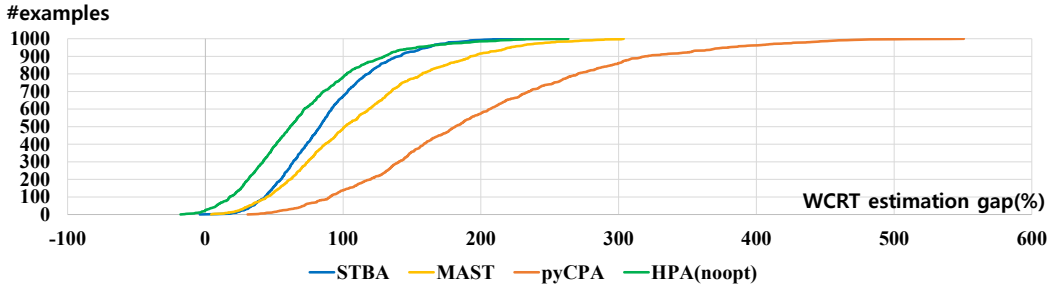
In the first set of experiments, we generate 1000 examples of "DAG_MIX" and compare the result with STBA and pyCPA. MAST and pyCPA_TC are excluded since both techniques are not applicable for non-linear graphs. For each example, we analyze the worst-case response times of all task graphs and compare the maximum among them. The WCRT estimation gap is computed as the ratio of the difference between the estimated WCRTs over the estimated WCRT from the optimized HPA. Figure 3.18 (a) shows the experimental result for "DAG_MIX" examples. The graph in the figure means the number of examples that shows smaller WCRT estimation gap than the corresponding x-axis value. In the figure, we also show the result of HPA without optimization technique (HPA(no opt.)). STBA and pyCPA show on average 31.10% and 205.87% looser bounds than the proposed technique. It is observed that our optimization technique increases the estimation accuracy on average by 10.97%.

For comparison with MAST, we make another 1000 examples of "LIN_MIX". pyCPA_TC is excluded since it does not support non-preemptive scheduling. Figure 3.18 (b) shows the comparison results. Note that the average estimation gap of STBA increases to 87.51% since there is no intra-graph interference from which STBA can get bene-



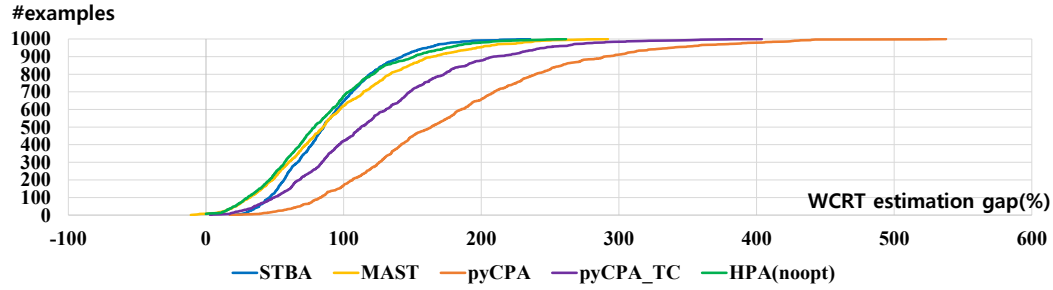
	Win	Tie	Lose	Max(%)	Min(%)	Avg(%)
vs STBA	1000	0	0	124.22	5.39	31.10
vs pyCPA	1000	0	0	561.87	39.23	205.87
vs HPA(no opt.)	521	479	0	112.04	0.00	10.97

(a) Comparison result for 1000 examples of “DAG_MIX”



	Win	Tie	Lose	Max(%)	Min(%)	Avg(%)
vs STBA	999	0	1	230.79	-3.98	87.52
vs MAST	1000	0	0	303.69	4.22	111.45
vs pyCPA	1000	0	0	550.60	30.71	197.63
vs HPA(no opt.)	974	5	21	263.46	-18.03	69.16

(b) Comparison result for 1000 examples of “LIN_MIX”



	Win	Tie	Lose	Max(%)	Min(%)	Avg(%)
vs STBA	1000	0	0	235.43	19.93	90.15
vs MAST	992	1	7	292.02	-10.97	93.21
vs pyCPA	1000	0	0	537.66	17.32	177.41
vs pyCPA_TC	1000	0	0	403.84	3.06	123.63
vs HPA(no opt.)	993	7	0	261.57	0.00	85.60

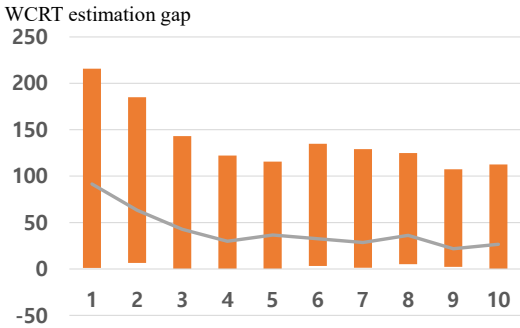
(c) Comparison result for 1000 examples of “LIN_FPP”

Figure 3.18: WCRT estimation gap between the proposed technique and the other approaches for three example sets “DAG_MIX”, “LIN_MIX”, and “LIN_FPP”

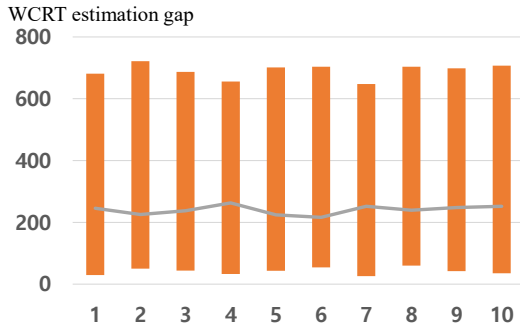
fits most. Note that optimization results in looser bounds in 21 out of 1000 examples compared to HPA(no opt.). While the optimization heuristic helps to remove duplicate preemptions, it may produce a longer WCRT by moving preemptions from ancestors to the target task even in case multiple preemptions are not redundant. Hence, the final implementation of HPA selects the better result between HPA and HPA(no opt.). Again HPA produces tighter bounds than the other approaches in almost all examples even without optimization. And the benefit of optimization increases significantly to 69.16% on average, compared with the previous experiment.

In the third experiment with synthetic examples, we use 1000 examples of "LIN_FPP" and all four reference techniques are used for comparison. The comparison results are shown in Figure 3.18 (c). MAST and pyCPA show slightly better estimations than the previous experiment with "LIN_MIX", which implies that they suffer from the over-estimation of the non-preemptive blocking delay. pyCPA_TC shows better WCRT estimation than pyCPA because they optimize pyCPA by decomposing a linear graph into a set of task-chains each of which executes continuously in one PE. However, since pyCPA_TC is yet based on the compositional approach, the over-estimation still remains.

Finally, we perform the experiment that confirms the effect of graph topology onto the estimation performance. In this experiment, each example has 3 graphs each of which contains 20 tasks. We generates 10 experiment sets that have different degrees of graph topology from 1 to 10. The topology degree value indicates the maximum number of child tasks. For instance, the graph is always linear transaction if degree value is 1. If degree value is 5, the graph is generated with the restriction that a task should have less than 5 child tasks. Each experiment set has 1000 examples and we measure the maximum, minimum, and average WCRT estimation gap of the reference technique compared to the proposed technique. Figure 3.19 shows the experimental result. The x axis and y axis indicate the graph topology degree value and the WCRT estimation gap, respectively. The upper bound and lower bound of the bar graph show the maximum and minimum



(a) WCRT estimation gap of STBA to HPA while varying the degree of graph topology



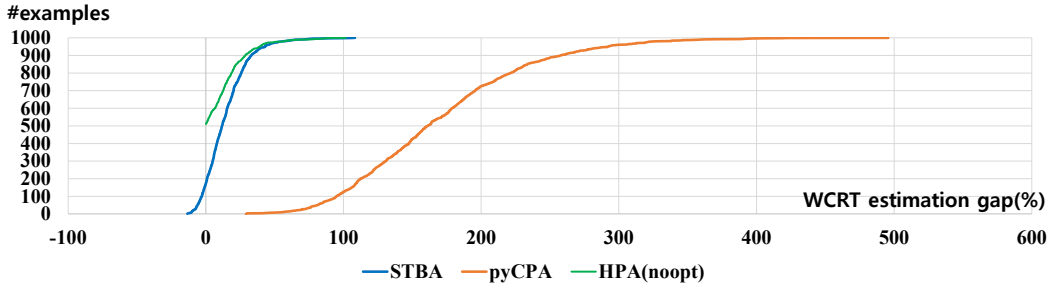
(b) WCRT estimation gap of pyCPA to HPA while varying the degree of graph topology

Figure 3.19: WCRT estimation gap of STBA and pyCPA compared to the proposed technique while varying the degree of graph topology

values, respectively, while the linear graph shows the average values. As we can see in Figure 3.19 (a), the estimation gap of STBA reduces as the degree of graph topology increases, since the inter-graph interference computation of the proposed technique tends to be overestimated when a task has multiple parent tasks. However, despite of the overestimation comes from the complicated graph topology, the proposed technique shows on average 26.51% tighter bound than STBA when the degree value is 10. The estimation gap of pyCPA is not affected by the degree of graph topology, as shown in Figure 3.19 (b).

3.7.4 Experiment for Arbitrary Deadline Model

To examine the performance of the proposed technique for the arbitrary deadline model, we generate 1000 synthetic examples of "DAG_MIX", but each graph in examples has a deadline greater than the period. Specifically, we restrict examples to have only graphs whose estimated WCRTs are larger than period, where WCRT is estimated by the proposed technique. The comparison results for the examples are summarized in Figure 3.20. The average estimation gap of STBA and pyCPA decreases to 14.26% and 170.30% respectively, compared to the experiment with the restricted deadline model. Since we always increase the schedule time bounds of a task instance as the instance



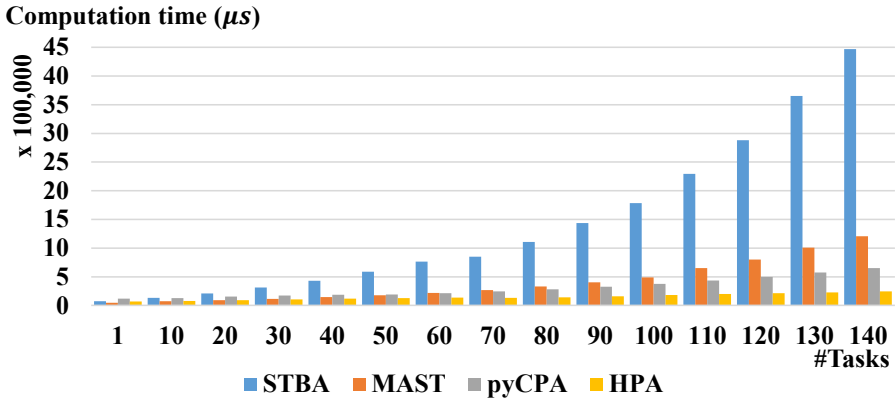
	Win	Tie	Lose	Max(%)	Min(%)	Avg(%)
vs STBA	826	0	174	108.32	-13.46	14.26
vs pyCPA	1000	0	0	495.73	29.08	170.30
vs HPA(no opt.)	489	511	0	101.54	0.00	9.50

Figure 3.20: WCRT estimation gap between the proposed technique and the other approaches for 1000 synthetic examples when deadline is not restricted

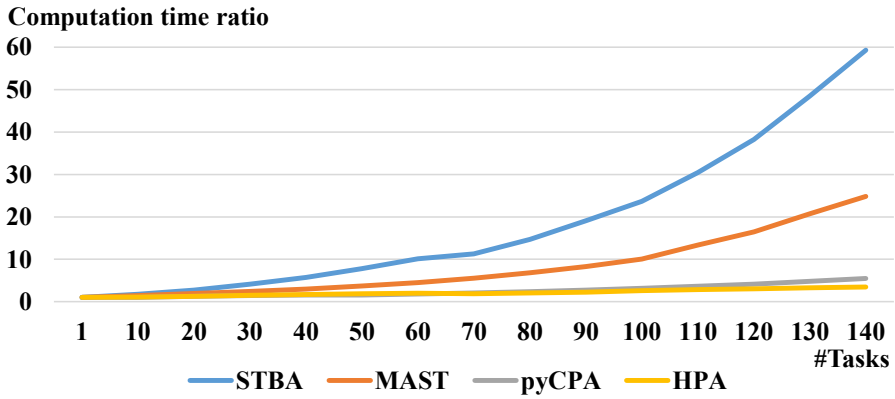
index increases, the proposed technique may over-estimate WCRT when many graph instances are involved. The proposed technique shows loose bound than STBA for 174 examples, but the estimation gap is at most 13.46%. Despite the conservativeness of the proposed technique, it shows tighter WCRT estimation than the reference techniques on average.

3.7.5 Comparison of the Computation Time

Finally, we conduct an experiment that measures the computation time of each technique while varying the number of tasks. The analysis is performed on a system with i7 3.40 GHz quad-core CPU and 8GB RAM. For each number of tasks, we generate 1000 synthetic examples. Since MAST supports the chain-structured graph only, we used different graphs with the same number of tasks for the MAST technique. Figure 3.21 (a) shows the average computation times of four techniques varying the number of tasks. It reveals that the proposed technique consumes smaller computation time than the other techniques in all cases. Even though we could not find any quantitative formula for the convergence speed of the proposed technique, the experimental result confirms that the iterative process of the proposed technique reaches to a fixed point fast.



(a) Computation time of four techniques while varying the number of tasks



(b) Computation time ratio of four techniques while varying the number of tasks.

Figure 3.21: An experimental result for comparison with the reference techniques in terms of computation time while varying the number of tasks

We plot the normalized computation time ratio of each technique in Figure 3.21 (b) to observe the scalability of the analysis techniques. All time values of each technique is normalized by the value of that technique for the case the number of task is 1. In Figure 3.21 (b), STBA shows the poorest scalability. The proposed technique shows the similar measure of scalability as pyCPA that is known as the representative scalable technique.

Chapter 4

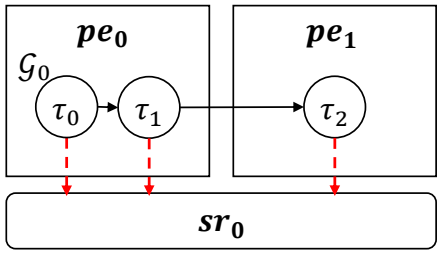
Shared Resource Contention Analysis

4.1 Problem Definition

We formally describe the system model and the task model assumed in this chapter.

A system consists of a set of processing elements (PEs) $\mathcal{PE}$ and a set of shared resources (SRs) $\mathcal{SR}$. In Figure 4.1 (a), the example system consists of two PEs pe_0 and pe_1, and one SR sr_0. We assume that all PEs use partitioned scheduling so that task mapping is not varying at run time. The scheduling policy of a PE is a fixed-priority preemptive scheduling ($\mathcal{P}$) or a fixed-priority non-preemptive scheduling ($\mathcal{NP}$). When a task attempts to access a shared resource, it busy-waits on its mapped PE without being preempted until the resource request is processed. SRs are globally accessed by the tasks executed in all PEs. When there are resource conflicts, resource requests should be arbitrated. Controller Area Network (CAN) bus communication protocol is a representative example. When shared resource uses non-preemptive arbitration policy such as CAN bus, we assume that the priority of a resource request inherits the priority of a task that issues the request.

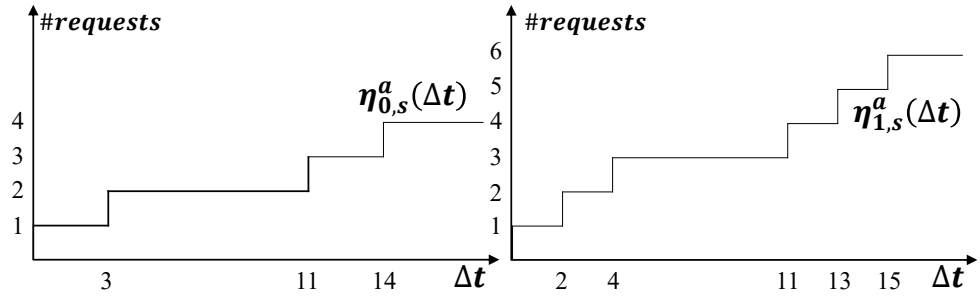
An input application, $\mathcal{G}$, is represented as an acyclic task graph. If a task has more than one input edge, it is released after all predecessor tasks are completed. An application $\mathcal{G}$ can be initiated periodically or sporadically, characterized by $T_{\mathcal{G}}$ where $T_{\mathcal{G}}$ represents the period of $\mathcal{G}$ or the minimum initiation interval for sporadic activation. For sporadic activation, $T_{\mathcal{G}}$ denotes the minimum initiation interval. Task graph $\mathcal{G}$ is given



(a) System Model

$\mathcal{G}_0$	$D_{\mathcal{G}_0} = 20$		$T_{\mathcal{G}_0} = 11$	
	M_i	PR_i	$[C_i^l, C_i^u]$	$\tau_{i,s}^w$
τ_0	pe_0	1	[6,8]	1
τ_1	pe_0	0	[6,10]	1
τ_2	pe_1	2	[10,10]	1

(b) Task Model



(c) Two functions $\eta_{0,s}^a(\Delta t)$ and $\eta_{1,s}^a(\Delta t)$ that represent the maximum numbers of resource accesses from two tasks τ_0 and τ_1

Figure 4.1: Examples that present the system model and the task model assumed in this work

relative deadline $D_{\mathcal{G}}$ to meet once activated. We support an arbitrary deadline model; the deadline may be greater than the period of an application.

A task is a basic mapping unit onto a processing element. We assume that task mapping is given and fixed. The processing element that the task τ_i is mapped to is denoted by M_i. For each task τ_i, the varying execution time is represented as a tuple $[C_i^l, C_i^u]$ indicating the lower and the upper bound on the mapped PE, which are given bounds analyzed with the ideal assumption that there is no shared resource contention. We assume that all tasks in the whole system have distinct priorities to make the scheduling order deterministic. The priority of the task τ_i is denoted by PR_i. The task graph that task τ_i belongs to is denoted by $\mathcal{G}_{\tau_i}$. Figure 4.1 (b) shows the information defined above for an example task set in Figure 4.1 (a).

The access pattern of each task to a shared resource can be represented by an event stream model [85] which has been used to model task activations in compositional analy-

sis approach [51]. Event stream model does not specify when each event occurs. Instead, it describes the upper and lower event arrival functions $\eta^+(\Delta t)$ and $\eta^-(\Delta t)$ which specify the maximum and the minimum number of events that occur in the event stream during any time interval of length Δt, respectively. Inverse functions $\delta^-(n)$ and $\delta^+(n)$ represent the minimum and maximum time windows in which n events can be observed in the stream. Both functions can be straightforwardly converted to each other. The event stream model is useful to summarize the complicated resource access patterns from multiple tasks on a processor with bounds. In this work, we only use upper bound functions to express the resource access patterns. Note that we assume resource access pattern of a task can be given conservatively from the task execution analysis, regardless of the run-time schedule. As long as the assumption holds, the proposed contention analysis technique can be applied to any shared resource.

We define the upper bound function for the number of resource accesses from τ_i to a shared resource sr_s, $\eta^a_{i,s}(\Delta t)$, with Definition 4.1.1. From now on, the superscript of a function represents a specific approach where it can be a, e, f, E, and F.

Definition 4.1.1. *$\eta^a_{i,s}(\Delta t)$ is the maximum number of resource accesses that may be issued by task τ_i to a shared resource sr_s within a time window of size Δt.*

Figure 4.1 (c) plots $\eta^a_{i,s}(\Delta t)$ for two tasks τ_0 and τ_1. For instance, $\eta^a_{0,s}(11) = 3$ means that there may be at most three resource accesses from τ_0 in any time interval of length 11. In addition to $\eta^a_{i,s}(\Delta t)$, we define $w_{i,j}$, the maximum access duration among all accesses of task τ_i to shared resource sr_j. We summarize the terms and notations in Table 4.1.

With the given information of task mapping and task execution profiles for task execution times and shared resource accesses, we aim to compute a tight upper bound of resource access delay each task experiences during the execution. To this end, we derive an upper bound function for each (pe_i, sr_j) pair that finds the maximum resource demand generated from pe_i to sr_j during the response time.

Table 4.1: Terms and notations

Notation	Description
$\mathcal{PE}$	a set of processing elements
pe_i	a processing element with its index i
$\mathcal{SR}$	a set of shared resources
sr_i	a shared resource with its index i
$\mathcal{P}$	the set of preemptive processing elements
$\mathcal{N}$	the set of non-preemptive processing elements
$\mathcal{G}_i$	a task graph with its index i
$T_\mathcal{G}$	an initiation interval of the task graph $\mathcal{G}$
$J_\mathcal{G}$	the maximum initiation jitter of the task graph $\mathcal{G}$
$D_\mathcal{G}$	a deadline of the task graph $\mathcal{G}$
τ_i	a task with its index i
M_i	the mapped processing element of a task τ_i
C_i^l	the best-case execution time of a task τ_i
C_i^u	the worst-case execution time of a task τ_i
PR_i	the priority of the task τ_i
$\eta_{i,s}^a(\Delta t)$	the maximum number of accesses from τ_i to sr_s within time window Δt
$w_{i,s}$	the maximum resource access duration among all accesses of task τ_i to sr_s

4.2 Review of Reference Technique

In this section, we review the reference technique proposed in [10]. With an example task set in Figure 4.1, we discuss the limitation of the reference technique and present the key idea of the proposed technique. Note that the reference technique [10] only supports independent task model and shared resource arbitrated by a non-preemptive policy. So tasks in Figure 4.1 are regarded as single tasks without dependency edges in this section.

In the system model of interest, a task cannot be preempted by other tasks while it accesses a resource. A task is not interfered by the resource accesses from the same PE except the case that it is released during the lower priority task is accessing a resource. Then the worst-case interference by the resource accesses from the same PE to a task τ_i can be computed easily as the maximum value among resource access duration of lower priority tasks: $max(\forall_{sr_s} \forall_{\{\tau_j | PR_j < PR_i, M_i = M_j\}} w_{j,s})$. Similarly, since the reference technique assumes that shared resources use a non-preemptive arbitration policy, at most one lower

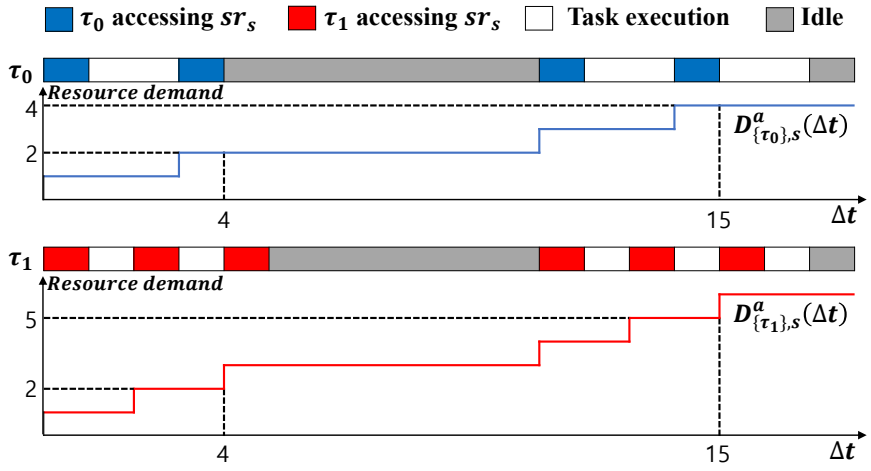


Figure 4.2: Execution profiles and resource demand curves of individual tasks τ_0 and τ_1

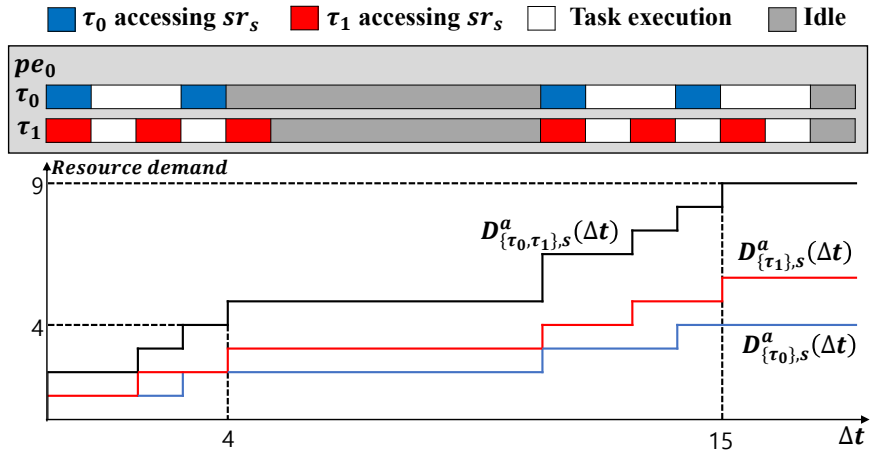


Figure 4.3: Execution profile and resource demand curve of two tasks in pe_0 assuming tasks can be executed in parallel

priority access from another PE may contribute to each resource access delay and the worst-case amount of such delay can be easily computed as the maximum access duration. Then we need to estimate the contribution of higher priority tasks mapped on other PEs onto the worst-case access delay in order to compute the resource access delay that a task τ_i experiences.

Consider a task set $T_{c,p}$ which includes all tasks that are mapped in PE pe_p and

may contribute to the SR access delay of the target task τ_c. When shared resource uses non-preemptive arbitration, all tasks in $T_{c,p}$ has higher priority than target task τ_c. If requests are arbitrated in FIFO fashion, $T_{c,p}$ includes all tasks in PE pe_p. The upper bound of resource demand from $T_{c,p}$ can be conservatively computed by summing up every resource demand of each task in $T_{c,p}$.

$$D^a_{T_{c,p},s}(\Delta t) = \sum_{\tau_i \in T_{c,p}} \eta^a_{i,s}(\Delta t) \cdot w_{i,s} \tag{4.1}$$

However, $D^a_{T_{c,p},s}(\Delta t)$ computes a loose bound of resource demand since it assumes all tasks in $T_{c,p}$ can run in parallel on a single processor. Figure 4.2 shows the task execution profiles that produce the worst-case resource demands for two tasks τ_0 and τ_1 of Figure 4.1. Figure 4.3 shows $D^a_{T_{c,p},s}(\Delta t)$ computation for two tasks τ_0 and τ_1. Note that both tasks access the shared resource from the beginning. If we sum these demand curves naively, we obtain the demand curve in Figure 4.3 with associated task execution profile. $D^a_{\{\tau_0,\tau_1\},s}(\Delta t)$ gives infeasible resource demand for a small range of time window Δt assuming that both τ_0 and τ_1 can be executed simultaneously, which is impossible since one PE can be occupied by only one task at a time.

To compensate this drawback, the reference technique defines another resource request bound function based on PE occupation $\eta^e_{i,s}(t)$, which is defined as follows:

Definition 4.2.1. *$\eta^e_{i,s}(t)$ is the maximum number of resource accesses that may be issued by task τ_i to a shared resource sr_s within any time window that includes t amount of τ_i execution time.*

In this definition, t is the *net* amount of the execution time in a time window, which will be denoted *net* execution time hereafter. The *net* execution time may consist of multiple execution time intervals of the task that are not continuous due to preemption or periodic appearance. $\eta^e_{i,s}(t)$ can be computed by finding the maximum number of resource accesses within a time window of size t assuming τ_i is the only task and its

period is equal to the execution time. Note that the latter assumption implies an unrealistic scenario that there is no idle time in the time window between two consecutive iterations of a task ignoring the task period.

Then a time window Δt is distributed to multiple tasks in $T_{c,p}$ that may interfere the shared resource access of a given task and each allocated time amount is regarded as the net execution time of the associated task. There may be numerous possible schedules of tasks in $T_{c,p}$ that can be executed in a time window Δt. Let t_i be the amount of net execution time of $\tau_i \in T_{c,p}$ in a time window Δt. Although t_i varies by the schedule, the total execution time of tasks in $T_{c,p}$ cannot exceed the size of time window Δt. Hence in order to find the maximum resource demand from a task set $T_{c,p}$, we need to find a set of t_i values that maximizes the total resource demand under the constraint of $\sum_{\tau_i \in T_{c,p}} t_i = \Delta t$. Then a new formula, $D^e_{T_{c,p},s}(\Delta t)$ for the maximum resource demand from a task set $T_{c,p}$ can be computed as follows:

$$D^e_{T_{c,p},s}(\Delta t) = \max \left\{ \sum_{\tau_i \in T_{c,p}} \eta^e_{i,s}(t_i) \cdot w_{i,s} \;\middle|\; \sum_{\tau_i \in T_{c,p}} t_i = \Delta t \right\} \tag{4.2}$$

This formula indicates that we have to find out the worst-case time distribution of the time window to maximize the resource demand for a given time window Δt. Figure 4.4 and Figure 4.5 show $D^e_{T_{c,p},s}(\Delta t)$ computation with the same example in Figure 4.1. Figure 4.4 shows the resource demand bound curve based on PE occupation for each task. Note that two shared resource demands from each task are stitched together since the period of the task is ignored so that the last resource demand of the previous iteration appears just before the first resource demand of the current iteration. The maximum resource demand by distributing the time window is displayed in Figure 4.5; $D^e_{T_{c,p},s}(\Delta t)$ is maximized when $t_0 = 2$ since it is better to give more time to task τ_1 than to task τ_0.

By avoiding concurrent execution of two tasks, $D^e_{T_{c,p},s}(\Delta t)$ gives a tighter bound for a small range of Δt. But it gives a looser bound than $D^a_{T_{c,p},s}(\Delta t)$ as Δt increases since it ignores the period of each task. Note that both functions $D^a_{T_{c,p},s}(\Delta t)$ and $D^e_{T_{c,p},s}(\Delta t)$ are

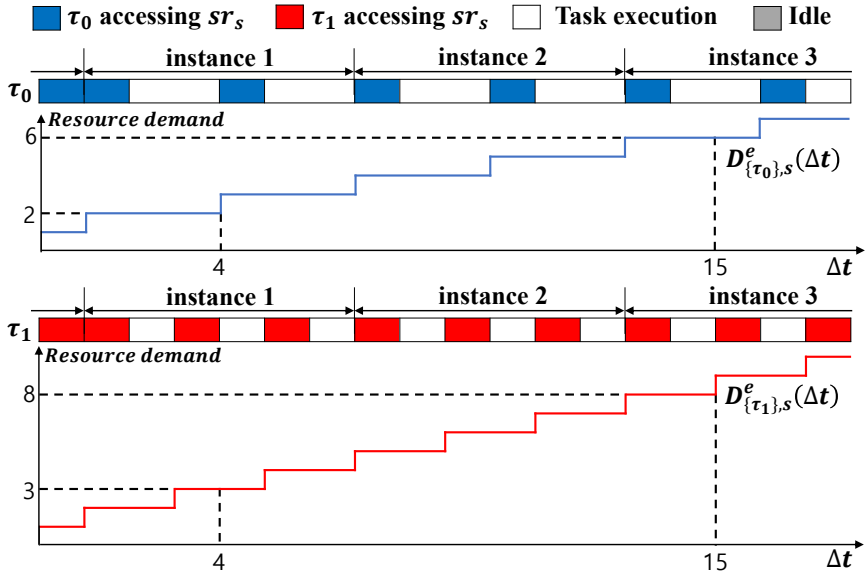


Figure 4.4: Execution profiles and resource demand curves of individual tasks τ_0 and τ_1 showing maximum resource demand per execution time

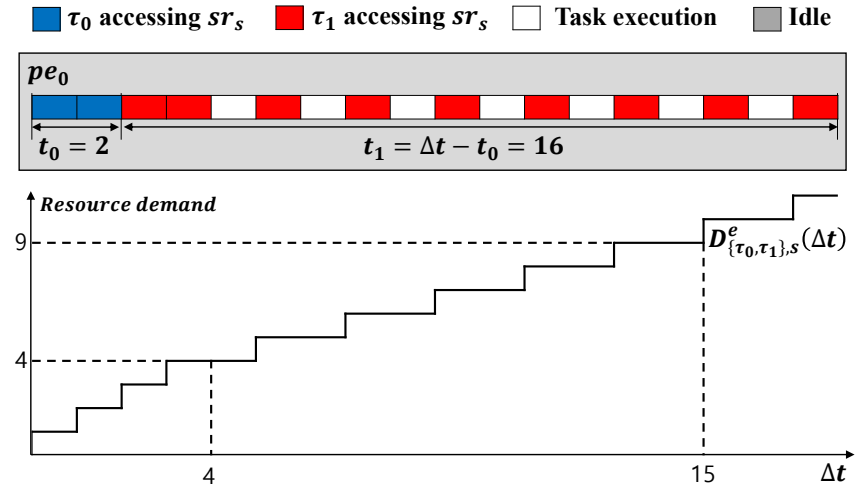


Figure 4.5: Execution profile and resource demand curve of two tasks in pe_0 when time window is distributed

proven to be conservative but give different estimation results. To make a tight estimation, the reference technique determines the final resource demand bound by taking the

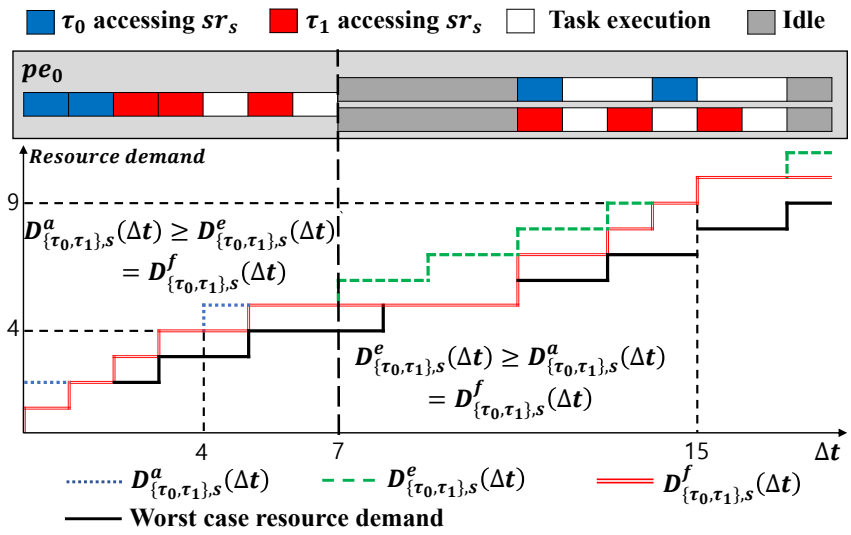


Figure 4.6: $D^f_{T_{c,p},s}(\Delta t)$ computation for an example task set and comparison with the worst-case resource demand

minimum between $D^a_{T_{c,p},s}(\Delta t)$ and $D^e_{T_{c,p},s}(\Delta t)$.

$$D^f_{T_{c,p},s}(\Delta t) = \min\left(D^a_{T_{c,p},s}(\Delta t), D^e_{T_{c,p},s}(\Delta t)\right) \tag{4.3}$$

Figure 4.6 shows $D^f_{T_{c,p},s}(\Delta t)$ computation result for the previous example. $D^e_{T_{c,p},s}(\Delta t)$ function shows a tighter bound than $D^a_{T_{c,p},s}(\Delta t)$ for small values of Δt, while $D^a_{T_{c,p},s}(\Delta t)$ is chosen for large values of Δt as expected. The resource demand bound is still overestimated as illustrated in Figure 4.6, since both functions assume impossible scheduling scenario of tasks. To remedy this drawback, the proposed technique considers the possible task schedules, especially the maximum execution time and the maximum number of task instances in a given time window Δt, which will be explained in subsection 4.3.1.

4.3 Shared Resource Contention Analysis

We explain how to analyze the shared resource demand bound in this section. The overall framework consists of two modules: SR contention modeling and WCRT analysis

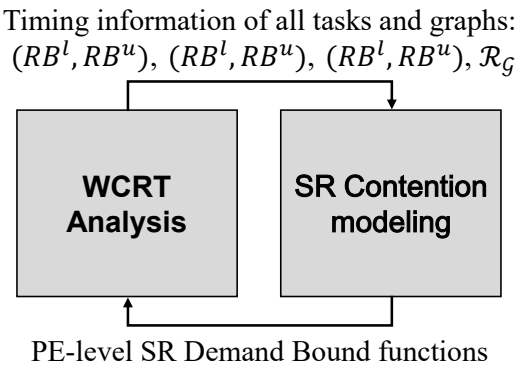


Figure 4.7: The overall iterative framework of the proposed WCRT analysis and SR contention modeling

as displayed in Figure 4.7. Since two modules are dependent on each other, the overall framework forms an iterative loop until converged. The SR contention modeling module computes the upper bound of resource demand from each PE that may block the access of a task to each shared resource. This module requires the timing information such as $(RB^l, RB^u), (SB^l, SB^u), (FB^l, FB^u)$, and $\mathcal{R}_\mathcal{G}$, which are obtained from the WCRT analysis module. The WCRT analysis module, which is based on the proposed HPA technique in Chapter 3, computes the worst-case response time of each task graph considering the shared resource contention using the SR demand bound functions generated from the SR contention modeling module. We briefly explain how to compute the worst-case resource access delay using the SR demand bound function in Section 4.4, for the shared resources with FIFO and fixed priority non-preemptive arbitrations as applicable examples.

In this section, we focus on the analysis of the worst-case shared resource demand from each PE, which corresponds to SR contention modeling module. We first propose an enhanced technique from the reference technique in subsection 4.3.1, with a restriction of having only independent tasks. The proposed technique always shows tigher bound than the reference technique by considering the possible task schedules. In subsection 4.3.2, we propose a novel SR contention analysis technique supporting dependent tasks.

4.3.1 SR Contention Modeling for Independent Tasks

As explained in Section 4.2, the reference technique does not consider the feasibility of task schedules that interfere the shared resource access. To overcome this limitation, we consider the feasibility of task schedule to derive a new resource demand bound function. In this subsection, we propose an enhanced technique from the reference technique that considers independent tasks. For simplicity, we assign task and graph indices to satisfy following constraints: $\mathcal{V}_{\mathcal{G}_i} = \{\tau_i\}$ and $PR_i > PR_j$ if $i < j$.

Considering the Bounds of Net Execution Time: When a task appears periodically, there exist the minimum and the maximum bound of net execution time that a task may take within any time window. The time window distribution found by $D^e_{T_{c,p,s}}(\Delta t)$ may rely on the impossible task schedule since it ignores the minimum and the maximum net execution time bound. Thus we define two functions $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$ that represent the minimum and the maximum execution time amount a task τ_i may take within a time window Δt, respectively. To maximize the number of accesses within a time window, we assume task execution scenarios that a task is invoked with its BCET C_i^l.

Figure 4.8 illustrates two scheduling patterns of task τ_i that correspond to $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$, respectively. In the figure, a dashed rectangle indicates the time window Δt, and the task is assumed to be invoked periodically. The start time of a task may be delayed to $\mathcal{R}_{\mathcal{G}_i} - C_i^l$ in the worst-case by preemption or shared resource contention, which is represented as the grey area in the execution profile.

For a task τ_i to take the minimum *net* execution time in the time window, the worst-case interval between two consecutive job instances should be considered. The worst-case interval is observed when an instance finishes its execution as soon as possible with response time of C_i^l and the start times of all subsequent instances are maximally delayed to $\mathcal{R}_{\mathcal{G}_i} - C_i^l$ as shown in Figure 4.8 (a). Then the minimum *net* execution time is found when the time window starts immediately after the finish time of the first instance as illustrated in Figure 4.8 (a). Now we formulate the amount of *net* ex-

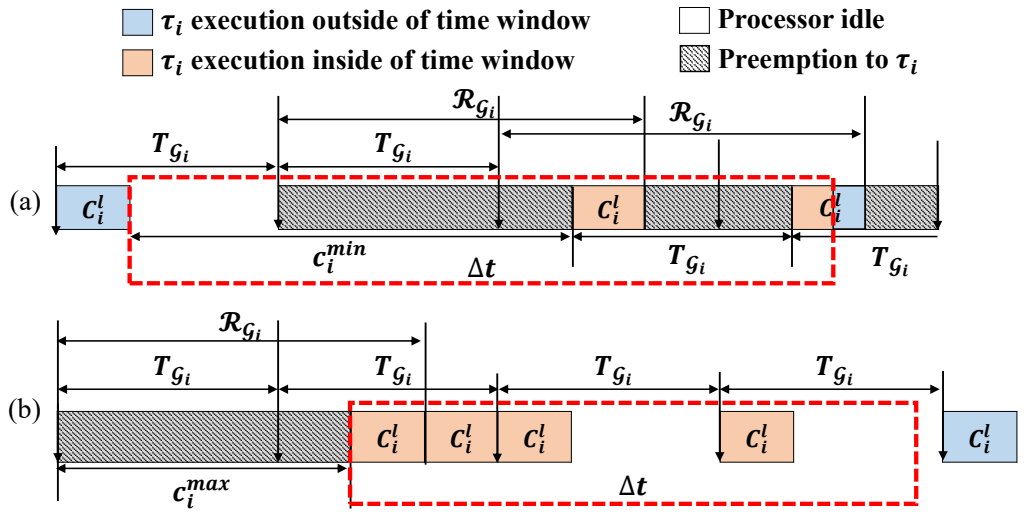


Figure 4.8: The minimum execution time scenario (a) and the maximum execution time scenario (b) in a time window Δt

ecution time in this case. At first, the amount of full job execution is considered. The worst-case interval between two job instances denoted as c_i^{min} is computed as the difference between the finish time of first instance and the start time of second instance: $c_i^{min} = (\mathcal{R}_{\mathcal{G}_i} - C_i^l) - (-T_{\mathcal{G}_i} + C_i^l) = T_{\mathcal{G}_i} + \mathcal{R}_{\mathcal{G}_i} - 2 \cdot C_i^l$. Since the job instance of τ_i appears periodically after c_i^{min} from the start of time window, the number of job instances that fully included can be computed as $\lfloor \max(0, \Delta t - c_i^{min})/T_{\mathcal{G}_i} \rfloor$. One job instance may be partially included at the end of the time window and the partial execution time can be computed as $\min(C_i^l, \max(0, \Delta t - c_i^{min}) \bmod T_{\mathcal{G}_i})$. In summary, we can derive the function $t_i^{min}(\Delta t)$ as follows:

$$t_i^{min}(\Delta t) = C_i^l \cdot \left\lfloor \frac{\max(0, \Delta t - c_i^{min})}{T_{\mathcal{G}_i}} \right\rfloor + \min(C_i^l, \max(0, \Delta t - c_i^{min}) \bmod T_{\mathcal{G}_i}) \qquad (4.4)$$

On the other hand, we should consider the bursty execution of task τ_i in order to compute the maximum execution time in time window Δt. If an instance starts as late as possible to finish at its worst-case response time $\mathcal{R}_{\mathcal{G}_i}$ and subsequent instances start immediately after the previous instance finishes, the bursty scheduling pattern may ap-

pear. The execution time amount is maximized in time window Δt when the time window starts at the onset of the bursty executions of the task, as illustrated in Figure 4.8 (b). The maximum amount of execution time $t_i^{max}(\Delta t)$ is derived as follows:

$$t_i^{max}(\Delta t) = \min\left(\Delta t, C_i^l \cdot \left\lfloor \frac{\Delta t + c_i^{max}}{T_{\mathcal{G}_i}} \right\rfloor + \min(C_i^l, (\Delta t + c_i^{max}) \bmod T_{\mathcal{G}_i})\right) \tag{4.5}$$

where $c_i^{max} = \mathcal{R}_{\mathcal{G}_i} - C_i^l$. The first term and the second term indicate fully included execution and partially included execution, respectively. The conservativeness is summarized as the following lemma.

Lemma 4.3.1. *$t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$ are conservative minimum and maximum bounds of execution time amount that a task τ_i may take within a time window of size Δt, respectively.*

Proof. Assume there exists time window that execution time of a task τ_i is less than $t_i^{min}(\Delta t)$ or larger than $t_i^{max}(\Delta t)$. It means that for both cases there exists a task instance of τ_i whose response time is larger than its worst-case response time $\mathcal{R}_{\mathcal{G}_i}$, which is contradiction to definition of $\mathcal{R}_{\mathcal{G}_i}$. □

With $t_i^{max}(\Delta t)$ and $t_i^{min}(\Delta t)$, we can estimate a tighter upper bound denoted $D^E_{T_{c,p},s}(\Delta t)$ than $D^e_{T_{c,p},s}(\Delta t)$ of the reference technique. When we distribute the time amount Δt to a task set $T_{c,p}$, we should consider the constraint that a task τ_i can be assigned the bounded *net* execution time between $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$. Then the resource demand bound function $D^E_{T_{c,p},s}(\Delta t)$ can be formulated as follows:

$$D^E_{T_{c,p},s}(\Delta t) = \max\left\{ \sum_{\tau_i \in T_{c,p}} \eta^e_{i,s}(\min(t_i, t_i^{max}(\Delta t))) \cdot w_{i,s} \;\middle|\; \begin{array}{c} \sum_{\tau_i \in T_{c,p}} t_i = \Delta t, \\ \forall_{\tau_i \in T_{c,p}} t_i \geq t_i^{min}(\Delta t) \end{array} \right\} \tag{4.6}$$

Considering the Maximum Number of Instances in a Time Window: The upper bound estimation can be further improved by considering the maximum number of in-

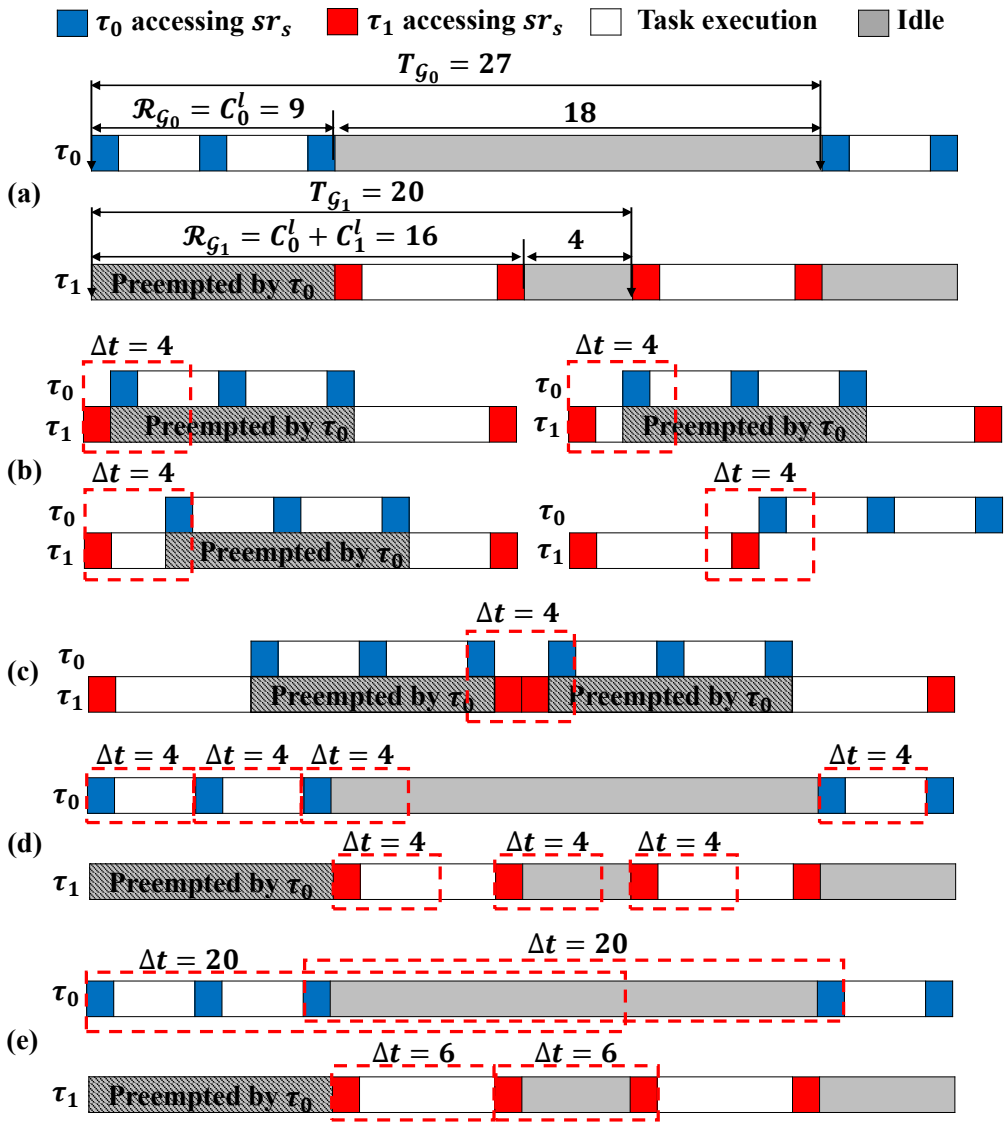


Figure 4.9: Resource demand depends on the number of instances laid in a time window Δt: (a) two example tasks τ_0 and τ_1, (b) four actual worst-case schedules that make maximum resource demand within a time window of size 4, (c) $D^E_{\{\tau_0,\tau_1\},s}(\Delta t)$ relies on the impossible scenario, (d) all time windows of size 4 have one resource request, (e) required minimum time window length to include two resource requests.

stances within a given time window of Δt. Remind that two instances are stitched together by ignoring the period in the computation of $\eta^e_{i,s}(t)$. The resource demand computed by $\eta^e_{i,s}(t)$ is possible only when the time window Δt is large enough to cover the periodic

task instances. Otherwise, it gives an over-estimation result.

Consider two tasks τ_0 and τ_1 in Figure 4.9 (a). For a time window of size 4, the actual worst-case resource demand is 2 as illustrated with four worst-case schedules in Figure 4.9 (b). On the other hand, the total resource demand from two tasks is estimated to 4 in $D^E_{T_{c,p},s}(\Delta t)$; each task has two instances in the time window, executes 2 time units, and accessing sr_0 twice as Figure 4.9 (c). It is obvious that there is no such an execution scenario in reality since two instances from one task cannot lie on the time window of size 4. As illustrated in Figure 4.9 (d), each task can access sr_0 at most once and only one task instance can lie in any time window of size 4. In order for each task to access sr_0 twice with 2 time units of net execution, the time window should be large, up to 20 for task τ_0 and 6 for task τ_1 to cover two task instances as shown in Figure 4.9 (e). It tells that we should consider the maximum number of instances within a given time window of Δt.

Suppose that the maximum number of task instances within a time window of size Δt is known and let n be that number. Then for a tight estimation of maximum resource demand within Δt, only the resource accesses from n task instances need to be considered. For example, resource accesses from only one instance should be considered for the example in Figure 4.9 when $\Delta t = 4$. Here we define a new resource access function $\eta^{e(n)}_{i,s}(t)$ as follows.

Definition 4.3.1. *$\eta^{e(n)}_{i,s}(t)$ is the maximum number of resource accesses that may be issued from n instances of a task τ_i to a shared resource sr_s when the net execution time of τ_i does not exceed t time units.*

$\eta^{e(n)}_{i,s}(t)$ can be computed in a similar way as $\eta^{e}_{i,s}(t)$; find the maximum number of resource accesses within a time window of size t assuming there are only n task instances of τ_i which are released in a bursty fashion. Figure 4.10 shows four functions $\eta^{e(1)}_{0,s}(t)$, $\eta^{e(2)}_{0,s}(t)$, $\eta^{e(3)}_{0,s}(t)$, and $\eta^{e(\infty)}_{0,s}(t)$ for the task τ_0 in Figure 4.9. In case there are two task instances in a time window Δt, the *net* execution time of 2 (1 per one task instance) can

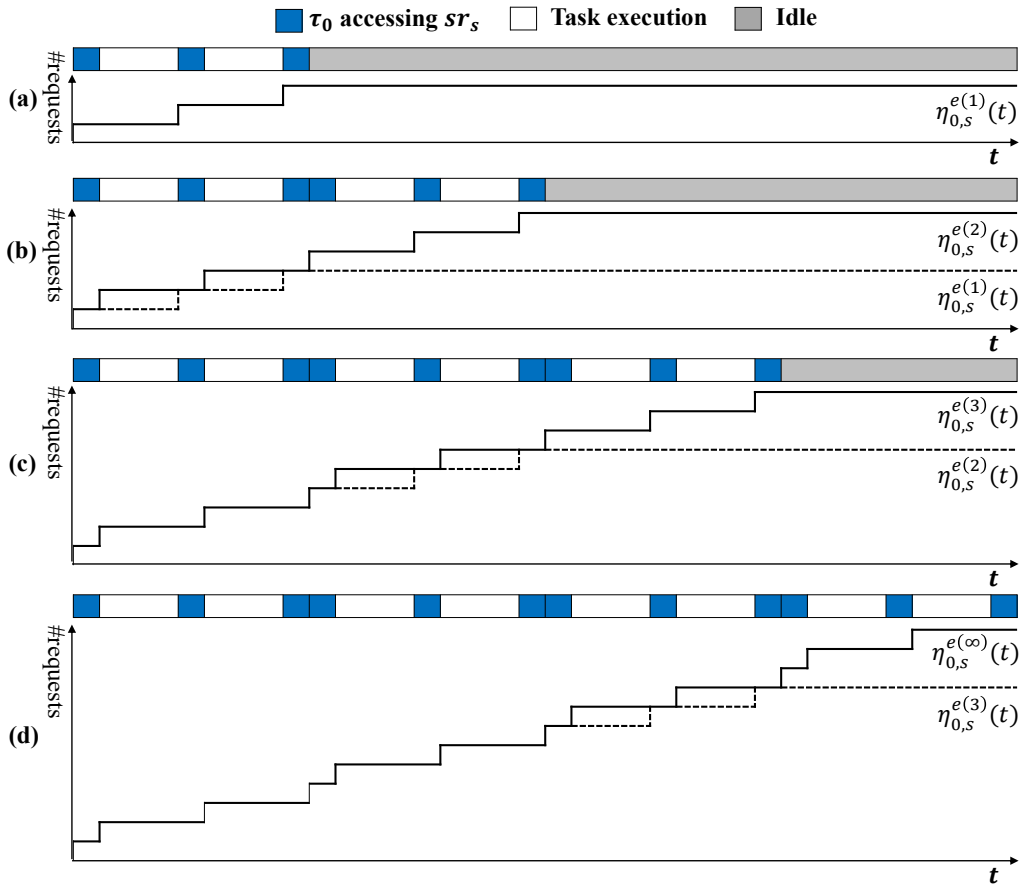


Figure 4.10: $\eta_{i,s}^{e(n)}(t)$ examples: (a) $\eta_{0,s}^{e(1)}(t)$ function graph, (b) $\eta_{0,s}^{e(2)}(t)$ function graph, (c) $\eta_{0,s}^{e(3)}(t)$ function graph, (d) $\eta_{0,s}^{e(\infty)}(t)$ function graph

make 2 resource accesses. Similarly, five resource accesses from *net* execution time of 11 is possible only when there are three or more instances in a time window. It is obvious that when there are infinite number of instances ($n = \infty$), $\eta_{0,s}^{e(\infty)}(t)$ produces the same result as $\eta_{i,s}^{e}(t)$.

Now we compute how many instances may exist in a time window Δt. Figure 4.11 shows the same task schedule scenario of burst execution as Figure 4.8 (b) and the dashed rectangle indicates the time window to achieve the maximum execution time $t_i^{max}(\Delta t)$ in a time window Δt. In order to cover the task instances as many as possible in the time window, with the same bursty task execution schedule, we shift the time window to the

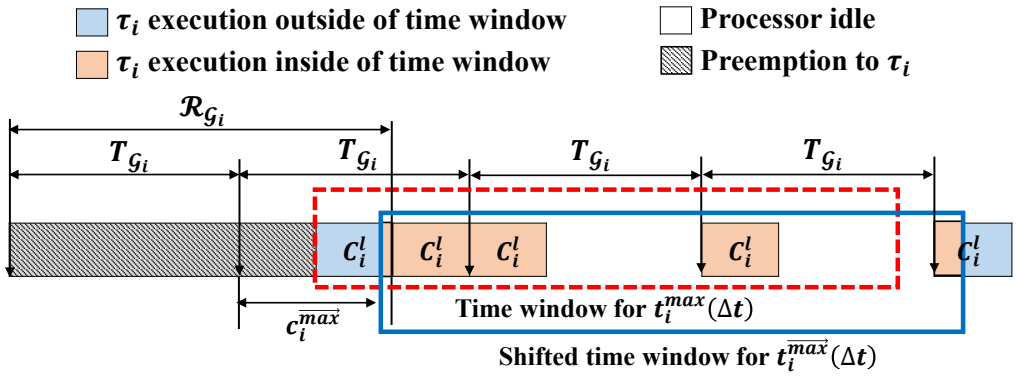


Figure 4.11: The shifted time window by $C_i^l - 1$ from the time window for $t_i^{max}(\Delta t)$

right direction. If the shift amount is greater than or equal to C_i^l, the first task instance becomes outside of the time window, making the number of instances decreases. Hence the shift amount should be less than C_i^l. On the other hand, we need to shift the time window as much as possible to include the instances at the right side of the time window. In summary, to make the maximum number of task instances that may lie in the time window, the time window should be shifted by $C_i^l - 1$. In the figure, the shifted rectangle contains one more instance of the task than the dashed rectangle.

Note that when the number of task instances laid in the time window is maximized, the *net* execution time may be smaller than that for the case when the net execution time in the time window is maximized. We explain this situation with an example in Figure 4.9 (e). As shown in the figure, at most two instances of τ_0 may lie in the time window of size 20. So let the number of instances n be 2 and compute $\eta_{0,s}^{e(2)}(t)$. Then, the *net* execution time t cannot be larger than 2; there is no possible schedule that two τ_0 instances execute larger than 2 time units in any time window of size 20 as illustrated with the second time window of τ_0 in Figure 4.9 (e). On the other hand, if the number of instances in the time window is 1, the *net* execution time can be as large as $t_0^{max}(20) = 9$. Hence we need to compare two cases to find the maximum possible resource demand; (1) the case the number of task instances is maximized and (2) the case the total execution time is

maximized. When $\Delta t = 20$ and $t = 2$, the first case makes larger resource demand than the second case. When $\Delta t = 20$ and $t = 9$, resource demand of the second case is 3 while the first case makes 2 since the *net* execution time is bounded by 2.

Therefore we need to compute the maximum *net* execution time for the first case and the maximum number of task instances for the second case. The number of instances for the second case can be computed as $n_i(\Delta t) = \left\lceil \frac{t_i^{max}(\Delta t)}{C_i^l} \right\rceil$. We denote the maximum *net* execution time for the first case $\overrightarrow{t_i^{max}}(\Delta t)$, where arrow indicates that the time window is shifted by $C_i^l - 1$ to maximize the number of instances. Then $\overrightarrow{t_i^{max}}(\Delta t)$ can be formulated as follows:

$$\overrightarrow{t_i^{max}}(\Delta t) = \min\left(\Delta t, C_i^l \cdot \left\lfloor \frac{\Delta t + \overrightarrow{c_i^{max}}}{T_{\mathcal{G}_i}} \right\rfloor + \min(C_i^l, (\Delta t + \overrightarrow{c_i^{max}}) \bmod T_{\mathcal{G}_i})\right) \tag{4.7}$$

where $\overrightarrow{c_i^{max}} = \mathcal{R}_{\mathcal{G}_i} - T_{\mathcal{G}_i}$. The only difference from equation (4.5) is that c_i^{max} is changed to $\overrightarrow{c_i^{max}}$. We denote the maximum number of τ_i instances laid in the time window Δt as $\overrightarrow{n_i}(\Delta t)$ and $\overrightarrow{n_i}(\Delta t)$ can be computed from $\overrightarrow{t_i^{max}}(\Delta t)$ to be $\left\lceil \frac{\overrightarrow{t_i^{max}}(\Delta t) - 1}{C_i^l} \right\rceil + 1$. The conservativeness of $\overrightarrow{n_i}(\Delta t)$ and $\overrightarrow{t_i^{max}}(\Delta t)$ is summarized as Lemma 4.3.2 and Lemma 4.3.3.

Lemma 4.3.2. *The maximum number of τ_i instances laid in a time window Δt is $\overrightarrow{n_i}(\Delta t)$.*

Proof. The maximum number of τ_i instances can be found in the bursty execution scenario depicted in Figure 4.11 and the time window Δt starts from the first instance of the bursty execution. Let $d_{i,k}$ be the distance between k-th instance and $(k+1)$-th instance of τ_i where 0-th instance is the first instance in the bursty execution of τ_i. Let $\Delta(n)$ be the maximum shifting amount of time window to miss n left instances and $N(\Delta)$ be the number of newly included instances when time window is shifted by Δ. Then $\Delta(n) = n \cdot C_i^l + \sum_{k=0}^{n} d_{i,k} + (C_i^l - 1)$. $N(\Delta(n)) \leq n + 1$ and $\forall_{n_1 \leq n_2} N(\Delta(n_1)) - n_1 \geq N(\Delta(n_2)) - n_2$ since $\forall_{k_1 \leq k_2} d_{i,k_1} \leq d_{i,k_2}$. Hence the maximum number of instances can be founded when time window is shifted by $\Delta(0)$, which is $\overrightarrow{n_i}(\Delta t)$. □

Lemma 4.3.3. *The maximum amount of net execution time of τ_i in a time window Δt when the number of instances is $\overrightarrow{n_i}(\Delta t)$ is $\overrightarrow{t_i^{max}}(\Delta t)$.*

Proof. Let $t_i^{exec}(\Delta t)$ be the execution time in time window Δt. According to the proof of Lemma 4.3.2, the maximum number of instances is found when time window is shifted by $\Delta(0)$. For shifting amount $\Delta > \Delta(0)$, $t_i^{exec}(\Delta t) \leq \overrightarrow{t_i^{max}}(\Delta t)$ since the distances between two tasks are monotonically increase. for shifting amount $\Delta \leq \Delta(0)$, $t_i^{exec}(\Delta t) = t_i^{max}(\Delta t)$ if $N(\Delta) = 0$ and $t_i^{exec}(\Delta t) = \overrightarrow{t_i^{max}}(\Delta t)$ if $N(\Delta) = 1$. Since $t_i^{max}(\Delta t) \geq \overrightarrow{t_i^{max}}(\Delta t)$, the maximum number of instances when $t_i^{exec}(\Delta t) > \overrightarrow{t_i^{max}}(\Delta t)$ is $n_i(\Delta t)$. □

PE-level Resource Demand Bound Function Derivation: Finally, we summarize our resource demand bound function $D^F_{T_{c,p},s}(\Delta t)$. When we distribute the time amount Δt to a task set $T_{c,p}$, we should consider the constraint that a task τ_i can be assigned the bounded *net* execution time between $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$. And, for a given Δt, we consider two cases where the number of instances of a task τ_i is $n_i(\Delta t)$ or $\overrightarrow{n_i}(\Delta t)$ as the demand bound function of each individual task, $D^E_{i,s}(t_i, \Delta t)$. In summary, the new resource demand bound function is formulated as follows:

$$D^F_{T_{c,p},s}(\Delta t) = \max \left\{ \sum_{\tau_i \in T_{c,p}} D^E_{i,s}(t_i, \Delta t) \;\middle|\; \begin{array}{c} \sum\limits_{\tau_i \in T_{c,p}} t_i = \Delta t, \\ \forall_{\tau_i \in T_{c,p}} t_i \geq t_i^{min}(\Delta t) \end{array} \right\} \tag{4.8}$$

$$D^E_{i,s}(t_i, \Delta t) = \max \begin{pmatrix} \eta_{i,s}^{e(n_i(\Delta t))}(\min(t_i, t_i^{max}(\Delta t))) \\ \eta_{i,s}^{e(\overrightarrow{n_i}(\Delta t))}(\min(t_i, \overrightarrow{t_i^{max}}(\Delta t))) \end{pmatrix} \cdot w_{i,s} \tag{4.9}$$

$D^F_{T_{c,p},s}(\Delta t)$ can be obtained by the max-plus convolution of individual demand bound functions of equation (4.9) in polynomial time since the max-plus convolution has associative property and commutative property.

Theorem 4.3.1. *$D^F_{T_{c,p},s}(\Delta t)$ function always finds a tighter upper bound of resource demand than $D^f_{T_{c,p},s}(\Delta t)$ function for any time window Δt, or $\forall_{\Delta t} D^F_{T_{c,p},s}(\Delta t) \leq D^f_{T_{c,p},s}(\Delta t)$.*

Proof. First, we prove $\forall_{\Delta t} D^F_{T_{c,p},s}(\Delta t) \leq D^e_{T_{c,p},s}(\Delta t)$. Let t_i^e and t_i^F denote the allocated time in $D^e_{T,s}(\Delta t)$ and $D^F_{T,s}(\Delta t)$ respectively. Then,

$$D^F_{T_{c,p},s}(\Delta t) = \sum_{\tau_i \in T_{c,p}} \max \begin{pmatrix} \eta_{i,s}^{e(n_i(\Delta t))}(\min(t_i^F, t_i^{max}(\Delta t))) \\ \eta_{i,s}^{e(\overrightarrow{n_i}(\Delta t))}(\min(t_i^F, \overrightarrow{t_i^{max}}(\Delta t))) \end{pmatrix} \cdot w_{i,s}$$

$$\leq \sum_{\tau_i \in T_{c,p}} \max \begin{pmatrix} \eta_{i,s}^{e(\infty)}(\min(t_i^F, t_i^{max}(\Delta t))) \\ \eta_{i,s}^{e(\infty)}(\min(t_i^F, \overrightarrow{t_i^{max}}(\Delta t))) \end{pmatrix} \cdot w_{i,s}$$

$$= \sum_{\tau_i \in T_{c,p}} \eta_{i,s}^{e(\infty)}(\min(t_i^F, t_i^{max}(\Delta t))) \cdot w_{i,s} = \sum_{\tau_i \in T_{c,p}} \eta_{i,s}^{e}(\min(t_i^F, t_i^{max}(\Delta t))) \cdot w_{i,s}$$

$$\leq \sum_{\tau_i \in T_{c,p}} \eta_{i,s}^{e}(t_i^F) \cdot w_{i,s} \leq \sum_{\tau_i \in T_{c,p}} \eta_{i,s}^{e}(t_i^e) \cdot w_{i,s} = D^e_{T_{c,p},s}(\Delta t)$$

The last inequality comes from the fact that t_i^e maximizes $D^e_{T_{c,p},s}(\Delta t)$ by definition.

Second, we prove $\forall_{\Delta t} D^F_{T_{c,p},s}(\Delta t) \leq D^a_{T_{c,p},s}(\Delta t)$.

$$D^F_{T_{c,p},s}(\Delta t) = \sum_{\tau_i \in T_{c,p}} D^E_{i,s}(t_i^F, \Delta t) \leq \sum_{\tau_i \in T_{c,p}} D^E_{i,s}(\Delta t, \Delta t)$$

$$= \sum_{\tau_i \in T_{c,p}} \max \begin{pmatrix} \eta_{i,s}^{e(n_i(\Delta t))}(\min(\Delta t, t_i^{max}(\Delta t))) \\ \eta_{i,s}^{e(\overrightarrow{n_i}(\Delta t))}(\min(\Delta t, \overrightarrow{t_i^{max}}(\Delta t))) \end{pmatrix} \cdot w_{i,s}$$

$$= \sum_{\tau_i \in T_{c,p}} \max \begin{pmatrix} \eta_{i,s}^{e(n_i(\Delta t))}(t_i^{max}(\Delta t)) \\ \eta_{i,s}^{e(\overrightarrow{n_i}(\Delta t))}(\overrightarrow{t_i^{max}}(\Delta t)) \end{pmatrix} \cdot w_{i,s} = D^a_{T_{c,p},s}(\Delta t)$$

according to Definition 4.1.1. □

Theorem 4.3.2. *$D^F_{T_{c,p},s}(\Delta t)$ function makes a conservative upper bound of demand to shared resource sr_s that may occur from a task set $T_{c,p}$.*

Proof. For any time window of size Δt, although the *net* execution time of each task varies by the schedule, the total execution time of tasks in $T_{c,p}$ cannot exceed the size of

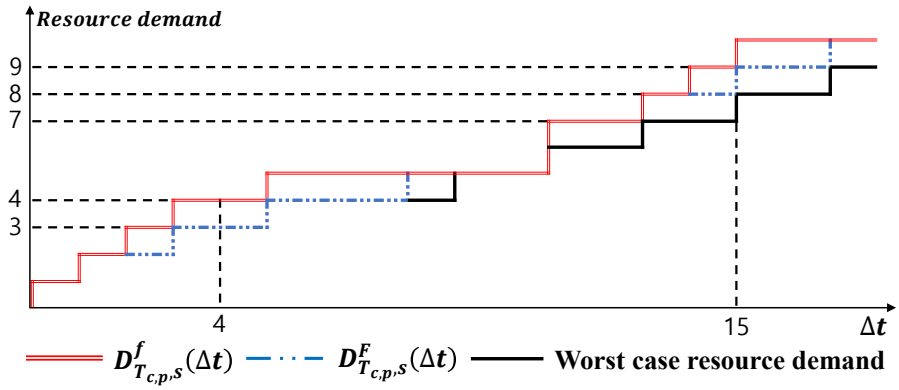


Figure 4.12: $D^F_{T_{c,p},s}(\Delta t)$ computation for an example task set and demand comparison with actual worst-case resource demand

time window Δt. Since each task τ_i should execute at least $t_i^{min}(\Delta t)$ in any time window Δt according to Lemma 4.3.1, the problem is subject to the constraints $\sum_{\tau_i \in T_{c,p}} t_i = \Delta t$ and $\forall_{\tau_i \in T_{c,p}} t_i \geq t_i^{min}(\Delta t)$.

When a task τ_i is assigned t_i to be executed in a time window Δt, it cannot take more than $t_i^{max}(\Delta t)$ in any time window Δt (Lemma 4.3.1) and its execution cannot appear more than $\overrightarrow{n_i}(\Delta t)$ times as stated in Lemma 4.3.2. When the number of instances is maximized as $\overrightarrow{n_i}(\Delta t)$, the execution time is bounded by $\overrightarrow{t_i^{max}}(\Delta t) \leq t_i^{max}(\Delta t)$ as stated in Lemma 4.3.3. Hence $D^E_{i,s}(t_i, \Delta t)$ finds the maximum resource demand since it considers both cases when the execution time and the number of instances in a time window are maximized. Consequently, $D^F_{T_{c,p},s}(\Delta t)$ is conservative since it finds the worst-case time distribution to tasks that maximizes the total resource demand. □

Figure 4.12 shows the resource demand bound curve $D^F_{T_{c,p},s}(\Delta t)$ for the example of Figure 4.1. Compared with $D^f_{T_{c,p},s}(\Delta t)$ shown in Figure 4.6, it makes tighter estimation to the actual worst-case resource demand. We summarize the computation process in Table 4.2 for $\Delta t = 4$ and $\Delta t = 15$ where the difference between two techniques is evident.

Table 4.2: $D^F_{T_{c,p},s}(\Delta t)$ computation for $\Delta t = 4$ and $\Delta t = 15$

$\Delta t = 4$					$\Delta t = 15$				
τ_0	$n_0(\Delta t)$	$t_0^{max}(\Delta t)$	$\overrightarrow{n_0}(\Delta t)$	$\overrightarrow{t_0^{max}}(\Delta t)$	τ_0	$n_0(\Delta t)$	$t_0^{max}(\Delta t)$	$\overrightarrow{n_0}(\Delta t)$	$\overrightarrow{t_0^{max}}(\Delta t)$
	1	4	1	1		2	8	2	7
τ_1	$n_1(\Delta t)$	$t_1^{max}(\Delta t)$	$\overrightarrow{n_1}(\Delta t)$	$\overrightarrow{t_1^{max}}(\Delta t)$	τ_1	$n_1(\Delta t)$	$t_1^{max}(\Delta t)$	$\overrightarrow{n_1}(\Delta t)$	$\overrightarrow{t_1^{max}}(\Delta t)$
	1	4	1	1		2	9	2	7
$D^F_{T_{c,p},s}(\Delta t) = \eta^{e(1)}_{0,s}(1) + \eta^{e(1)}_{1,s}(3) = 1+2 = 3$					$D^F_{T_{c,p},s}(\Delta t) = \eta^{e(2)}_{0,s}(6) + \eta^{e(2)}_{1,s}(9) = 3+5 = 8$				

4.3.2 SR Contention Modeling for Dependent Tasks

In this subsection, we propose a novel conservative modeling technique of SR contention supporting dependent tasks. As same as the previous subsection, we derive a SR demand bound function $D^F_{T_{c,p},s}(\Delta t)$ per PE by considering the mapped tasks all together since tasks mapped to a same PE cannot be executed in parallel. In addition, we consider the dependency of tasks in this subsection.

For each task τ_c, each PE pe_p, and each shared resource sr_s, the SR contention modeling module derives $D^F_{T_{c,p},s}(\Delta t)$ considering a set of tasks that may contribute to the resource access delay of task τ_c and execute in parallel with τ_c in pe_p. It consists of three submodules: Separation Analysis, Task Clustering, and PE-level SR Demand Bound Function Derivation, as shown in Figure 4.13. In the Separation Analysis, we compute the minimum distance between two tasks in the same task graph and check whether they can be simultaneously executed or not. In case two tasks are completely separated, meaning that they cannot be simultaneously executed at all times, we need not consider the resource demand of the other task. In the Task Clustering, we make a cluster of tasks that will be executed sequentially to consider their resource demands all together. Finally PE-level SR Demand Bound functions are derived considering the task execution order of each cluster in the last sub-module. Now we explain each submodule in detail.

Separation Analysis: Two tasks may be completely separated due to dependency in a

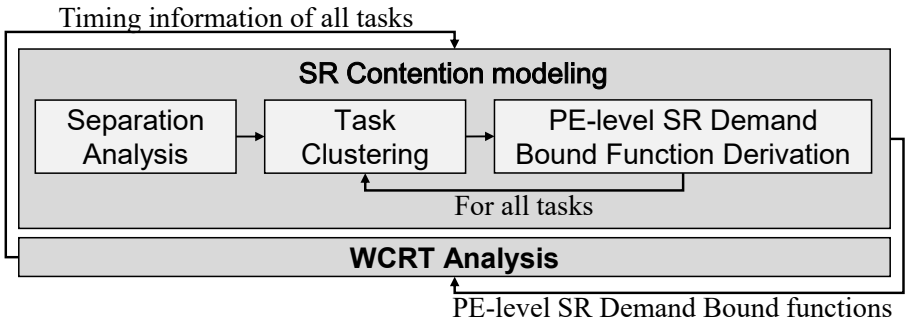


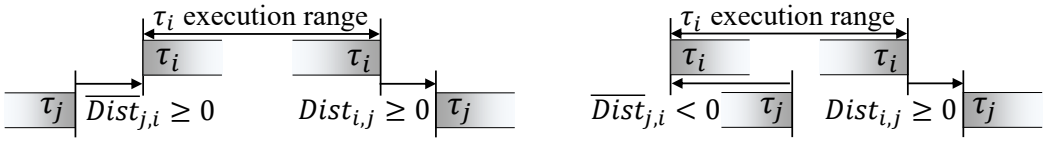
Figure 4.13: The overall flow of the SR contention modeling for dependent tasks

task graph. If two tasks have a parent-child relationship, they are definitely separated. Between two tasks without direct dependency, we still can check if they are completely separated based on two pairs of time bound information, (RB_i^l, RB_i^u) and (FB_i^l, FB_i^u), for each task τ_i. Separation analysis is accomplished by computing the minimum possible scheduling distance between two tasks in the same task graph.

We define $Dist_{i,j}$ as the minimum distance from the finish time of τ_i to the release time of τ_j. Note that the sign of $Dist_{i,j}$ means the direction. If it is non-negative, the finish time of τ_i is no later than the release time of τ_j. Otherwise, τ_i may be simultaneously executed with τ_j. Since the task release and finish times are computed based on the graph release time, only time bounds of tasks in the same task graph are comparable. $Dist_{i,j}$ is formulated with the following equation:

$$Dist_{i,j} = \begin{cases} RB_i^l - FB_i^l, & \text{if } i = j \\ \max\limits_{\tau_s \in succ(\tau_i)} \left(FB_s^l - RB_s^l + Dist_{s,j} \right), & \text{else if } \tau_i \in ancestors(\tau_j) \\ RB_j^l - FB_i^u, & \text{else if } \mathcal{G}_{\tau_i} = \mathcal{G}_{\tau_j} \\ -\infty, & \text{otherwise} \end{cases} \tag{4.10}$$

where $succ(\tau_i)$ and $ancestors(\tau_j)$ denote a set of immediate successors of τ_i and a set of ancestors of τ_j, respectively. $Dist_{i,i}$ is set to $RB_i^l - FB_i^l$, which is the negative value of the best-case response time of τ_i. If task τ_i is an ancestor of τ_j, the minimum distance can be



(a) τ_i and τ_j are separated when $Dist_{i,j} \geq 0$ and $\overline{Dist}_{j,i} \geq 0$

(b) The previous instance of τ_j may overlap with τ_i when $\overline{Dist}_{j,i} < 0$

Figure 4.14: $Dist_{i,j}$ and $\overline{Dist}_{j,i}$ computation for separation analysis

computed recursively by computing the minimum distance from each successor of τ_i. To compute the minimum distance, we add the best-case response time of each successor to the distance from the successor to τ_j. In case there are multiple dependency paths from task τ_i to τ_j, we choose the maximum value. In case two tasks τ_i and τ_j belong to the same task graph and have no direct dependency, we compute the minimum distance by the difference of earliest release time of τ_j and the latest finish time of τ_i for conservative computation. If no aforementioned case is satisfied, the distance is set to a large negative value, implying that there is no dependency between two tasks if they belong to different task graphs.

In case the deadline is larger than the period of an application, the instances from different iterations may affect each other. Although a task instance has a positive distance from another task in the current iteration, its execution may overlap with the task of the previous iteration. In order to examine the possibility of overlap between task instances that belongs to different iterations, we define another distance variable $\overline{Dist}_{j,i}$ to indicate the minimum distance from the finish time of τ_j of the previous iteration to the release time of τ_i in the current iteration as follows:

$$\overline{Dist}_{j,i} = \begin{cases} RB_i^l - \left(FB_j^u - T_{\mathcal{G}_{\tau_j}}\right), & \text{if } \mathcal{G}_{\tau_i} = \mathcal{G}_{\tau_j} \\ -\infty, & \text{otherwise} \end{cases} \tag{4.11}$$

If both $Dist_{i,j}$ and $\overline{Dist}_{j,i}$ are non-negative, there is no overlap between two tasks as illustrated in Figure 4.14 (a). If $\overline{Dist}_{j,i}$ is negative while $Dist_{i,j}$ is positive, the previous

Algorithm 2 Task clustering algorithm with three input parameters τ_c, pe_p, sr_s

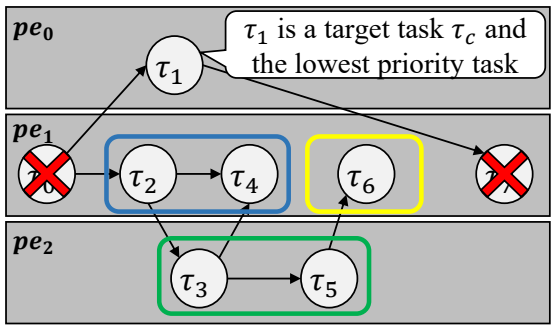
1: $Cand_{c,p,s} \leftarrow \left\{ \tau_i \;\middle|\; \begin{array}{c} \tau_c \in T_{c,p}, w_{i,s} > 0, \\ \left(Dist_{c,i} < 0 \; or \; \overline{Dist}_{i,c} < 0\right), \left(Dist_{i,c} < 0 \; or \; \overline{Dist}_{c,i} < 0\right) \end{array} \right\}$
2: Sort $Cand_{c,p,s}$ in topological order, $CS_{c,p,s} \leftarrow \emptyset$
3: **while** $Cand_{c,p,s} \neq \emptyset$ **do**
4: $\quad \tau_l \leftarrow$ leftmost task in a set $Cand_{c,p,s}$
5: $\quad \tau_r \leftarrow \tau_l, C \leftarrow \{\tau_l\}, Cand_{c,p,s} \leftarrow Cand_{c,p,s} - \{\tau_l\}$
6: $\quad$ **repeat**
7: $\quad\quad R \leftarrow \{\tau_i | \tau_i \in Cand_{c,p,s}, Dist_{r,i} \geq 0, \overline{Dist}_{i,l} \geq 0\}$
8: $\quad\quad \tau_r \leftarrow \tau_k$ that $\tau_k \in R$ and $Dist_{r,k} = \min_{\tau_i \in R} Dist_{r,i}$
9: $\quad\quad$ **if** $R \neq \emptyset$ **then**
10: $\quad\quad\quad$ Move τ_r from $Cand_{c,p,s}$ to the rightmost in C
11: $\quad\quad$ **end if**
12: $\quad$ **until** $R = \emptyset$
13: $\quad$ Add task cluster C to task cluster set $CS_{c,p,s}$
14: **end while**
15: **return** $CS_{c,p,s}$

iteration may affect the execution of τ_i, which is the case of Figure 4.14 (b).

Task Clustering: In the task clustering module sketched in Algorithm 2, we first collect tasks into a set $Cand_{c,p,s}$ that can be simultaneously executed with the target task τ_c in pe_p (condition $\tau_i \in T_{c,p}$) and have accesses to sr_s (line 1). If $\left(Dist_{c,i} \geq 0, \overline{Dist}_{i,c} \geq 0\right)$ or $\left(Dist_{i,c} \geq 0, \overline{Dist}_{c,i} \geq 0\right)$, τ_i is always separated from τ_c so that it does not contribute to the shared resource contention of τ_c.

Among the chosen tasks, we make a cluster of tasks that will be executed in order in any case; tasks in the cluster do not interfere with each other. By considering a cluster as a whole, we can derive a tighter bound on the overall resource demand than considering the resource demand of each task separately since we can analyze the distance between resource accesses issued in the cluster.

From the candidate task set $Cand_{c,p,s}$, we make an ordered task cluster C in the execution order of tasks. After a task is chosen and moved to the cluster C (lines 4 and 5), we find a task that is separated from the current ordered cluster C and has the closest distance from the rightmost task in cluster C (lines 7-8). If such a task is found, it is



(a) A task graph and PE mapping

	(RB_i^l, RB_i^u)	(FB_i^l, FB_i^u)
τ_0	(0,0)	(10,10)
τ_1	(10,10)	(60,60)
τ_2	(10,10)	(20,20)
τ_3	(20,20)	(30,30)
τ_4	(30,30)	(40,50)
τ_5	(30,30)	(35,70)
τ_6	(35,70)	(45,80)
τ_7	(60,60)	(70,80)

(b) Time bound information

Figure 4.15: An example of Task Clustering

moved from $Cand_{c,p,s}$ to cluster C (lines 9-11). This procedure is repeated until there is no separated task (line 12). Task Clustering procedure finishes when all candidate tasks cluster (line 3).

Consider the example of Figure 4.15 where tasks are mapped to three PEs. For simple illustration, we assume that every task accesses the same resource sr_s and the deadline is not greater than the period so that $\overline{Dist}_{i,j}$ is always positive. Let task τ_1 be the target task τ_c and the lowest priority task. τ_0 and τ_7 are excluded from $Cand_{1,1,s}$ since they have parent-child relationship with τ_1. Then τ_2 is selected as the first element of cluster. $Dist_{2,4} = \max(FB_3^l - RB_3^l + Dist_{3,4}, FB_4^l - RB_4^l + Dist_{4,4}) = \max(10 + (10-10), 0) = 10$ and $Dist_{2,6} = RB_6^l - FB_2^u = 35 - 20 = 15$ so that τ_4 and τ_6 are in R. Since τ_4 is closer than τ_6, τ_4 is added to the cluster. τ_6 constructs its own cluster since $Dist_{4,6} = RB_6^l - FB_4^u = 35 - 50 = -15$. Similarly for pe_2, τ_3 and τ_5 cluster together because $Dist_{3,5} = 0$.

PE-level SR Demand Bound Function Derivation: After all task clusters are formed, we construct the PE-level SR demand bound function by distributing the time window Δt to each task cluster for execution.

Since one PE can be occupied by only one task at a time, there are plenty of task execution scenarios that fill a time window Δt. Then the problem of computing the maximum shared resource demand bound issued in a PE within a time window Δt becomes

the problem that finds the combination of task cluster execution amounts to produce the maximum shared resource demand among all possible combinations. The PE-level SR Demand Bound function is formulated as follows:

$$D^F_{T_{c,p},s}(\Delta t) = \max \left\{ \sum_{C \in CS_{c,p,s}} D^E_{C,s}(t_C, \Delta t) \;\middle|\; \sum_{C \in CS_{c,p,s}} t_C = \Delta t \right\} \tag{4.12}$$

where $CS_{c,p,s}$, t_C, and $D^E_{C,s}(t_C, \Delta t)$ denote a task cluster set obtained from Task Clustering of Algorithm 2 with arguments (τ_c, pe_p, sr_s), a time amount allocated for execution of task cluster C, and a function that finds the maximum shared resource demand bound when task cluster C executes t_C amount of time in a time window Δt, respectively. $D^F_{T_{c,p},s}(\Delta t)$ can be obtained by the max-plus convolution of individual functions $D^E_{C,s}(t_C, \Delta t)$ per cluster in polynomial time since the max-plus convolution has associative property and commutative property. Then the remaining problem is to derive a function $D^E_{C,s}(t_C, \Delta t)$ for each cluster.

Let $\tau_{C[i]}$ be the i-th task in cluster C and n be the number of tasks in cluster C. Since a cluster C includes a set of tasks that is executed sequentially, the worst-case cluster execution can be represented as a weighted cyclic graph as shown in Figure 4.16 (a). An edge has a weight that means the distance between two tasks. Note that we assume that two iterations of cluster execution have the minimum inter-iteration distance $\overline{Dist}_{C[n],C[1]}$ to obtain the worst-case resource demand. The subsequent iterations appear periodically as indicated by the feedback arc from the rightmost task to the leftmost task in the cluster, with the edge weight $T_{\mathcal{G}_{\tau_{C[1]}}} - \sum_{i=1}^{n} C^l_{C[i]} - \sum_{i=1}^{n-1} Dist_{C[i],C[i+1]}$.

Note that the minimum inter-iteration distance can be negative. In this case, cluster C consists of only one task. Then some task instances may appear in a bursty fashion as shown in Figure 4.16 (b). Let the number of burst instances x. Then the first positive distance, δ in Figure 4.16 (b), can be computed as $T_{\mathcal{G}_{\tau_{C[1]}}} \cdot x - \left\{ \left(FB^u_{C[1]} - RB^l_{C[1]} \right) + (x-1) \cdot C^l_{C[1]} \right\}$. Since it is positive value, the number of burst instances becomes $x = \lceil \left(\left(FB^u_{C[1]} - \right.\right.$

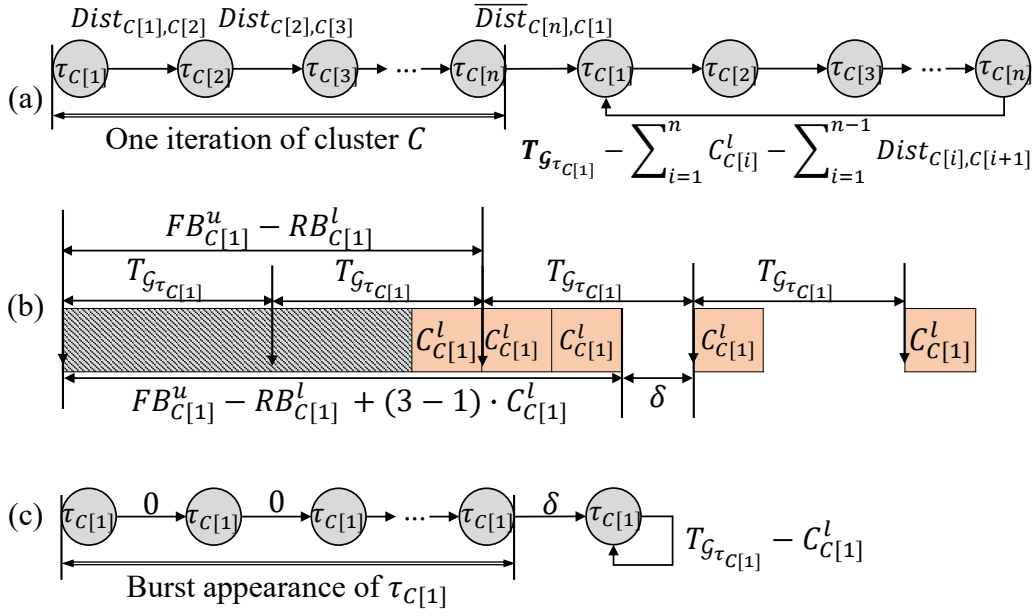


Figure 4.16: The cluster execution flow graph with weighted edge: (a) a weighted cyclic graph for a cluster of multiple tasks, (b) finding minimum distances in bursty execution schedule of a cluster of one task, (c) a weighted cyclic graph for a cluster of one task.

$RB^l_{C[1]}) - C^l_{C[1]}) / (T_{\mathcal{G}_{\tau_{C[1]}}} - C^l_{C[1]})\rceil$. The subsequent instances appear periodically. Finally, the corresponding execution flow graph for burst execution is modeled as Figure 4.16 (c).

With the information of cluster execution flow graph described above, we can compute $D^E_{C,s}(t_C, \Delta t)$ that is the maximum resource demand issued by tasks in a cluster C when cluster C executes t_C time amount within a time window Δt. Since the absolute resource access times are not known in our application model, we assume the worst-case pattern of shared resource accesses; all accesses of the first task in time window may be concentrated at the end of execution while the resource accesses in subsequent tasks are concentrated at the start of execution, as shown in Figure 4.17 (a). With $\delta^{e(1)}_{i,s}(n)$, which is the inverse function of $\eta^{e(1)}_{i,s}(t)$, we can compute the minimum execution time of τ_i for generating n resource accesses. In Figure 4.17 (a), $\tau_{C[i]}$ generates maximum three resource accesses during one job execution ($\eta^{e(1)}_{C[i],s}\left(C^l_{C[i]}\right) = 3$). Then, we consider three resource access points in $\tau_{C[i]}$ execution, $C^l_{C[i]} - \left(\delta^{e(1)}_{C[i],s}(3) + w_{C[i],s}\right)$,

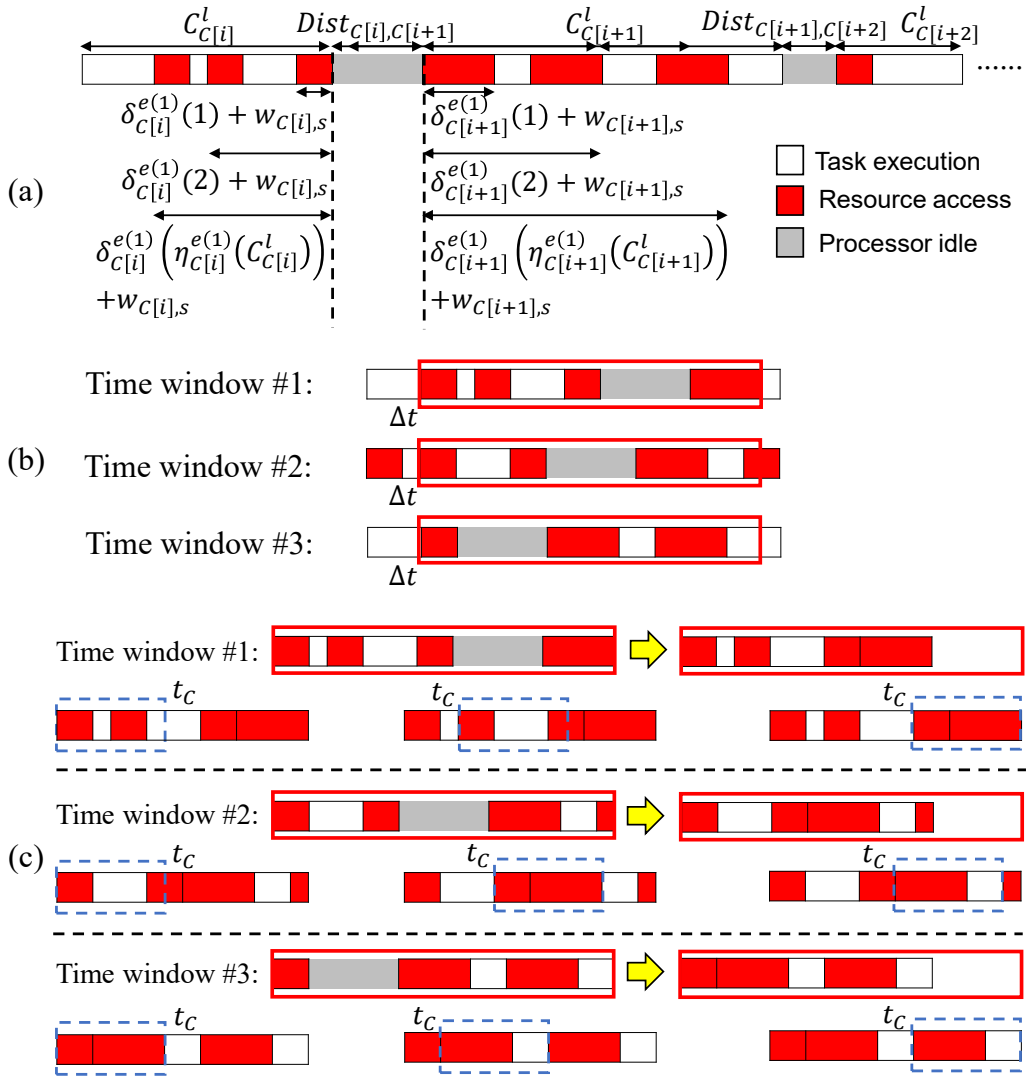


Figure 4.17: Maximum resource demand computation for execution time amount t_C and a time window Δt: (a) concentrated resource accesses at the end and the start of execution, (b) setting candidate starting points of time window at all resource access points, (c) resource demand computation when C executes the allocated time t_C in the candidate time window.

$C^l_{C[i]} - \left(\delta^{e(1)}_{C[i],s}(2) + w_{C[i],s}\right)$, and $C^l_{C[i]} - \left(\delta^{e(1)}_{C[i],s}(1) + w_{C[i],s}\right)$, which generate 3, 2, and 1 resource accesses at the end of $\tau_{C[i]}$ execution with minimum execution time, respectively.

We set all resource access points as candidate starting points of time window, as

illustrated in Figure 4.17 (b). Consider task $\tau_{C[i]}$ in Figure 4.17. Since it has three resource accesses, there are three candidate starting points associated with task $\tau_{C[i]}$. For each time window, we should consider the amount of resource demand when cluster C executes the allocated time t_C. To this end, we ignore idle time between tasks in the candidate time window, as illustrated in Figure 4.17 (c). Then we move a sub time window of lenfth t_C from left to right, and computes the resource demand within sub time window using $\eta_{i,s}^{e(1)}(\Delta t)$. For instance of candidate time window #1, we can find the maximum resource demand when sub time window is at the rightmost position. Finally, $D_{C,s}^{E}(t_C, \Delta t)$ is chosen as the maximum resource demand among all candidate time windows.

4.4 Worst-case SR Access Delay Estimation

In this section, we show how to consider the worst-case shared resource access delay in the WCRT Analysis module, assuming two arbitration policies: fixed-priority non-preemptive (FPNP) arbitration and first-in-first-out (FIFO) arbitration. There are two key ideas for modeling shared resource contention. First, we use the PE-level SR Demand Bound functions. Second, even though the worst-case resource contention is computed for each resource access, the total amount of resource contention during the response time is bounded. Note that although this section only explains how we incorporate the worst-case SR access delay estimation into the response time analysis of a single task, it can be easily extended to the proposed HPA technique. In subsection 4.4.1, we derive the upper bound of the delay time per one access. Then we derive the response time analysis equation considering the total amount of resource access delay in subsection 4.4.2.

4.4.1 Resource Access Delay Estimation Per One Access

FPNP arbitration: Consider one resource access from a task τ_t to a shared resource sr_s. Since a resource serves all accesses in a non-preemptive fashion, the resource access can be delayed by one resource access from a remote lower priority task if it is served by sr_s.

Then the blocking time by the remote lower priority task is computed as follows:

$$BT1_{t,s} = \max_{\tau_i \in \mathcal{G}_{\tau_t}} w_{i,s} \tag{4.13}$$

where $\mathcal{G}_{\tau_t} = \{\tau_i | M_i \neq M_t, PR_i < PR_t, \big(Dist_{t,i} < 0 \; or \; \overline{Dist}_{i,t} < 0\big), \big(Dist_{i,t} < 0 \; or \; \overline{Dist}_{t,i} < 0\big)\}$, a set of remote lower priority tasks that can be executed simultaneously with τ_t.

If access demands from remote higher priority tasks are pending for the shared resource, the resource access from task τ_t should wait until all higher priority resource accesses are served. This is modeled with the following recursive equation:

$$BT2_{t,s} = \sum_{\forall pe_p \neq M_t} D^F_{T_{t,p},s}(BT1_{t,s} + BT2_{t,s} + 1) \tag{4.14}$$

where $T_{t,p} = \{\tau_i | M_i = pe_p, PR_i > PR_t\}$, a set of remote higher priority tasks in pe_p. We need not consider the task separation in $T_{t,p}$ since the Task Clustering algorithm which is invoked in $D^F_{T_{t,p},s}(\Delta t)$ equation makes separated cluster set.

FIFO arbitration: When a task accesses a shared resource with FIFO arbitration policy, it can be blocked by at most one access per each PE regardless of the task priority. Hence $BT1_{t,s}$ is set to zero and we consider all remote priority tasks in $BT2_{t,s}$ computation as follows.

$$BT2_{t,s} = \sum_{\forall pe_p \neq M_t} \max_{\tau_i \in T_{c,p}} w_{i,s} \tag{4.15}$$

where $T_{t,p} = \{\tau_i | M_i = pe_p, \big(Dist_{t,i} < 0 \; or \; \overline{Dist}_{i,t} < 0\big), \big(Dist_{i,t} < 0 \; or \; \overline{Dist}_{t,i} < 0\big)\}$, a set of all remote tasks that can be executed simultaneously with τ_t in pe_p. We choose one access of maximum duration per each PE.

The total resource access delay of one access for both arbitation policy becomes $BT1_{t,s} + BT2_{t,s} + w_{t,s}$. Note that we device the access delay into $BT1_{t,s}$ and $BT2_{t,s}$ in order to bound the total resource access delay within the response time, which will be explained in next subsection.

4.4.2 Response Time Analysis with SR Access Delay

In this subsection, we derive the response time analysis equation that incorporates the worst-case SR access delay for fixed-priority preemptive scheduling and fixed-priority non-preemptive scheduling.

Preemptive: A higher priority task cannot preempt a lower priority task if the lower priority task is accessing the shared resource, according to our problem model. To account for the worst-case blocking time due to a low priority task that is mapped to the same processor, we choose the maximum among the response times of local lower priority resource accesses as follows:

$$RT1_t = \max_{\tau_i \in \mathcal{H}_{\tau_t}} \max_{\forall_{sr_s}(w_{i,s} \neq 0)} (BT1_{i,s} + BT2_{i,s} + w_{i,s}) \tag{4.16}$$

where $\mathcal{H}_{\tau_t} = \{\tau_i | M_i = M_t, PR_i < PR_t, \big(Dist_{t,i} < 0 \; or \; \overline{Dist}_{i,t} < 0\big), \big(Dist_{i,t} < 0 \; or \; \overline{Dist}_{t,i} < 0\big)\}$. Next, we compute the maximum amount of task executions within the WCRT as $RT2_t$. For simplicity, we define a function $\eta_i^+(\Delta t)$ to indicate the maximum number of executions of task τ_i during a time window Δt. Then, $RT2_t$ can be expressed by the following equation:

$$RT2_t = C_t^u + \sum_{\tau_i \in I_{\tau_t}} \eta_i^+(\mathcal{R}_{\tau_t}) \cdot C_i^u \tag{4.17}$$

where $I_{\tau_t} = \{\tau_i | M_i = M_t, PR_i > PR_t, \big(Dist_{t,i} < 0 \; or \; \overline{Dist}_{i,t} < 0\big), \big(Dist_{i,t} < 0 \; or \; \overline{Dist}_{t,i} < 0\big)\}$ and $\mathcal{R}_{\tau_t} = FB_t^u - RB_t^u$. Finally, we compute the worst-case SR access delay due to the tasks that are mapped to the other PEs. Since all resource accesses issued during the response time of τ_t may experience resource contention, the amount of contention is

$$\begin{aligned} RT3_t = & \sum_{\forall sr_s} \left(\eta_{t,s}^{e(1)}(C_t^u) \cdot (w_{t,s} + BT1_{t,s}) + \sum_{\tau_i \in I_{\tau_t}} (\eta_i^+(\mathcal{R}_{\tau_t}) \cdot \eta_{i,s}^{e(1)}(C_i^u) \cdot (w_{i,s} + BT1_{i,s})) \right) \\ & + \sum_{\forall sr_s} \min \begin{pmatrix} \sum_{\forall_{pe_p \neq M_t}} D_{T_{t,p},s}^F(\mathcal{R}_{\tau_t}), \\ \eta_{t,s}^{e(1)}(C_t^u) \cdot BT2_{t,s} + \sum_{\tau_i \in I_{\tau_t}} (\eta_i^+(\mathcal{R}_{\tau_t}) \cdot \eta_{i,s}^{e(1)}(C_i^u) \cdot BT2_{i,s}) \end{pmatrix} \end{aligned} \tag{4.18}$$

Note that the second term implies that the total amount of resource contention time by remote higher priority tasks is bounded by the maximum resource demand in the time window $\mathcal{R}_{\tau_t}$.

Then, the response time of a task τ_t can be computed by summing up all those three components: $\mathcal{R}_{\tau_t} = RT1_t + RT2_t + RT3_t$.

Non-preemptive: When task τ_t is scheduled with a non-preemptive scheduling policy, at most one execution among all lower priority tasks in the same processor can block the execution of task τ_t. The blocking time by a lower priority task is computed as follows:

$$RT1_t = \max_{\tau_i \in \mathcal{H}_{\tau_t}} \left(C_i^u + \sum_{\forall_{sr_s}} \eta_{i,s}^{e(1)}(C_i^u) \cdot BT1_{i,s} + \sum_{\forall_{sr_s}} \min \begin{pmatrix} \sum_{\forall_{pe_p \neq M_i}} D^F_{T_{i,p},s}(RT1_t), \\ \eta_{i,s}^{e(1)}(C_i^u) \cdot BT2_{i,s} \end{pmatrix} \right) \tag{4.19}$$

Note that we bound the total resource contention due to remote higher priority tasks by $\sum_{\forall_{pe_p \neq M_i}} D^F_{T_{i,p},s}(RT1_t)$. Since a task execution will not be preempted by any other task after it starts, we need to consider the interference before its start time. Then the maximum amount of task executions within the WCRT can be computed as follows:

$$RT2_t = C_t^u + \sum_{\tau_i \in I_{\tau_t}} (\eta_i^+(\overleftarrow{\mathcal{R}_{\tau_t}}) \cdot C_i^u) \tag{4.20}$$

where $\overleftarrow{\mathcal{R}_{\tau_t}} = \mathcal{R}_{\tau_t} - C_t^u + 1$. The amount of contention during the response time is modeled similarly to the equation (4.18) as follows:

$$\begin{aligned} RT3_t = & \sum_{\forall sr_s} \left(\eta_{t,s}^{e(1)}(C_t^u) \cdot (w_{t,s} + BT1_{t,s}) + \sum_{\tau_i \in I_{\tau_t}} (\eta_i^+(\overleftarrow{\mathcal{R}_{\tau_t}}) \cdot \eta_{i,s}^{e(1)}(C_i^u) \cdot (w_{i,s} + BT1_{i,s})) \right) \\ & + \sum_{\forall sr_s} \min \begin{pmatrix} \sum_{\forall_{pe_p \neq M_t}} D^F_{T_t,p,s}(\mathcal{R}_{\tau_t}), \\ \eta_{t,s}^{e(1)}(C_t^u) \cdot BT2_{t,s} + \sum_{\tau_i \in I_{\tau_t}} (\eta_i^+(\overleftarrow{\mathcal{R}_{\tau_t}}) \cdot \eta_{i,s}^{e(1)}(C_i^u) \cdot BT2_{i,s}) \end{pmatrix} \end{aligned} \tag{4.21}$$

In summary, the response time of task τ_t in a non-preemptive processing element can be obtained by summing those three components: $\mathcal{R}_{\tau_t} = RT1_t + RT2_t + RT3_t$.

4.5 Experiments

4.5.1 Comparison with the Reference Technique

The proposed technique is compared to the reference technique through extensive experiments using benchmarks and synthetic examples. We use our own implementation of the reference technique in [10] to support our system model. For fair comparison, we used the same WCRT analysis method as [10]. Thus, for both reference and proposed techniques, the experimental results are affected only by the resource demand bound function. Note that all tasks in experiments in this subsection are independent since the reference technique does not support dependent tasks.

Experiment with Benchmarks: We assume that each core has a private/partitioned cache of 32KB size and shares a main memory. The shared resource access demands or last-level cache misses are obtained by profiling with TQSim simulator [86]. We conduct experiments with benchmark tasks obtained from MediaBench benchmark suites [87]. They are media applications such as JPEG, MPEG, and GSM which usually have real-time constraints and are triggered periodically or sporadically. Among them, we choose four representative benchmarks that have different request density and intensity. High request density implies that the resource requests are concentrated in a small range of execution time. High request intensity means that a task has many resource requests compared to its execution time. We summarize the profiled task information in Table 4.3 and the request upper bound function for each task in Figure 4.18. In this experiment, all tasks are mapped to one fixed-priority preemptive processor. Period of each task is set to 7 times of its execution time and deadline is given 100 times of its execution time. It is confirmed that all tasks are schedulable by schedulability analysis.

Figure 4.19 shows the estimated resource demand bound functions of the reference technique $D^{f}_{T_{c,p},s}(\Delta t)$ and the proposed technique $D^{F}_{T_{c,p},s}(\Delta t)$, assuming four benchmark tasks are mapped onto one processor ($T_{c,p} = \{\tau_0, \tau_1, \tau_2, \tau_3\}$). The graph shows that two

Table 4.3: Profiled task information(unit:cycle)

Task	Benchmark	C_i^l	$w_{i,s}$	Description
τ_0	mpeg2decode	536,308,261	200	low request intensity, low request density
τ_1	gsmdecode	122,895,400	200	low request intensity, high request density
τ_2	jpegencode	30,405,671	200	high request intensity, low request density
τ_3	jpegdecode	7,685,892	200	high request intensity, high request density

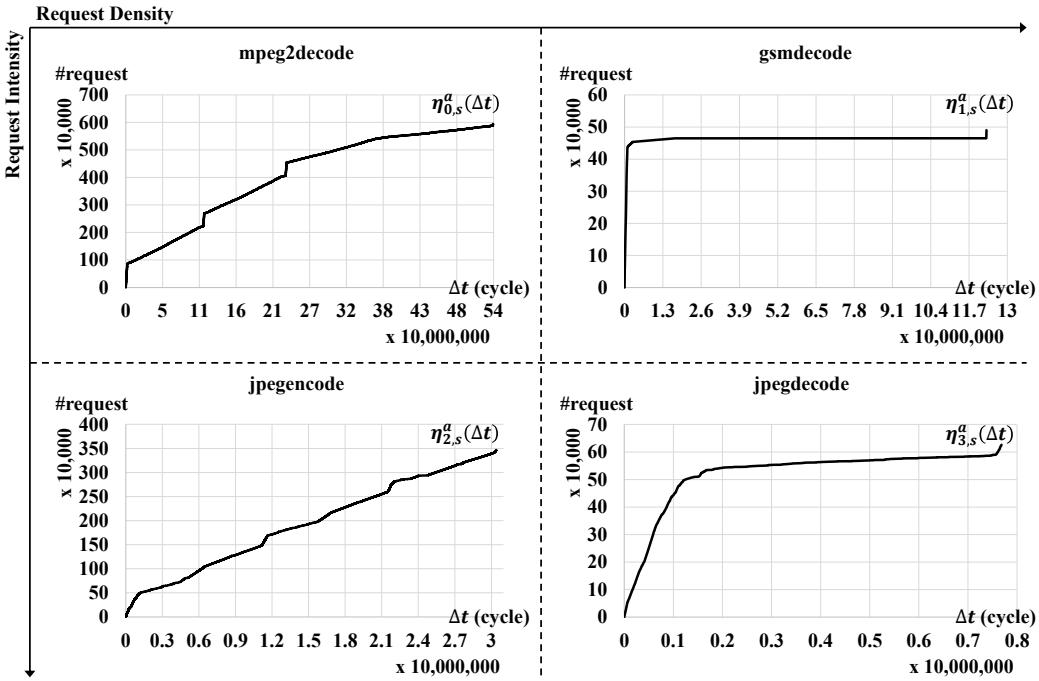


Figure 4.18: Profiled resource request upper bound functions

techniques show almost the same estimation when the time window size is less than 30,000,000, while the proposed technique shows tighter estimation than the reference technique as the time window size becomes larger. When time window size is small, both techniques show similar estimation since the reference technique tends to find a feasible schedule in the time window despite of ignorance of periodic appearance. However, the drawback of the reference technique becomes evident as the time window size grows. It confirms the superiority of the proposed technique to the reference technique.

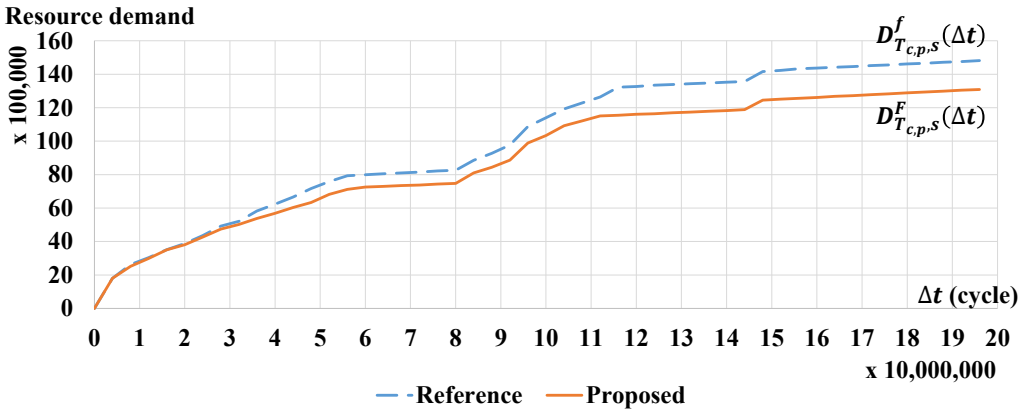


Figure 4.19: Estimated resource demand bound functions of the reference technique $D^f_{T_{c,p},s}(\Delta t)$ and the proposed technique $D^F_{T_{c,p},s}(\Delta t)$

In the second set of experiments, we perform experiments with several benchmark combinations to examine how the performance of our technique is affected by resource access patterns of tasks in $T_{c,p}$. We plot the experimental results of four representative combinations in Figure 4.20. The graphs in Figure 4.20 show the estimated resource demand bound functions of the reference technique $D^f_{T_{c,p},s}(\Delta t)$ and the proposed technique $D^F_{T_{c,p},s}(\Delta t)$ for four combinations. The first combination in Figure 4.20 (a) uses four tasks that are all mpeg2decode benchmarks. When all tasks in $T_{c,p}$ are equal, the advantage of our technique over the reference technique is not significant. It is because both techniques distribute the time window to tasks similarly in case all tasks have the same resource access patterns. The second combination in Figure 4.20 (b) consists of two mpeg2decode and two gsmdecode benchmarks. Even though gsmdecode has different resource access pattern of high request density, two techniques show same estimation results. It is preferred to include request-dense execution in the time window in order to maximize the total resource demand. However, since all tasks are low request intensive tasks and have large execution time, the estimation tendency follows the case of Figure 4.20 (a). On the other hand, when high request intensive tasks jpegencode and jpegdecode are used with mpegdecode as shown in Figure 4.20 (c) and (d), the estimation of our technique benefits

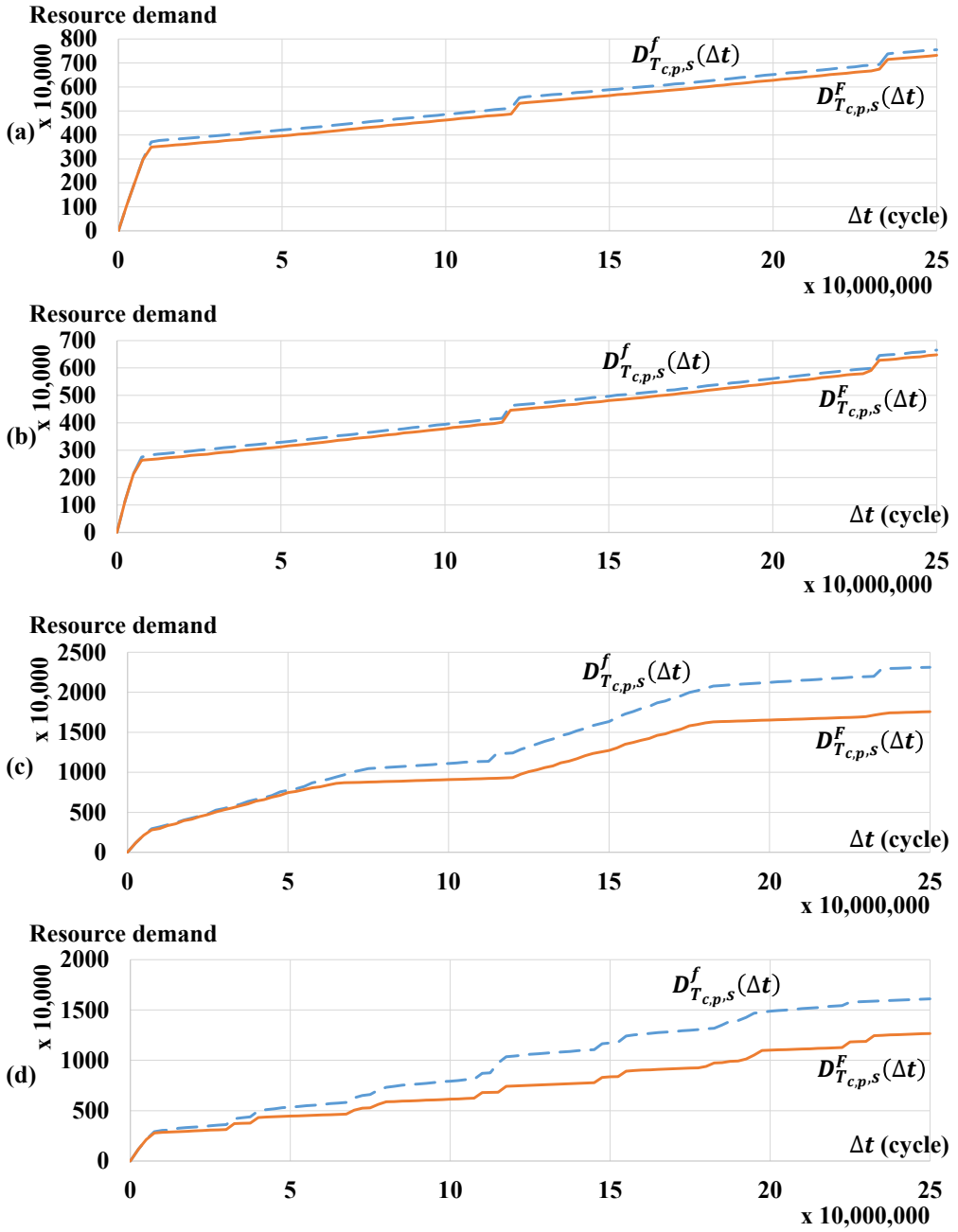


Figure 4.20: Estimated resource demand bound functions of the reference technique $D^f_{T_{c,p,s}}(\Delta t)$ and the proposed technique $D^F_{T_{c,p,s}}(\Delta t)$ for four benchmark combinations: (a) four mpeg2decode benchmarks, (b) two mpeg2decode benchmarks and two gsmdecode benchmarks, (c) two mpeg2decode benchmarks and two jpegencode benchmarks, and (d) two mpeg2decode benchmarks and two jpegdecode benchmarks

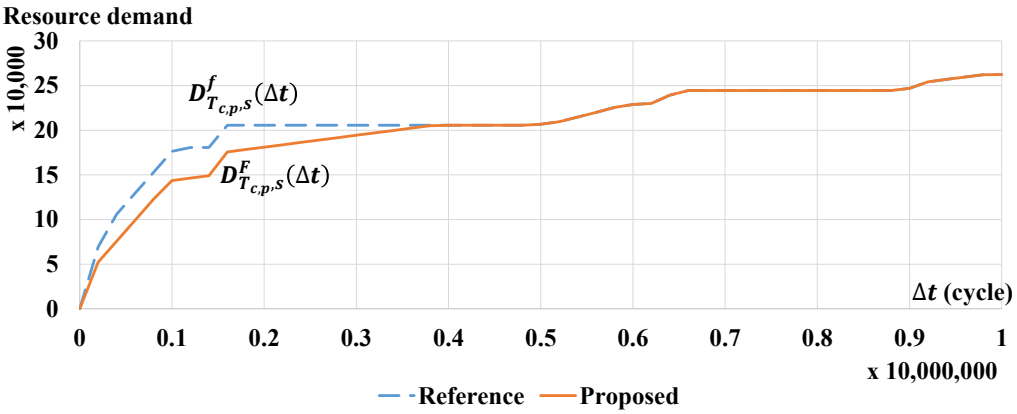


Figure 4.21: Estimated resource demand bound functions of the reference technique $D^{f}_{T_{c,p},s}(\Delta t)$ and the proposed technique $D^{F}_{T_{c,p},s}(\Delta t)$ for four synthetic benchmark examples with equal execution times

more than the reference technique. This is due to time distribution; intensive tasks occupy the time window greedily in the reference technique, which may be infeasible, while our technique limits the maximum feasible execution time in the time window.

In the third experiment, we generate four synthetic tasks that inherit the memory access pattern of the original four benchmarks, while the execution time of each synthetic task is equally fixed to 1,000,000 cycles. Then the resource demand bound function of each task is scaled accordingly with the ratio $r = \frac{\text{original execution time}}{1{,}000{,}000}$: original function $y = f(x)$ becomes $y = \frac{1}{r} \cdot f(r \cdot x)$. Four synthetic tasks are all mapped onto one processor. Figure 4.21 shows the experimental results that the proposed technique shows tighter estimation than the reference technique for the time window of size less than 4,000,000 while both techniques estimate the same bound for the large time window size. The proposed technique shows tighter estimation for the time window less than 4,000,000 since the reference technique $D^{e}_{T_{c,p},s}(\Delta t)$ fills the time window with intensive tasks greedily regardless of the schedule. On the other hand, for the time window larger than 4,000,000, the reference technique $D^{a}_{T_{c,p},s}(\Delta t)$ does not show the overestimation since the time window is enough to include all task executions. Hence the reference technique estimates the resource demand as tight as the proposed technique.

Table 4.4: Experiment Setup: synthetic example parameter ranges

#tasks	C_i^l	$n_{i,s}$	$w_{i,s}$	$d_{i,s}$
[6,10]	[150, 200]	[3,12]	[1,5]	random value satisfying $n_{i,s} \cdot (w_{i,s} + d_{i,s}) - d_{i,s} < C_i^l$

Priority	Period & Deadline
random	random value that makes the system schedulable

Experiment with Synthetic Examples: The objective of experiments in this part is to analyze the effect of task parameters to the estimation performance. To this end, we generate synthetic examples with some parameters to be controlled. Note that we intentionally allow unrealistic parameter settings.

In order to express a resource access pattern in a simple way, rather than using an event stream model, we introduce two variables $n_{i,s}$ and $d_{i,s}$ which represent the maximum number of resource accesses and the minimum distance between two accesses to shared resource sr_s during one job execution of task τ_i, respectively. Then the request upper bound function for the resource access pattern can be derived from 3-tuple $(n_{i,s}, w_{i,s}, d_{i,s})$. Even though the event stream model is more general and expressive, it is cumbersome to make random examples. Thus we used 3-tuple models to generate random access patterns easily.

Synthetic example is generated according to the following rules: The number of tasks is determined between 6 and 10. We assume that the system consists of two fixed-priority preemptive processing elements and focus on the situation where the lowest priority task running on a processing element experiences the maximum arbitration delay to access a shared resource by the higher priority tasks running on the other processing element. For each task, execution time is fixed ($C^l = C^u$) and is randomly selected in the range of [150, 200]. The maximum number of resource accesses $n_{i,s}$ is chosen between 3 and 12 and the maximum access duration $w_{i,s}$ between 1 and 5. The minimum shared resource access distance $d_{i,s}$ is also chosen randomly under the constraint that all resource

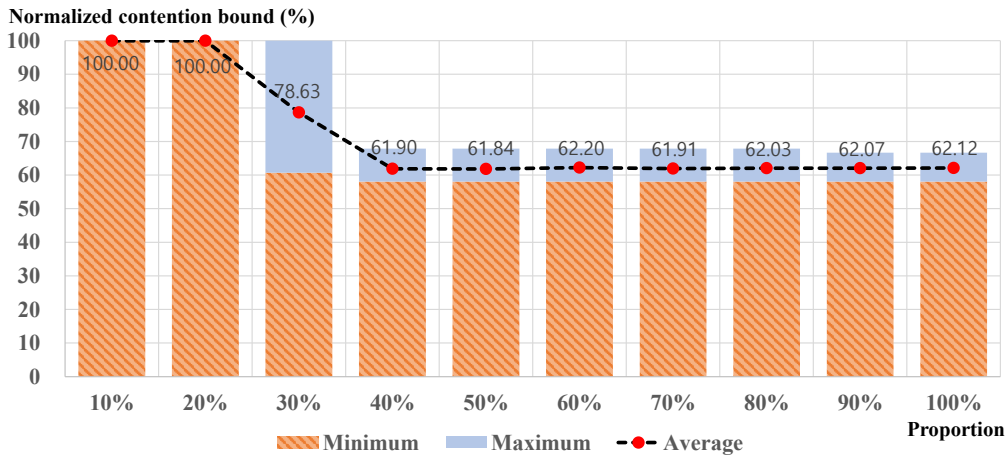


Figure 4.22: Normalized shared resource contention bound while changing the proportion of the resource access section in the task execution

accesses occur within the execution time of the task. The priority is randomly assigned to generated tasks. When we make a synthetic example, we have to make the example schedulable. The periods and deadlines are initially assigned as the task execution time. Until the system becomes schedulable, which can be checked by schedulability analysis, we increase task periods and deadlines by random amounts. We summarize the experimental setup for selecting synthetic example parameters in Table 4.4. The experiments are based on those parameter ranges basically. When some parameter setting is changed in an experiment, we will state that parameter setting before the experiment.

We conduct experiments with synthetic examples to investigate how the performance of the proposed technique is affected by the memory access pattern. Figure 4.22 shows the comparison result while changing the proportion of the resource access section during task execution, assuming the accesses occur with 10 time unit distance and each access takes 1 time unit. The x-axis means the proportion of the access section and the number of accesses is determined by the section length; if the proportion is 10%, $n_{i,s} = \frac{0.1 \cdot C_i^l + d_{i,s}}{w_{i,s} + d_{i,s}}$. We generate 100 synthetic examples for each proportion and plot the maximum and minimum normalized contention bound ratios to the reference technique

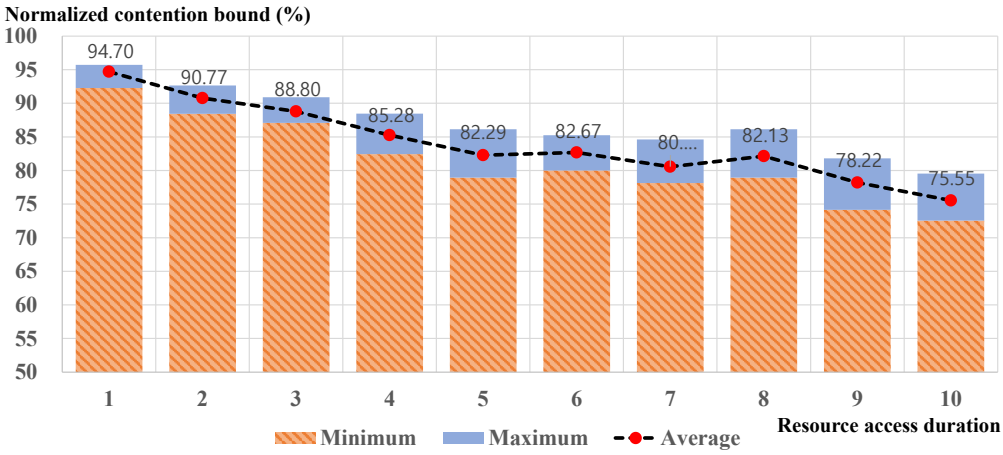


Figure 4.23: Normalized shared resource contention bound while increasing the resource access duration and the resource access distance

with the bar graphs and the average is plotted with the line graph. Contention bound is obtained by the difference between the WCRT without shared resource contention and the WCRT with shared resource contention. Then the contention bound obtained from our technique is normalized by that from the reference technique. The result implies that the performance gap between our technique and the reference technique increases as the accesses take larger portion of execution. For the small portion cases below 20%, the cause of overestimation of the reference technique by ignoring the idle time between two consecutive executions was completely hidden since the allocated time slot to each task became smaller than the task execution time due to the large portion of the task execution time without SR accesses.

In the next experiment, we vary the access pattern from frequent accesses with short duration to sparse accesses with long duration with the same proportion of resource access section, 50% of the task execution time. We make frequent accesses by setting a small value to the resource access distance $d_{i,s}$, and short duration by a small value of resource access duration $w_{i,s}$. While preserving the proportion of resource access section, we gradually increase the resource access duration from 1 to 10 and set the resource ac-

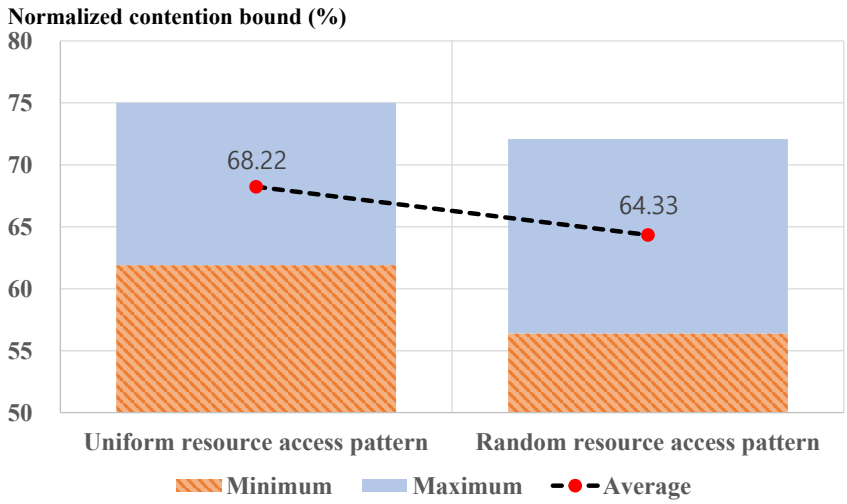


Figure 4.24: Normalized shared resource contention bound for two sets of examples with uniform resource access pattern and random resource access pattern

cess distance to be twice of the resource access duration. For instance, let the execution time $C_i^l = 100$ and we want to preserve 50% of the execution time as resource access section. If the access duration is 1 (short duration) then the distance $d_{i,s} = 2 \cdot w_{i,s} = 2$ and the number of resource accesses $n_{i,s} = \lfloor \frac{50+d_{i,s}}{w_{i,s}+d_{i,s}} \rfloor = 18$ (frequent accesses). If the access duration is 10 (long duration), then the distance $d_{i,s} = 2 \cdot w_{i,s} = 20$ and the number of resource accesses $n_{i,s} = \lfloor \frac{50+d_{i,s}}{w_{i,s}+d_{i,s}} \rfloor = 2$ (sparse accesses). In this way, the total amount of memory demand from one job execution ($w_{i,s} \cdot n_{i,s}$) is preserved. 100 synthetic examples are generated for each configuration and the maximum, minimum, and average normalized contention bound ratios are displayed in Figure 4.23. The x-axis means the resource access duration and the y-axis indicates the normalized contention bound to the reference technique. The graph shows that the performance advantage of the proposed technique becomes more evident for the sparser resource access pattern with longer duration.

In the previous two experiments, all tasks commonly have the same access duration and the access distance values. The last experiment with synthetic examples is conducted with two set of examples. The first example set has uniform resource access patterns;

Table 4.5: Configurations for verification of the key techniques

Configuration	Task Clustering	PE-level SR Demand Bound function
Config1	Not used	Not used
Config2	Not used	Used
Config3	Used	Not used
Config4	Used	Used

resource access duration, resource access distance, and resource access section are set to 1, 10, and 50% of execution, respectively. On the other hand, the resource access pattern of second example set is randomly generated. Figure 4.24 shows the comparison result from two set of random examples. The result shows that the proposed technique gives tighter upper bound than the reference technique when the tasks have different resource access patterns. Remind that function $D^{e}_{T_{c,p},s}(\Delta t)$ does not consider the processor idle time and the task with frequent memory access pattern is likely to fill the most of time window, which is the main cause of overestimation. This tendency stands out in the result of second set of examples.

4.5.2 Performance Verification of SR Contention Analysis for Dependent Tasks

In the first set of experiments, we vary the configurations of the proposed SR contention analysis technique for dependent tasks as shown in Table 4.5 to verify the effectiveness of two key modules of the proposed technique: Task Clustering and PE-level SR Demand Bound function. If PE-level SR Demand Bound function is not used, the Task-level(or Cluster-level) SR Demand Bound functions are simply summated for computing resource contention. If Task Clustering is not used, the tasks are regarded as independent tasks so that each task constructs its own cluster.

We use 100 synthetic examples that have dependent tasks. The number of PEs varies from 3 to 5, and the number of SRs from 3 to 4. The scheduling policy of a PE is either $\mathcal{P}$ or $\mathcal{N}$. The number of task graphs varies from 2 to 3 and the number of total tasks varies

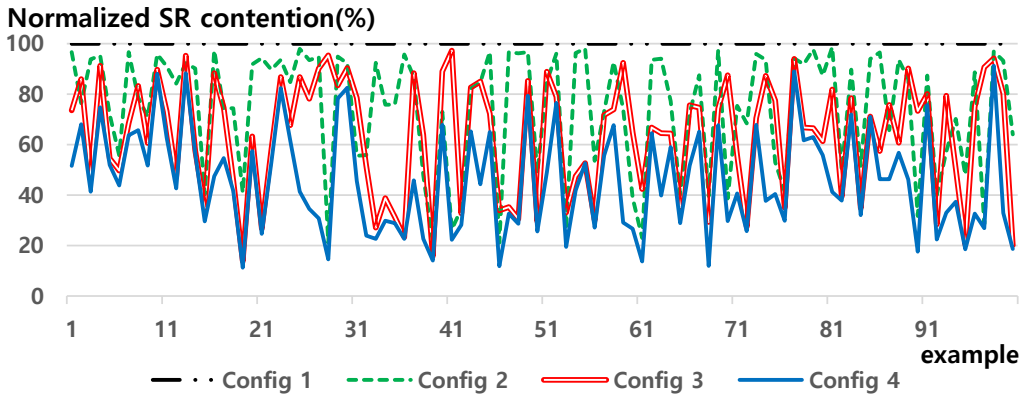


Figure 4.25: Normalized SR contention graph for four configurations

Table 4.6: The best, worst, and average performance

Configuration	Min	Max	Avg
Config2	21.05%	98.67%	74.58%
Config3	13.79%	97.20%	63.28%
Config4	11.19%	91.25%	45.68%

from 20 to 30. C^l and C^u of each task are randomly selected in the range of $[1000, 1500]$ and $[C^l, C^l \times 1.5]$. SRs that a task accesses are also randomly selected. When a task accesses a shared resource, the maximum number of accesses $n_{i,s}$ varies from 10 to 40 and the maximum access duration $w_{i,s}$ from 1 to 5. The minimum access distances $d_{i,s}$ and the periods T_G are randomly chosen.

The normalized shared resource contention for the generated 100 examples are displayed in Figure 4.25, while the results are normalized by the result of Config1. The shared resource contention is computed by the difference of the estimated WCRT with resource contention and the estimated WCRT without resource contention. From the results of Config2 and Config3 that use only one of two techniques, we observe that one technique does not dominate the other since they reduce different parts of overestimation. The task clustering technique works well if the remote tasks that may induce contentions are serially executed or separated. On the other hand, the PE-level SR Demand Bound function reduces overestimation by considering the possible resource demand from PE.

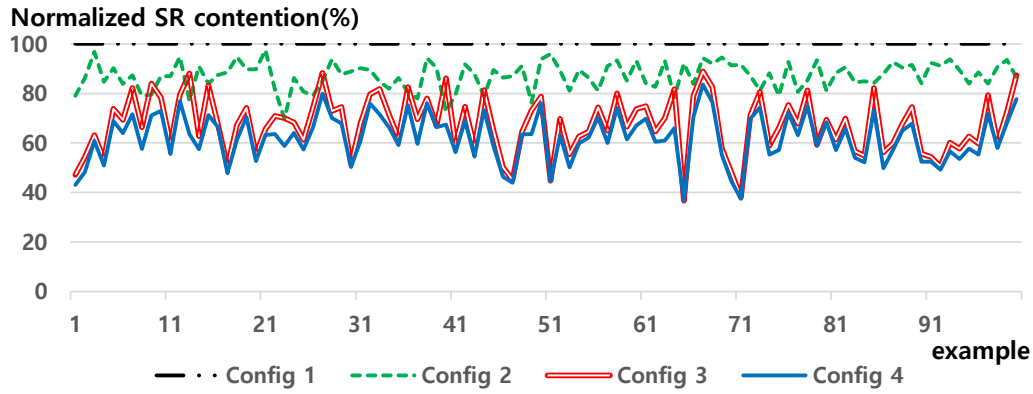


Figure 4.26: Normalized SR contention graph for four configurations with sparse resource access pattern

On average, however, the task clustering method gives more performance gain. Table 4.6 presents the best, worst, and average performance estimation ratio of Config2, Config3, and Config4, respectively, compared to Config1. Config2 and Config3 reduce the estimation by 25.42% and 36.72%, respectively on average. The proposed technique reduces the estimation of shared resource contention dramatically, on average by 54.32%.

The next set of experiments is conducted to examine the effect of resource access pattern to the estimation performance by making memory access request more sparse but with longer duration. The number of maximum accesses varies from 1 to 2 and the maximum access duration from 100 to 250. Figure 4.26 shows that the task clustering technique becomes the dominant source of performance improvement in almost all examples. Moreover the performance gap between Config1 and Config4 is reduced. If the resource accesses are infrequent and have long access duration, tasks in a PE may demand a resource continuously without pause. In that case, the overall resource demand from a PE becomes very close to the simple summation of the resource demands of all tasks, making the PE-level SR Demand Bound derivation meaningless.

In the last set of experiments, we analyze the worst-case response time of parallelized viola-jones object detection as a real-life example. After the initialization step (τ_0), each detection task (τ_1 - τ_{17}) independently proceeds using cascaded classifiers on

Table 4.7: Profiled task information (unit of bcet and wcet:cycle)

Task	C_i^l	C_i^u	$n_{i,s}$	Task	C_i^l	C_i^u	$n_{i,s}$
τ_0	28,119,576	28,120,030	8,536	τ_{10}	17,566,664	28,655,096	60,764
τ_1	541,960,280	763,061,294	2,475,671	τ_{11}	11,347,916	18,238,489	50,159
τ_2	372,602,070	531,171,589	1,248,607	τ_{12}	7,136,979	11,560,672	48,977
τ_3	258,487,562	380,000,067	888,365	τ_{13}	4,650,264	6,989,643	11,785
τ_4	181,527,732	284,875,102	727,991	τ_{14}	2,740,420	5,371,356	11,017
τ_5	126,202,815	201,385,108	567,993	τ_{15}	1,714,636	3,546,796	14,028
τ_6	85,613,575	139,997,737	522,943	τ_{16}	1,183,579	1,598,812	6,073
τ_7	57,669,176	94,305,609	233,658	τ_{17}	889,162	946,998	4,199
τ_8	38,615,013	63,730,841	146,647	τ_{18}	14,962,401	15,198,446	5,492
τ_9	25,695,213	43,626,677	100,296				

Table 4.8: Two task mapping configurations

Configuration		Tasks(index)	Configuration		Tasks(index)
Mapping1	pe_0	0, 1	Mapping2	pe_0	0, 1, 5, 12
	pe_1	2		pe_1	2, 7, 10, 14, 16, 17
	pe_2	3, 4		pe_2	3, 9, 11, 13, 15
	pe_3	5-17, 18		pe_3	4, 6, 8, 18

each downsampled integral image from 640x480 image resolution. After detection process finished, the results are combined in the finalization step (τ_{18}). We profile the task information of execution cycle and cache miss pattern using TQSim simulator [86] with a system that has a 32KB L1 cache. We profile $n_{i,s}$, $w_{i,s}$, and $d_{i,s}$ for the cache miss pattern. The information is summarized in Table 4.7. $w_{i,s}$ and $d_{i,s}$ for all tasks are commonly 200 and 1, respectively.

With the profiled task information, we estimate the worst-case response time when tasks are executed on a quad-core system. Figure 4.27 shows the estimated WCRTs while varying the task mapping. No SR contention means that the WCRT is estimated without considering shared resource contention. Two mapping configurations Mapping1 and Mapping2 are described in Table 4.8. The priorities are assigned by the following rule: $PR_i > PR_j$ if $(i < j, M_i = M_j)$ or $(M_i = pe_k, M_j = pe_l, k < l)$. If a traditional analysis technique that does not consider shared resource contention is used, Mapping1 would be

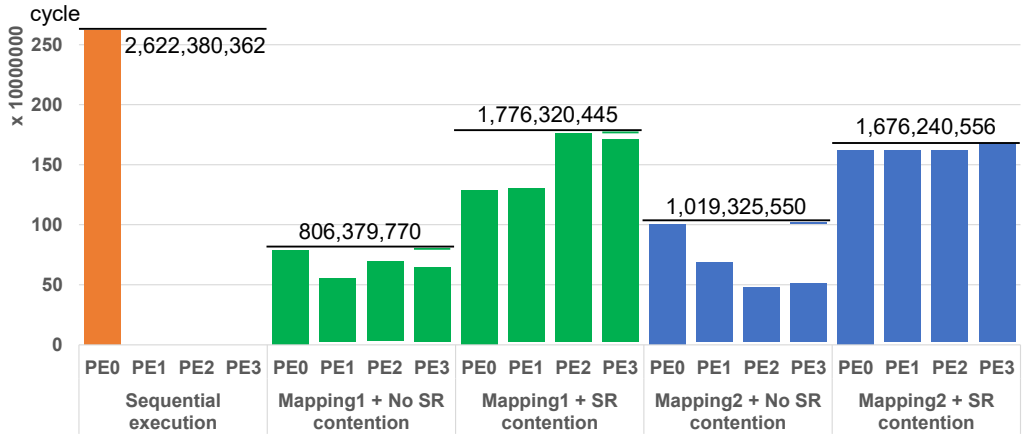


Figure 4.27: Estimated WCRTs of a viola-jones object detection example while task mapping is varied

regarded as better than Mapping2 as shown in Figure 4.27. On the other hand, the WCRT of Mapping2 considering shared resource contention is smaller than that of Mapping1. This implies the importance of considering shared resource contention when exploring the design space of a multi-core system. For both mapping configuration, the estimated WCRTs are almost half of the WCRT when the application is sequentially executed.

4.6 Case Study : End-to-end Latency Analysis of the Engine Management System Benchmark

In this section, we study how the proposed SR contention analysis can be successfully used for real-life systems. Recently, Bosch GmbH initiated a challenge for an industry-strength benchmark, a full-blown performance model of a modern engine management system [88]. The complexity of modern automotive systems has drastically grown due to the introduction of multi-core execution platform and a large quantity of innovative functionalities such as powertrain systems, chassis control applications, and Advanced Driver Assistance Systems (ADAS). In the engine management system, a global memory is shared among four symmetric cores connected by full-connectivity crossbar

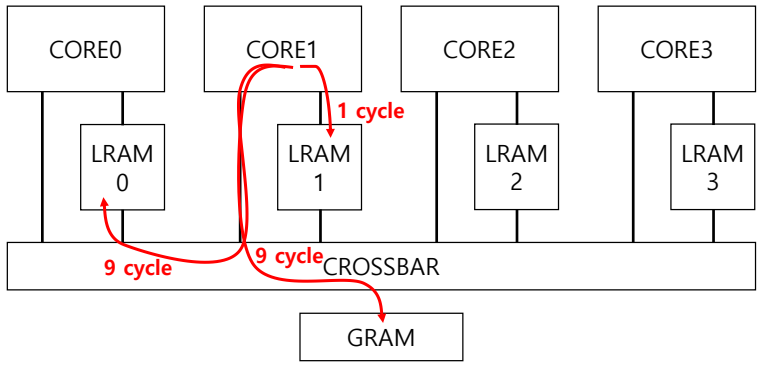


Figure 4.28: Microcontroller architecture in the engine management benchmark

network. In order to precisely estimate the end-to-end response times of safety-relevant automotive applications, the worst-case arbitration delay in the shared global memory needs to be analyzed.

To this end, we propose a timing analysis technique that computes the scheduling time bounds of each task considering the task dependency similarly to the scheduling time bound analysis (STBA). Then we estimate the upper bound of the end-to-end latencies of cause-effect chains in the engine management system benchmark by considering the schedule time bounds, where a cause-effect chain is an end-to-end process flow from a sensor to an actuator. To estimate the memory contention delay during the latency, we compute the worst-case resource demand from each processing element by the proposed shared resource contention analysis technique.

4.6.1 Engine Management System Benchmark

In the benchmark [88], the system contains a simplified microcontroller architecture with four symmetric cores as shown in Figure 4.28. Each core pe_i has its own local memory sr_i where $0 \leq i < 4$, and a crossbar network is used for the interconnection among cores and a global memory sr_4. The system-wide frequency is 200 MHz.

A task τ_i is a basic mapping unit onto a core, and task-to-core mapping is fixed and given. The core τ_i is mapped to is denoted by M_i. A task is invoked either periodically or

sporadically. I_P and I_S denote a set of periodic tasks and a set of sporadic tasks, respectively. The minimum and the maximum initiation interval for each task τ_i are denoted as T_i^l and T_i^u. Then $T_i^l = T_i^u$ if $\tau_i \in I_P$. All tasks are simultaneously initiated at the system activation time. The basic timing requirement for task τ_i is to finish execution before its deadline denoted by D_i and D_i is equal to T_i^l.

A distinct priority is assigned to each task by giving a unique index in the descending priority order; task τ_i has a higher priority than τ_j if $i < j$. A task τ_i is scheduled by either preemptive or cooperative fixed priority scheduling policy. S_P and S_C denote a set of preemptive tasks and a set of cooperative tasks, respectively. A task $\tau_i \in S_P$ can preempt lower priority tasks at any time, whereas a task $\tau_i \in S_C$ can preempt lower priority cooperative tasks at the boundary of runnable executions [89].

A task τ_i consists of a set of runnables $\{r_{i,j} \mid 1 \leq j \leq |\tau_i|\}$ where runnable $r_{i,j}$ is a unit of execution and $|\tau_i|$ means the number of runnables in the task. Runnables in a task are executed sequentially on the mapped core in the ascending index order. The lower and the upper bound of the execution time of $r_{i,j}$, denoted $c_{i,j}^l$ and $c_{i,j}^u$, are specified assuming that code is executed directly from core-exclusive flashes without contention. Note that memory access delay is not included in the execution times. The runnables are assumed to read all required data at the beginning of their execution and write back the results after execution is completed. We assume that when a runnable attempts to access a memory, no preemption is allowed until the resource request is processed.

A cause-effect chain CEC_i defines a chain of runnables that are connected by the read/write dependency with labels. Note that there are no cyclic dependencies between tasks within a cause-effect chain. Due to the potential different task periods, data may get lost (undersampling) or get duplicated (oversampling). An end-to-end (E2E) latency of a cause-effect chain is defined as the maximum time duration between the first input that may be undersampled and the first output generated from the corresponding or later input. This semantic is same as the *reaction time constraint* of the AUTOSAR [90]. Figure 4.29

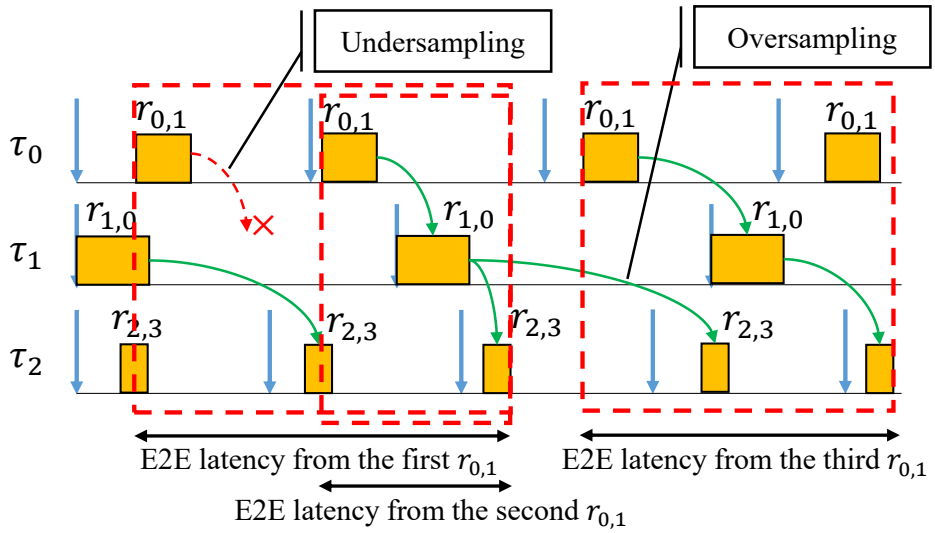


Figure 4.29: End-to-end latency of an example cause-effect chain

shows end-to-end latencies from three stimulus runnable instances in an example cause-effect chain $\{r_{0,1}, r_{1,0}, r_{2,3}\}$. Since we are concerned about reaction time, the second $r_{1,0}$ instance is regarded as the reaction of the first $r_{0,1}$ instance. The third $r_{2,3}$ instance is the first response to the second $r_{1,0}$ instance so that final reaction of the first $r_{0,1}$ instance is generated by the third $r_{2,3}$ instance.

Data is specified by a set of labels and all labels are assumed to be stored in the global memory. Read and write accesses have symmetric memory access times. When accessing the global memory, crossbar transfer takes 8 cycles and access to global memory takes 1 cycle. The accesses are assumed to be arbitrated according to the FIFO policy in the global memory. Later in subsection 4.6.4, we enable the mapping of labels in local memories and find the label mapping that optimizes the end-to-end latencies. The local memory access time is 1 cycle and FIFO arbitration is used as same as the global memory.

We aim to conservatively estimate the upper bound of the response time of each task τ_i, denoted as $\mathcal{R}_{\tau_i}$, and end-to-end latency of cause-effect chain CEC_i, denoted as $\mathcal{R}_{CEC_i}$,

Table 4.9: Terms and notations

Notation	Description
τ_i	a task with its index i
I_P	a set of periodic tasks
I_S	a set of sporadic tasks
M_i	the mapped processing element of a task τ_i
T_i^l	the minimum initiation interval of task τ_i
T_i^u	the maximum initiation interval of task τ_i
D_i	the deadline of task τ_i. $D_i = T_i^l$
S_P	a set of preemptive tasks
S_C	a set of cooperative tasks
$r_{i,j}$	j-th runnable in task τ_i
$\lvert\tau_i\rvert$	the number of runnables in task τ_i
$c_{i,j}^l$	the lower bound of execution time of $r_{i,j}$
$c_{i,j}^u$	the upper bound of execution time of $r_{i,j}$
CEC_i	a cause-effect chain of index i
$\mathcal{R}_{\tau_i}$	the worst-case response time of task τ_i
$\mathcal{R}_{CEC_i}$	the end-to-end latency of cause-effect chain CEC_i

as tightly as possible. Table 4.9 summarizes the notations declared above.

4.6.2 Schedule Time Bound Analysis

In this subsection, we derive two schedule time bounds $LB_c^s(i,j)$ and $UB_c^f(i,j)$ which are the lower and the upper bound of the latency between the release time of a runnable $r_{c,i}$ to the start time and the finish time of a runnable $r_{c,j}$, respectively, where $1 \leq i \leq j \leq |\tau_c|$.

Start time lower bound: For $LB_c^s(i,j)$ computation, we consider the minimum interference from the other runnables. Assuming that no task is blocked by a low priority task, $LB^s(r_{c,i}, r_{c,j})$ can be formulated as follows:

$$LB_c^s(i,j) = \sum_{k=i}^{j-1} c_{c,k}^l + \sum_{\tau_h \in hp(\tau_c)} \left\lceil \frac{\max(0, LB_c^s(i,j) - (T_h^u - C_h^l) + 1)}{T_h^u} \right\rceil \cdot C_h^l \tag{4.22}$$

where $hp(\tau_c) = \{\tau_h | M_h = M_c, c > h\}$, $C_h^l = \sum_{k=1}^{|\tau_h|} c_{h,k}^l$.

Finish time upper bound: For $UB_c^f(i,j)$ computation, we have to compute the maximum interference among tasks. We formulate $UB_c^f(i,j)$ for a preemptive task $\tau_c \in S_P$ as follows:

$$UB_c^f(i,j) = \sum_{k=i}^{j} c_{c,k}^u + \sum_{\tau_h \in hp(\tau_c)} \left\lceil \frac{UB_c^f(i,j)}{T_h^l} \right\rceil \cdot C_h^u \tag{4.23}$$

where $C_h^u = \sum_{k=1}^{|\tau_h|} c_{h,k}^u$. For a cooperative task τ_c, the release of τ_c can be blocked by at most one runnable execution of a lower priority task mapped on the same core. Higher priority cooperative tasks released after the start time of a runnable $r_{c,j}$ do not affect on the finish time. We need to formulate the upper bound of the latency between the release time of a runnable $r_{c,i}$ to the start time of a runnable $r_{c,j}$, denoted $UB_c^s(i,j)$ as follows:

$$UB_c^s(i,j) = B_c + \sum_{k=i}^{j-1} c_{c,k}^u + \sum_{\tau_h \in hp(\tau_c)} \left\lceil \frac{UB_c^s(i,j)+1}{T_h^l} \right\rceil \cdot C_h^u \tag{4.24}$$

The first term B_c indicates the maximum blocking from a lower priority task. $B_c = \max_{r_{l,k} \in \cup lp(\tau_c)} c_{l,k}^u$ where $lp(\tau_c) = \{\tau_l | M_l = M_c, c < l, \tau_l \in S_C\}$, only when $i = 1$ since any lower priority task cannot start after the first runnable starts. If $i > 1$, $B_c = 0$. Note that $UB_c^s(i,j)+1$ is used in the third term to include the higher priority tasks released between the finish of the $(j-1)$-th runnable and the start of the (j)-th runnable. Then $UB_c^f(i,j)$ can be estimated as follows:

$$UB_c^f(i,j) = B_c + \sum_{k=i}^{j} c_{c,k}^u + \sum_{\tau_h \in hp(\tau_c) \cap S_P} \left\lceil \frac{UB_c^f(i,j)}{T_h^l} \right\rceil \cdot C_h^u + \sum_{\tau_h \in hp(\tau_c) \cap S_C} \left\lceil \frac{UB_c^s(i,j)}{T_h^l} \right\rceil \cdot C_h^u \tag{4.25}$$

All requests of higher priority preemptive tasks within $UB_c^f(i,j)$ are accounted while the requests of higher priority cooperative tasks after $r_{c,j}$ starts are excluded.

Finally, we can compute the estimated end-to-end latency of a task τ_c as $\mathcal{R}_{\tau_c} =$

$UB_c^f(1,|\tau_i|)$.

During the schedule time bound analysis explained above, the worst-case arbitration delay in the global memory should be analyzed. Since memory accesses are arbitrated according to the FIFO policy and a core is assumed to be blocked during memory access, a naive way to estimate the worst-case access delay is to assume that every access experiences blocking of maximum queued accesses from all other cores (one access per each core). To find a tighter bound of memory access delay, however, we analyze the maximum number of memory accesses issued by tasks in each core within any time window of size Δt, using the SR contention analysis technique proposed in Section 4.3. We can derive the memory access pattern of a task by the event stream model since the read/write accesses occur only at the beginning and the completion of runnable execution. Then we estimate the worst-case arbitration delay as explained in Section 4.4, using the PE-level SR demand bound function $D^F_{T_{c,p},s}(\Delta t)$.

4.6.3 End-to-end Latency of a Cause-Effect Chain

We define two variables $BCST(r_{c,i}) = LB_c^s(1,i)$ and $WCFT(r_{c,i}) = UB_c^f(1,i)$ for brevity, which mean a lower bound of start time of $r_{c,i}$ and an upper bound of finish time of $r_{c,i}$, respectively.

Figure 4.30 shows three example cause-effect chains and the activation patterns of five tasks are summarized in Figure 4.30 (a). A cause-effect chain CEC_0 in Figure 4.30 (b) consists of four runnables in the same task τ_3. In this case, we have to analyze how many task instances are involved in the chain. If the (i+1)-th runnable of the chain has a smaller index than the i-th runnable, labels written by the i-th runnable will be read by the (i+1)-th runnable in the next task instance. Hence the number of the instances involved in the chain is computed by counting how many times runnable indices decrease in the task sequence. In Figure 4.30 (b), two task instances are involved in the chain since index decrease appears only once in the chain ($r_{3,272} \rightarrow r_{3,107}$). If one task instance covers the

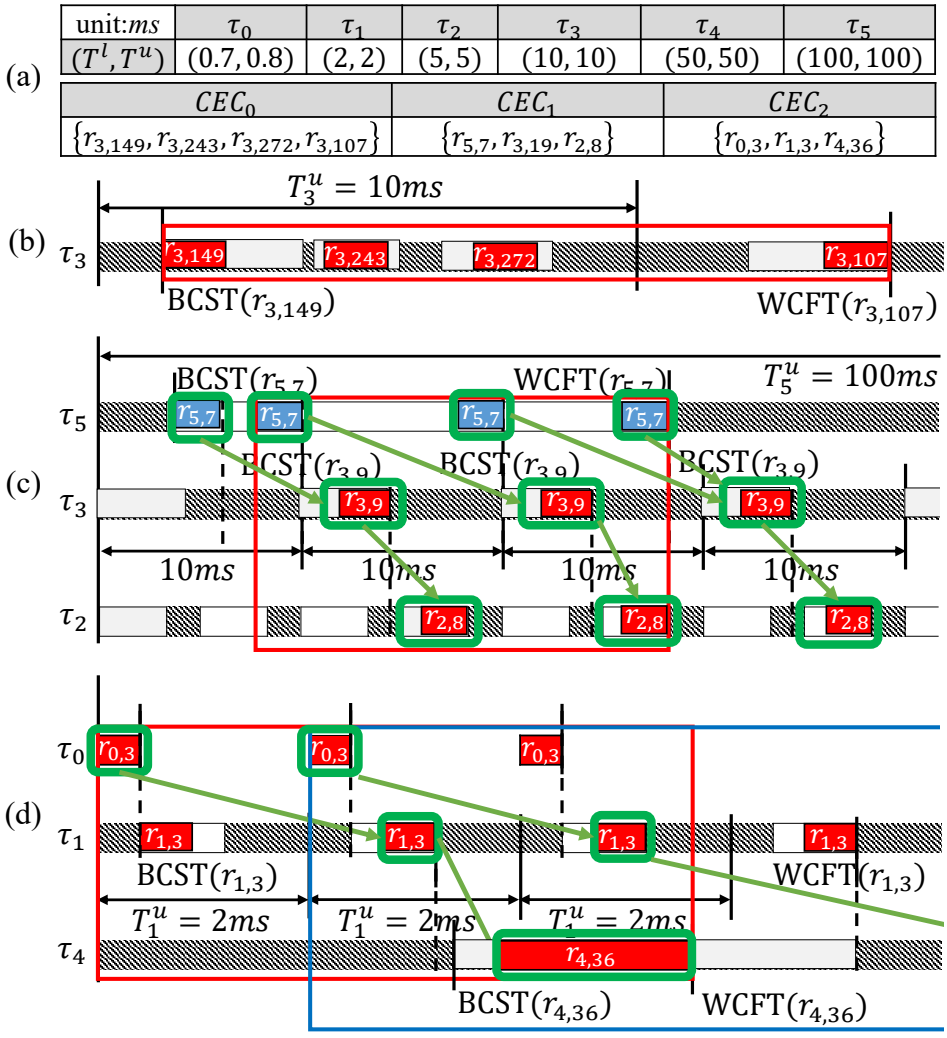


Figure 4.30: End-to-end latency computation of three example cause-effect chains CEC_0 (b), CEC_1 (c), and CEC_2 (d). A white box indicates the schedule time bound of a runnable while a red or a blue box is a runnable execution.

cause-effect chain, the end-to-end latency can be computed as $UB_c^f(b,e)$ where $r_{c,b}$ and $r_{c,e}$ are the first and the last τ_c runnables in the chain. Otherwise, the worst-case end-to-end latency becomes the distance from the BCST of the first runnable to the WCFT of the last runnable plus the task period multiplied by the count of index decreases in the chain, which gives $T_3^u + WCFT(r_{3,107})$ - $BCST(r_{3,149})$ for the example of Figure 4.30 (b).

A cause-effect chain CEC_1 in Figure 4.30 (c) consists of three runnables with differ-

ent activation patterns. In this case, we consider the schedule range of the first runnable $[BCST(r_{5,7}), WCFT(r_{5,7})]$ and examine all possible BCSTs of the second runnable $r_{3,19}$ that may appear after the first runnable. In the example of Figure 4.30 (c), there are three possible BCSTs of $r_{3,19}$. If we consider a pair of runnables only, the worst-case scenario is that the second runnable starts just before the first runnable finishes and the label written by the first runnable is read by the second runnable at the latest in the next task instance. Based on this observation we define a set of starting points of the first runnable as shown in blue color in the figure. The set includes the schedule of the first runnable whose finish time coincides with a possible BCST of the second runnable as well as the earliest and the latest schedule within the schedule bound.

For the subsequent pair of runnables, for instance the second and the third runnables in the example of Figure 4.30 (c), we need to consider the schedule time bound of the successor and the WCFT of the predecessor. If the WCFT of the predecessor lies in the schedule time bound of the successor, the label written by the predecessor should be read by the successor runnable at the latest in the next task instance. For each candidate starting point of the first runnable in the chain, the figure shows the longest cause-effect chain by green arrows where red and blue boxes mean the executions of runnables. Among all candidate starting points, we find one that gives the worst-case chain latency that is represented by a red bounding box in the figure, which corresponds to the second candidate starting point.

In this example, we consider a single runnable involved in each task. In case more than one runnable of the same task is included in the chain, we group them as a sub-chain. Then, a cause-effect chain consists of a sequence of sub-chains where each sub-chain consists of a set of runnables in the same task. If the worst-case latency of the sub-chain spans more than one task instance like the case of Figure 4.30 (b), we need to consider only one starting point for the sub-chain for the second case.

The third case shown in Figure 4.30 (d) is the case that the cause-effect chain starts

with a sporadic task: the first runnable in CEC_2 belongs to a sporadic task τ_0. Since the sporadic task may start anytime, we find the worst-case scenario in which the finish time of $r_{0,3}$ is aligned with the best-case start time of the first $r_{1,3}$ instance. Then the end-to-end latency from $r_{0,3}$ to $r_{1,3}$ is bounded by $UB_0^f(3,3) + T_1^u + WCFT(r_{1,3}) - BCST(r_{1,3})$. Note that we need to check only one starting point, which makes the finish time of the sub-chain be aligned with the best-case start time of next sub-chain, unlike the case of periodic tasks in Figure 4.30 (b). We repeat this computation for all task instances of the first periodic task in the chain within the hyper-period of tasks. In Figure 4.30 (d), τ_1 is the first periodic task. If we repeat computation for all τ_1 instances, the maximum latency occurs with the third τ_1 instance since labels written by the third $r_{1,3}$ instance is missed by the first $r_{4,36}$ instance.

Now we summarize the proposed technique for the estimation of the end-to-end latency of a cause-effect chain with Algorithm 3. At first, if the chain starts with sporadic tasks, we compute the end-to-end latency d_len of those sporadic sub-chains (lines 2-7). Then for the first periodic sub-chain, we examine all instances of the first peridic sub-chain within the hyperperiod of the chain. (lines 8-23). For each instance, we find all candidate starting points and compute the latency from the starting point to the end time of the chain (lines 9-21). If the chain starts with a sporadic task or a sub-chain that spans more than one task instance, we need to consider only one starting point which is the BCST of the runnable. Otherwise, we find all candidate starting points as Figure 4.30 (c). From each starting point, we find the end point of the chain (lines 13-19).

4.6.4 Label Mapping Decision

In this subsection, we determine a label-to-memory mapping to optimize the end-to-end latencies with a greedy algorithm. If a label is mapped to a local memory sr_i, the accesses to the label from core pe_i do not need the crossbar transfer delay (8 cycles) which is larger than the worst-case arbitration delay (4 cycles).

Algorithm 3 Algorithm to compute the end-to-end latency of a cause-effect chain

```
1: E2E ← 0, d_len ← 0
2: if the first sub-chain is in a sporadic task then
3:     d_len ←end-to-end latency of the first sub-chain
4:     while all sporadic sub-chains before the first periodic sub-chain do
5:         d_len ← d_len + (one period) + (WCFT of the last runnable) − (BCST of the
           first runnable)
6:     end while
7: end if
8: for all instances of the first periodic sub-chain within hyperperiod do
9:     find all candidate starting points of the first runnable
10:    for all candidate starting points do
11:        start ← (candidate starting point)
12:        end ←corresponding end point
13:        for all sub-chains after the first periodic sub-chain do
14:            if sub-chain is in a sporadic task then
15:                end ← end + (one period) + (WCFT of the last runnable) − (BCST of the
                   first runnable)
16:            else
17:                end ←minimum WCFT among runnable instances whose BCST is no
                   smaller than end
18:            end if
19:        end for
20:        E2E ← max(E2E, end − start)
21:    end for
22: end for
23: return E2E + d_len
```

Algorithm 4 presents a pseudo code of the proposed greedy algorithm to determine label-to-memory mapping. Initially labels are mapped to a global memory sr_4 (line 8). At first, we compute each fitness value of a mapping of $L[i]$ to sr_j, $F[i][j]$ (line 9). The fitness value is higher if $L[i]$ is more frequently accessed from pe_j. Then we determine a mapping of each label (lines 11-22). We select the most beneficial mapping according to the fitness values (line 12). Since we assume a limited local memory size, the label $L[l]$ can be mapped to sr_m in case sr_m has enough memory size (lines 13-18). The progress is repeated until there is no mapping that optimize the memory access delay (line 11).

Algorithm 4 Greedy algorithm to determine label-to-memory mapping

Input : a set of labels L, an array of label sizes S_L and local memory size s
Output : an array of label mapping M

$S_M \leftarrow$ one dimensional array of size 4
$F \leftarrow$ two dimensional array of size $|L| \times 4$
▷ $S_L[i]$ *is a label size of* $L[i]$
▷ $M[i]$ *is a memory a label* $L[i]$ *is mapped to*
▷ $S_M[i]$ *indicates available memory size of* sr_i
▷ $F[i][j]$ *is a fitness value of a mapping of* $L[i]$ *to* sr_j
for $0 \leq i < |L|, 0 \leq j < 4$ **do**
    $M[i] \leftarrow sr_4$, $S_M[j] \leftarrow s$
    $F[i][j] \leftarrow \sum_{M_k = pe_j} \frac{\#\text{accesses of } \tau_k \text{ to } L[i]}{T_k^l}$
end for
while $\exists_{i,j} F[i][j] > 0$ **do**
    find indices l and m that $F[l][m] = \max_{i,j} F[i][j]$
    if $S_M[m] \geq S_L[l]$ **then**
        $S_M[m] \leftarrow S_M[m] - S_L[l]$
        $M[l] = sr_m$
        for $0 \leq s < 4$ **do**
            $F[l][s] \leftarrow 0$
        end for
    else
        $F[l][m] \leftarrow 0$
    end if
end while

4.6.5 Benchmark Analysis Result

The estimated end-to-end latencies of all tasks and cause-effect chains from the proposed technique are summarized in Table 4.10. In the table, we use three configurations. In the second column, we estimate the end-to-end latency assuming there is no shared resource contention while the worst-case memory access delay is incorporated in the third and fourth columns. The end-to-end latencies in the third column are measured assuming all labels are in global memory, while labels are mapped into local memories using Algorithm 4 for those in the fourth column.

Even without SR access delay, 6 out of 21 tasks in the benchmark are unschedulable according to our analysis results since core utilizations are too high: utilizations are 97%,

Table 4.10: End-to-end latencies of tasks and cause-effect chains specified in the provided system model (unit: cycle)

Task		w/o SR cont.	w/ SR cont. w/o local mem	w/ SR w/ local mem
CORE0	ISR_10 (τ_0)	6,068	6,308	6,112
	ISR_5 (τ_1)	57,704	58,256	57,785
	ISR_6 (τ_2)	63,894	64,698	63,996
	ISR_4 (τ_3)	137,054	138,278	137,206
	ISR_8 (τ_4)	261,725	263,843	261,973
	ISR_7 (τ_5)	530,598	534,453	531,061
	ISR_11 (τ_6)	853,378	859,207	854,081
	ISR_9 (τ_7)	unschedulable	unschedulable	unschedulable
CORE1	Task_1ms (τ_{11})	152,870	156,345	153,588
	Angle_Sync (τ_{12})	unschedulable	unschedulable	unschedulable
CORE2	Task_2ms (τ_{13})	80,817	82,425	81,188
	Task_5ms (τ_{14})	267,180	270,252	267,900
	Task_20ms (τ_{16})	3,709,404	3,760,278	3,719,254
	Task_50ms (τ_{17})	7,973,611	unschedulable	7,992,287
	Task_100ms (τ_{18})	unschedulable	unschedulable	unschedulable
	Task_200ms (τ_{19})	unschedulable	unschedulable	unschedulable
	Task_1000ms (τ_{20})	unschedulable	unschedulable	unschedulable
CORE3	ISR_1 (τ_8)	7,011	7,383	7,066
	ISR_2 (τ_9)	10,560	11,160	10,635
	ISR_3 (τ_{10})	15,347	16,247	15,448
	Task_10ms (τ_{15})	unschedulable	unschedulable	unschedulable

Cause-effect chain	w/o SR cont.	w/ SR cont. w/o local mem	w/ SR w/ local mem
EffectChain_1	unschedulable	unschedulable	unschedulable
EffectChain_2	unschedulable	unschedulable	unschedulable
EffectChain_3	17,817,190	unschedulable	17,835,552

133.5%, 106.8%, and 117.9% for each core. There is even a task that has the worst-case execution time larger than its deadline (Task_10ms). End-to-end latencies of cause-effect chains cannot be analyzed due to the runnables in unschedulable tasks. We claim that the worst-case execution time should be decreased to make the system schedulable.

Results show that the portion of the memory access delay in the end-to-end latency is not significant. Task_50ms becomes unschedulable when memory access delay is not ignored. Because of the memory access delay, its end-to-end latency becomes over

Table 4.11: Scaled worst-case execution times for schedulable system and end-to-end latencies (unit: cycle)

Task		WCET	E2E Latency	Deadline
CORE0 (96%)	ISR_10 (τ_0)	5,825	5,867	140,000
	ISR_5 (τ_1)	49,570	55,472	180,000
	ISR_6 (τ_2)	5,942	61,434	220,000
	ISR_4 (τ_3)	70,233	131,706	300,000
	ISR_8 (τ_4)	58,345	251,485	340,000
	ISR_7 (τ_5)	62,375	509,791	980,000
	ISR_11 (τ_6)	58,729	814,042	1,000,000
	ISR_9 (τ_7)	71,133	896,985	1,200,000
CORE1 (71%)	Task_1ms (τ_{11})	108,537	109,164	200,000
	Angle_Sync (τ_{12})	540,360	1,197,321	1,332,000
CORE2 (93%)	Task_2ms (τ_{13})	75,159	75,511	400,000
	Task_5ms (τ_{14})	173,317	249,170	1,000,000
	Task_20ms (τ_{16})	1,947,129	3,383,696	4,000,000
	Task_50ms (τ_{17})	573,714	7,358,075	10,000,000
	Task_100ms (τ_{18})	1,751,743	19,908,947	20,000,000
	Task_200ms (τ_{19})	25,758	19,933,318	40,000,000
	Task_1000ms (τ_{20})	25,511	19,958,468	200,000,000
CORE3 (83%)	ISR_1 (τ_8)	5,819	5,869	1,900,000
	ISR_2 (τ_9)	2,945	8,834	1,900,000
	ISR_3 (τ_{10})	3,973	12,832	1,900,000
	Task_10ms (τ_{15})	1,944,313	1,986,787	2,000,000

Cause-effect chain	E2E Latency	Deadline
EffectChain_1	2,269,514	4,000,000
EffectChain_2	2,628,493	22,400,000
EffectChain_3	13,888,054	10,540,000

8,000,000 and one more preemption of Task_20ms whose worst-case execution time is 2,093,688 occurs, making the latency larger than the deadline. Almost all read/write accesses go to local memory after label-to-memory mapping is done so that the memory access delay decreases accordingly. Task_50ms is barely schedulable after label-to-memory mapping.

We conducted additional experiment to find maximum execution times of tasks satisfying all task deadlines. For each core, we scale down all worst-case execution times of mapped tasks by the same percentage and find the maximum percentages that make

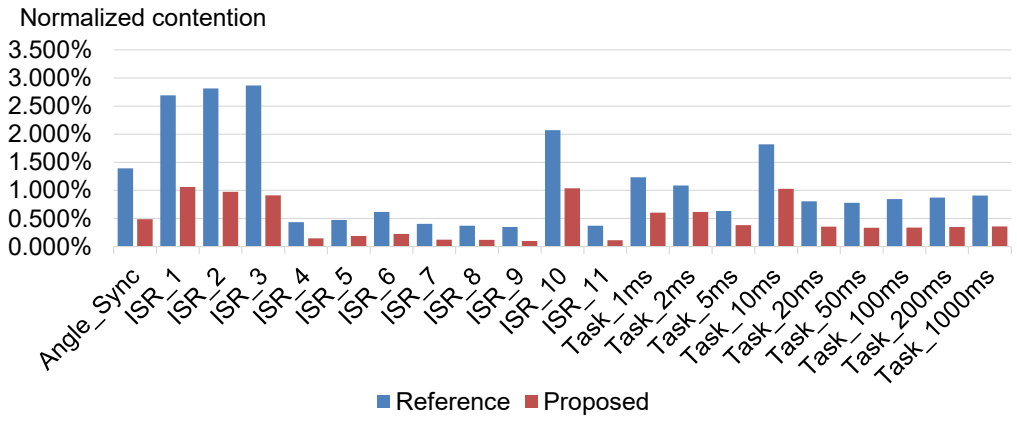


Figure 4.31: Comparison result with the reference technique.

all estimated end-to-end latencies of tasks below deadlines. Table 4.11 summarizes the scaled worst-case execution times and the end-to-end latencies considering memory access delay after label-to-memory mapping. Note that the percentage decrease for each core is proportional to the utilization of the core. Now all cause-effect chains have finite end-to-end latencies, but EffectChain_3 has the end-to-end latency larger than the deadline.

Finally, we measure the amount of memory access delay estimated by the proposed technique and the reference technique [10]. Figure 4.31 shows the comparison result. In figure, the y-axis shows the percentage of the memory access delay within the estimated worst-case response time. As shown in the figure, the reference technique shows 2.45 times larger access delay estimation than the proposed technique, although the memory access delay is not significant.

Chapter 5

Fault-Tolerant Mixed-Criticality Systems

5.1 Problem Definition

We present a formal description of the system model we assume in this chapter, followed by the formulation of the optimization problem.

5.1.1 Application

Applications are represented as a set of directed acyclic task graphs $\mathcal{A}$ and a task graph $\mathcal{G} \in \mathcal{A}$ is characterized with a set of vertices $\mathcal{V}_\mathcal{G}$ and a set of directed edges $\mathcal{E}_\mathcal{G}$, i.e., $\mathcal{G} := \langle \mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G} \rangle$. $\mathcal{V}_\mathcal{G}$ represents a set of all tasks in $\mathcal{G}$ and $\mathcal{E}_\mathcal{G}$ denotes a set of execution dependencies from a task to another in $\mathcal{G}$. That is, if there is an execution dependency from τ_1 to τ_2 in $\mathcal{G}$, $\{\tau_1, \tau_2\} \subset \mathcal{V}_\mathcal{G}$ and $\langle \tau_1, \tau_2 \rangle \in \mathcal{E}_\mathcal{G}$. With an execution dependency $\langle \tau_s, \tau_d \rangle$, τ_d can only be executed after the execution of τ_s. If a task is associated with more than one incoming edges, it is released after all the predecessor tasks complete their executions. An application $\mathcal{G}$ can be initiated either periodically or sporadically. In periodic activation, the application is assumed to be released every $T_\mathcal{G}$ time units. For sporadic activation, it can be interpreted that $T_\mathcal{G}$ denotes the minimum initiation time interval between two consecutive activations. Task graph $\mathcal{G}$ is given a relative deadline $D_\mathcal{G}$, by which each instance of released tasks should be completed. For simplicity, the task graph that τ_i belongs to is denoted by $\mathcal{G}_{\tau_i}$.

5.1.2 Reliability and Task Dropping

We distinguish between "droppable" and "non-droppable" applications. Even in case of fault occurrences non-droppable applications must always be executed completely, whereas droppable tasks might be dropped by the scheduler when necessary. To specify this distinction in the model, non-droppable applications have a reliability constraint $rel_{\mathcal{G}} \in (0,1]$ which denotes the maximum tolerable probability of fault occurrence. The lower reliability constraint $rel_{\mathcal{G}}$ means higher requirement on the fault tolerance. Each droppable task graph $\mathcal{G}$ is associated with the relative importance of the service $sev_{\mathcal{G}}$, while setting $rel_{\mathcal{G}}$ to 1. $sev_{\mathcal{G}}$ is a non-negative, non-zero real value ($sev_{\mathcal{G}} \in \mathbb{R}, sev_{\mathcal{G}} > 0$). This is to relatively quantify importances of multiple applications in a system. Note that $sev_{\mathcal{G}}$ is a part of specification, i.e., should be set by the system designer. If we decide not to drop a droppable application $\mathcal{G}$ in a faulty status, it contributes as much as $sev_{\mathcal{G}}$ to the system in terms of quality of service (QoS). For non-droppable task graphs, $sev_{\mathcal{G}}$ values are just set to 0.

For each droppable task graph $\mathcal{G}$, a dropping factor $drp_{\mathcal{G}} (0 \leq drp_{\mathcal{G}} \leq 1)$ is to be decided. If task τ in a droppable task graph $\mathcal{G}$ exceeds $drp_{\mathcal{G}}$ times of its WCET, it is dropped by scheduler. It can be seen as partial task dropping as the WCET is reduced by dropping factor $drp_{\mathcal{G}}$ instead of being zero. In the case that $drp_{\mathcal{G}}$ equals zero, $\mathcal{G}$ will be completely dropped in case of faults. For non-droppable tasks, $drp_{\mathcal{G}}$ values are fixed to 1. When partial dropping is applied by fixing the dropping factor $drp_{\mathcal{G}}$, the QoS of the graph $\mathcal{G}$ should be properly scaled as well using quality-of-service function $qos_{\mathcal{G}}(x) (0 \leq qos_{\mathcal{G}}(x) \leq 1)$. QoS function is to quantify the QoS degradation from partial dropping, and is also given by the system designer as a part of the specification. For instance, if application $\mathcal{G}$ is decided to be partially dropped by 50% ($drp_{\mathcal{G}} = 0.5$), it contributes as much as $qos_{\mathcal{G}}(0.5) \cdot sev_{\mathcal{G}}$ to the system in terms of QoS. Then the quality of service of a system is defined as $\frac{\sum_{\mathcal{G} \in \mathcal{A}} (qos_{\mathcal{G}}(drp_{\mathcal{G}}) \cdot sev_{\mathcal{G}})}{\sum_{\mathcal{G} \in \mathcal{A}} sev_{\mathcal{G}}}$. It is worthwhile to mention that our method is not specific to a certain form of time-QoS model. The QoS only needs to

be a monotonically increasing function of time. We believe this is a valid model in many applications. In an any-time algorithm, for instance, it can return a valid result even if it is interrupted before its completion [27]. Further, the quality of result is enhanced if it spends more execution time. The quality of vision-based object tracking algorithm, as another example, can be enhanced if enlarged search space is used at the cost of longer execution time [91]. In this case, as a finite set of image resolutions are used, the QoS forms a staircase function of the execution time.

Let us take an example of a vehicle system with two low-criticality applications: media player and car navigation. One may set $sev_{\mathcal{G}}$ values to 100 and 10 for car navigation and media player, respectively, as car navigation is considered to be 10 times more important than media player. Further, let us assume that the applications are anytime algorithms and $qos_{\mathcal{G}}(x)$ is given as $1-e^{(-3x)}$ and x^3 for car navigation and media player, respectively. Then, the overall QoS of the system is quantified as $\frac{100\cdot(1-e^{(-3\cdot drp_n)})+10\cdot {drp_m}^3}{100+10}$, where drp_n and drp_m denote the dropping factor for car navigation and media player, respectively.

5.1.3 Architecture

A multi-core system, which we target in this chapter, consists of a set of processing elements (PEs) $\mathcal{PE}$, whose elements are connected through an on-chip communication fabric such as a shared bus, crossbar switch, or a network-on-chip. In this work, we assume that faults in the communication links are transparent at system level, as typically they are protected by low-level error-resilient techniques such as error-correction code or protocol-level retransmission. Many other system-level reliability researches also assume such error-free communication architectures [15, 77, 18, 14, 21, 22]. Each PE $pe \in \mathcal{PE}$ is associated with a constant fault rate per time unit λ_{pe} and we assume that faults are independent of each other. Although the fault-triggering events such as radiation particle or data noise might cause errors in a burty fashion, the time unit by which the fault

probability is defined (e.g. millisecond) is long so that short-term bursty errors are considered as a single fault event that occurs in a time unit in our analysis model. In a single time unit, the system may detect the fault occurrence and do something to handle the effect of the previous source event. Further, at task scheduling, the granularity of time scale gets even coarser, several tens or hundreds milliseconds, making inter-dependency between the events be likely to disappear. Thus, we believe it is reasonable to assume independence between faults. Power behavior of the PE pe is characterized by its static power $stat_{pe}$ and dynamic power dyn_{pe}. Static power is always consumed whenever a PE is chosen to be used regardless of the schedule. On the other hand, dynamic power is closely related to the schedule as it is only paid when a task is executed.

5.1.4 Mapping/Scheduling

A task is a basic mapping unit of an application to PEs and we take a partitioned scheduling policy. In other words, tasks do not move from a PE to another at runtime. When Ω is defined to be a set of all constituent tasks in $\mathcal{A}$, i.e., $\Omega := \bigcup_{\mathcal{G} \in \mathcal{A}} \mathcal{V}_{\mathcal{G}}$, a mapping decision can be seen as determining a function $map : \Omega \rightarrow \mathcal{PE}$. When task τ_i is mapped on pe, for instance, $M_i = pe$. While the proposed technique in this chapter does not rely on any specific scheduling policy, we assume that all PEs schedule the assigned tasks under fixed-priority preemptive scheduling policy, following the assumption in the baseline WCRT analysis technique. The proposed technique may be extended to other scheduling policy if there exists a WCRT analysis technique that supports both directed acyclic task graph and the scheduling policy, which remains as a future work. In the fixed-priority scheduling, each task τ_i is assigned a unique priority PR_i by the system designer. A task τ_i has variable execution time for a single invocation, whose range is represented by a tuple $[C^l, C^u]$ indicating the best-case execution time (BCET) and the worst-case execution time (WCET) on the mapped PE, respectively. These bounds can be obtained by existing WCET analysis techniques such as [92].

5.1.5 Worst-Case Response Time

Once the mapping decision is made, we can analyze the WCRTs of the task graphs. WCRT of a task graph $\mathcal{G}$ is defined to be the time difference between the latest finish time and the earliest release time among tasks and denoted by $\mathcal{R}_{\mathcal{G}}$.

5.1.6 Hardening

A task belonging to a non-droppable application can be hardened by one of the following hardening techniques:

- **Re-execution:** The re-execution scheme assumes that a fault is locally detected at the end of the task execution. Such detection is realized by means of special hardware assistances [15] or by compile-time software support [93]. The fault detection overhead o^d should be paid at each execution of τ, in addition to the re-execution overhead itself. If a fault is detected, before re-executing the same instance, all the stateful variables are rolled-back to the lastly stored state by paying the roll-back overhead o^r. If task τ is to be re-executed at k times, the upper bound of its execution time should be revised to $(C^u + o^d) \times (k+1) + k \cdot o^r$.

- **Active replication:** Active replication refers to a hardening technique that makes multiple replicas of a task, mapped on different PEs. The degree of replication is usually an odd number larger than two to enable majority voting. Contrary to the re-execution scheme, replication requires the task graph topology to be modified, but the execution time bound $[C^l, C^u]$ is preserved for each replica. The replicated tasks are connected to a single task, whose duty is to perform detection or majority voting. The execution time variation of the voting task is assumed to be $[0, o^v]$.

- **Passive replication:** In passive replication, on the contrary to the active replication scheme, not all the cloned tasks are proactively executed, but only on the request of voters. To be more concrete, when the voter detects a tie in the results from

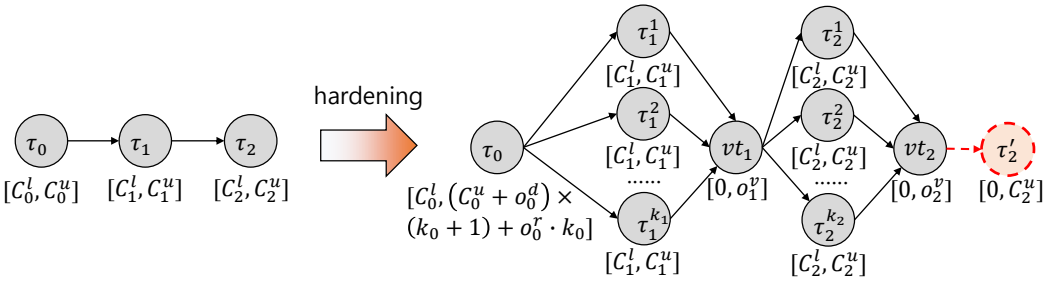


Figure 5.1: A hardening example of a simple task graph.

even number of active replicas, one passively replicated instance is activated. It saves resources compared with the active activation scheme, but with potentially increased WCRT.

A hardening example is depicted in Figure 5.1, where three tasks in an example task graph are hardened. Task τ_0 is chosen to be re-executed and the WCET, in turn, becomes $(C_0^u + o_0^d) \times (k_0 + 1) + o_0^r \cdot k_0$ where k_0 indicates the number of re-executions of τ_0. Task τ_1 is hardened by active replication with k_1 replicas. Finally, τ_2 choose the passive replication scheme with k_2 active replicas. The lower bound of the passive replica τ_2''s execution time is zero since it is only initiated when voter task vt_2 detects a tie. It is worthwhile to mention that a hardening decision may require modifications of the original task graph. For the ease of presentation, we simply denote the modified task graph set, task graph, and task set as $\mathcal{A}^m$, $\mathcal{G}^m$, and Ω^m, respectively in the rest of this chapter.

Table 5.1 summarizes all the notations declared above.

5.1.7 Problem Formulation

Using the notations defined above, We formulate the optimization problem of fault-tolerant mixed-criticality multi-core systems as follows:

- **Input:** Given a set of applications $\mathcal{A}$ and a multi-core architecture $\mathcal{PE}$,
- **Output:** Determine the hardening technique that transforms $\mathcal{A}$ into $\mathcal{A}^m$, the map-

Table 5.1: Terms and notations

	Description		Description
$\mathcal{A}$	a set of task graphs	M_i	the processing element τ_i is mapped on
$\mathcal{G}$	a task graph	PR_i	a priority of τ_i
τ	a task	C_i^l	the best-case execution time of τ_i
Ω	a set of all tasks in $\mathcal{A}$	C_i^u	the worst-case execution time of τ_i
$\mathcal{A}^m$	the modified task graph set	o_i^d	the fault detection overhead of τ_i
$\mathcal{G}^m$	the modified task graph	o_i^r	the roll-back overhead of τ_i
Ω^m	a set of all tasks in $\mathcal{A}^m$	o_i^v	the voting overhead of τ_i
$\mathcal{PE}$	a set of processing elements	λ_{pe}	a constant fault rate of pe
pe	a processing element	$stat_{pe}$	leakage power consumption of pe
$\mathcal{G}_\tau$	the task graph τ belongs to	dyn_{pe}	dynamic power consumption of pe
$\mathcal{V}_\mathcal{G}$	a set of tasks in $\mathcal{G}$	$rel_\mathcal{G}$	a reliability constraint of $\mathcal{G}$
$\mathcal{E}_\mathcal{G}$	a set of edges in $\mathcal{G}$	$sev_\mathcal{G}$	importance of the service of $\mathcal{G}$
$T_\mathcal{G}$	an activation period of $\mathcal{G}$	$drp_\mathcal{G}$	a dropping factor of $\mathcal{G}$
$D_\mathcal{G}$	a relative deadline of $\mathcal{G}$	$qos_\mathcal{G}$	a quality-of-service function of $\mathcal{G}$
$\mathcal{R}_\mathcal{G}$	the worst-case response time of $\mathcal{G}$		

ping decision M_i for all $\tau_i \in \Omega^m$ onto $\mathcal{PE}$, the scheduling decision PR_i for all $\tau_i \in \Omega^m$, and dropping factors $drp_\mathcal{G}$ for all $\mathcal{G} \in \mathcal{A}$.

- **Constraints:** The output results are only valid as long as $\mathcal{R}_\mathcal{G} \leq D_\mathcal{G}$ and the failure probability is less than or equal to $rel_\mathcal{G}$ for all $\mathcal{G} \in \mathcal{A}$ as formulated in equation (5.10).

- **Objectives:** In this work, we optimize three design criteria: the quality-of-service, estimated power consumption in the normal states, and estimated power consumption in the faulty states. Power in the normal states is a good indicator of average behavior while that of the faulty states is to reduce the peak power. Peak power is a non-negligible design concern in energy harvesting devices [94] or guaranteeing the worst-case temperature [95]. Remind that the QoS metric is defined as $\frac{\sum_{\mathcal{G}\in\mathcal{A}}(qos_\mathcal{G}(drp_\mathcal{G})\cdot sev_\mathcal{G})}{\sum_{\mathcal{G}\in\mathcal{A}} sev_\mathcal{G}}$. We estimate power consumption in the normal states by $\sum_{pe\in\mathcal{PE}}(stat_{pe} + dyn_{pe}\cdot u_{pe})$ where $u_{pe} = \sum_{\{\tau_i|\tau_i\in\Omega^m, M_i=pe\}} \frac{C_i^u}{T_{\mathcal{G}_{\tau_i}}}$, assuming passive replicas and re-executions not invoked. In contrast, in the faulty states, it is esti-

mated by considering dropping factor in $u_{pe} = \sum_{\{\tau_i | \tau_i \in \Omega^m, M_i = pe\}} \frac{drp_{\mathcal{G}\tau_i} \cdot C_i^u}{T_{\mathcal{G}\tau_i}}$, and in this case all passive replicas and re-executions are assumed to be invoked. Note that active replicas always contribute the power consumption both in the normal and faulty states.

5.2 Review of the Existing Fault Probability Analysis

In order to certify the fulfillment of the reliability constraint, we need to quantify the fault probability of a mapping/hardening decision. In this section, we introduce how the existing fault probability analysis techniques that consider re-execution [20, 21] or replication [21, 15] can be extended to quantify the reliability considering multiple faults. We take Poisson distribution for modeling the probability of multiple fault events as it is popularly used for the fault modeling in embedded systems [78, 79]. It is worthwhile to mention that the proposed technique is no specific to Poisson distribution. What we only assume is that faults are *independent* events. That is, a fault event does not affect the occurrence probability of another. Once this assumption holds true, other probability distributions can be applied to the proposed technique without loss of generality.

Derived from the basic form of Poisson distribution, the probability that processor pe experiences n faults within t time units can be calculated as follows:

$$P(pe,t,n) = \frac{(t \cdot \lambda_{pe})^n \cdot e^{-t \cdot \lambda_{pe}}}{n!}. \tag{5.1}$$

In what follows, how each hardening technique lowers the fault probability of individual tasks, based on the above probability equation. For simplicity, we denote the fault probability of τ_i within t time units as $F(\tau_i,t) = 1 - P(M_i,t,0)$.

Re-execution: If the re-execution technique is applied to task τ_t by k_t times, it only fails if all the $k_t + 1$ execution instances experience at least one fault each. Then, the fault

probability of τ_t becomes

$$F(\tau_t, C_t^u + o_t^d + o_t^r)^{k_t+1} \tag{5.2}$$

Active replication: If τ_t is actively replicated by k_t times, it fails if

1. more than half, i.e., $\lceil \frac{k_t}{2} \rceil$, replicas fail, or

2. less than half replicas fail but faults occur during the voter execution.

k_t is an odd number to enable majority voting. We define $H(\tau_t, n)$ as a set of all cases that n replicas of τ_t fail, and each element of $H(\tau_t, n)$, h_i, denotes sets of faulty replicas. Let us take an example where task τ_t is actively replicated three times $(\tau_t^1, \tau_t^2, \tau_t^3)$. Then, we have $H(\tau_t, 2) = \{\{\tau_t^1, \tau_t^2\}, \{\tau_t^1, \tau_t^3\}, \{\tau_t^2, \tau_t^3\}\}$. The probability that n replicas of τ_t fail is

$$FR(\tau_t, n) = \sum_{h_i \in H(\tau_t, n)} \left(\prod_{\tau_t^j \in h_i} \left(F(\tau_t^j, C_t^u) \right) \prod_{\tau_t^j \notin h_i} \left(1 - F(\tau_t^j, C_j^u) \right) \right) \tag{5.3}$$

where τ_t^j is j-th replica of τ_t. With $FR(\tau_t, n)$ we can derive the fault probability of τ_t. The probability of case 1 is

$$PA_{case_1} = \sum_{n=\lceil \frac{k_t}{2} \rceil}^{k_t} FR(\tau_t, n) \tag{5.4}$$

The probability of case 2 is

$$PA_{case_2} = (1 - PA_{case_1}) \cdot F(vt_t, o_t^v) \tag{5.5}$$

where vt_t is the voter task appended to the replicas of τ_t. Therefore, the fault probability of τ_t, in active replication, is

$$PA_{case_1} + PA_{case_2}. \tag{5.6}$$

Passive replication: Task τ_t with the passive replication policy of k_t active replicas fails if

1. more than $\frac{k_t}{2}$ replicas fail, or

2. $\frac{k_t}{2}$ active replicas fail, no faults occur during voting, but the passive replica fails, or

3. less than or equal to $\frac{k_t}{2}$ active replicas fail, but faults occur during the voter execution.

In case of passive replication, k_t is an even number. The probability of case 1, PP_{case_1} is identical to PA_{case_1} and the probability of case 2 is

$$PP_{case_2} = FR(\tau_t, \frac{k_t}{2}) \times (1 - F(vt_t, o_t^v)) \times F(\tau_t', C_t^u) \tag{5.7}$$

where τ_t' denotes a passive replica of τ_t. At last, the probability of case 3 is

$$PP_{case_3} = \left(\sum_{n=0}^{\frac{k_t}{2}} FR(\tau_t, n) \right) \times F(vt_t, o_t^v) \tag{5.8}$$

To sum up, the fault probability of τ_t in passive replication is

$$PP_{case_1} + PP_{case_2} + PP_{case_3}. \tag{5.9}$$

The success probability of a task graph is the product of success possibilities of all the tasks in the graph since the application successes if and only if all the tasks successfully complete their executions. Therefore, the reliability constraint of an application $\mathcal{G}$ can be formulated as follows:

$$1 - \prod_{\tau \in \mathcal{V}_\mathcal{G}} (1 - R(\tau)) \leq rel_\mathcal{G} \tag{5.10}$$

where $R(\tau)$ is the fault probability of task τ.

During the design space exploration of hardening decision, the degrees of replication or re-execution are selected and tested whether the selected hardening decision satisfies equation (5.10) or not. Once the decision is confirmed to satisfy the reliability

constraint, the WCRT is analyzed with that decision. However, this probability quantification results in an overestimation of WCRT as stated with an example in Figure 2.5. In subsection 5.3.3, we will show how such overestimations can be mitigated by quantifying the failure probability in an alternative way.

5.3 Proposed Optimization Technique

5.3.1 Overall Optimization Framework

Thus far, the maximum degree of re-executions has been statically fixed at design time [22, 20]. As discussed with the example of Figure 2.5, this reliability quantification may result in the overestimation of WCRT analysis. Alternatively, in the proposed technique, we first take an entire schedule, as a whole, to figure out how many faults could stochastically occur during a single run. Then, we consider in which distribution of faults the worst-case scenario arises.

Let us take a task graph example illustrated in Figure 5.2 (a), mapped on two different PEs. If we evaluate the reliability of the tasks individually using equation (5.10), the worst case would be the case that *each* of the tasks mapped on pe_0 (pe_1) experiences one fault (two faults), triggering one instance (two instances) of re-executions, as described in Figure 5.2(b) and (c). From the perspective of the entire schedule, however, this is an overestimated result as the probability that it has two faults in pe_0 and six faults in pe_1 over a single schedule run would be extremely low ($P(pe_0,46,2) \cdot P(pe_1,46,6) = 1.16 \times 10^{-33}$), and certainly smaller than the given reliability constraint. Actually considering two faults per PE over a single schedule is enough for reliability constraint. How to find the number of faults per PE will be explained later in subsection 5.3.3.1. Now, the problem becomes how to figure out when each of two faults occurs in the worst-case scenario. Note that it is virtually impossible to examine all possible fault distributions to determine the WCRT due to many uncertain behaviors. In the proposed technique, we aim

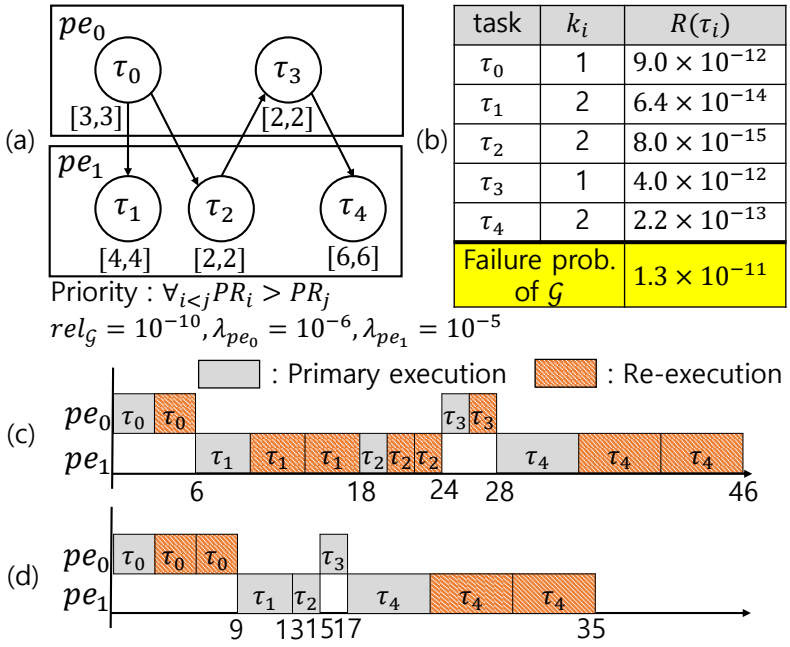


Figure 5.2: A WCRT Analysis example of a simple task graph: (a) an example graph, (b) probability analysis result, (c) exaggerated WCRT when each task assumes worst-case fault experience independently, and (d) actual WCRT when faults occur in the largest execution in each PE.

to find the upper bound of the WCRT with respect to the given number of re-executions.

Figure 5.3 shows the overall flow of the proposed optimization technique, based on the genetic algorithm (GA). As shown in the leftmost flowchart in Figure 5.3, it follows the typical procedure of GA optimizations, i.e., repetition of generation-evaluation-replacement. This will be elaborated in subection 5.3.2. In the GA-based design space exploration (DSE) framework, we aim at optimizing the system as formulated in subection 5.1.7. Each individual of the population maintained in the GA engine needs to be analyzed in terms of WCRT and reliability, as depicted in the evaluation part in Figure 5.3. Note that each solution, to be analyzed, is fixed in mapping/scheduling and hardening decisions and analyzed in the analysis part of the framework. The rightmost flowchart in Figure 5.3 illustrates the overall procedure of the proposed analysis technique, which will be presented in detail in subsection 5.3.3 and 5.3.4.

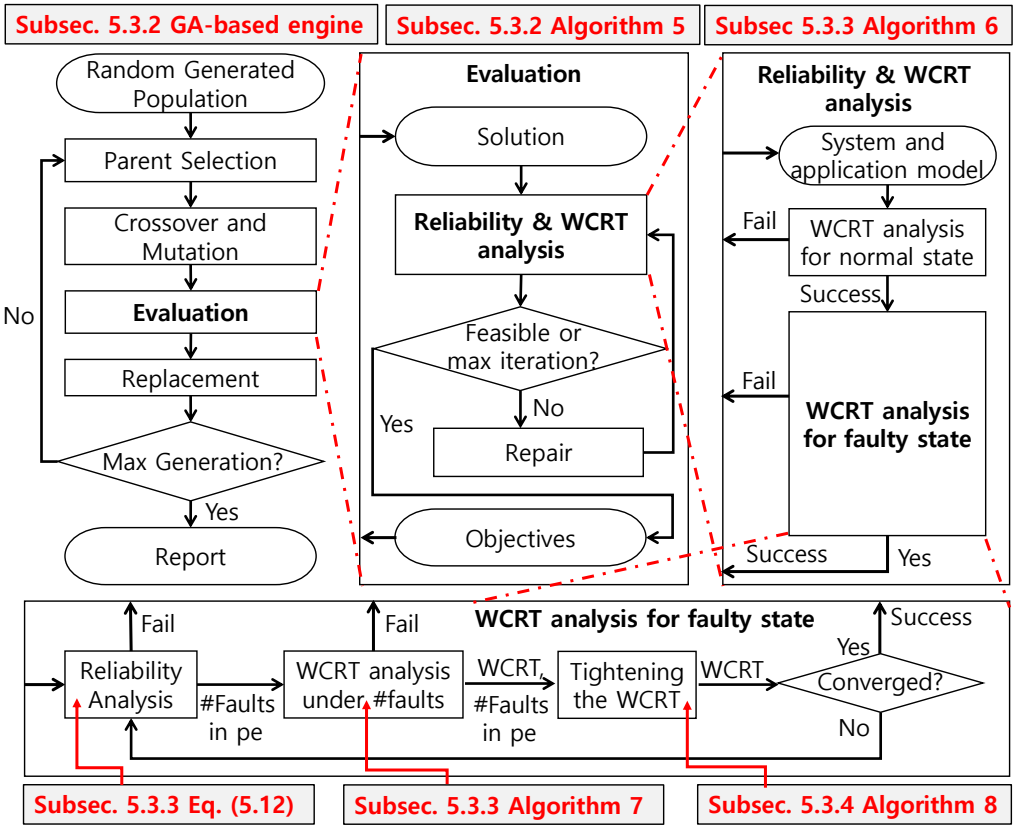
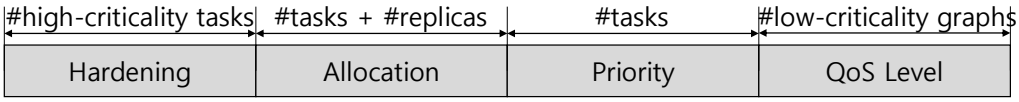


Figure 5.3: The overall flow of the proposed optimization technique.

5.3.2 Part I: GA based DSE Engine

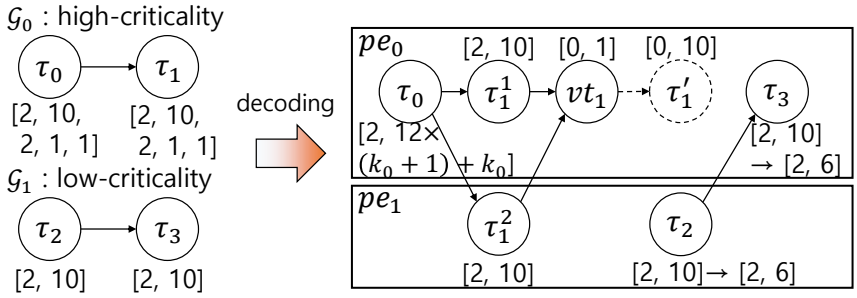
The goal of our design is to maximize the quality-of-service in the normal and faulty states, while minimizing the estimated power consumption, as declared as the objectives in Section 5.1.7.

Genotype structure: Initially, individuals of the population are generated randomly. Figure 5.4 (a) illustrates the gene structure of the individual designed to solve the proposed problem. First, the hardening policies of all high criticality tasks are determined in the 'Hardening' part. The 'Allocation' part determines the mapping of all tasks and their replicas. As a scheduling decision, priority of each task is decided in the 'Priority' part. We encode the priority of each task as a unique integer value for deterministic priority



(a) Design of genotype

Hardening			Allocation							Priority				QoS Level
τ_0	τ_1		τ_0	τ_1^1	τ_1^2	vt_1	τ_1'	τ_2	τ_3	τ_0	τ_1	τ_2	τ_3	$\mathcal{G}_1$
0	2	2	0	0	1	0	0	1	0	0	2	1	3	3



(b) An example genotype and its decoding

Figure 5.4: Design of the proposed genotype structure and a simple decoding example.

order. Finally, the 'QoS Level' part is for the dropping decision of each low criticality application. An integer value between 0 and 5 is associated with each task graph in this part, where 0 and 5 denote complete and no dropping respectively. If a task graph is assigned 2 here, for example, each task belonging to this graph is executed by only 40% in faulty state.

Figure 5.4(b) demonstrates a solution example represented in the proposed chromosome structure and how it is decoded. Five values annotated below each task denote BCET, WCET, detection overhead, roll-back overhead, and voting overhead, respectively. For instance, $C_0^l = 2$, $C_0^u = 10$, $o_0^v = 2$, $o_0^r = 1$, and $o_0^d = 1$. Hardening decisions should be made for tasks τ_0 and τ_1 as they belong to the high-criticality graph $\mathcal{G}_0$. Task τ_0 uses re-execution policy (first gene value 0) while task τ_1 uses passive replication (first gene value 2) with 2 replicas (second gene). In the 'Allocation' part, the mapping decisions of all tasks and replicas are made as shown in the boxes on the bottom-right. We determine only the priorities of original tasks as scheduling decisions. In 'QoS Level',

the low-criticality graph $\mathcal{G}_1$ is decided to limit execution budgets of τ_2 and τ_3 as 60% of their original WCETs once the system gets into a faulty state.

Crossover/Mutation: At each generation of the GA, new offsprings are generated by means of crossover and mutation operations. We apply single point crossover for each of 'Hardening'. 'Allocation', 'Priority', and 'QoS Level' separately. However, 'Hardening' and 'Allocation' share the same pivot as they are highly dependent upon each other. For instance, if the crossover pivot point for 'Hardening' is right after τ_0 in an individual, the pivot point of 'Allocation' must also be right after τ_0.

In case of mutations, each gene is randomly changed with a fixed mutation rate. In some cases, a mutation may require a post-processing in order to keep the validity of the solution. If 'Hardening' is mutated to have more replicas, for example, new genes need to be set up for the newly generated replicas in 'Allocation'. Such newly generated genes are filled with valid random values. Post-processing is also necessary for 'Priority'. Suppose that the priority value of τ_0 is mutated from 0 to 2 in the example of Figure 5.4(b). Then, we remove the duplicated priority numbers by shifting the priority values of τ_1 and τ_2 to 1 and 0.

Repairing/Evaluation: During the evaluation process, each offspring is decoded and verified whether it is a valid solution or not, in terms of the reliability and WCRT. It is common in electronic design automation (EDA) problems that local search algorithms based on domain-specific knowledge are incorporated into the GA engine [96]. In the proposed technique, we also apply such local search heuristic to repair invalid solutions. A solution is invalid if it violates either reliability or deadline constraint. To make such solutions valid, we repair them as follows: first, in case of reliability violation, we repeatedly add two more replicas to the task of the highest fault probability in order to decrease the failure probability. If the replication degree gets bigger than the number of processing elements, the hardening technique for the task is changed to re-execution. This procedure repeats until the reliability is satisfied or no task to be replicated is left. In the

case that all the tasks are hardened by re-execution, we can eventually find the numbers of re-execution that satisfy the reliability.

Secondly, we repair the deadline violations, which have various causes: the number of replicas, dropping, mapping, or priorities. In this repair procedure, we aim to repair a solution more likely to be feasible, since we cannot *exactly* tell whether it is feasible or not without WCRT analysis. In case replicas are excessively generated, we reduce the number of replicas as much as possible while satisfying the reliability. Since the active replication and the passive replication require at least 3 and 2 replicas, we find tasks that have more than 3 replicas and remove 2 replicas from each task only if the reliability preserves after removal. The violation may also come from ill-assigned priorities. We sort the feasible applications in descending order of slack time, i.e., $D_{\mathcal{G}} - \mathcal{R}_{\mathcal{G}}$, and choose a random number of applications in front of the sorted list. Then, we enforce them to have priorities lower than that of infeasible applications. Similarly, we slightly tweak mapping decisions to have evenly distributed workload over the processing elements. We randomly select tasks and move each task to a processing element that has lower utilization $\sum \frac{C^u}{T_{\mathcal{G}}}$. Finally, if a low-criticality application may interfere the infeasible application, we randomly degrade the dropping factor of that application.

The above repair procedure does not always succeed; it just enhances the chance to produce feasible solutions. Hence we repeat *evaluation-and-repair* procedure until the solution becomes feasible or iteration number reaches a given limit. Though repeating many iterations of such procedure helps to find a feasible solution, it requires large computational overhead. In case that this repair procedure eventually fails, we penalize the infeasible solutions by setting the quality of service to $-\infty$, normal power consumption to ∞, and critical power consumption to ∞.

Note that all the objectives can be computed *without* WCRT evaluation, which is the most time-consuming task in the proposed GA. In order to speed up the GA procedure, we do not always perform the WCRT analysis. Instead, a solution is evaluated in terms

Algorithm 5 Evaluation of a genotype

```
 1: Solution S = decode(genotype)
 2: for iteration < MAXITERATION do
 3:    Compute multi-objectives O from S
 4:    if O is dominated by pareto solutions then
 5:       return O
 6:    end if
 7:    WCRT Analysis of S                                  ▷ Algorithm 6
 8:    if S is feasible then
 9:       return O
10:    else
11:       repair(genotype)
12:       S = decode(genotype)
13:    end if
14: end for
15: return infeasible objectives O = (−∞, ∞, ∞)
```

of WCRT only if it is not dominated by all pareto solutions[1]. By doing so, we eliminate unnecessary evaluations of sub-optimal solutions while keeping the pareto solutions intact. Algorithm 5 shows the evaluation process. If a solution is dominated by any pareto solution found so far, the evaluation terminates without further verifications (lines 4-6). If it is infeasible in WCRT (lines 7-8), we repeatedly perform repairing (line 11) until it reaches the iteration limit (line 2).

5.3.3 Part II: Proposed WCRT Analysis

In this subsection, we explain how a mapping/scheduling/hardening decision can be analyzed in terms of WCRT.

Algorithm 6 sketches the proposed iterative WCRT analysis. We first analyze the WCRTs of all applications without faults using the proposed WCRT analysis in Chapter 3 (line 1). In case all applications are feasible in normal state (lines 2-4), we analyze the WCRT of high criticality applications in faulty state. For each high-criticality application (line 5), the WCRT is initialized to the length of the longest path of the task graph,

[1]Solution A is said to dominate B if the fitness values of A are better than those of B in all metrics of optimization. If there is any metric in which B is better than A, dominance relation does not exist.

Algorithm 6 Top-most WCRT analysis loop

```
1: Conventional WCRT analysis for normal state
2: if deadline violation exists then
3:    return FAIL
4: end if
5: for each high criticality application G do
6:    R_G = longest path of G;                                  ▷ Initialization
7:    repeat
8:       Determine ∀pe ∈ PE, k_pe s.t. equation (5.12) holds true and Σ_{pe∈PE} k_pe is
         minimized                  ▷ Maximum number of faults for each pe ∈ PE
9:       if k_pe is not found then
10:         return FAIL
11:      end if
12:      re-estimate R_G with k_pes                              ▷ Algorithm 7
13:      if R_G > D_G then
14:         return FAIL
15:      end if
16:   until R_G converged;
17: end for
18: return SUCCESS
```

i.e., no interferences or re-executions considered. Then, the maximum number of faults to be tolerated on pe, k_{pe}, is calculated for each PE, with respect to the current WCRT value (line 8). How to determine k_{pe} will be explained in subection 5.3.3.1 in more detail. In turn, the updated number of faults that can possibly occur during the WCRT may affect the WCRT back. Thus, we need to re-estimate the WCRT with the new maximum number of faults k_{pe} (line 12). In order to estimate the WCRT considering the interferences between tasks in faulty state, it uses Algorithm 7 which will be shown in subsection 5.3.3.2. As these two values, k_{pe} and $\mathcal{R}_{\mathcal{G}}$, are dependent upon each other, these calculations are repeated until the WCRT value converges (line 16). Due to the fixed-point convergence loop (lines 7-16), the time complexity of the proposed algorithm is not formally proven to be polynomial. However, in practice, the number of loop iteration is very little, making the analysis time affordable. In our experiments which will be shown in Section 5.4, the maximum number of iterations is only two since k_{pe} is usually small.

5.3.3.1 Maximum Number of Faults

How can we obtain the maximum number of faults that can be tolerated within the WCRT under the given reliability constraint? Let us suppose that k_{pe} errors can maximally occur in each $pe \in \mathcal{PE}$ over the time window of $\mathcal{R}_\mathcal{G}$ and the application always successfully finishes before $\mathcal{R}_\mathcal{G}$. Then, from the perspective of the entire schedule, the probability that application $\mathcal{G}$ successfully finishes within $\mathcal{R}_\mathcal{G}$ becomes

$$\prod_{pe \in \mathcal{PE}} \sum_{i=0}^{k_{pe}} P(pe, \mathcal{R}_\mathcal{G}, i). \tag{5.11}$$

when we assume fault occurrences are independent events.

Besides the entire schedule, let us focus on the individual tasks. They are not reliable if the applied hardening techniques fail. Note that here we need to consider the active and passive replications only since a task hardened by re-execution will be successful without the limit on the re-execution counts in the proposed technique. Since the number of faults in each PE is limited to k_{pe}, the fault probability $F(\tau_s, t)$ is revised to consider the finite number of errors: the original possibility with unlimited errors $1 - P(M_s, t, 0)$ is replaced with $\sum_{i=1}^{k_{M_t}} P(M_s, t, i)$. Then, the reliability constraint of the graph becomes

$$1 - \left(\prod_{pe \in \mathcal{PE}} \sum_{i=0}^{k_{pe}} P(pe, \mathcal{R}_\mathcal{G}, i) - \sum_{\tau_s \in REP} R'(\tau_s) \right) \leq rel_\mathcal{G}, \tag{5.12}$$

where REP is a set of tasks hardened by replication and $R'(\tau_i)$ is the fault probability of τ_i under the limited numbers of faults k_{pe}. Then, one can obtain the value of k_{pe} that represents the maximum number of faults in each pe to satisfy the reliability constraint $rel_\mathcal{G}$.

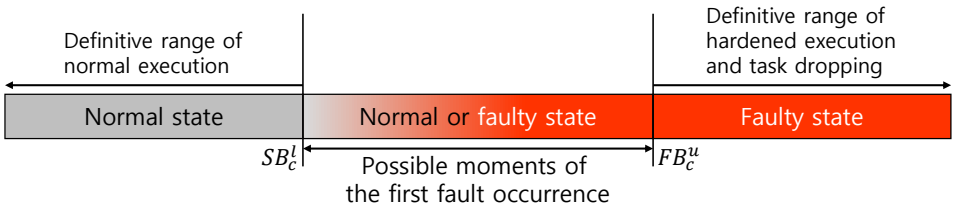


Figure 5.5: Classification of WCRT analysis modes according to the faulty state entering moment.

5.3.3.2 Calculating WCRT under the Given Number of Faults

Next, we show how to calculate the WCRT of a high-criticality application when the maximum number of faults to be tolerated in each PE is given. As a first step, we apply the WCRT analysis technique in Chapter 3 to get the release time bound (RB^l, RB^u), starting time bound (SB^l, SB^u), and finishing time bound (FB^l, FB^u). Hardening decisions are closely related to these bounds. For instance, without re-execution, the finishing time bound of τ_0 in Figure 5.2 is $(3,3)$. But, with one re-exeuction, it will get $(3,6)$.

Faulty state: In the calculation of tight WCRTs, it is important to know when the system enters faulty state, i.e., when the first fault occurs and dropping of low-criticality applications starts. It is because both high- and low-criticality applications are significantly influenced by that event: low-criticality applications may be partially dropped out of the scheduler while the high-criticality ones may endure additional overhead incurred by re-execution or passive replication. To accurately model such effects, the estimation of the WCRT should be differently performed before and after the faulty state entering moment. Let us suppose that the task detects the first fault, and triggers the faulty state, is τ_c. Then, we can distinguish the WCRT analysis mode into three based on the execution of τ_c as depicted in Figure 5.5. That is, we can assure that no instance of re-execution or passive replica executes before the earliest starting time of τ_c, SB_c^l('Normal state' in the figure). Similarly, droppable applications may partially exist in the system after FB_c^u ('Faulty state'). During the possible execution time of τ_c, $[SB_c^l, FB_c^u]$, we have to be open

to any possibility in the analysis ('Normal or faulty state').

Interference from low-criticality applications: As an analysis result, upper and lower bounds of release, starting, and finishing times should be derived. Among them, let us first consider the upper bounds of a task τ_t: RB_t^u, SB_t^u, and FB_t^u. These are the worst cases in terms of interference. Thus, all possible interference from low-criticality (droppable) applications that can appear before FB_c^u should be taken into account. Conversely, when calculating the lower bounds, RB_t^l, SB_t^l, and FB_t^l, we need to minimize the interference from the droppable applications. We consider partially dropped execution time $drp_{\mathcal{G}} \cdot C^u$ that appears after SB_c^l. In short, when considering the worst-case interference, all droppable tasks that may occur before FB_c^u are fully executed assuming the faulty state begins at FB_c^u. For the best case, we suppose that the fault occurs at SB_c^l and all appearance of droppable tasks after SB_c^l are partially dropped according to their dropping factors.

Interference from high-criticality applications: On the contrary, high-criticality applications can cause interference throughout the entire running time of the target application as they are never dropped. However, after entering into faulty state, they cause additional interferences such as re-executions or passive replications. In order to precisely keep track of the worst-case interference, it is necessary to scrutinize every possible re-execution scenario. In the proposed technique, we let the high-criticality tasks, after entering into the faulty state, have re-execution instances as many times as the number of faults to be tolerated. When the number of faults is k_{M_t}, the WCET of a task τ_t, in the faulty state, is adjusted to

$$(C_t^u + o_t^d) \times (k_{M_t} + 1) + k_{M_t} \cdot o_t^r \tag{5.13}$$

Note that the BCET remains intact so that the analysis still considers the cases that lesser number of faults occur.

Algorithm 7 summarizes the WCRT calculation procedure in pseudo code. It is structured in two nested loops. In the outer loop, it examine all possible cases that the first fault can possibly occur (line 2). Once it is fixed, it adjusts the WCETs according

Algorithm 7 WCRT analysis of $\mathcal{G}$ with the maximum number of faults for each *pe* given as k_{pe}.

```
1:  R_G = 0;                                                    ▷ Initialization
2:  for each task τ_i ∈ G^m do
3:      repeat
4:          τ_c := τ_i;                          ▷ set the first fault occurring task as τ_i
5:          Update WCETs according to k_pes and equation (5.13);
6:          Compute the time bounds of all tasks in G^m;
7:      until all time bounds converged
8:      if R_G < max_{τ_s ∈ V_{G^m}} FB^u_s then
9:          R_G = max_{τ_s ∈ V_{G^m}} FB^u_s                    ▷ update WCRT
10:     end if
11: end for
12: return R_G
```

to equation (5.13) with respect to the given k_{pe}s and τ_c (line 5). Then, the WCRT analysis technique in Chapter 3 is invoked at line 6. This procedure, in turn, repeats until the six time bounds explained above converge. At the end of this loop, it is checked that if the newly analyzed WCRT is the largest one ever observed (line 8). If so, the WCRT is updated properly (line 9).

5.3.4 Tightening the WCRT Estimation

The WCRT analysis presented in the previous subsection is very pessimistic as it assumes the maximum number of re-execution occurs. In this subsection, we present a post-processing of the analysis that tightens the WCRT estimates by removing such *unnecessary* re-executions. Note that it is virtually impossible to examine all possible combinations in practice, since the number of possible cases increases exponentially when the problem size grows. Instead, we deductively assemble a re-execution trace that upper-bounds the worst-case scenario. In what follows, we argue and prove that deferring re-executions in time always results in a worse, or equal, response time. That is, we have to distribute the re-execution instances as late as possible in the schedule to obtain the WCRT.

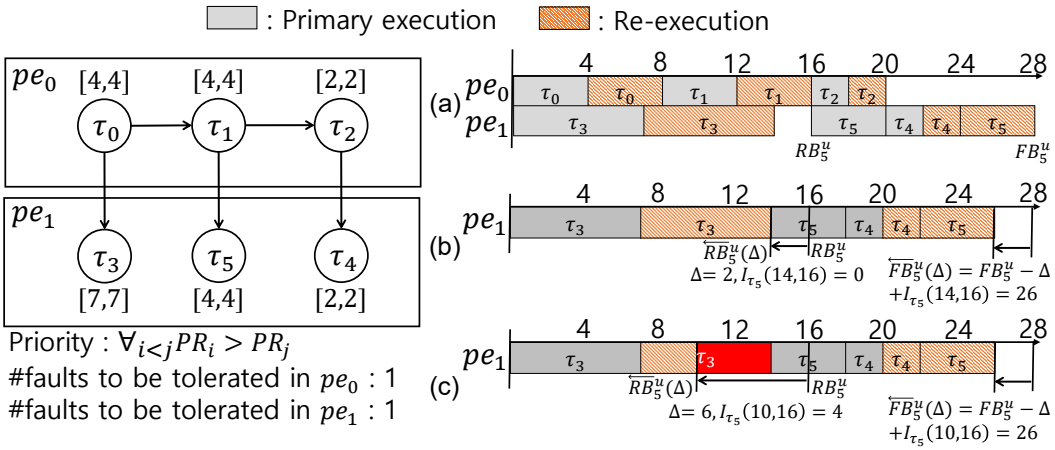


Figure 5.6: An example to show the effect of earlier release time on the finish time: (a) time bound schedule after algorithm 6, (b) reduce the worst-case release time of τ_5 by $\Delta = 2$, (c) reduce the worst-case release time of τ_5 by $\Delta = 6$.

Lemma 5.3.1. ***(Earlier release time)*** *If the worst-case release time RB^u of a task τ is adjusted Δ time units ahead, the reduction of its worst-case finish time FB^u is no greater than Δ.*

Proof. Let $\overleftarrow{RB}^u(\Delta)$ be the adjusted worst-case release time of τ, i.e., $\overleftarrow{RB}^u(\Delta) = RB^u - \Delta$. There are additional preemptions or blocking introduced in the time interval $[RB^u - \Delta, RB^u]$ if the release time is reduced to $RB^u - \Delta$. Let us define $I_\tau(x,y)$ as the maximum amount of preempted time from the higher priority tasks than τ in the time interval $[x,y]$. Then, $I_\tau(x,y)$ is at least 0 if τ is not preempted at all, and at most $y-x$ if the entire time interval is taken by the execution of higher priority tasks, i.e., $0 \leq I_\tau(x,y) \leq y-x$. Using this term, we can calculate the adjusted finishing time of τ as $\overleftarrow{FB}^u(\Delta) = FB^u - \Delta + I_\tau[RB^u - \Delta, RB^u]$. Thus, the reduction of the finishing time, $FB^u - \overleftarrow{FB}^u(\Delta)$, is $\Delta - I_\tau[RB^u - \Delta, RB^u]$ is positive and less than or equal to Δ. Hence, the lemma holds. □

Figure 5.6 illustrates two examples that show how the finishing time is adjusted in case of changes in release time. In the first example, Figure 5.6(b), the worst-case release time of τ_5 is reduced by 2. In this case, τ_5 does not experience preemption in the the time

interval $[14,16]$ ($I_{\tau_5}[14,16] = 0$), thus the finishing time is only reduced by $\Delta = 2$. On the other hand, in the second example (Figure 5.6(c)), we have $\Delta = 6$ and the amount of preemption should be calculated for the time interval $[10,16]$. As shown in the schedule, τ_3 may preempts τ_5 at most 4 time units making $I_{\tau_5}[10,16] = 4$. Therefore, the worst-case finishing time is reduced by $\Delta - I_{\tau_5}(10,16) = 6-4 = 2$.

Lemma 5.3.2. ***(Earlier release time of preceding tasks)*** *If the worst-case release time of a preceding task τ_s of τ_d is adjusted Δ time units ahead, the reduction of τ_d's worst-case finish time is no greater than Δ.*

Proof. Since the release time of the successor is defined as the maximum among the finishing times of its preceding tasks, the reduction in the release time of τ_d is less than or equal to the reduction of FB_s^u. By Lemma 5.3.1, in turn, the reduction of FB_s^u is no greater than Δ. Hence, the lemma. □

Theorem 5.3.1. ***(Deferring re-executions)*** *Given a time bound schedule with a fixed number of re-executions, shifting a re-execution instance to a later task in the schedule never makes the response time shorter.*

Proof. If a re-execution instance, which is Δ time units long, is removed from the time bound of task τ_a, the worst-case release time of its successor task τ_b is reduced by the length of the re-execution instance Δ or less because τ_b may have many predecessors. Then, by Lemma 5.3.1 and 5.3.2, the worst-case finish time of τ_b also reduced by Δ or less. Then, if we place the re-execution instance of length Δ to the successor task τ_b, it increases the finishing time of τ_b by Δ. In total, by shifting a re-execution instance from τ_a to τ_b, the increment of τ_b's worst-case finish time is Δ while its decrement is less than or equal to Δ. So, the response time never decreases. □

By Theorem 5.3.1, we can conclude that shifting re-execution instances inductively to the last task in the schedule results in the response time of larger or equal length, which

Algorithm 8 Rearranging re-executions to estimate the WCRT

1: **for** each $pe \in \mathcal{PE}$ **do**
2: $MTI_{pe} := 0$; ▷ *Initialization of maximum time intervals*
3: **end for**
4: Calculate the critical path $[\tau_1, ..., \tau_n]$;
5: **for** $i = 1, ..., n$ **do** ▷ *Step I: Removal of all re-executions*
6: Reduce the release time of τ_i as much as possible;
7: $MTI_{M_i} := \max(MTI_{M_i}$, maximum re-execution length in $\tau_i)$;
8: Remove the re-executions in τ_i and adjust its finishing time;
9: **end for**
10: **for** each $pe \in \mathcal{PE}$ **do** ▷ *Step II: Rearranging re-executions*
11: Find out $\tau_{last,pe}$;
12: Increase the finishing time of $\tau_{last,pe}$ by $MTI_{pe} \times k_{pe}$, and update time bounds of successor tasks of $\tau_{last,pe}$;
13: **end for**

is the key idea of the proposed WCRT estimation method. Algorithm 8 shows how re-execution instances are removed and redistributed in the schedule while guaranteeing the WCRT. It consists of two distinct steps. In the first step (line 5-9), it removes all the re-execution instances from the schedule. Then, in the second, the re-executions are placed in the last task of the critical path on each pe, $\tau_{last,pe}$, as many as necessary (line 10-13).

In the first step, it starts by calculating the critical path of the given schedule in $[\tau_1, ..., \tau_n]$ (line 4). Iterating the critical path from the source node to the sink, it first adjusts the release time of the task ahead in time as much as possible (line 6). Then, it examines that the task is associated with any re-execution. If so, it removes them from the schedule and adjusts the finishing time appropriately (line 8). During this removal procedure, the maximum time interval MTI_{pe} of the re-execution instances should be recorded (line 7) for each pe. Later, in the second step, this time interval will be added in the schedule to the newly placed position.

Finally, in the second step, the faults are re-distributed in the last task of the critical path on each PE, $\tau_{last,pe}$, by the maximum number of faults to be tolerated (lines 10-13). In order to keep the conservativeness of the estimation, the distributed re-execution instances are assumed to have a time interval of MTI_{pe} on pe.

In the following, we prove that the response time estimate obtained by Algorithm 8 always guarantees the worst case.

Theorem 5.3.2. ***(Conservativeness)*** *The response time obtained by the schedule where re-executions are rearranged by Algorithm 8 is larger than or equal to the WCRT.*

Proof. We use the method of reduction to absurdity to prove the theorem. Let the WCRT obtained by Algorithm 8 be denoted as $WCRT_a$. Let us suppose that we have a schedule s whose response time is $WCRT_s$ and larger than $WCRT_a$. Now, we can repeatedly shift each re-execution of schedule s to the successors. By doing so, it eventually ends up with a new schedule s' which is exactly the same as the one derived by Algorithm 8. Note that it is exactly the case that no re-execution instance can be shifted further. By Theorem 5.3.1, we know that during the schedule transformation from s to s' the WCRT never decreases, i.e., $WCRT_s \leq WCRT_{s'}$. As we added the largest time interval when rearranging the re-execution in line 12 of Algorithm 8, $WCRT_{s'} \leq WCRT_a$. Thus, we arrive at $WCRT_s \leq WCRT_a$ and this contradicts the supposition. □

We revisit the proposed method with an example given in Figure 5.7. In this example, the critical path, highlighted in red, can be represented in $[\tau_0, \tau_2, \tau_3, \tau_5, \tau_6]$. A naive estimation would result in the exaggerated WCRT estimate as shown in Figure 5.7(a). After applying the first step of Algorithm 8, the schedule would look like Figure 5.7(b) where no re-executions are triggered. Finally, the re-executions are placed as late as possible in the schedule as a result of the second step of Algorithm 8 which results in the schedule of Figure 5.7(c).

Note that Algorithm 8 only considers the critical path, thus re-executions in non-critical paths are still there as τ_1 in Figure 5.7 (c), affecting the adjustment of the release times in the critical path. Such side effects can be mitigated by applying the algorithm for the subcritical paths recursively. If the release time reduction is bounded by the finish time of the other predecessor which have excessive re-executions, Algorithm 8 is recursively

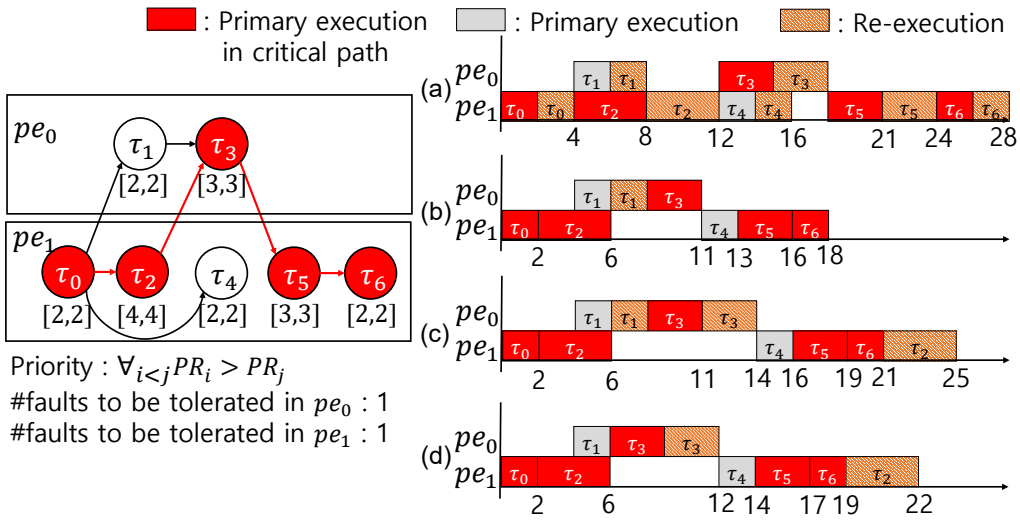


Figure 5.7: An example to show the process of Algorithm 8: (a) pessimistic WCRT estimation with excessive re-executions, (b) removal of re-executions in the critical path, (c) re-placement of re-executions to the end of the schedule, (d) applying the algorithm for the subcritical path

called to the critical path from the source task to the predecessor to eliminate excessive re-execution of the other path. Figure 5.7 (d) shows the schedule after recursive call for τ_1. Since this approach increases the estimation accuracy at the cost of huge analysis time, the recursive depth should be carefully decided to make an efficient choice on the tradeoff between the accuracy and analysis time.

5.4 Experiments

5.4.1 Environment

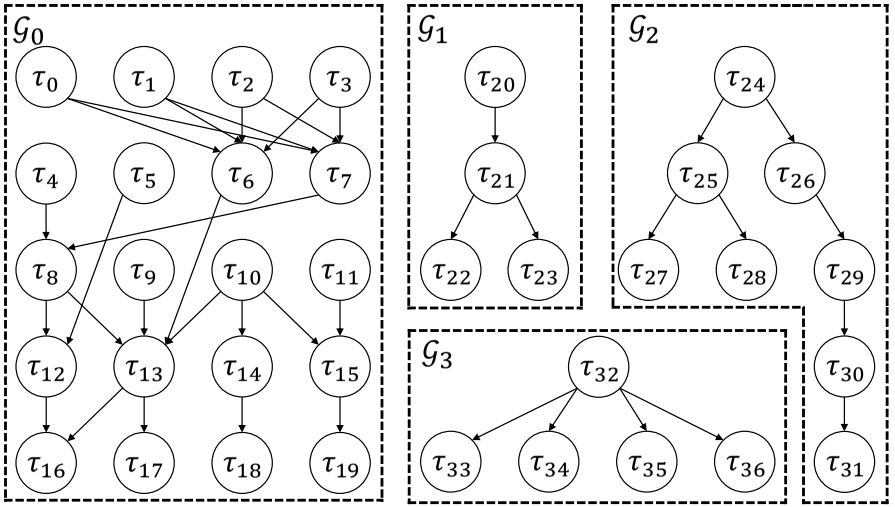
System configurations: In experiments, we use synthetic examples and two real-life benchmarks. Commonly in all synthetic examples and benchmarks, the target architecture is a multi-core system with 4-8 cores, where each core has the fault rate λ_{pe} of 868 FIT (Failure-in-Time, i.e., expected failures for one billion device-hours of operation) according to [97]. The power dissipation of each core is set to $stat_{pe} = 0.5W$

and $dyn_{pe} = 0.9W$. These parameters are taken from [22] and we set the reliability constraint to $rel_{\mathcal{G}} = 1.0 \times 10^{-10}$ for all high-criticality applications, which is 10 times smaller than the highest level of an existing safety standard, ISO-26262 [13]. We use a quadratic service-of-quality function $qos_{\mathcal{G}}(x) = 1-(x-1)^2$, as an example of a concave downward and monotonically increasing function. In many anytime algorithms, QoS is modeled using such a monotonically increasing function of execution time [98, 99, 100, 101]. Note that we may use other QoS function as long as it is monotonically increasing. We use constant values for roll-back overhead $o^r = 2ms$ and voting overhead $o^v = 1ms$ while detection overhead can vary between 10 ms and $\frac{3}{4} \cdot C^u$ ms.

Benchmarks: We borrow two real-life benchmarks, a cruise control application *Cruise* from [102] and a control benchmark *DT-med* "medium distributed non-preemptive real-time CORBA application" from [103]. To increase the benchmark complexity and uncertainty, we add three synthetic applications to *Cruise*, and multiply the invocation period and execution time in *DT-med* by 20 times. Graph topology and the detailed information of each benchmark is illustrated in Figure 5.8.

Synthetic examples: We generate the synthetic examples as follows: ten applications (task graphs) are generated in each example and a half of them is assumed to be of high-criticality. The importance of the service $sev_{\mathcal{G}}$ for droppable applications is 10. Each application consists of 10 tasks, where BCET C^l varies in [20 ms, 50 ms] and variation of C^u is within $[C^l, C^l \times 1.5]$.

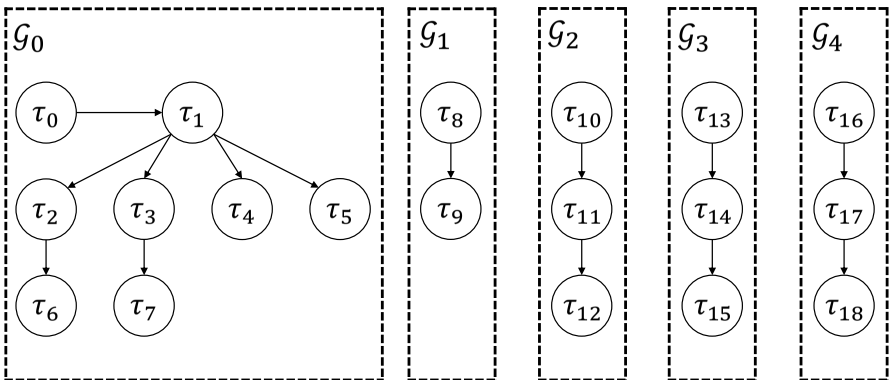
To make the tasks have dependencies, a random number of edges are arbitrarily assigned between tasks. In doing so, we keep task graphs acyclic. That is, an edge (τ_s, τ_d) is removed if it makes a cycle in the topology or there exists another path from τ_s to τ_d. In the 100 randomly generated examples, we used in the experiments, each graph has on average 10.733 edges. To make the generated example schedulable, the periods and deadlines are initially assigned as the latencies of the longest paths of the applications. We repeatedly multiply the periods and deadlines by 2 until all applications are analyzed



graph	period	deadline	Criticality	Importance of service
$\mathcal{G}_0$	1000	600	HIGH	0
$\mathcal{G}_1$	1000	1000	HIGH	0
$\mathcal{G}_2$	1000	1000	LOW	20
$\mathcal{G}_3$	1000	1000	LOW	10

task	[BCET,WCET]	task	[BCET,WCET]	task	[BCET,WCET]	task	[BCET,WCET]
τ_0	[29,29]	τ_{10}	[21,21]	τ_{20}	[24,24]	τ_{30}	[41,41]
τ_1	[29,29]	τ_{11}	[25,25]	τ_{21}	[40,40]	τ_{31}	[45,45]
τ_2	[29,29]	τ_{12}	[36,36]	τ_{22}	[45,45]	τ_{32}	[15,15]
τ_3	[29,29]	τ_{13}	[29,29]	τ_{23}	[19,19]	τ_{33}	[13,13]
τ_4	[43,43]	τ_{14}	[43,43]	τ_{24}	[18,18]	τ_{34}	[47,47]
τ_5	[25,25]	τ_{15}	[36,36]	τ_{25}	[35,35]	τ_{35}	[25,25]
τ_6	[43,43]	τ_{16}	[21,21]	τ_{26}	[12,12]	τ_{36}	[34,34]
τ_7	[21,21]	τ_{17}	[21,21]	τ_{27}	[27,27]		
τ_8	[43,43]	τ_{18}	[21,21]	τ_{28}	[19,19]		
τ_9	[25,25]	τ_{19}	[14,14]	τ_{29}	[29,29]		

(a) Graph topology and the detailed information of the benchmark example *Cruise*



graph	period	deadline	Criticality	Importance of service
$\mathcal{G}_0$	600	370	HIGH	0
$\mathcal{G}_1$	600	500	HIGH	0
$\mathcal{G}_2$	600	350	LOW	10
$\mathcal{G}_3$	600	450	LOW	30
$\mathcal{G}_4$	600	360	LOW	20

task	[BCET,WCET]	task	[BCET,WCET]	task	[BCET,WCET]	Task	[BCET,WCET]
τ_0	[20,20]	τ_5	[40,40]	τ_{10}	[40,40]	τ_{15}	[30,30]
τ_1	[50,50]	τ_6	[10,10]	τ_{11}	[50,50]	τ_{16}	[30,30]
τ_2	[50,50]	τ_7	[10,10]	τ_{12}	[10,10]	τ_{17}	[60,60]
τ_3	[60,60]	τ_8	[40,40]	τ_{13}	[30,30]	τ_{18}	[10,10]
τ_4	[10,10]	τ_9	[40,40]	τ_{14}	[30,30]		

(b) Graph topology and the detailed information of the benchmark example *DT-med*

Figure 5.8: Graph topology and the detailed information of two benchmark examples [Unit: ms]

as feasible by the proposed WCRT analysis. Unless specified otherwise, the hardening technique, mapping, priority, and dropping factor are randomly assigned.

GA engine: We use an open-source framework OPT4J [104] as GA engine and SPEA-II [105] is chosen as population selector for the multi objective optimization of the three objectives, QoS, power consumption in the normal state, and power consumption in the faulty state, as defined in Section 5.1.7. The population size, the number of parent individuals, and the number of offsprings are set to 300, 25, and 25, respectively. At each generation, 25 offsprings are generated by crossover and mutation from 25 randomly selected parent individuals. Crossover and mutation rates are 95% and 10%. Individuals with inferior fitness values are replaced with new offsprings by SPEA-II. The maximum number of generations is set to 5,000. With the above parameters and configuration, we could achieve the converged solution for all experiments performed in this section.

5.4.2 Comparison of the WCRT estimation techniques

We first show the effectiveness of the proposed WCRT analysis technique in comparative evaluations with the state-of-the art [22]. In this experiment, we use 100 synthetic examples and two benchmarks. For each example, 100 random schedules and 100 optimal schedules are generated. In "random schedule", mapping, priority, hardening technique, and QoS level are randomly assigned, while "optimal schedule" is optimized by the proposed GA-based DSE engine. We measure the ratio of the WCRT estimation by the reference technique to proposed one for each schedule. That is, the ratio of 130 means that the WCRT estimate from the reference technique is 30 % larger than that of the proposed technique.

Tightness of WCRT estimations: Figure 5.9 shows the distribution of the ratios in percent in box-plots for 100 different mappings for each case. In the box-plots, two ending points of the vertical lines indicate the maximum and minimum ratios. Top and bottom of the boxes correspond to 25- and 75-percentile points while the lines in the middle of

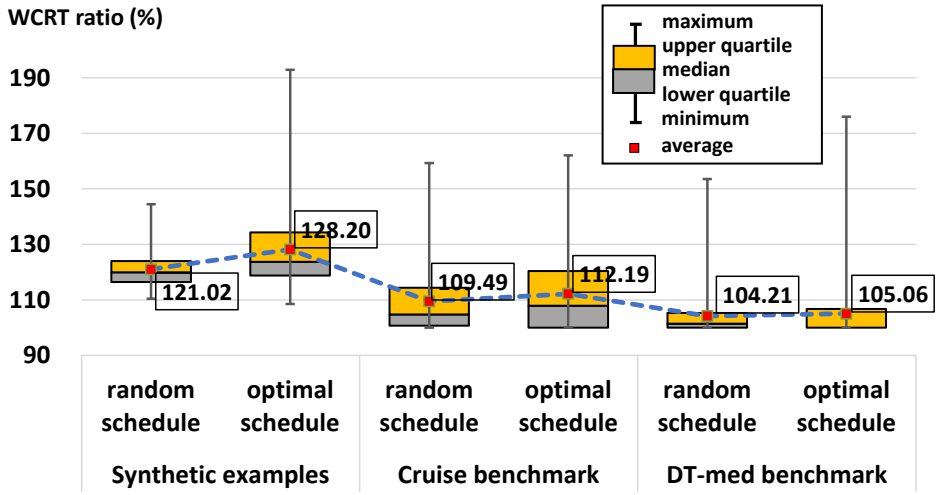


Figure 5.9: Comparison with the reference technique using synthetic examples and two benchmarks

the boxes represent median values. We highlight the average ratios as red dots.

We first show how conservative the reference technique is compared to the proposed one with the synthetic examples. For random schedules, the proposed technique shows 21.02% tighter WCRT estimate on average. These gaps get bigger up to 28.20% on average in case of optimal schedules. This difference is attributed to how often the re-execution scheme is adopted as hardening technique. The benefit of the proposed WCRT estimate is more prominent in optimal schedules, where re-execution is preferred in favor of allocating more resources.

We could observe the same tendency for the real-life benchmarks. The proposed technique shows better performance on the optimal schedule than random schedule for both benchmarks. Here, again, it is observed that optimal schedules likely to have more tasks hardened by re-execution: 87.45% and 90.2% of the high criticality tasks in *Cruise* and *DT-med* optimal schedules used re-execution. We observe relatively lower average gain in real-life benchmarks compared to the synthetic examples. Since our optimization approach rearranges re-executions in the critical path, graphs with less number of tasks in the critical path like *DT-med* have relatively smaller benefit in the proposed technique.

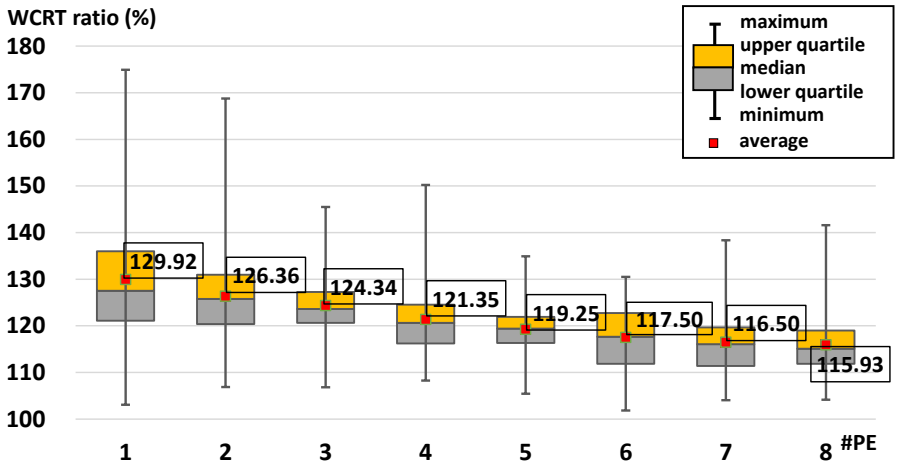


Figure 5.10: Estimation ratio and estimation time while varying the number of processing elements from 1 to 8

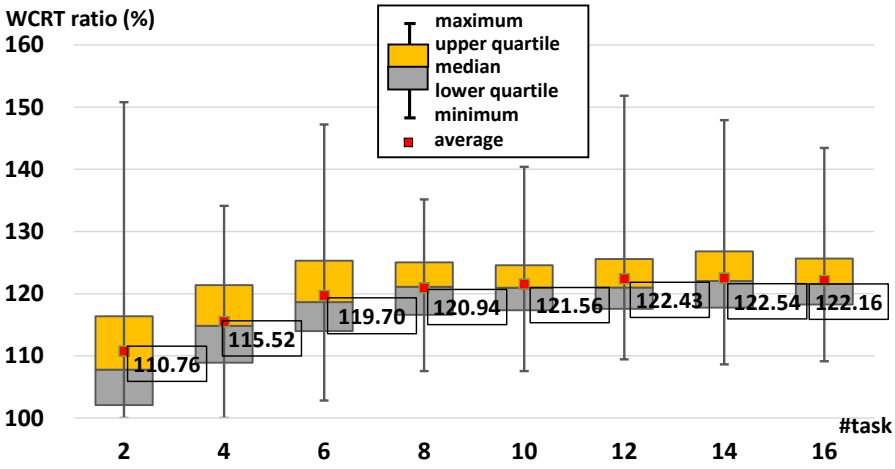


Figure 5.11: Estimation ratio and estimation time while varying the number of tasks in each graph from 2 to 16

Scalability: We verify the scalability of the proposed WCRT estimation technique by measuring how the WCRT estimation ratio scales as the number of processing elements and the number of tasks in each graph grow. We vary the number of processing elements from 1 to 8, for each of which we generate 100 synthetic examples and measure the WCRT ratio to the reference technique. As shown in Figure 5.10, the gap decreases as

the number of PEs increases, since it results in increases of the total number of faults in all PEs that should be tolerated, i.e., $\sum_{pe \in \mathcal{PE}} k_{pe}$.

We perform the same scalability experiment varying the number of tasks in each graph from 2 to 16 while fixing the number of processing elements as 4. As shown in Figure 5.11, we observe that the gap increases as the number of tasks increases, but the gain is saturated around 122% when the number of tasks is above 12. Note that the key idea of the proposed technique is to reduce the number of unnecessary re-executions during WCRT estimation. Thus, the ratio is highly dependent upon the number of faults. It is obvious that the number of faults to be tolerated on each PE is small as the number of tasks per graph is small. This is the reason why the gain gets bigger as the number of tasks per graph grows. On the other hand, the gain is converged to a certain ratio as the number of re-executions is fixed as a finite number in hardening decision.

5.4.3 Multi-objective DSE

Scheduling difficulty and optimality: Overestimated WCRT may cause inefficiency in the system design, especially when the deadline is close to the scheduling make-span. In this experiment, we observe how toughness of scheduling affects the DSE result. That is, starting from the deadline equal to the period (ratio of 1.0), we repeatedly reduce the deadline and perform the optimization until the reference technique fails to find out a feasible solution.

Table 5.2 summarizes the experimental results. We show the three objectives (quality of service (QoS), power consumption in normal states (PC_N), power consumption in faulty states (PC_F)) of a representative solution and the number of pareto solutions found during the DSE. For the representative solutions, we choose the one with the biggest QoS value in each case. When the deadline is equal to the period (ratio of 1.0), the scheduling is relatively less difficult, thus, both the proposed and reference techniques find the solutions of similar optimality (maximum QoS and small power consumptions). As the

Table 5.2: The effect of WCRT estimation on the DSE results of two benchmarks *Cruise* and *DT-med*

Cruise	deadline ratio	QoS	PC_N	PC_F	#pareto
proposed	1.000	100%	2.80W	3.83W	27
	0.600	100%	3.32W	4.32W	7
	0.400	100%	4.30W	5.33W	8
	0.200	100%	5.30W	6.33W	1
	0.196	92%	5.30W	6.24W	2
reference [22]	1.000	100%	2.85W	3.81W	14
	0.600	97.33%	3.31W	4.28W	8
	0.400	100%	4.34W	5.32W	4
	0.200	85.33%	5.30W	6.21W	1
	0.196	52%	5.30W	6.11W	1
DT-med	deadline ratio	QoS	PC_N	PC_F	#pareto
proposed	1.000	100%	2.70W	3.49W	6
	0.600	100%	3.20W	3.93W	15
	0.400	100%	3.70W	4.49W	8
	0.300	100%	3.70W	4.49W	7
	0.234	98%	4.20W	4.96W	12
reference [22]	1.000	98.67%	2.70W	3.46W	8
	0.600	100%	3.20W	3.99W	6
	0.400	100%	3.70W	4.49W	9
	0.300	100%	4.20W	4.99W	10
	0.234	94.00%	4.20W	4.90W	11

deadline ratio gets smaller making the optimization more challenging, power consumptions tend to increase. It is straightforward because more PEs are required for scheduling to satisfy the harder timing deadline.

The merit of the proposed technique is evidently shown when the deadline is very small, i.e., the length of the critical path, 0.196 for *Cruise* and 0.234 of *DT-med*. In the case of the deadline ratio 0.196 in *Cruise*, the power consumption does not increase anymore, as it already uses all available cores in the ratio of 0.2. In this case, low-criticality graphs are aggressively dropped in order to fulfill the deadline constraint. Due to the pessimism of WCRT estimation, the reference technique only finds a solution with QoS of

Table 5.3: The effect of hardening decision on the DSE results of two benchmarks *Cruise* and *DT-med*

Cruise	QoS	PC_N	PC_F	percentages of hardening techniques
configA	100%	6.27W	6.27W	(0%, 100%, 0%)
configAP	98.67%	5.37W	5.74W	(0%, 37.5%, 62.5%)
configAPR	97.33%	2.85W	3.77W	(75%, 16.67%, 8.33%)
DT-med	QoS	PC_N	PC_F	percentages of hardening techniques
configA	96%	4.94W	4.85W	(0%, 100%, 0%)
configAP	100%	4.46W	4.94W	(0%, 40%, 60%)
configAPR	96.67%	2.71W	3.36W	(70%, 10%, 20%)

52 % for *Cruise*. Meanwhile, the proposed technique helps DSE engine to detect feasibility accurately, and leads to find solutions with higher quality than the reference technique. Similarly to the previous experiments, *DT-med* shows similar tendency with a relatively small difference.

Effect of hardening decision: In this experiment, we examine the effect of hardening techniques, by restricting the selection of hardening techniques in the DSE engine. Table 5.3 summarizes the solutions of three different hardening configurations for *Cruise* and *DT-med*, respectively: Only active replication is used in “configA”, passive replication is additionally considered in “configAP”, and all the three hardening techniques are allowed in ”configAPR”. Three numbers of the last columns in the tables show the percentages of usage of re-execution, active replication, and passive replication, respectively. It is trivial that “configA” shows the worst power consumption. Estimated power consumption gets slightly enhanced by allowing passive replication, as shown in “configAP”. At last, “configAPR” shows the best solutions since the effect of enhanced WCRT analysis is particularly powerful in re-execution.

Effect of partial dropping: We conduct an experiment to verify how the proposed partial dropping enhances the service-power trade-off. We only show PC_F and QoS to enhance readability. The pareto-front of power-service pairs as output of the DSE is shown in

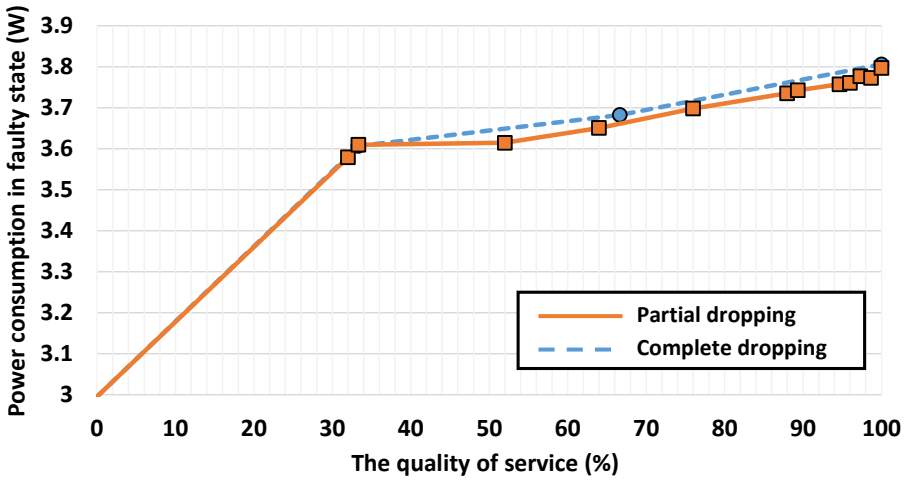


Figure 5.12: Service-power pareto-optimum graph for *Cruise*

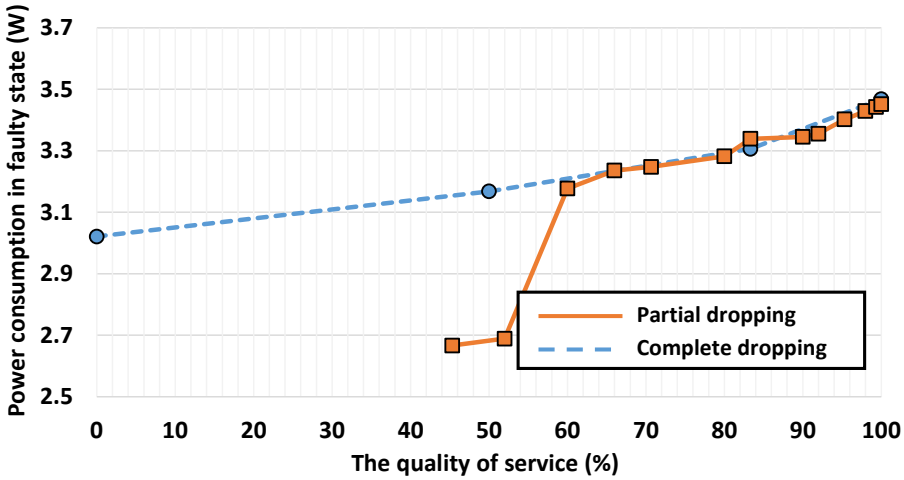


Figure 5.13: Service-power pareto-optimum graph for *DT-med*

Figure 5.12 and 5.13 for *Cruise* and *DT-med*, respectively. The y-axis denotes the power consumption in the faulty state. The plots show the pareto solutions with partial dropping by red line, and solutions with only task dropping by blue dashed line. As shown in the result, we can find far more feasible solutions, 9 more for both *Cruise* and *DT-med* when the partial dropping is applied. This is especially useful when the system has finer requirements on the quality of service or power consumption. For example, if the power

Table 5.4: The DSE result for four meta-heuristic algorithms

meta heuristic	*Cruise*			*DT-med*		
	QoS	PC_N	PC_F	QoS	PC_N	PC_F
SPEA-II	100%	2.80W	3.83W	100%	2.70W	3.47W
NSGA-II	100%	2.81W	3.82W	88.67%	2.20W	2.82W
SA	100%	3.32W	4.31W	96%	3.20W	3.90W
DE	96%	2.80W	3.76W	96%	2.2W	2.87W

consumption in the faulty state is required to be less than 3.78 for *Cruise* and 3.28 for *DT-med*, solutions with QoS of 98.67% and 70.67% can be found by partial dropping for *Cruise* and *DT-med*, respectively. In contrast, the achieved QoS levels are only 66.67% and 50% with complete dropping for *Cruise* and *DT-med*, respectively.

5.4.4 Optimization Engine

Selection of meta-heuristic: In order to prove the general applicability of the proposed technique, we conduct an experiment that compares the optimality of four different meta-heuristics algorithms supported by OPT4J [104]: SPEA-II [105], NSGA-II [106], simulated annealing (SA), and differential evolution (DE). For a fair comparison, we adjust the number of iterations of meta-heuristic engines to have an equal number of solution evaluations. For instance, the number of iterations for SA is 125,000 since the number of generations and the number of offsprings in the GA-based engines are 5,000 and 25 respectively. Table 5.4 summarizes the DSE result. Similarly to the previous experiments, we only show one representative solution with the biggest QoS for each case: Two GA-based engines, SPEA-II and NSGA-II, show slightly better optimality than the other two. Since SA searches the problem space by repeatedly traversing a neighbor solution, it is more likely to prematurely converge to local optima. DE shows similar optimality as the GA-based ones. But, it is known to be particularly effective in case that geographical distance between genes has some correlations in the gene values, which is not the case in the proposed problem.

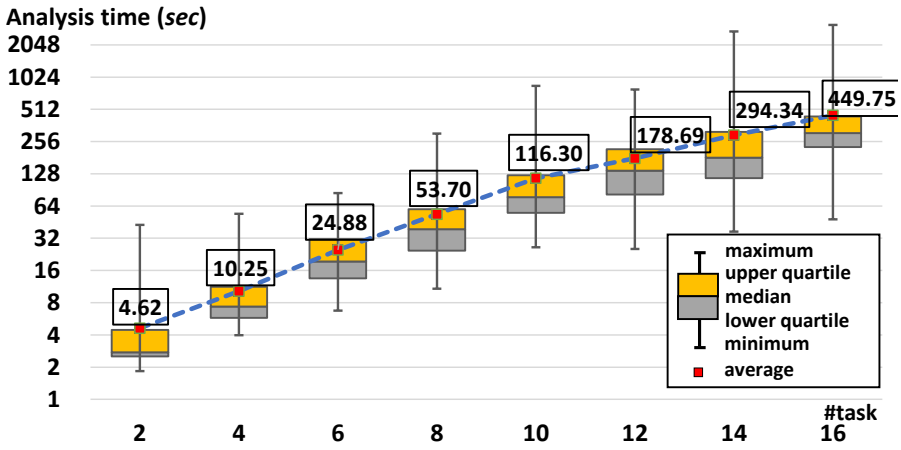


Figure 5.14: Logscale optimization time of the proposed GA engine while varying the number of tasks in each graph from 2 to 16

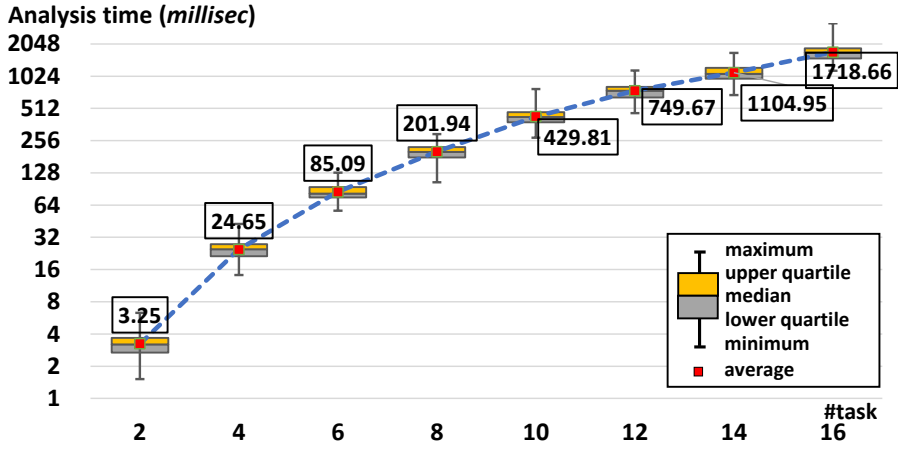


Figure 5.15: Logscale optimization time of the proposed WCRT analysis while varying the number of tasks in each graph from 2 to 16

Optimization time: Finally, we verify how the analysis and optimization times scale as the number of tasks in each graph grows from 2 to 16 while fixing the number of graphs and the number of PEs as 10 and 4, respectively. For each case, we generate 100 synthetic examples and perform the optimization separately. Figure 5.14 shows the distribution of the optimization time taken in the proposed framework (in log scale) in box-plots. Similarly to the previous plots, two ending points of vertical lines denote the maximum

and minimum optimization times while boxes in the plot surround 25- and 75-percentile points and the horizontal line in the box denotes the median. Average optimization times are denoted as red dots. As shown in the plot, the optimization time grows exponentially to the size of input task graphs. It varies from several seconds to several minutes, which we believe affordable in an offline optimization. In the most complex example of 160 tasks (16 task graphs with each having 10 nodes), it takes about 7.5 minutes on average.

In order to conduct in-depth analysis on the optimization time, we measure the WCRT analysis time separately and illustrate in Figure 5.15 in the same format as before. Again, it is shown that the analysis time exponentially grows to the size of input due to the inevitable time complexity inherited by the existing WCRT analysis technique. However, it is noticeable that many of WCRT analyses have been skipped by the selective evaluation scheme proposed in Section 5.3.2. On average, the WCRT evaluation is invoked only 262 times out of 125,000 evaluations. Although the proposed optimization is affordable for even in the biggest example, how to further reduce the WCRT analysis time remains as an important future work.

Chapter 6

Multi-mode Dataflow

6.1 Problem Definition

An MMDF is represented by a mode transition graph (MTG) and a dataflow for each mode. An MTG is defined as $< Mode, Trans >$ where $Mode$ represents a set of modes and $Trans$ represents a set of mode transitions. $(m_p, m_n) \in Trans$ means a transition from the previous mode m_p to the next mode m_n. Each mode m_i is defined as a dataflow graph $< \mathcal{V}_i, \mathcal{E}_i >$, where $\mathcal{V}_i$ is a set of tasks in mode m_i and $\mathcal{E}_i$ is a set of dependencies between tasks in $\mathcal{V}_i$. $(\tau_p, \tau_c) \in \mathcal{E}_i$ is a parent-child relationship between two tasks $\tau_p \in \mathcal{V}_i$ and $\tau_c \in \mathcal{V}_i$. A task may belongs to more than one mode. The scheduler checks whether the mode transition of the MMDF occurs at every period T. If the mode transition (m_p, m_n) occurs, the new mode m_n is scheduled instead of the previous mode m_p. Otherwise, the scheduler executes the current mode m_p again. All modes of the MMDF must finish before a given deadline D and we assume $D \leq 2T$.

When the dataflow of mode m_i is executed, $\tau_j \in \mathcal{V}_i$ can be executed only after all parent tasks have finished. The execution time variation of τ_j is represented by a tuple (C_j^l, C_j^u) that denotes the best-case execution time and the worst-case execution time, respectively. Each task τ_j is assigned a unique priority PR_j and is statically mapped to a processor M_j. All processors are assumed to use a preemptive scheduler. If a task belongs to both modes m_p and m_n but the processor mapping is different, it becomes executable

Table 6.1: Terms and notations

Notation	Description
$Mode$	a set of modes of the MMDF application
$Trans$	a set of mode transitions of the MMDF application
m_i	a mode of the MMDF application with its index i
$\mathcal{V}_i$	a set of tasks in the mode m_i
$\mathcal{E}_i$	a set of edges in the mode m_i
T	an initiation interval of the MMDF application
D	a deadline of the MMDF application ($D \leq 2T$)
RT	a set of real-time tasks
τ_i	a task with its index i
C_i^l	the best-case execution time of a task τ_i
C_i^u	the worst-case execution time of a task τ_i
PR_i	the priority of the task τ_i
M_i	the mapped processor of a task τ_i
MC_i	the processor migration cost of a task τ_i

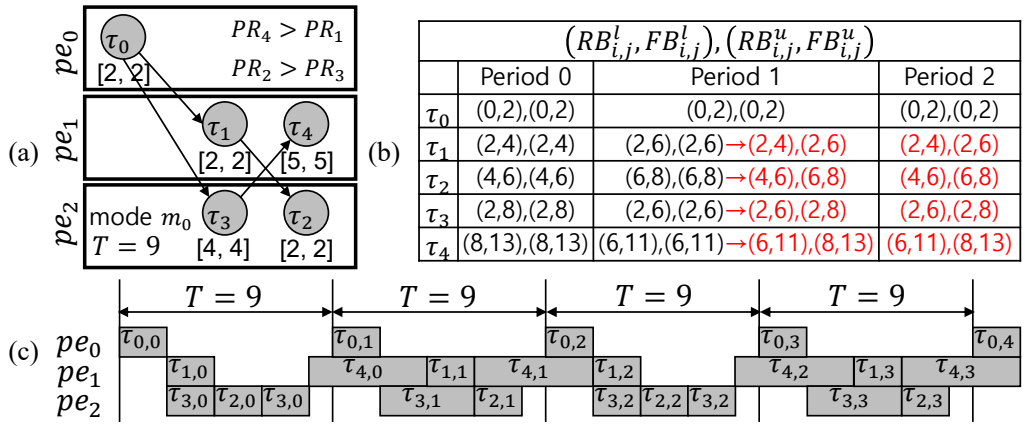
after paying the processor migration cost MC_j when the mode transition (m_p, m_n) occurs.

We assume a set of real-time tasks denoted by RT shares a multiprocessor with the MMDF application. Each independent task $\tau_j \in RT$ has execution time variation (C_j^l, C_j^u), static processor mapping M_j, and static priority PR_j. Real-time independent tasks do not have mode transition, and the priorities of tasks in RT are assumed to be higher than that of MMDF. That is, $\forall_{\tau_i \in RT, \tau_j \in Mode} PR_i > PR_j$.

Table 6.1 summarizes the notations declared above.

6.2 Proposed Technique

The proposed technique consists of three steps. At first, we analyze the schedulability of individual modes without mode transition in subsection 6.2.1. When the first step passes, we analyze whether the deadline constraint of the MMDF is satisfied or not for each mode transition (m_p, m_n), which will be explained in subsection 6.2.2. Finally, since the mode-to-mode interference can be accumulated when the mode transition occurs consecutively, we verify the schedulability for repeated mode transition in a con-



$(RB^l_{i,j}, FB^l_{i,j}), (RB^u_{i,j}, FB^u_{i,j})$

	Period 0	Period 1	Period 2
τ_0	(0,2),(0,2)	(0,2),(0,2)	(0,2),(0,2)
τ_1	(2,4),(2,4)	(2,6),(2,6)→(2,4),(2,6)	(2,4),(2,6)
τ_2	(4,6),(4,6)	(6,8),(6,8)→(4,6),(6,8)	(4,6),(6,8)
τ_3	(2,8),(2,8)	(2,6),(2,6)→(2,6),(2,8)	(2,6),(2,8)
τ_4	(8,13),(8,13)	(6,11),(6,11)→(6,11),(8,13)	(6,11),(8,13)

Figure 6.1: Schedule time bound computation through period extension for repeated single mode execution

servative manner in subsection 6.2.3. The MMDF always satisfies its deadline constraint when it passes all three steps.

6.2.1 Step 1: WCRT Analysis for Single Mode Repetition

As the first step of the proposed technique, we analyze the worst-case response time when each mode m_i is repeatedly executed without mode transition. Since $D \leq 2T$, mode executions in the previous period and the next period may overlap and interfere with each other. Therefore, we repeatedly expand job instances in the next period and analyze the schedule time bounds until the all bound values are converged.

We explain the process of analyzing worst-case performance by repeating the graph expansion through the single mode example in Figure 6.1 (a). In the figure, $\tau_{i,j}$ indicates the j-th τ_i job instance. When we add the job instances of the next period, we add an edge between two consecutive job instances of the same task $(\tau_{i,j}, \tau_{i,j+1})$ in order to guarantee the execution order. For the generated job instances, the schedule time bounds are analyzed using the baseline WCRT analysis technique in Chapter 3. Table in Figure 6.1 (b) shows the analyzed schedule time bounds. Here, the schedule time bounds of each period are a relative value with respect to the start time of the period of all job instances in three

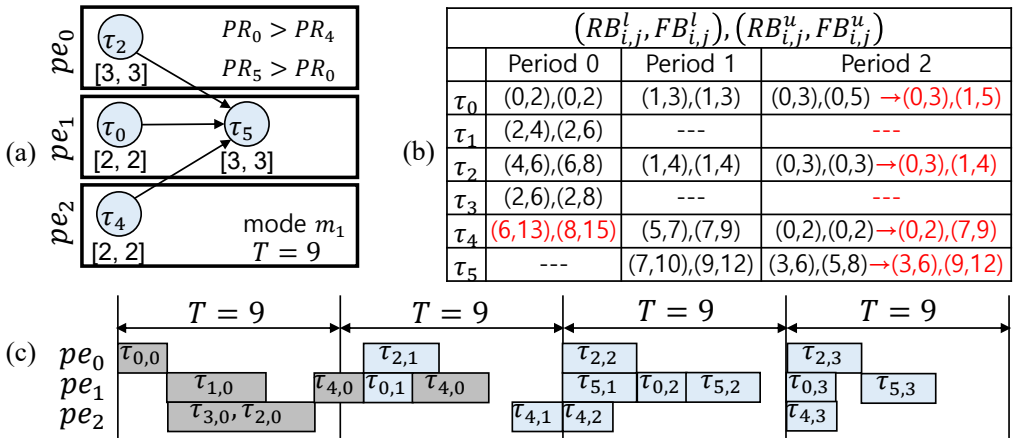


Figure 6.2: Worst-case performance analysis for a single mode transition

periods. W repeat the analysis until the schedule time bounds of all tasks no longer grow. As shown in Figure 6.1 (c), the extended analysis may not be terminated if the schedule time bounds fluctuate for each period extension. This problem can be solved by modifing the schedule time bounds of each period to always include the schedule time bounds of the previous period, as shown as the modified schedule time bounds of job instances in period 1 and 2 in Figure 6.1 (b).

6.2.2 Step 2: WCRT Analysis for One Mode Transition

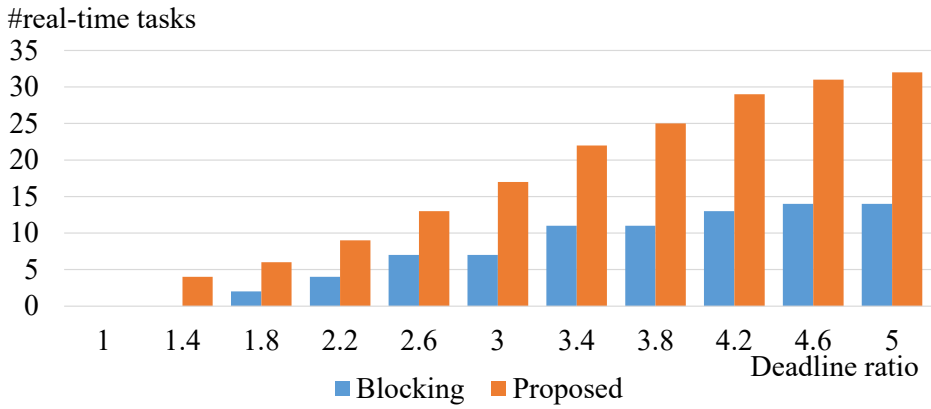
After passing the first step in subsection 6.2.1, we perform the worst-case performance analysis for one occurrence of each mode transition $(m_p, m_n) \in Trans$. The basic flow is almost the same as in subsection 6.2.1. In this step, the first period is generated from the dataflow of m_p, and the consecutive periods are expanded from the dataflow of m_n. To guarantee the worst-case transition from the previous mode m_p, the schedule time bounds of tasks in m_p is initially set to the converged bounds in the previous analysis, which cover time bounds of all iterations. The schedule time bounds of tasks in m_n are always compensated to include the schedule time bounds of the previous job instance to prevent oscillation of the time bounds. Figure 6.2 (c) shows executions of job instances

when mode transition occurs from m_0 in Figure 6.1 (a) to m_1 in Figure 6.2 (a). Figure 6.2 (b) summarizes the schedule time bounds of three periods when the mode transition occurs at the second period. Note that we need to consider the processor migration cost when the mode transition occurs. If τ_i belongs to both modes and the processor mapping is changed, the job instance of τ_i in the first m_n mode execution becomes ready only when the migration cost MC_i is paid after the completion time of the τ_i job instance in the previous m_p mode execution. In Figure 6.2 (c), $\tau_{0,1}$ and $\tau_{2,1}$ is migrated at the time of the mode transition since they finish earlier than the mode transition, while $\tau_{4,1}$ is migrated at the finishing time of $\tau_{4,0}$.

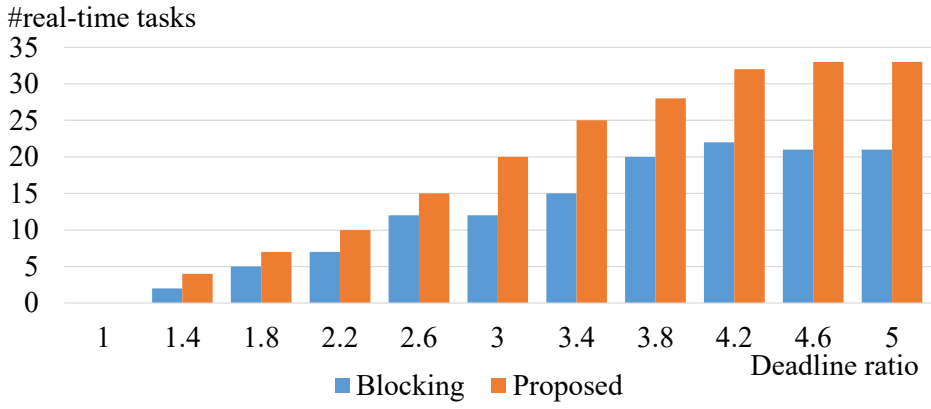
6.2.3 Step 3: WCRT Analysis for Repeated Mode Transitions

If the procedure in subsection 6.2.2 is passed successfully, it can be guaranteed that the deadline is satisfied for one occurrence of each mode transition (m_p, m_n). If mode transition occurs repeatedly, the worst-case response time may be further increased due to the accumulation of interference from mode transitions. instead of testing all possible mode transition scenarios, we use a conservative approach to estimate the worst-case response time.

If we analyze (m_p, m_n) for one mode transition in the procedure of subsection 6.2.2, the schedule time bounds of m_n will increase considering the interference caused by interference from mode m_p. Once all mode transitions have been analyzed, the extended schedule time bounds of all modes are analyzed. Now, we repeat the procedure in subsection 6.2.2 again. When analyzing each mode transition (m_p, m_n), we set the schedule time bounds of the previous mode m_p to include the schedule time bounds recorded in the previous iteration. Since m_p has the largest schedule time bounds that accumulate the interference from multiple mode transitions, we can analyze the worst-case performance of the repeated mode transition from the worst-case schedule bounds of m_p. By repeating this process until the time bounds are converged, we can analyze the worst-case



(a) Experimental result of H.264 decoder



(b) Experimental result of MP3 decoder

Figure 6.3: Experimental result for two benchmarks H.264 decoder and MP3 decoder

performance conservatively for repeated mode transitions.

6.3 Experiments

To verify the performance of the proposed technique, we conduct experiments with two benchmark H.264 decoder and MP3 decoder in [29]. The mapping and scheduling of each benchmark are determined by the technique proposed in [29]. In the experiment, we create a set of randomly generated real-time tasks and use a genetic algorithm (GA) to find the maximum number of real-time tasks that can share the system with the benchmark example while satisfying the deadline constraints. A chromosome is designed to

indicate the real-time tasks to be included in the system. Then fitness of the chromosomes is computed as the number of real-time tasks that are included when the deadline satisfaction is confirmed through the proposed technique. In the experiment, we compare the proposed technique with the model "Blocking" approach that the next mode becomes ready to execute only after all job instances in the previous mode finish.

We use following random configuration. The number of real-time tasks is 100, and the execution time is determined in the range of [50, 1000]. The period and deadline of the real-time tasks are equal and given in the range [10000, 50000]. The period of the benchmark is determined as the schedule latency without real-time tasks, and the deadline of the benchmark is fixed to $D = 2T$.

Figure 6.3 shows the experimental results obtained for two benchmarks. We vary the period and deadline of the benchmark by multiplying the ratio listed in x-axis. The y-axis represents the maximum number of real-time tasks found by the GA. Both benchmarks show that the blocking method includes fewer tasks, due to the fact that blocking method results in the larger response time. In the case of MP3 decoder, the performance difference is slightly small since the given schedule is not well pipelined.

Chapter 7

Conclusion

In this dissertation, we addressed a very challenging problem that is to tightly estimate the worst-case response time of an application that is given as a task graph in a distributed embedded system. We propose a hybrid performance analysis (HPA) method that combines two techniques ingeniously: the scheduling time bound analysis for intra-graph interference analysis and the response time analysis for inter-graph interference analysis. It finds a conservative and tight WCRT bound, considering task dependency, execution time variation, an arbitrary mixture of fixed-priority preemptive and non-preemptive processing elements, and input jitters. Experimental results show that it produces tighter bounds than the state-of-the-art techniques such as STBA, MAST, and pyCPA. And the proposed technique is scalable in terms of the number of tasks and faster than the other approaches.

Next, we propose a novel conservative modeling technique of shared resource contention supporting dependent tasks. For independent tasks, the proposed technique improves estimation performance significantly by considering the scheduling pattern of tasks, especially the maximum number of instances and the maximum execution time in a given time window. For dependent tasks, we identify the tasks that will not incur any access contention in any case by analyzing the schedule distance between tasks. And we cluster tasks that will be scheduled in a fixed order in each processing element. Finally

we obtain the PE-level SR demand bound by finding the worst combination of the SR demand from task clusters in each PE. In the experiments, the proposed technique shows performance improvement of 24.62% on average for random examples. Through experiments we investigated how the performance is affected by the resource access pattern. Experimental results with synthetic examples and a real-life example confirm the significance of considering the data dependency in the modeling of shared resource contention. At last, we studied the industrial-strength engine management benchmark to declare the applicability of the proposed shared resource contention analysis.

In Chapter 5, we propose a novel optimization technique of fault-tolerant mixed-criticality multi-core systems with enhanced WCRT guarantees and partial task dropping. We improve the analysis in order to reduce the pessimism of WCRT estimates by considering the maximum number of faults to be *maximally* tolerated. Being guided by the enhanced WCRT estimation, we could better explore the design space of hardening, mapping/scheduling, and quality-of-service of the multi-core mixed-criticality systems. The effectiveness of the proposed technique is verified by extensive experiments with synthetic and real-life benchmarks.

Finally, we propose a method for analyzing the worst-case performance when MMDF application is executed in multiprocessor with other real-time tasks. We propose a technique to conservatively analyze the worst-case response time of an MMDF considering the accumulated mode-to-mode interference. We conduct an experiment that finds the maximum number of real-time tasks that can share the system with the benchmark example while satisfying the deadline constraints.

There are many future research topics to extend the proposed technique for more general cases, eg., global scheduling, shared resource with other arbitration policies, multiple levels of criticality, etc. The proposed WCRT analysis for an MMDF application has a restriction that it does not support a system with multiple MMDFs, which is also left as a future work.

Bibliography

[1] Aske Brekling, Michael R. Hansen, and Jan Madsen. Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(1–2):1 – 19, 2008. The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006).

[2] J. Kim, H. Oh, H. Ha, S. H. Kang, J. Choi, and S. Ha. An ilp-based worst-case performance analysis technique for distributed real-time embedded systems. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 363–372, Dec 2012.

[3] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System level performance analysis–the symta/s approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

[4] Kai Lampka, Simon Perathoner, and Lothar Thiele. Analytic real-time analysis and timed automata: a hybrid methodology for the performance analysis of embedded real-time systems. *Design Automation for Embedded Systems*, 14(3):193–227, 2010.

[5] Hoeseok Yang, Sungchan Kim, and Soonhoi Ha. An milp-based performance analysis technique for non-preemptive multitasking mpsoc. *IEEE transactions on computer-aided design of integrated circuits and systems*, 29(10):1600–1613, 2010.

[6] Jinwoo Kim, Hyunok Oh, Junchul Choi, Hyojin Ha, and Soonhoi Ha. A novel analytical method for worst case response time estimation of distributed embedded systems. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–10, May 2013.

[7] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, Dec 1989.

[8] Mircea Negrean, Simon Schliecker, and Rolf Ernst. Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In

Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09, pages 524–529, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[9] M. Negrean and R. Ernst. Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, pages 191–200, June 2012.

[10] Simon Schliecker and Rolf Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, January 2011.

[11] Cristian Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE micro*, 23(4):14–19, 2003.

[12] H. Gall. Functional safety iec 61508 / iec 61511 the impact to certification and the user. In *2008 IEEE/ACS International Conference on Computer Systems and Applications*, pages 1027–1031, March 2008.

[13] International Organization for Standardization. International standard 26262: Road vehicles functional safety, 2011.

[14] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems. In *Design, Automation and Test in Europe*, pages 864–869 Vol. 2, March 2005.

[15] Philip Axer, Maurice Sebastian, and Rolf Ernst. Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 149–158. ACM, 2011.

[16] Peter van Stralen and Andy Pimentel. A safe approach towards early design space exploration of fault-tolerant multimedia mpsocs. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 393–402. ACM, 2012.

[17] C. Bolchini and A. Miele. Reliability-driven system-level synthesis for mixed-critical embedded systems. *IEEE Transactions on Computers*, 62(12):2489–2502, Dec 2013.

[18] J. Huang, J. O. Blech, A. Raabe, C. Buckl, and A. Knoll. Reliability-aware design optimization for multiprocessor embedded systems. In *2011 14th Euromicro Conference on Digital System Design*, pages 239–246, Aug 2011.

[19] A. Jhumka, S. Klaus, and S. A. Huss. A dependability-driven system-level design approach for embedded systems. In *Design, Automation and Test in Europe*, pages 372–377 Vol. 1, March 2005.

[20] Pengcheng Huang, Hoeseok Yang, and Lothar Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 131:1–131:6, New York, NY, USA, 2014. ACM.

[21] S. H. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele. Reliability-aware mapping optimization of multi-core systems with mixed-criticality. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–4, March 2014.

[22] Shin-haeng Kang, Hoeseok Yang, Sungchan Kim, Iuliana Bacivarov, Soonhoi Ha, and Lothar Thiele. Static mapping of mixed-critical applications for fault-tolerant mpsocs. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 31:1–31:6, New York, NY, USA, 2014. ACM.

[23] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

[24] Sanjoy K Baruah, Vincenzo Bonifaci, Gianlorenzo D?Angelo, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *European Symposium on Algorithms*, pages 555–566. Springer, 2011.

[25] Luyuan Zeng, Pengcheng Huang, and Lothar Thiele. Towards the design of fault-tolerant mixed-criticality systems on multicores. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 6. ACM, 2016.

[26] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. Pyramid methods in image processing. *RCA engineer*, 29(6):33–41, 1984.

[27] Yash Vardhan Pant, Houssam Abbas, Kartik Mohta, Truong X Nghiem, Joseph Devietti, and Rahul Mangharam. Co-design of anytime computation and robust control. In *Real-Time Systems Symposium, 2015 IEEE*, pages 43–52. IEEE, 2015.

[28] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 404–411, July 2011.

[29] Hanwoong Jung, Hyunok Oh, and Soonhoi Ha. Multiprocessor scheduling of a multi-mode dataflow graph considering mode transition delay. *ACM Trans. Des. Autom. Electron. Syst.*, 22(2):37:1–37:25, January 2017.

[30] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209, Dec 1990.

[31] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, April 1994.

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

[33] J. C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 26–37, Dec 1998.

[34] E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, Nov 2004.

[35] J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*, pages 328–339, 1999.

[36] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 149–160, Dec 2007.

[37] Nan Guan, M. Stigge, Wang Yi, and Ge Yu. New response time bounds for fixed priority multiprocessor scheduling. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 387–397, Dec 2009.

[38] Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536 – 546, 2011. Special Issue on Multiprocessor Real-time Scheduling.

[39] K. Tindell, A. Burns, and A. J. Wellings. Analysis of hard real-time communications. *Real-Time Syst.*, 9(2):147–171, September 1995.

[40] Rodolfo Pellizzoni and Giuseppe Lipari. Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling. *Journal of Computer and System Sciences*, 73(2):186 – 206, 2007. Special Issue: Real-time and Embedded Systems.

[41] B. Bodin, A. Munier-Kordon, and B.D. de Dinechin. Periodic schedules for cyclo-static dataflow. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on*, pages 105–114, Oct 2013.

[42] Mohamed Bamakhrama and Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 195–204, New York, NY, USA, 2011. ACM.

[43] Mohamed A. Bamakhrama and Todor P. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Des. Autom. Embedded Syst.*, 17(2):221–249, June 2013.

[44] J. Spasic, Di Liu, E. Cannella, and T. Stefanov. Improved hard real-time scheduling of csdf-modeled streaming applications. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2015 International Conference on*, pages 65–74, Oct 2015.

[45] H.I. Ali, B. Akesson, and L.M. Pinho. Generalized extraction of real-time parameters for homogeneous synchronous dataflow graphs. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 701–710, March 2015.

[46] Michael GonzÃlez Harbour. Mast: Modeling and anlysis suite for real-time applications, 2001.

[47] R. Henia and R. Ernst. Improved offset-analysis using multiple timing-references. In *Proceedings of the Design Automation Test in Europe Conference*, volume 1, pages 6 pp.–, March 2006.

[48] P. S. Kurtin, J. P. H. M. Hausmans, and M. J. G. Bekooij. Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[49] J. Schlatow and R. Ernst. Response-time analysis for task chains in communicating threads. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–10, April 2016.

[50] N. Guan, C. Gu, M. Stigge, Q. Deng, and W. Yi. Approximate response time analysis of real-time task graphs. In *2014 IEEE Real-Time Systems Symposium*, pages 304–313, Dec 2014.

[51] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the symta/s approach. *IEE Proceedings - Computers and Digital Techniques*, 152(2):148–166, Mar 2005.

[52] B.B. Brandenburg, J.M. Calandrino, A. Block, H. Leontyev, and J.H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pages 342–353, April 2008.

[53] Bjorn B. Brandenburg. Improved analysis and evaluation of real-time semaphore protocols for p-fp scheduling. In *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '13, pages 141–152, Washington, DC, USA, 2013. IEEE Computer Society.

[54] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, January 2012.

[55] Alexander Wieder and Björn B. Brandenburg. On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Proceedings of the 2013 IEEE 34th Real-Time Systems Symposium*, RTSS '13, pages 45–56, Washington, DC, USA, 2013. IEEE Computer Society.

[56] R. Mancuso, R. Pellizzoni, M. Caccamo, Lui Sha, and Heechul Yun. Wcet(m) estimation in multi-core systems using single core equivalence. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 174–183, July 2015.

[57] B.C. Ward, J.L. Herman, C.J. Kenna, and J.H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 157–167, July 2013.

[58] A. Schranzhofer, R. Pellizzoni, J. J. Chen, L. Thiele, and M. Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *Design Automation Conference*, pages 332–337, June 2010.

[59] Hyoseung Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 145–154, April 2014.

[60] Sebastian Altmeyer, Robert I. Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. A generic and compositional framework for multicore response time analysis. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, pages 129–138, New York, NY, USA, 2015. ACM.

[61] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *2009 30th IEEE Real-Time Systems Symposium*, pages 57–67, Dec 2009.

[62] D. Dasari and V. Nelis. An analysis of the impact of bus contention on the wcet in multicores. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1450–1457, June 2012.

[63] Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, pages 1–51, 2015.

[64] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 741–746, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[65] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 239–243, Washington, DC, USA, 2007. IEEE Computer Society.

[66] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, RTSS '11, pages 34–43, Washington, DC, USA, 2011. IEEE Computer Society.

[67] S. Baruah and B. Chattopadhyay. Response-time analysis of mixed criticality systems with pessimistic frequency specification. In *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 237–246, Aug 2013.

[68] S. Baruah and Z. Guo. Scheduling mixed-criticality implicit-deadline sporadic task systems upon a varying-speed processor. In *2014 IEEE Real-Time Systems Symposium*, pages 31–40, Dec 2014.

[69] M. Jan, L. Zaourar, and M. Pitel. Maximizing the execution rate of low criticality tasks in mixed criticality system. In *Proc. WMC RTSS*, pages 43–48, 2013.

[70] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Fault tolerant scheduling of mixed criticality real-time tasks under error bursts. *Procedia Computer Science*, 46:1148 – 1155, 2015.

[71] A. Thekkilakattil, R. Dobrin, and S. Punnekkat. Mixed criticality scheduling in fault-tolerant distributed real-time systems. In *2014 International Conference on Embedded Systems (ICES)*, pages 92–97, July 2014.

[72] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 97–102, March 2016.

[73] J. Lin, A. M. K. Cheng, D. Steel, and M. Y.-C. Wu. Scheduling mixed-criticality real-time tasks with fault tolerance. In *Proc. 2nd Workshop Mixed Criticality Conjuction IEEE RTSS*, 2014.

[74] Biao Hu, Kai Huang, Pengcheng Huang, Lothar Thiele, and Alois Knoll. On-the-fly fast overrun budgeting for mixed-criticality systems. In *Embedded Software (EMSOFT), 2016 International Conference on*, pages 1–10. IEEE, 2016.

[75] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger. Tolerating transient faults in mars. In *[1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*, pages 466–473, June 1990.

[76] A. Burns, R. Davis, and S. Punnekkat. Feasibility analysis of fault-tolerant real-time task sets. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 29–33, Jun 1996.

[77] Jia Huang, Andreas Raabe, Kai Huang, Christian Buckl, and Alois Knoll. A framework for reliability-aware design exploration on mpsoc based systems. *Design Automation for Embedded Systems*, 16(4):189–220, 2012.

[78] Junlong Zhou, Min Yin, Zhifang Li, Kun Cao, Jianming Yan, Tongquan Wei, Mingsong Chen, and Xin Fu. Fault-tolerant task scheduling for mixed-criticality real-time systems. *Journal of Circuits, Systems and Computers*, 26(01):1750016, 2017.

[79] Dakai Zhu, Rami Melhem, and Daniel Mossé. The effects of energy management on reliability in real-time embedded systems. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 35–40. IEEE, 2004.

[80] L. T. X. Phan, S. Chakraborty, and P. S. Thiagarajan. A multi-mode real-time calculus. In *2008 Real-Time Systems Symposium*, pages 59–69, Nov 2008.

[81] Junchul Choi and Soonhoi Ha. Worst-case response time analysis of a synchronous dataflow graph in a multiprocessor system with real-time tasks. *ACM Trans. Des. Autom. Electron. Syst.*, 22(2):36:1–36:26, January 2017.

[82] Ti-Yen Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1125–1136, Nov 1998.

[83] Jonas Diemer and Philip Axer. Python implementation of compositional performance analysis, 2012.

[84] Sander Stuijk, Marc Geilen, and Twan Basten. Sdf3:sdf for free, 2006.

[85] T. Henriksson, P. van der Wolf, A. Jantsch, and A. Bruce. Network calculus applied to verification of memory access performance in socs. In *Embedded Systems for Real-Time Multimedia, 2007. ESTIMedia 2007. IEEE/ACM/IFIP Workshop on*, pages 21–26, Oct 2007.

[86] S. Kang and S. Ha. Tqsim: Timed qemu-based simulator, Jan 2013.

[87] Chunho Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on*, pages 330–335, Dec 1997.

[88] 2016 formals methods and timing verification (fmtv) challenge, co-located with the 7th waters. https://waters2016.inria.fr/challenge/.

[89] AUTOSAR. Autosar specification of rte software.

[90] AUTOSAR. Autosar specification of timing extensions.

[91] Hyung-Chan An, Hoeseok Yang, and Soonhoi Ha. A formal approach to power optimization in cpss with delay-workload dependence awareness. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 35(5):750–763, May 2016.

[92] M. Damschen, L. Bauer, and J. Henkel. Timing analysis of tasks on runtime reconfigurable processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(1):294–307, Jan 2017.

[93] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. Control-flow checking by software signatures. *IEEE transactions on Reliability*, 51(1):111–122, 2002.

[94] Yi Xiang and Sudeep Pasricha. A hybrid framework for application allocation and scheduling in multicore systems with energy harvesting. In *Proceedings of the 24th edition of the great lakes symposium on VLSI*, pages 163–168. ACM, 2014.

[95] Thidapat Chantem, X Sharon Hu, and Robert P Dick. Temperature-aware scheduling and assignment for hard real-time applications on mpsocs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(10):1884–1897, 2011.

[96] Rabeh Ayari, Imane Hafnaoui, Giovanni Beltrame, and Gabriela Nicolescu. Schedulability-guided exploration of multi-core systems. In *Proceedings of the 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype*, pages 121–127. ACM, 2016.

[97] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[98] Mark Boddy and Thomas Dean. Solving time-dependent planning problems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'89, pages 979–984, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[99] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 09 1996.

[100] Arnaud Giacometti and Arnaud Soulet. Anytime algorithm for frequent pattern outlier detection. *International Journal of Data Science and Analytics*, 2(3):119–130, Dec 2016.

[101] Benjamin Arai, Gautam Das, Dimitrios Gunopulos, and Nick Koudas. Anytime measures for top-k algorithms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 914–925. VLDB Endowment, 2007.

[102] Nagarajan Kandasamy, John P. Hayes, and Brian T. Murray. Dependable communication synthesis for distributed embedded systems. *Reliability Engineering & System Safety*, 89(1):81 – 92, 2005. Safety, Reliability and Security of Industrial Computer Systems.

[103] G. Madl et al. Tutorial for the open-source dream tool. In *Univ. California, Irvine, CA, CECS Tech. Rep*, 2006.

[104] Martin Lukasiewycz, Michael Glaß, Felix Reimann, and Jürgen Teich. Opt4j: A modular framework for meta-heuristic optimization. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1723–1730, New York, NY, USA, 2011. ACM.

[105] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm. In *EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control With Applications to Industrial Problems*, Sep 2001.

[106] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

요 약

실시간 응용을 실행하기 위한 내장형 시스템을 설계할 때 응용이 가진 실시간 제약조건이 항상 보장되는지 판단해야만 한다. 응용이 후보 시스템 아키텍쳐에 배치된 후에 시스템의 성능을 예측함으로써 이러한 실시간 제약조건의 만족 여부를 판단할 수 있다. 성능 예측 속도가 빠를수록 제한된 시간 안에 더 많은 시스템 아키텍쳐와 응용 배치를 탐색해볼 수 있고, 성능 예측의 정확도가 높을수록 시스템의 최소 필요 성능을 정확히 예측할 수 있어 시스템 구축 비용을 절약할 수 있다. 그러나 분산 내장형 시스템에서 특히 응용 내의 작업들 간에 복잡한 실행 순서 관계가 존재하는 경우 응용의 최악 응답 시간을 정확하면서도 빠르게 예측하는 것은 여전히 어려운 문제로 남아있다.

단일 코어의 속도가 한계점에 도달하면서 싱글코어 시스템으로는 더 이상 점진적인 성능 증가를 기대할 수 없게 되었기에, 멀티코어 시스템은 범용 시스템에서 뿐만 아니라 실시간 내장형 시스템에서도 대중적으로 사용되는 설계 요소가 되었다. 일반적으로 멀티코어 시스템의 설계에서는 L2 캐시, 공유 메모리, 그리고 다양한 주변 장치들을 코어들이 공유하게 된다. 응용들이 여러 코어에서 실행될 때 이러한 시스템 요소들에 동시적으로 접근이 발생할 수 있기 때문에 동적 시간에 비결정적으로 발생하는 동시 접근들에 의해서 응용은 비확적정인 접근 지연을 겪을 수 있다. 응용들이 공유하는 자원이 존재하는 시스템에서의 최악 응답 시간 분석에서는 실시간성의 보장을 위하여 이러한 비확정적인 접근 지연을 보수적으로 고려할 필요가 있다.

반도체 생산 기술의 지속적인 발전으로 시스템이 저전력/고주파수 환경에서 실행됨에 따라 결함 동작 확률이 점점 높아져 시스템의 신뢰성의 보장 또한 중요한 이슈가 되고 있다. 시스템의 결함은 크게 영구적 결함과 일시적 결함으로 분류할 수 있는데, 영구적 결함이 발생한 경우 시스템의 일부분이 영구적으로 손상을 입어 사용이 불가능해지는 반면, 일시적 결함의 경우 시스템의 기능 일부가 일시적으로 오동작을 일으킬 수 있으며, 이러한 일시적 결함이 응용 실행 오류의 주요 원인이라는 것으로 알려져 있다. 다양한 신뢰성 요구조건을 가진 응용들이 실행되는 혼합 안전도 시스템에서는 결함이 발생했을 때애도 신뢰성 조건을 만족시키기 위하여 일반적으로 더 많은 하드웨어 자원을 사용하거나 실행 오류 지점을 재실행하는 방식으로 실행 오류를 피하고

신뢰성을 보장하고 있다.

시스템이 복잡해짐에 따라 동적 상황에서 응용의 동작이 변화하도록 응용이 설계될 수 있다. 예를 들면, 영상 복호화 작업은 입력으로 들어오는 암호화된 영상 프레임의 종류에 따라서 다른 동작을 실행할 수 있다. 또한 가용 자원이 부족한 경우 응용들의 서비스 품질 요구사항에 맞춰 시스템이 특정 응용들을 저품질로 동작하도록 변경할 수 있다. 이렇게 동적으로 변화하는 응용이 존재하면 이를 고려한 최악 응답 시간 분석이 필요하게 된다.

실시간 분산 내장형 시스템에서 실행되는 응용의 실시간성을 보장하기 위하여 이 논문에서는 그래프 구조로 내부 작업의 순서가 정의된 응용의 최악 성능을 예측하는 기술들을 제안한다. 먼저 이 논문은 기반 기술로써 응답시간 분석법 (response time analysis, RTA) 과 스케쥴 시간 경계 분석법 (schedule time bound analysis, STBA)을 혼합한 혼합 성능 분석법 (hybrid performance analysis, HPA)을 제안한다. 제안하는 혼합 접근법은 그래프 더 빠르고 정확한 예측을 위하여 그래프 내부 작업간의 간섭과 그래프 외부로부터의 간섭을 다르게 분석한다. 그래프 내부 작업간들 간의 선점 발생 여부는 STBA를 사용하여 분석하는 반면, 다른 그래프 응용으로부터의 간섭은 RTA에 기반한 기법을 통해 분석한다.

기반 최악 성능 분석 기술을 사용하여 멀티코어 실시간 내장현 시스템이 가지는 여러 설계 요소들을 고려한 세가지 확장 기술을 제안한다. 첫번째 확장 기술은 공유 자원 접근 지연 시간의 상한선을 모델링하는 방법이다. 제안하는 기법은 각 코어로부터 발생할 수 있는 최악의 자원 접근 요청량을 이벤트 스트림 모델 (event stream model)을 이용하여 모델링하며, 작업들의 스케쥴 패턴과 작업간의 의존관계를 고려하여 정밀하게 자원 접근 요청량을 예측한다. 또한 제안하는 공유 자원 접근 지연 시간 분석법의 효용성을 보여주기 위하여 Bosch GmbH에서 실제 엔진 관리 시스템을 프로파일링하여 제공한 벤치마크에 대해 분석을 진행한다.

두번째 확장 기술로 결함 감내 혼합 안전도 시스템을 고려한 최악 응답 시간 분석 기술을 제안한다. 일반적으로 결함 감내 시스템에서는 응용의 작업들은 신뢰성을 향상시키기 위해 결함 여부를 판단 후 재실행하거나, 복제본을 여러 하드웨어에서 실

행하는 방식을 사용한다. 또한 혼합 안전도의 스케쥴러 정책에 따라서 낮은 안전도 요구사항을 가진 작업들의 경우 상황에 따라 실행이 취소될 수 있다. 이러한 결함 감내 기법이나 혼합 안전도 스케쥴링 정책에 의한 불확정성은 최악 응답 시간 분석을 매우 어렵게 만들게 된다. 제안하는 기법은 기존 기술들이 가지는 지나친 보수성을 줄이기 위하여 응용이 신뢰성 조건을 만족하기 위해 응답 시간 동안 전체적으로 감내해야할 최대 결함 수를 고려하는 기법을 제안한다. 또한 낮은 안전도를 가진 태스크의 부분 실행을 지원하는 혼합 안전 스케쥴링을 지원하도록 기법을 확장한다. 이 기법을 이용하여 응용들의 결함 기법, 스케쥴링, 서비스 품질 수준의 최적 해를 탐색하는 설계 공간 탐색을 진행한다.

마지막 확장 기술은 동적으로 동작이 변화하는 응용에 대한 최악 응답 시간 분석 기법이다. 응용은 모드 전환이 일어날 때 다른 데이터플로우 (dataflow)로 실행이 전환되는 멀티모드 데이터플로우 (multi-mode dataflow, MMDF)로 모델링됨을 가정한다. 또한 모드 전환 시에 작업이 다른 코어로 매핑이 바뀌는 경우 이주 비용 (migration cost)이 발생하는 것을 고려한다. 제안하는 기법은 모드 전환시에 서로 다른 두 개의 모드가 동시에 실행됨으로써 발생하는 모드간 간섭을 고려하여 전체적인 멀티모드 응용의 최악 응답 시간을 분석한다.

제안하는 각 기법들의 성능은 벤치마크 응용들과 랜덤 생성된 합성 예제들을 이용하여 기존 기법들과의 비교를 통해 확인한다.

주요어 : 멀티코어 시스템, 실시간 시스템, 성능 분석, 응답 시간 분석, 최악 응답 시간, 분할 스케줄링, 태스크 그래프, 데이터 의존성, 공유 자원, 결함 감내, 혼합 안전성
학번 : 2014-30320